Advanced Topics in Types and Programming Languages

Advanced Topics in Types and Programming Languages

Benjamin C. Pierce, editor

The MIT Press
Cambridge, Massachusetts
London, England

Contents

Preface

Brief overview of the whole book.

0.0.1 Acknowledgements

Amal Ahmed, Karl Crary, Derek Dreyer, Matthias Felleisen, Robby Findler, Kathleen Fisher, Nadji Gauthier, Michael Hicks, Xavier Leroy, William Lovas, Yitzhak Mandelbaum, Martin Müller, Simon Peyton Jones, Alan Schmitt, Peter Sewell, Vincent Simonet, Eijiro Sumii, David Swasey, Joe Vanderwaart, Yanling Wang, Geoff Washburn, Keith Wansbrough, Dinghao Wu,

PART I

ML-style Systems

1 *The Essence of ML Type Inference*

By François Pottier and Didier Rémy

1.1 Preliminaries

1.1.1 Names and renaming

Mathematicians and computer scientists use *names* to refer to arbitrary or unknown objects in the statement of a theorem, to refer to the parameters of a function, *etc.* Names are convenient because they are understandable by humans; nevertheless, they can be tricky. An in-depth treatment of the difficulties associated with names and renaming is beyond the scope of the present chapter: we encourage the reader to study Gabbay and Pitts' excellent series of papers (Gabbay and Pitts, 2002; Pitts, 2002b). Here, we merely recall a few notions that are used throughout this chapter. Consider, for instance, an inductive definition of the abstract syntax of a simple programming language, the pure λ-calculus:

$$\mathtt{t} ::= \mathtt{z} \mid \lambda \mathtt{z}.\mathtt{t} \mid \mathtt{t}\ \mathtt{t}$$

Here, the *meta-variable* $\mathtt{z}$ ranges over an infinite set of *variables*—that is, names—while the meta-variable $\mathtt{t}$ ranges over *terms*. As usual in mathematics, we write "the variable $\mathtt{z}$" and "the term $\mathtt{t}$" instead of "the variable denoted by $\mathtt{z}$" and "the term denoted by $\mathtt{t}$". The above definition states that a term may be a variable $\mathtt{z}$, a pair of a variable and a term, written $\lambda\mathtt{z}.\mathtt{t}$, or a pair of terms, written $\mathtt{t}_1\ \mathtt{t}_2$. However, this is not quite what we need. Indeed, according to this definition, the terms $\lambda\mathtt{z}_1.\mathtt{z}_1$ and $\lambda\mathtt{z}_2.\mathtt{z}_2$ are distinct, while we would like them to be a single mathematical object, because we intend $\lambda\mathtt{z}.\mathtt{z}$ to mean "the function that maps $\mathtt{z}$ to $\mathtt{z}$"—a meaning that is independent of the name $\mathtt{z}$. To achieve this effect, we complete the above definition by stat-

---

Code for this chapter may be found at `http://pauillac.inria.fr/~remy/mlrow/`.

ing that the construction $\lambda z.t$ *binds* z within t. One may also say that λz is a *binder* whose *scope* is t. Then, $\lambda z.t$ is no longer a pair: rather, it is an *abstraction* of the variable z within the term t. Abstractions have the property that the identity of the bound variable does not matter; that is, $\lambda z_1.z_1$ and $\lambda z_2.z_2$ are the same term. Informally, we say that terms are considered equal modulo α-*conversion*. Once the position and scope of binders are known, several standard notions follow, such as the set of *free variables* of a term t, written $fv(t)$, and the *capture-avoiding substitution* of a term t_1 for a variable z within a term t_2, written $[z \mapsto t_1]t_2$. For conciseness, we write $fv(t_1, t_2)$ for $fv(t_1) \cup fv(t_2)$. A term is said to be *closed* when it has no free variables.

A *renaming* is a total bijective mapping from variables to variables that affects only a finite number of variables. The sole property of a variable is its identity, that is, the fact that it is distinct from other variables. As a result, at a global level, all variables are interchangeable: if a theorem holds in the absence of hypotheses about any particular variable, then any renaming of it holds as well. We often make use of this fact. When proving a theorem T, we say that a hypothesis H may be assumed *wihout loss of generality* (*w.l.o.g.*) if the theorem T follows from the theorem $H \Rightarrow T$ via a renaming argument, which is usually left implicit.

If $\bar{z}_1$ and $\bar{z}_2$ are sets of variables, we write $\bar{z}_1 \# \bar{z}_2$ as a shorthand for $\bar{z}_1 \cap \bar{z}_2 = \varnothing$, and say that $\bar{z}_1$ is *fresh* for $\bar{z}_2$ (or vice-versa). We say that $\bar{z}$ is fresh for t if and only if $\bar{z} \# \mathrm{fv}(t)$ holds.

In this chapter, we work with several distinct varieties of names: program variables, memory locations, and type variables, the latter of which may be further divided into *kinds*. We draw names of different varieties from disjoint sets, each of which is infinite.

1.2 What is ML?

The name "ML" appeared during the late seventies. It then referred to a general-purpose programming language that was used as a meta-language (whence its name) within the theorem prover LCF (Gordon, Milner, and Wadsworth, 1979b). Since then, several new programming languages, each of which offers several different implementations, have drawn inspiration from it. So, what does "ML" stand for today?

For a semanticist, "ML" might stand for a programming language featuring first-class functions, data structures built out of products and sums, mutable memory cells called *references*, exception handling, automatic memory management, and a call-by-value semantics. This view encompasses the Standard ML (Milner, Tofte, and Harper, 1990) and Caml (Leroy, 2000) fami-

lies of programming languages. We refer to it as *ML-the-programming-language*.

For a type theorist, "ML" might stand for a particular breed of type systems, based on the simply-typed λ-calculus, but extended with a simple form of polymorphism introduced by `let` declarations. These type systems have decidable type inference; their type inference algorithms crucially rely on first-order unification and can be made efficient in practice. In addition to Standard ML and Caml, this view encompasses programming languages such as Haskell (Hudak, Peyton Jones, Wadler, Boutel, Fairbairn, Fasel, Guzman, Hammond, Hughes, Johnsson, Kieburtz, Nikhil, Partain, and Peterson, 1992) and Clean (Brus, van Eekelen, van Leer, and Plasmeijer, 1987), whose semantics is rather different—indeed, it is pure and lazy—but whose type system fits this description. We refer to it as *ML-the-type-system*. It is also referred to as *Hindley and Milner's type discipline* in the literature.

For us, "ML" might also stand for the particular programming language whose formal definition is given and studied in this chapter. It is a core calculus featuring first-class functions, local definitions, and constants. It is equipped with a call-by-value semantics. By customizing constants and their semantics, one may recover data structures, references, and more. We refer to this particular calculus as *ML-the-calculus*.

Why study ML-the-type-system today, such a long time after its initial discovery? One may think of at least two reasons.

First, its treatment in the literature is often cursory, because it is considered either as a simple extension of the simply-typed λ-calculus (TAPL Chapter 9) or as a subset of Girard and Reynolds' System F (TAPL Chapter 23). The former view is supported by the claim that local definitions, which distinguish ML-the-type-system from the simply-typed λ-calculus, may be understood as a simple textual expansion facility. However, this view only tells part of the story, because it fails to give an account of the *principal types* property enjoyed by ML-the-type-system, leads to a naïve type inference algorithm whose time complexity is exponential even for non-contrived programs, and breaks down when the language is extended with side effects, such as state or exceptions. The latter view is supported by the fact that every type derivation within ML-the-type-system is also a valid type derivation within an implicitly-typed variant of System F. Such a view is correct, but again fails to give an account of type inference for ML-the-type-system, since type inference for System F is undecidable (Wells, 1999).

Second, existing accounts of type inference for ML-the-type-system (Milner, 1978; Damas and Milner, 1982; Tofte, 1988; Leroy, 1992; Lee and Yi, 1998; Jones, 1999) usually involve heavy manipulations of type substitutions. Such an ubiquitous use of type substitutions is often quite obscure. Furthermore, actual implementations of the type inference algorithm do *not* explicitly ma-

nipulate substitutions; instead, they extend a standard first-order unification algorithm, where terms are updated in place as new equations are discovered (Huet, 1976). Thus, it is hard to tell, from these accounts, how to write an efficient type inference algorithm for ML-the-type-system. Yet, in spite of the increasing speed of computers, efficiency remains crucial when ML-the-type-system is extended with expensive features, such as Objective Caml's object types (Rémy and Vouillon, 1998) or polymorphic methods (Garrigue and Rémy, 1999).

For these reasons, we believe it is worth giving an account of ML-the-type-system that focuses on *type inference* and strives to be at once *elegant* and *faithful* to an efficient implementation, such as Rémy's (1992a). In this presentation, we forego type substitutions and instead put emphasis on *constraints*, which offer a number of advantanges. First, constraints allow a modular presentation of type inference as the combination of a constraint generator and a constraint solver. Such a decomposition allows reasoning separately about *when* a program is correct, on the one hand, and *how* to check whether it is correct, on the other hand. It has long been standard in the setting of the simply-typed λ-calculus (TAPL Chapter 22). In the setting of ML-the-type-system, such a decomposition is provided by the reduction of typability problems to acyclic semi-unification problems (Henglein, 1993; Kfoury, Tiuryn, and Urzyczyn, 1994). This approach, however, was apparently never used in actual implementations of ML-the-programming-language, although it did find applications in the closely related area of program analysis (Fähndrich, Rehof, and Das, 2000). In this chapter, we give a constraint-based description of a "classic" implementation of ML-the-type-system, which is based on first-order unification and on a mechanism for creating and instantiating principal *type schemes*. Second, it is often natural to define and implement the solver as a constraint rewriting system. Then, the constraint language allows reasoning not only about correctness—is every rewriting step meaning-preserving?—but also about low-level implementation details, since constraints *are* the data structures manipulated throughout the type inference process. For instance, describing unification in terms of *multi-equations* (Jouannaud and Kirchner, 1991) allows reasoning about the sharing of nodes in memory, which a substitution-based approach cannot account for. Last, constraints are more general than type substitutions, and allow smoothly extending of ML-the-type-system with recursive types, rows, subtyping, first-order unification under a mixed prefix, and more.

Before delving into the details of this new presentation of ML-the-type-system, however, it is worth recalling its standard definition. Thus, in what follows, we first define the syntax and operational semantics of the programming language ML-the-calculus, and equip it with a type system, known as

x, y	::=	*Identifiers:*		m	*Memory location*
	z	*Variable*		$\lambda z.t$	*Function*
	m	*Memory location*		$c\ v_1 \ldots v_k$	*Data*
	c	*Constant*			$c \in \mathcal{Q}^+ \wedge k \leq a(c)$
t	::=	*Expressions:*		$c\ v_1 \ldots v_k$	*Partial application*
	x	*Identifier*			$c \in \mathcal{Q}^- \wedge k < a(c)$
	$\lambda z.t$	*Function*	$\mathcal{E}$	::=	*Evaluation Contexts:*
	$t\ t$	*Application*		$[]$	*Empty context*
	$\texttt{let}\ z = t\ \texttt{in}\ t$	*Local definition*		$\mathcal{E}\ t$	*Left side of an application*
v, w	::=	*Values:*		$v\ \mathcal{E}$	*Right side of an application*
	z	*Variable*		$\texttt{let}\ z = \mathcal{E}\ \texttt{in}\ t$	*Local definition*

Figure 1-1: Syntax of ML-the-calculus

Damas and Milner's type system.

1.2.1 ML-the-calculus

The syntax of ML-the-calculus is defined in Figure 1-1. It is made up of several syntactic categories.

Identifiers group several kinds of names that may be referenced in a program: variables, memory locations, and constants. We let x and y range over identifiers. *Variables*—sometimes also called *program variables* to avoid ambiguity—are names that may be bound to values using λ or `let` binding forms; in other words, they are names for function parameters or local definitions. We let z and f range over program variables. We sometimes write _ for a program variable that does not occur free within its scope: for instance, $\lambda\_.t$ stands for $\lambda z.t$, provided z is fresh for t. *Memory locations* are names that represent memory addresses. Memory locations never appear in *source programs*, that is, programs that are submitted to a compiler. They only appear during execution, when new memory blocks are allocated. *Constants* are fixed names for primitive values and operations, such as integer literals and integer arithmetic operations. Constants are elements of a finite or infinite set $\mathcal{Q}$. They are never subject to α-conversion. Program variables, memory locations, and constants belong to distinct syntactic classes and may never be confused.

The set of constants $\mathcal{Q}$ is kept abstract, so most of our development is independent of its concrete definition. We assume that every constant c has a nonnegative integer *arity* $a(c)$. We further assume that $\mathcal{Q}$ is partitioned into

subsets of *constructors* $\mathcal{Q}^+$ and *destructors* $\mathcal{Q}^-$. Constructors and destructors differ in that the former are used to *form* values, while the latter are used to *operate* on values.

1.2.1 EXAMPLE [INTEGERS]: For every integer n, one may introduce a nullary constructor $\hat{n}$. In addition, one may introduce a binary destructor $\hat{+}$, whose applications are written infix, so $t_1 \mathbin{\hat{+}} t_2$ stands for the double application $\hat{+}\, t_1\, t_2$ of the destructor $\hat{+}$ to the expressions t_1 and t_2. □

Expressions—also known as *terms* or *programs*—are the main syntactic category. Indeed, unlike procedural languages such as C and Java, functional languages, including ML-the-programming-language, suppress the distinction between expressions and statements. Expressions include identifiers, λ-abstractions, applications, and local definitions. The λ-*abstraction* $\lambda z.t$ represents the function of one parameter named z whose result is the expression t, or, in other words, the function that maps z to t. Note that the variable z is bound within the term t, so (for instance) $\lambda z_1.z_1$ and $\lambda z_2.z_2$ are the same object. The *application* $t_1\ t_2$ represents the result of calling the function t_1 with actual parameter t_2, or, in other words, the result of applying t_1 to t_2. Application is left-associative, that is, $t_1\ t_2\ t_3$ stands for $(t_1\ t_2)\ t_3$. The construct $\texttt{let}\ z = t_1\ \texttt{in}\ t_2$ represents the result of evaluating t_2 after binding the variable z to t_1. Note that the variable z is bound within t_2, but not within t_1, so for instance $\texttt{let}\ z_1 = z_1\ \texttt{in}\ z_1$ and $\texttt{let}\ z_2 = z_1\ \texttt{in}\ z_2$ are the same object. The construct $\texttt{let}\ z = t_1\ \texttt{in}\ t_2$ has the same meaning as $(\lambda z.t_2)\ t_1$, but is dealt with in a more flexible way by ML-the-type-system. To sum up, the syntax of ML-the-calculus is that of the pure λ-calculus, extended with memory locations, constants, and the `let` construct.

Values form a subset of expressions. They are expressions whose evaluation is completed. Values include identifiers, λ-abstractions, and applications of constants, of the form $c\ v_1\ \ldots\ v_k$, where k does not exceed c's arity if c is a constructor, and k is smaller than c's arity if c is a destructor. In what follows, we are often interested in closed values, that is, values that do not contain any free program variables. We use the meta-variables v and w for values.

1.2.2 EXAMPLE: The integer literals $\ldots, \widehat{-1}, \hat{0}, \hat{1}, \ldots$ are nullary constructors, so they are values. Integer addition $\hat{+}$ is a binary destructor, so it is a value, and so is every partial application $\hat{+}\ v$. Thus, both $\hat{+}\ \hat{1}$ and $\hat{+}\ \hat{+}$ are values. An application of $\hat{+}$ to two values, such as $\hat{2}\hat{+}\hat{2}$, is not a value. □

1.2.3 EXAMPLE [PAIRS]: Let $(\cdot,\cdot)$ be a binary constructor. If t_1 are t_2 are expressions, then the double application $(\cdot,\cdot)\ t_1\ t_2$ may be called the *pair* of t_1 and

t_2, and may be written (t_1, t_2). By the definition above, (t_1, t_2) is a value if and only if t_1 and t_2 are both values. □

Stores are finite mappings from memory locations to closed values. A store μ represents what is usually called a heap, that is, a collection of data structures, each of which is allocated at a particular address in memory and may contain pointers to other elements of the heap. ML-the-programming-language allows overwriting the contents of an existing memory block—an operation sometimes referred to as a *side effect*. In the operational semantics, this effect is achieved by mapping an existing memory location to a new value. We write $\varnothing$ for the empty store. We write $\mu[m \mapsto v]$ for the store that maps m to v and otherwise coincides with μ. When μ and μ' have disjoint domains, we write $\mu\mu'$ for their union. We write $dom(\mu)$ for the domain of μ and $range(\mu)$ for the set of memory locations that appear in its codomain.

The operational semantics of a purely functional language, such as the pure λ-calculus, may be defined as a rewriting system on expressions. Because ML-the-calculus has side effects, however, we define its operational semantics as a rewriting system on *configurations*. A configuration t/μ is a pair of an expression t and a store μ. The memory locations in the domain of μ are considered bound within t/μ, so (for instance) $m_1/(m_1 \mapsto \hat{0})$ and $m_2/(m_2 \mapsto \hat{0})$ are the same object. In what follows, we are often interested in *closed* configurations, that is, configurations t/μ such that t has no free program variables and every memory location that appears within t or within the range of μ is in the domain of μ. If t is a closed source program, its evaluation begins within an empty store—that is, with the configuration $t/\varnothing$. Because source programs do not contain memory locations, this is a closed configuration. Furthermore, we shall see that all reducts of a closed configuration are closed as well. Please note that, instead of separating expressions and stores, it is possible to make store fragments part of the syntax of expressions; this idea, proposed in (Crank and Felleisen, 1991), is reminiscent of the encoding of reference cells in process calculi (Turner, 1995; Fournet and Gonthier, 1996).

A *context* is an expression where a single subexpression has been replaced with a *hole*, written $[]$. *Evaluation contexts* form a strict subset of contexts. In an evaluation context, the hole is meant to highlight a point in the program where it is valid to apply a reduction rule. Thus, the definition of evaluation contexts determines a reduction strategy: it tells where and in what order reduction steps may occur. For instance, the fact that $\lambda z.[]$ is not an evaluation context means that the body of a function is never evaluated—that is, not until the function is applied. The fact that $t\ \mathcal{E}$ is an evaluation context only if t is a value means that, to evaluate an application $t_1\ t_2$, one should fully

$$(\lambda z.t)\ v \longrightarrow [z \mapsto v]t \qquad \text{(R-BETA)}$$

$$\texttt{let}\ z = v\ \texttt{in}\ t \longrightarrow [z \mapsto v]t \qquad \text{(R-LET)}$$

$$\frac{t/\mu \xrightarrow{\delta} t'/\mu'}{t/\mu \longrightarrow t'/\mu'} \qquad \text{(R-DELTA)}$$

$$\frac{t/\mu \longrightarrow t'/\mu' \quad dom(\mu'') \# dom(\mu') \quad range(\mu'') \# dom(\mu' \setminus \mu)}{t/\mu\mu'' \longrightarrow t'/\mu'\mu''} \qquad \text{(R-EXTEND)}$$

$$\frac{t/\mu \longrightarrow t'/\mu'}{\mathcal{E}[t]/\mu \longrightarrow \mathcal{E}[t']/\mu'} \qquad \text{(R-CONTEXT)}$$

Figure 1-2: Semantics of ML-the-calculus

evaluate t_1 before attempting to evaluate t_2. More generally, in the case of a multiple application, it means that arguments should be evaluated from left to right. Of course, other choices could be made: for instance, defining $\mathcal{E} ::= \ldots \mid t\ \mathcal{E} \mid \mathcal{E}\ v \mid \ldots$ would enforce a right-to-left evaluation order, while defining $\mathcal{E} ::= \ldots \mid t\ \mathcal{E} \mid \mathcal{E}\ t \mid \ldots$ would leave the evaluation order unspecified, effectively allowing reduction to alternate between both subexpressions, and making evaluation nondeterministic. The fact that $\texttt{let}\ z = v\ \texttt{in}\ \mathcal{E}$ is not an evaluation context means that the body of a local definition is never evaluated—that is, not until the definition itself is reduced. We write $\mathcal{E}[t]$ for the expression obtained by replacing the hole in $\mathcal{E}$ with the expression t.

Figure 1-2 defines first a relation $\longrightarrow$ between configurations, then a relation $\longrightarrow$ between *closed* configurations. If $t/\mu \longrightarrow t'/\mu'$ or $t/\mu \longrightarrow t'/\mu'$ holds, then we say that the configuration t/μ *reduces* to the configuration t'/μ'; the ambiguity involved in this definition is benign. If $t/\mu \longrightarrow t'/\mu$ holds for every store μ, then we write $t \longrightarrow t'$ and say that the reduction is *pure*.

The semantics need not be deterministic. That is, a configuration may reduce to two different configurations. In fact, our semantics is deterministic only if the relation $\xrightarrow{\delta}$, which is a parameter to our semantics, is itself deterministic. As explained above, the semantics could also be made nondeterministic by a different choice in the definition of evaluation contexts.

The key reduction rule is R-BETA, which states that a function application $(\lambda z.t)\ v$ reduces to the function body, namely t, where every occurrence of the formal argument z has been replaced with the actual argument v. The λ construct, which prevented the function body t from being evaluated, disappears, so the new term may (in general) be further reduced. Because ML-the-calculus adopts a *call-by-value* strategy, rule R-BETA is applicable only if the

actual argument is a value $\mathtt{v}$. In other words, a function cannot be invoked until its actual argument has been fully evaluated. Rule R-LET is very similar to R-BETA. Indeed, it specifies that $\mathtt{let\ z = v\ in\ t}$ has the same behavior, with respect to reduction, as $(\lambda \mathtt{z.t})\ \mathtt{v}$. We remark that substitution of a value for a program variable throughout a term is expensive, so R-BETA and R-LET are never implemented literally: they are only a simple *specification*. Actual implementations usually employ *runtime environments*, which may be understood as a form of *explicit substitutions* (Abadi, Cardelli, Curien, and Lévy, 1991; Thérèse Hardin and Pagano, 1998). Please note that our choice of a call-by-value reduction strategy is fairly arbitrary, and has essentially no impact on the type system; the programming language Haskell (Hudak, Peyton Jones, Wadler, Boutel, Fairbairn, Fasel, Guzman, Hammond, Hughes, Johnsson, Kieburtz, Nikhil, Partain, and Peterson, 1992), whose reduction strategy is known as *lazy* or *call-by-need*, also relies on Hindley and Milner's type discipline.

Rule R-DELTA describes the semantics of constants. It states that a certain relation $\xrightarrow{\delta}$ is a subset of $\longrightarrow$. Of course, since the set of constants is unspecified, the relation $\xrightarrow{\delta}$ must be kept abstract as well. We require that, if $\mathtt{t}/\mu \xrightarrow{\delta} \mathtt{t}'/\mu'$ holds, then

(i) $\mathtt{t}$ is of the form $\mathtt{c\ v}_1\ \ldots\ \mathtt{v}_n$, where $\mathtt{c}$ is a destructor of arity n; and

(ii) $\mathit{dom}(\mu)$ is a subset of $\mathit{dom}(\mu')$.

Condition (i) ensures that δ-reduction concerns full applications of destructors only, and that these are evaluated in accordance with the call-by-value strategy. Condition (ii) ensures that δ-reduction may allocate new memory locations, but not deallocate existing locations. In particular, a "garbage collection" operator, which destroys unreachable memory cells, cannot be made available as a constant. Doing so would not make much sense anyway in the presence of R-EXTEND. Condition (ii) allows proving that, if $\mathtt{t}/\mu$ reduces to $\mathtt{t}'/\mu'$, then $\mathit{dom}(\mu)$ is a subset of $\mathit{dom}(\mu')$; this is left as an exercise to the reader.

Rule R-EXTEND states that any valid reduction is also valid in a larger store. The initial and final stores μ and μ' in the original reduction are both extended with a new store fragment μ''. The rule's second premise requires that the domain of μ'' be disjoint with that of μ' (and, consequently, also with that of μ), so that the new memory locations are indeed undefined in the original reduction. (They may, however, appear in the image of μ.) The last premise ensures that the new memory locations in μ'' do not accidentally carry the same names as the locations allocated during the original reduction step, that is, the locations in $\mathit{dom}(\mu' \setminus \mu)$.

Rule R-CONTEXT completes the definition of the operational semantics by defining $\longrightarrow\!\!\!\!\!\rightarrow$, a relation between closed configurations, in terms of $\longrightarrow$. The rule states that reduction may take place not only at the term's root, but also deep inside it, provided the path from the root to the point where reduction occurs forms an evaluation context. This is how evaluation contexts determine an evaluation strategy. As a purely technical point, because $\longrightarrow\!\!\!\!\!\rightarrow$ relates closed configurations only, we do not need to require that the memory locations in $dom(\mu' \setminus \mu)$ be fresh for $\mathcal{E}$: indeed, every memory location that appears within $\mathcal{E}$ must be a member of $dom(\mu)$.

1.2.4 EXAMPLE [INTEGERS, CONTINUED]: The operational semantics of integer addition may be defined as follows:

$$\hat{k}_1 \mathbin{\hat{+}} \hat{k}_2 \xrightarrow{\delta} \widehat{k_1 + k_2} \qquad \text{(R-ADD)}$$

The left-hand term is the double application $\mathbin{\hat{+}}\ \hat{k}_1\ \hat{k}_2$, while the right-hand term is the integer literal $\hat{k}$, where k is the sum of k_1 and k_2. The distinction between object level and meta level (that is, between $\hat{k}$ and k) is needed here to avoid ambiguity. □

1.2.5 EXAMPLE [PAIRS, CONTINUED]: In addition to the pair constructor defined in Example 1.2.3, we may introduce two destructors π_1 and π_2 of arity 1. We may define their operational semantics as follows, for $i \in \{1, 2\}$:

$$\pi_i\ (v_1, v_2) \xrightarrow{\delta} v_i \qquad \text{(R-PROJ)}$$

Thus, our treatment of constants is general enough to account for pair construction and destruction; we need not build these features explicitly into the language. □

1.2.6 EXERCISE [BOOLEANS, RECOMMENDED, ★★]: Let `true` and `false` be nullary constructors. Let `if` be a ternary destructor. Extend the semantics with

$$\texttt{if true}\ v_1\ v_2 \xrightarrow{\delta} v_1 \qquad \text{(R-TRUE)}$$

$$\texttt{if false}\ v_1\ v_2 \xrightarrow{\delta} v_2 \qquad \text{(R-FALSE)}$$

Let us use the syntactic sugar $\texttt{if}\ t_0\ \texttt{then}\ t_1\ \texttt{else}\ t_2$ for the triple application of $\texttt{if}\ t_0\ t_1\ t_2$. Explain why these definitions do not quite provide the expected behavior. Without modifying the semantics of `if`, suggest a new definition of the syntactic sugar $\texttt{if}\ t_0\ \texttt{then}\ t_1\ \texttt{else}\ t_2$ that corrects the problem. □

1.2.7 EXAMPLE [SUMS]: Booleans may in fact be viewed as a special case of the more general concept of *sum*. Let $\texttt{inj}_1$ and $\texttt{inj}_2$ be unary constructors, called respectively *left* and *right injections*. Let `case` be a ternary destructor, whose semantics is defined as follows, for $i \in \{1, 2\}$:

$$\texttt{case}\ (\texttt{inj}_i\ v)\ v_1\ v_2 \xrightarrow{\delta} v_i\ v \qquad \text{(R-CASE)}$$

Here, the value $\texttt{inj}_i\ v$ is being scrutinized, while the values v_1 and v_2, which are typically functions, represent the two arms of a standard `case` construct. The rule selects an appropriate arm (here, v_i) based on whether the value under scrutiny was formed using a left or right injection. The arm v_i is executed and given access to the data carried by the injection (here, v). □

1.2.8 EXERCISE [★, ↛]: Explain how to encode `true`, `false` and the `if` construct in terms of sums. Check that the behavior of R-TRUE and R-FALSE is properly emulated. □

1.2.9 EXAMPLE [REFERENCES]: Let `ref` and ! be unary destructors. Let := be a binary destructor. We write $t_1 := t_2$ for the double application $:=\ t_1\ t_2$. Define the operational semantics of these three destructors as follows:

$$\texttt{ref}\ v/\varnothing \xrightarrow{\delta} m/(m \mapsto v) \quad \text{if } m \text{ is fresh for } v \qquad \text{(R-REF)}$$

$$!m/(m \mapsto v) \xrightarrow{\delta} v/(m \mapsto v) \qquad \text{(R-DEREF)}$$

$$m := v/(m \mapsto v_0) \xrightarrow{\delta} v/(m \mapsto v) \qquad \text{(R-ASSIGN)}$$

According to R-REF, evaluating `ref` v allocates a fresh memory location m and binds v to it. Because configurations are considered equal up to α-conversion of memory locations, the choice of the name m is irrelevant, provided it is chosen fresh for v, so as to prevent inadvertent capture of the memory locations that appear free within v. By R-DEREF, evaluating $!m$ returns the value bound to the memory location m within the current store. By R-ASSIGN, evaluating $m := v$ discards the value v_0 currently bound to m and produces a new store where m is bound to v. Here, the value returned by the assignment $m := v$ is v itself; in ML-the-programming-language, it is usually a nullary constructor $()$, pronounced *unit*. □

1.2.10 EXAMPLE [RECURSION]: Let `fix` be a binary destructor, whose operational semantics is:

$$\texttt{fix}\ v_1\ v_2 \xrightarrow{\delta} v_1\ (\texttt{fix}\ v_1)\ v_2 \qquad \text{(R-FIX)}$$

`fix` is a fixpoint combinator: it effectively allows recursive definitions of functions. Indeed, the construct $\texttt{letrec f} = \lambda z.t_1\ \texttt{in}\ t_2$ provided by ML-the-programming-language may be viewed as syntactic sugar for $\texttt{let f} = \texttt{fix}\ (\lambda f.\lambda z.t_1)\ \texttt{in}\ t_2$. □

1.2.11 EXERCISE [RECOMMENDED, ★★, ↛]: Assuming the availability of Booleans and conditionals, integer literals, subtraction, multiplication, integer comparison, and a fixpoint combinator, most of which were defined in previous

examples, define a function that computes the factorial of its integer argument, and apply it to $\hat{3}$. Determine, step by step, how this expression reduces to a value. □

It is straightforward to check that, if t/μ reduces to t'/μ', then t is not a value. In other words, values are irreducible: they represent completed computations. This fact is established by the next three lemmas.

1.2.12 LEMMA: If $t/\mu \longrightarrow t'/\mu'$ holds, then t is not a value. □

Proof: By structural induction on a derivation of $t/\mu \longrightarrow t'/\mu'$. It is clear, by definition of values, that the left-hand sides of R-BETA and R-LET are not values. For R-DELTA, the left-hand side must be of the form $c\ v_1\ \ldots\ v_n$, where c is a destructor of arity n, so it is not a value. For R-EXTEND, the result follows from the induction hypothesis. □

1.2.13 LEMMA: If $\mathcal{E}[t]$ is a value, then t is a value. □

Proof: By induction on the structure of the evaluation context.

∘ *Case* $[]$. Immediate.

∘ *Case* $\mathcal{E}\ t'$. Assume $\mathcal{E}[t]\ t'$ is a value. Then, by the definition of values, it must be of the form $c\ v_1\ \ldots\ v_k$, where k is bounded (strictly bounded, if c is a destructor) by c's arity. So, $\mathcal{E}[t]$ is $c\ v_1\ \ldots\ v_{k-1}$, which is a value as well. The result follows by the induction hypothesis.

∘ *Case* $v\ \mathcal{E}$. Assume $v\ \mathcal{E}[t]$ is a value. Then, by the definition of values, it must be of the form $c\ v_1\ \ldots\ v_k$. So, $\mathcal{E}[t]$ is v_k, that is, a value. The result follows by the induction hypothesis.

∘ *Case* $\texttt{let}\ z = \mathcal{E}\ \texttt{in}\ t'$. Immediate, since $\texttt{let}\ z = \mathcal{E}[t]\ \texttt{in}\ t'$ cannot be a value. □

1.2.14 LEMMA: If $t/\mu \longrightarrow\!\!\!\!\!\rightarrow t'/\mu'$ holds, then t is not a value. □

Proof: By R-CONTEXT, Lemma 1.2.12, and the contrapositive of Lemma 1.2.13. □

The converse, however, does not hold: if t/μ is irreducible with respect to $\longrightarrow\!\!\!\!\!\rightarrow$, then t is not necessarily a value. In that case, the configuration t/μ is said to be *stuck*. It represents a *runtime error*, that is, a situation that does not allow computation to proceed, yet is not considered a valid outcome. A closed source program t is said to *go wrong* if and only if the configuration $t/\varnothing$ reduces to a stuck configuration.

1.2.15 EXAMPLE: Runtime errors typically arise when destructors are applied to arguments of an unexpected nature. For instance, the expressions $\hat{\dagger}\ \hat{1}\ m$ and $\pi_1\ \hat{2}$ and $!\hat{3}$ are stuck, regardless of the current store. The program `let z =` $\hat{\dagger}\ \hat{\dagger}$ `in z 1` is not stuck, because $\hat{\dagger}\ \hat{\dagger}$ is a value. However, its reduct through R-LET is $\hat{\dagger}\ \hat{\dagger}\ 1$, which is stuck, so this program goes wrong. The primary purpose of type systems is to prevent such situations from arising. □

1.2.16 REMARK: The configuration $!m/\mu$ is stuck if m is not in the domain of μ. In that case, however, $!m/\mu$ is not closed. Because we consider $\longrightarrow$ as a relation between closed configurations only, this situation cannot arise. In other words, the semantics of ML-the-calculus never allows the creation of *dangling pointers*. As a result, no particular precautions need be taken to guard against them. Several strongly typed programming languages do nevertheless allow dangling pointers in a controlled fashion (Tofte and Talpin, 1997; Crary, Walker, and Morrisett, 1999b; DeLine and Fähndrich, 2001; Grossman, Morrisett, Jim, Hicks, Wang, and Cheney, 2002a). □

1.2.2 Damas and Milner's type system

ML-the-type-system was originally defined by Milner (1978). Here, we reproduce the definition given a few years later by Damas and Milner (1982), which is written in a more standard style: typing judgements are defined inductively by a collection of typing rules. We refer to this type system as DM.

To begin, we must define *types*. In DM, like in the simply-typed λ-calculus, types are first-order terms built out of *type constructors* and *type variables*. We begin with several considerations concerning the specification of type constructors.

First, we do not wish to fix the set of type constructors. Certainly, since ML-the-calculus has functions, we need to be able to form an arrow type $\mathtt{T} \to \mathtt{T}'$ out of arbitrary types $\mathtt{T}$ and $\mathtt{T}'$; that is, we need a binary type constructor $\to$. However, because ML-the-calculus includes an unspecified set of constants, we cannot say much else in general. If constants include integer literals and integer operations, as in Example 1.2.1, then a nullary type constructor int is needed; if they include pair construction and destruction, as in Examples 1.2.3 and 1.2.5, then a binary type constructor $\times$ is needed; and so on.

Second, it is common to refer to the parameters of a type constructor *by position*, that is, by numeric index. For instance, when one writes $\mathtt{T} \to \mathtt{T}'$, it is understood that the type constructor $\to$ has arity 2, that $\mathtt{T}$ is its *first* parameter, known as its *domain*, and that $\mathtt{T}'$ is its *second* parameter, known as

its *codomain*. Here, however, we refer to parameters *by names*, known as *directions*. For instance, we define two directions *domain* and *codomain* and let the type constructor $\rightarrow$ have arity $\{domain, codomain\}$. The extra generality afforded by directions is exploited in the definition of nonstructural subtyping (Example 1.3.9) and in the definition of rows (§1.11).

Last, we allow types to be classified using *kinds*. As a result, every type constructor must come not only with an arity, but with a richer *signature*, which describes the kinds of the types to which it is applicable and the kind of the type that it produces. A distinguished kind $\star$ is associated with "normal" types, that is, types that are directly ascribed to expressions and values. For instance, the signature of the type constructor $\rightarrow$ is $\{domain \mapsto \star, codomain \mapsto \star\} \Rightarrow \star$, because it is applicable to two "normal" types and produces a "normal" type. Introducing kinds other than $\star$ allows viewing some types as ill-formed: this is illustrated, for instance, in §1.11. In the simplest case, however, $\star$ is really the only kind, so the signature of a type constructor is nothing but its arity (a set of directions), and every term is a well-formed type, provided every application of a type constructor respects its arity.

1.2.17 DEFINITION: Let d range over a finite or denumerable set of *directions*. Let κ range over a finite or denumerable set of *kinds*. Let $\star$ be a distinguished kind. Let K range over partial mappings from directions to kinds. Let F range over a finite or denumerable set of *type constructors*, each of which has a *signature* of the form $\mathsf{K} \Rightarrow \kappa$. The domain of K is referred to as the *arity* of F, while κ is referred to as its *image kind*. We write κ instead of $\mathsf{K} \Rightarrow \kappa$ when K is empty. Let $\rightarrow$ be a type constructor of signature $\{domain \mapsto \star, codomain \mapsto \star\} \Rightarrow \star$. □

The type constructors and their signatures collectively form a *signature* $\mathcal{S}$. In the following, we assume that a fixed signature $\mathcal{S}$ is given and that every type constructor in it has *finite* arity, so as to ensure that types are machine representable. However, in §1.11, we shall explicitly work with several distinct signatures, one of which involves type constructors of denumerable arity.

A *type variable* is a name that is used to stand for a type. For simplicity, we assume that every type variable is branded with a kind, or, in other words, that type variables of distinct kinds are drawn from disjoint sets. Each of these sets of type variables is individually subject to α-conversion: that is, renamings must preserve kinds. Attaching kinds to type variables is only a technical convenience: in practice, every operation performed during type inference preserves the property that every type is well-kinded, so it is not necessary to keep track of the kind of every type variable. It is only necessary to check that all types supplied by the user, within type declarations, type annotations, or module interfaces, are well-kinded.

1.2.18 DEFINITION: For every kind κ, let $\mathcal{V}_\kappa$ be a disjoint, denumerable set of *type variables*. Let $\mathtt{X}$, $\mathtt{Y}$, and $\mathtt{Z}$ range over the set $\mathcal{V}$ of all type variables. Let $\bar{\mathtt{X}}$ and $\bar{\mathtt{Y}}$ range over finite sets of type variables. We write $\bar{\mathtt{X}}\bar{\mathtt{Y}}$ for the set $\bar{\mathtt{X}} \cup \bar{\mathtt{Y}}$ and often write $\mathtt{X}$ for the singleton set $\{\mathtt{X}\}$. We write $\mathit{ftv}(\mathtt{o})$ for the set of *free type variables* of an object $\mathtt{o}$. □

The set of types, ranged over by $\mathtt{T}$, is the free many-kinded term algebra that arises out of the type constructors and type variables. Types are given by the following inductive definition:

1.2.19 DEFINITION: A *type* of kind κ is either a member of $\mathcal{V}_\kappa$, or a term of the form $\mathtt{F}\{\mathtt{d}_1 \mapsto \mathtt{T}_1, \ldots, \mathtt{d}_n \mapsto \mathtt{T}_n\}$, where $\mathtt{F}$ has signature $\{\mathtt{d}_1 \mapsto \kappa_1, \ldots, \mathtt{d}_n \mapsto \kappa_n\} \Rightarrow \kappa$ and $\mathtt{T}_1, \ldots, \mathtt{T}_n$ are types of kind $\kappa_1, \ldots, \kappa_n$, respectively. □

As a notational convention, we assume that, for every type constructor $\mathtt{F}$, the directions that form the arity of $\mathtt{F}$ are implicitly ordered, so that we may say that $\mathtt{F}$ has signature $\kappa_1 \otimes \ldots \otimes \kappa_n \Rightarrow \kappa$ and employ the syntax $\mathtt{F}\ \mathtt{T}_1\ \ldots\ \mathtt{T}_n$ for applications of $\mathtt{F}$. Applications of the type constructor $\rightarrow$ are written infix and associate to the right, so $\mathtt{T} \rightarrow \mathtt{T}' \rightarrow \mathtt{T}''$ stands for $\mathtt{T} \rightarrow (\mathtt{T}' \rightarrow \mathtt{T}'')$.

In order to give meaning to the free type variables of a type, or, more generally, of a typing judgement, traditional presentations of ML-the-type-system, including Damas and Milner's, employ *type substitutions*. Most of our presentation avoids substitutions and uses *constraints* instead. However, we do need substitutions on a few occasions, especially when relating our presentation to Damas and Milner's.

1.2.20 DEFINITION: A *type substitution* θ (sometimes φ) is a total, kind-preserving mapping of type variables to types that is the identity everywhere but on a finite subset of $\mathcal{V}$, which we call the *domain* of θ and write $\mathit{dom}(\theta)$. The *range* of θ, which we write $\mathit{range}(\theta)$, is the set $\mathit{ftv}(\theta(\mathit{dom}(\theta)))$. A type substitution may naturally be viewed as a total, kind-preserving mapping of types to types. We write $\bar{\mathtt{X}} \# \theta$ for $\bar{\mathtt{X}} \# (\mathit{dom}(\theta) \cup \mathit{range}(\theta))$. We write $\theta \setminus \bar{\mathtt{X}}$ for the restriction of θ outside $\bar{\mathtt{X}}$, that is, the restriction of θ to $\mathcal{V} \setminus \bar{\mathtt{X}}$. □

If $\vec{\mathtt{X}}$ and $\vec{\mathtt{T}}$ are respectively a vector of *distinct* type variables and a vector of types of the same (finite) length, such that, for every index $\mathtt{i}$, $\mathtt{X}_\mathtt{i}$ and $\mathtt{T}_\mathtt{i}$ have the same kind, then $[\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}]$ denotes the substitution that maps $\mathtt{X}_\mathtt{i}$ to $\mathtt{T}_\mathtt{i}$ for every index $\mathtt{i}$ and is the identity elsewhere. The domain of $[\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}]$ is a subset of $\bar{\mathtt{X}}$, the set underlying the vector $\vec{\mathtt{X}}$. Its range is a subset of $\mathit{ftv}(\bar{\mathtt{T}})$, where $\bar{\mathtt{T}}$ is the set underlying the vector $\vec{\mathtt{T}}$. (These may be *strict* subsets: for instance, the domain of $[\mathtt{X} \mapsto \mathtt{X}]$ is the empty set, since this substitution is the identity.) Every substitution θ may be written under the form $[\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}]$, where $\bar{\mathtt{X}} = \mathit{dom}(\theta)$. Then, θ is *idempotent* if and only if $\bar{\mathtt{X}} \# \mathit{ftv}(\bar{\mathtt{T}})$ holds.

$$\frac{\Gamma(x) = S}{\Gamma \vdash x : S} \quad \text{(DM-VAR)} \qquad \frac{\Gamma \vdash t_1 : S \qquad \Gamma; z : S \vdash t_2 : T}{\Gamma \vdash \mathtt{let}\ z = t_1\ \mathtt{in}\ t_2 : T} \quad \text{(DM-LET)}$$

$$\frac{\Gamma; z : T \vdash t : T'}{\Gamma \vdash \lambda z.t : T \to T'} \quad \text{(DM-ABS)} \qquad \frac{\Gamma \vdash t : T \qquad \bar{X} \# \mathit{ftv}(\Gamma)}{\Gamma \vdash t : \forall \bar{X}.T} \quad \text{(DM-GEN)}$$

$$\frac{\Gamma \vdash t_1 : T \to T' \qquad \Gamma \vdash t_2 : T}{\Gamma \vdash t_1\ t_2 : T'} \quad \text{(DM-APP)} \qquad \frac{\Gamma \vdash t : \forall \bar{X}.T}{\Gamma \vdash t : [\vec{X} \mapsto \vec{T}]T} \quad \text{(DM-INST)}$$

Figure 1-3: Typing rules for DM

As pointed out earlier, types are first-order terms; that is, in the grammar of types, none of the productions *binds* a type variable. As a result, every type variable that appears within a type T appears *free* within T. This situation is identical to that of the simply-typed λ-calculus. Things become more interesting when we introduce *type schemes*. As its name implies, a type scheme may describe an entire family of types; this effect is achieved via *universal quantification* over a set of type variables.

1.2.21 DEFINITION: A type scheme S is an object of the form $\forall \bar{X}.T$, where T is a type of kind $\star$ and the type variables $\bar{X}$ are considered bound within T. Any type of the form $[\vec{X} \mapsto \vec{T}]T$ is called an *instance* of the type scheme $\forall \bar{X}.T$. □

One may view the type T as the trivial type scheme $\forall \varnothing.T$, where no type variables are universally quantified, so types of kind $\star$ may be viewed as a subset of type schemes. The type scheme $\forall \bar{X}.T$ may be viewed as a finite way of describing the possibly infinite family of its instances. Note that, throughout most of this chapter, we work with *constrained type schemes*, a generalization of DM type schemes (Definition 1.3.2).

Typing environments, or environments for short, are used to collect assumptions about an expression's free identifiers.

1.2.22 DEFINITION: An *environment* Γ is a finite ordered sequence of pairs of a program identifier and a type scheme. We write $\varnothing$ for the empty environment and ";" for the concatenation of environments. An environment may be viewed as a finite mapping from program identifiers to type schemes by letting $\Gamma(x) = S$ if and only if Γ is of the form $\Gamma_1; x : S; \Gamma_2$, where Γ_2 contains no assumption about x. The set of *defined program identifiers* of an environment Γ, written $\mathit{dpi}(\Gamma)$, is defined by $\mathit{dpi}(\varnothing) = \varnothing$ and $\mathit{dpi}(\Gamma; x : S) = \mathit{dpi}(\Gamma) \cup \{x\}$. □

To complete the definition of Damas and Milner's type system, there re-

mains to define *typing judgements*. A typing judgement takes the form $\Gamma \vdash \mathtt{t} : \mathtt{S}$, where $\mathtt{t}$ is an expression of interest, Γ is an environment, which typically contains assumptions about $\mathtt{t}$'s free program identifiers, and $\mathtt{S}$ is a type scheme. Such a judgement may be read: *under assumptions* Γ, *the expression* $\mathtt{t}$ *has the type scheme* $\mathtt{S}$. By abuse of language, it is sometimes said that $\mathtt{t}$ *has type* $\mathtt{S}$. A typing judgement is *valid* (or *holds*) if and only if it may be derived using the rules that appear in Figure 1-3. An expression $\mathtt{t}$ is *well-typed* within the environment Γ if and only if there exists some type scheme $\mathtt{S}$ such that the judgement $\Gamma \vdash \mathtt{t} : \mathtt{S}$ holds; it is *ill-typed* within Γ otherwise.

Rule DM-VAR allows fetching a type scheme for an identifier $\mathtt{x}$ from the environment. It is equally applicable to program variables, memory locations, and constants. If no assumption concerning $\mathtt{x}$ appears in the environment Γ, then the rule isn't applicable. In that case, the expression $\mathtt{x}$ is ill-typed within Γ. Assumptions about constants are usually collected in a so-called *initial environment* Γ_0. It is the environment under which closed programs are typechecked, so every subexpression is typechecked under some extension Γ of Γ_0. Of course, the type schemes assigned by Γ_0 to constants must be consistent with their operational semantics; we say more about this later (§1.7). Rule DM-ABS specifies how to typecheck a λ-abstraction $\lambda\mathtt{z}.\mathtt{t}$. Its premise requires the body of the function, namely $\mathtt{t}$, to be well-typed under an extra assumption that causes all free occurrences of $\mathtt{z}$ within $\mathtt{t}$ to receive a common type $\mathtt{T}$. Its conclusion forms the arrow type $\mathtt{T} \to \mathtt{T}'$ out of the types of the function's formal parameter, namely $\mathtt{T}$, and result, namely $\mathtt{T}'$. It is worth noting that this rule always augments the environment with a type $\mathtt{T}$—recall that, by convention, types form a subset of type schemes—but never with a nontrivial type scheme. Rule DM-APP states that the type of a function application is the codomain of the function's type, provided that the domain of the function's type is a valid type for the actual argument. Rule DM-LET closely mirrors the operational semantics: whereas the semantics of the local definition $\mathtt{let\ z = t_1\ in\ t_2}$ is to augment the *runtime* environment by binding $\mathtt{z}$ to the *value* of $\mathtt{t_1}$ prior to evaluating $\mathtt{t_2}$, the effect of DM-LET is to augment the *typing* environment by binding $\mathtt{z}$ to a *type scheme* for $\mathtt{t_1}$ prior to typechecking $\mathtt{t_2}$. Rule DM-GEN turns a type into a type scheme by universally quantifying over a set of type variables that do not appear free in the environment; this restriction is discussed in Example 1.2.23 below. Rule DM-INST, on the contrary, turns a type scheme into one of its instances, which may be chosen arbitrarily. These two operations are referred to as *generalization* and *instantiation*. The notion of type scheme and the rules DM-GEN and DM-INST are characteristic of ML-the-type-system: they distinguish it from the simply-typed λ-calculus.

1.2.23 EXAMPLE: It is unsound to allow generalizing type variables that appear free in the environment. For instance, consider the typing judgement $z : X \vdash z : X$ **(1)**, which, according to DM-VAR, is valid. Applying an unrestricted version of DM-GEN to it, we obtain $z : X \vdash z : \forall X.X$ **(2)**, whence, by DM-INST, $z : X \vdash z : Y$ **(3)**. By DM-ABS and DM-GEN, we then have $\varnothing \vdash \lambda z.\, z : \forall XY.X \to Y$. In other words, the identity function has unrelated argument and result types! Then, the expression $(\lambda z.z)\ \hat{0}\ \hat{0}$, which reduces to the stuck expression $\hat{0}\ \hat{0}$, has type scheme $\forall Z.Z$. So, well-typed programs may cause runtime errors: the type system is unsound.

What happened? It is clear that the judgement (1) is correct only because the type assigned to z is the *same* in its assumption and in its right-hand side. For the same reason, the judgements (2) and (3)—the former of which may be written $z : X \vdash z : \forall Y.Y$—are incorrect. Indeed, such judgements defeat the very purpose of environments, since they disregard their assumption. By universally quantifying over X *in the right-hand side only*, we break the connection between occurrences of X in the assumption, which remain free, and occurrences in the right-hand side, which become bound. This is correct only if there are in fact no free occurrences of X in the assumption. □

It is a key feature of ML-the-type-system that DM-ABS may only introduce a type T, rather than a type scheme, into the environment. Indeed, this allows the rule's conclusion to form the arrow type $T \to T'$. If instead the rule were to introduce the assumption $z : S$ into the environment, then its conclusion would have to form $S \to T'$, which is not a well-formed type. In other words, this restriction is necessary to preserve the stratification between types and type schemes. If we were to remove this stratification, thus allowing universal quantifiers to appear deep inside types, we would obtain an implicitly-typed version of System F (TAPL Chapter 23). Type inference for System F is undecidable (Wells, 1999), while type inference for ML-the-type-system is decidable, as we show later, so this design choice has a rather drastic impact.

1.2.24 EXERCISE [RECOMMENDED, ★]: Build a type derivation for the expression $\lambda z_1.\mathtt{let}\ z_2 = z_1\ \mathtt{in}\ z_2$. □

1.2.25 EXERCISE [RECOMMENDED, ★]: Let int be a nullary type constructor of signature $\star$. Let Γ_0 consist of the bindings $\hat{+} : \mathsf{int} \to \mathsf{int} \to \mathsf{int}$ and $\hat{k} : \mathsf{int}$, for every integer k. Can you find derivations of the following valid typing judgements? Which of these judgements are valid in the simply-typed λ-calculus,

where $\texttt{let z} = \texttt{t}_1 \texttt{ in t}_2$ is syntactic sugar for $(\lambda \texttt{z.t}_2)\ \texttt{t}_1$?

$$\Gamma_0 \vdash \lambda \texttt{z.z} : \mathsf{int} \to \mathsf{int}$$
$$\Gamma_0 \vdash \lambda \texttt{z.z} : \forall \texttt{X.X} \to \texttt{X}$$
$$\Gamma_0 \vdash \texttt{let f} = \lambda \texttt{z.z} \hat{+} \hat{1} \texttt{ in f } \hat{2} : \mathsf{int}$$
$$\Gamma_0 \vdash \texttt{let f} = \lambda \texttt{z.z in f f } \hat{2} : \mathsf{int}$$

Show that the expressions $\hat{1}\ \hat{2}$ and $\lambda \texttt{f.(f f)}$ are ill-typed within Γ_0. Could these expressions be well-typed in a more powerful type system? □

1.2.26 EXERCISE [★★★]: In fact, the rules shown in Figure 1-3 are not exactly Damas and Milner's original rules. In (Damas and Milner, 1982), the generalization and instantiation rules are:

$$\frac{\Gamma \vdash \texttt{t} : \texttt{S} \qquad \texttt{X} \notin \mathit{ftv}(\Gamma)}{\Gamma \vdash \texttt{t} : \forall \texttt{X.S}} \qquad \text{(DM-GEN')}$$

$$\frac{\Gamma \vdash \texttt{t} : \forall \bar{\texttt{X}}.\texttt{T} \qquad \bar{\texttt{Y}} \,\#\, \mathit{ftv}(\forall \bar{\texttt{X}}.\texttt{T})}{\Gamma \vdash \texttt{t} : \forall \bar{\texttt{Y}}.[\vec{\texttt{X}} \mapsto \vec{\texttt{T}}]\texttt{T}} \qquad \text{(DM-INST')}$$

where $\forall \texttt{X.S}$ stands for $\forall \texttt{X}\bar{\texttt{X}}.\texttt{T}$ if S stands for $\forall \bar{\texttt{X}}.\texttt{T}$. Show that the combination of DM-GEN' and DM-INST' is equivalent to the combination of DM-GEN and DM-INST. □

DM enjoys a number of nice theoretical properties, which have practical implications. First, under suitable hypotheses about the semantics of constants, about the type schemes that they receive in the initial environment, and—in the presence of side effects—under a slight restriction of the syntax of `let` constructs, it is possible to show that the type system is sound: that is, *well-typed (closed) programs do not go wrong*. This essential property ensures that programs that are accepted by the typechecker may be compiled without runtime checks. Furthermore, it is possible to show that there exists an algorithm that, given a (closed) environment Γ and a program t, tells whether t is well-typed with respect to Γ, and if so, produces a *principal* type scheme S. A principal type scheme is such that (i) it is valid, that is, $\Gamma \vdash \texttt{t} : \texttt{S}$ holds, and (ii) it is most general, that is, every judgement of the form $\Gamma \vdash \texttt{t} : \texttt{S}'$ follows from $\Gamma \vdash \texttt{t} : \texttt{S}$ by DM-INST and DM-GEN. (For the sake of simplicity, we have stated the properties of the type inference algorithm only in the case of a closed environment Γ; the specification is slightly heavier in the general case.) This implies that *type inference is decidable*: the compiler does not require expressions to be annotated with types. It also implies that, under a fixed environment Γ, all of the type information associated with an expression t may be summarized in the form of a single (principal) type scheme, which is very convenient.

1.2.3 Road map

Before proving the above claims, we first generalize our presentation by moving to a *constraint-based* setting. The necessary tools, namely the constraint language, its interpretation, and a number of constraint equivalence laws, are introduced in §1.3. In §1.4, we describe the standard constraint-based type system HM(X) (Odersky, Sulzmann, and Wehr, 1999). We prove that, when constraints are made up of equations between free, finite terms, HM(X) is a reformulation of DM. In the presence of a more powerful constraint language, HM(X) is an extension of DM. In §1.5, we propose an original reformulation of HM(X), dubbed PCB(X), whose distinctive feature is to exploit *type scheme introduction* and *instantiation* constraints. In §1.6, we show that, thanks to the extra expressive power afforded by these constraint forms, type inference may be viewed as a combination of constraint generation and constraint solving, as promised earlier. Then, in §1.7, we give a type soundness theorem. It is stated purely in terms of constraints, but—thanks to the results developed in the previous sections—applies equally to PCB(X), HM(X), and DM.

Throughout this core material, the syntax and interpretation of constraints are left partly unspecified. Thus, the development is *parameterized* with respect to them—hence the unknown X in the names HM(X) and PCB(X). We really describe a *family* of constraint-based type systems, all of which *share* a common constraint generator and a common type soundness proof. Constraint solving, however, cannot be independent of X: on the contrary, the design of an efficient solver is heavily dependent on the syntax and interpretation of constraints. In §1.8, we consider constraint solving in the particular case where constraints are made up of equations interpreted in a free tree model, and define a constraint solver on top of a standard first-order unification algorithm.

The remainder of this chapter deals with extensions of the framework. In §1.9, we explain how to extend ML-the-calculus with a number of features, including products, sums, references, recursion, algebraic data types, pattern matching, type annotations, and recursive types. In §1.10, we extend the constraint language with universal quantification and describe a number of extra features that require this extension, including a different flavor of type annotations, polymorphic recursion, and first-class universal and existential types. Last, in §1.11, we extend the type language with *rows* and use them to assign polymorphic type schemes to operations on records and variants.

σ ::=		*type scheme:*	
	$\forall \bar{X}[C].T$		
C, D ::=		*constraint:*	
	true	*truth*	
	false	*falsity*	
	$P\ T_1 \ldots T_n$	*predicate application*	
	$C \wedge C$	*conjunction*	
	$\exists \bar{X}.C$	*existential quantification*	
	def $x : \sigma$ in C	*type scheme introduction*	
	$x \preceq T$	*type scheme instantiation*	
C, D ::=		*Syntactic sugar for constraints:*	
	...	*As before*	
	$\sigma \preceq T$	*Definition 1.3.3*	
	let $x : \sigma$ in C	*Definition 1.3.3*	
	$\exists \sigma$	*Definition 1.3.3*	
	def Γ in C	*Definition 1.3.4*	
	let Γ in C	*Definition 1.3.4*	
	$\exists \Gamma$	*Definition 1.3.4*	

Figure 1-4: Syntax of type schemes and constraints

1.3 Constraints

In this section, we define the syntax and logical meaning of constraints. Both are partly unspecified. Indeed, the set of *type constructors* (Definition 1.2.17) must contain at least the binary type constructor $\rightarrow$, but might contain more. Similarly, the syntax of constraints involves a set of so-called *predicates* on types, which we require to contain at least a binary *subtyping* predicate $\leq$, but might contain more. (The introduction of subtyping, which is absent in DM, has little impact on the complexity of our proofs, yet increases the framework's expressive power. When subtyping is not desired, we interpret the predicate $\leq$ as equality.) The logical interpretation of type constructors and of predicates is left almost entirely unspecified. This freedom allows reasoning not only about Damas and Milner's type system, but also about a family of constraint-based extensions of it.

Type constructors other than $\rightarrow$ and predicates other than $\leq$ will never explicitly appear in the definition of our constraint-based type systems, precisely because the definition is parametric with respect to them. They can (and usually do) appear in the type schemes assigned to constructors and destructors in the initial environment Γ_0.

1.3.1 Syntax

We now define the syntax of constrained type schemes and of constraints, and introduce some extra constraint forms as syntactic sugar.

1.3.1 DEFINITION: Let P range over a finite or denumerable set of *predicates*, each of which has a *signature* of the form $\kappa_1 \otimes \ldots \otimes \kappa_n \Rightarrow \cdot$, where $n \geq 0$. For every kind κ, let $=_\kappa$ and $\leq_\kappa$ be distinguished predicates of signature $\kappa \otimes \kappa \Rightarrow \cdot$. $\square$

1.3.2 DEFINITION: The syntax of *type schemes* and *constraints* is given in Figure 1-4. It is further restricted by the following requirements. In the type scheme $\forall\bar{X}[C].T$ and in the constraint $x \preceq T$, the type T must have kind $\star$. In the constraint $P\,T_1 \ldots T_n$, the types $T_1, \ldots, T_n$ must have kind $\kappa_1, \ldots, \kappa_n$, respectively, if P has signature $\kappa_1 \otimes \ldots \otimes \kappa_n \Rightarrow \cdot$. We write $\forall\bar{X}.T$ for $\forall\bar{X}[\mathsf{true}].T$, which allows viewing DM type schemes as a subset of constrained type schemes. □

We write $T_1 =_\kappa T_2$ and $T_1 \leq_\kappa T_2$ for the binary predicate applications $= T_1\,T_2$ and $\leq T_1\,T_2$, and refer to them as equality and subtyping constraints, respectively. We often omit the subscript κ, so $T_1 = T_2$ and $T_1 \leq T_2$ are well-formed constraints whenever T_1 and T_2 have the same kind. By convention, $\exists$ and def bind tighter than $\wedge$; that is, $\exists\bar{X}.C \wedge D$ is $(\exists\bar{X}.C) \wedge D$ and $\mathsf{def}\ x : \sigma\ \mathsf{in}\ C \wedge D$ is $(\mathsf{def}\ x : \sigma\ \mathsf{in}\ C) \wedge D$. In $\forall\bar{X}[C].T$, the type variables $\bar{X}$ are bound within C and T. In $\exists\bar{X}.C$, the type variables $\bar{X}$ are bound within C. The sets of free type variables of a type scheme σ and of a constraint C, written $ftv(\sigma)$ and $ftv(C)$, respectively, are defined accordingly. In $\mathsf{def}\ x : \sigma\ \mathsf{in}\ C$, the identifier x is bound within C. The sets of free program identifiers of a type scheme σ and of a constraint C, written $fpi(\sigma)$ and $fpi(C)$, respectively, are defined accordingly. Please note that x occurs free in the constraint $x \preceq T$.

The constraint true, which is always satisfied, mainly serves to indicate the absence of a nontrivial constraint, while false, which has no solution, may be understood as the indication of a type error. Composite constraints include conjunction and existential quantification, which have their standard meaning, as well as *type scheme introduction* and *type scheme instantiation* constraints, which are similar to Gustavsson and Svenningsson's constraint abstractions (2001b). In order to be able to explain these last two forms, we must first introduce a number of derived constraint forms:

1.3.3 DEFINITION: Let σ be $\forall\bar{X}[D].T$. If $\bar{X}$ # $ftv(T')$ holds, then $\sigma \preceq T'$ (read: T' *is an instance of* σ) stands for the constraint $\exists\bar{X}.(D \wedge T \leq T')$. We write $\exists\sigma$ (read: σ *has an instance*) for $\exists\bar{X}.D$ and $\mathsf{let}\ x : \sigma\ \mathsf{in}\ C$ for $\exists\sigma \wedge \mathsf{def}\ x : \sigma\ \mathsf{in}\ C$. □

Constrained type schemes generalize Damas and Milner's type schemes, while this definition of instantiation constraints generalizes Damas and Milner's notion of instance (Definition 1.2.21). Let us draw a comparison. First, Damas and Milner's instance relation is binary (given a type scheme S and a type T, either T is an instance of S, or it isn't), and is purely syntactic. For instance, the type $Y \to Z$ is *not* an instance of $\forall X.X \to X$ in Damas and Milner's sense, because Y and Z are distinct type variables. In our presentation, on the other hand, $\forall X.X \to X \preceq Y \to Z$ is not an assertion; rather, it is a constraint, which by definition is $\exists X.(\mathsf{true} \wedge X \to X \leq Y \to Z)$. We later prove that it is

equivalent to $\exists X.(Y \leq X \wedge X \leq Z)$ and to $Y \leq Z$, and, if subtyping is interpreted as equality, to $Y = Z$. That is, $\sigma \preceq T'$ represents a condition on (the ground types denoted by) the type variables in $ftv(\sigma, T')$ for T' to be an instance of σ, in a logical, rather than purely syntactic, sense. Second, the definition of instantiation constraints involves subtyping, so as to ensure that any supertype of an instance of σ is again an instance of σ (see Lemmas 1.3.24 and 1.3.25). This is consistent with the purpose of subtyping, which is to allow supplying a subtype where a supertype is expected (TAPL Chapter 15). Third and last, every type scheme σ is now of the form $\forall \bar{X}[C].T$. The constraint C, whose free type variables may or may not be members of $\bar{X}$, is meant to restrict the set of instances of the type scheme $\forall \bar{X}[C].T$. This is evident in the instantiation constraint $\forall \bar{X}[C].T \preceq T'$, which by Definition 1.3.3 stands for $\exists \bar{X}.(C \wedge T \leq T')$: the values that $\bar{X}$ may assume are restricted by the requirement that C be satisfied. This requirement vanishes in the case of DM type schemes, where C is true. Our notions of constrained type scheme and of instantiation constraint are standard: they are exactly those of HM(X) (Odersky, Sulzmann, and Wehr, 1999).

Let us now come back to an explanation of type scheme introduction and instantiation constraints. In short, the construct def $x : \sigma$ in C binds the name x to the type scheme σ within the constraint C. If C contains a subconstraint of the form $x \preceq T$, where this occurrence of x is free in C, then this subconstraint acquires the meaning $\sigma \preceq T$. Thus, the constraint $x \preceq T$ is indeed an instantiation constraint, where the type scheme that is being instantiated is referred to by name. The constraint def $x : \sigma$ in C may be viewed as an *explicit substitution* of the type scheme σ for the name x within C. Later (§1.5), we use such explicit substitutions to supplant typing environments. That is, where Damas and Milner's type system augments the current typing environment (DM-ABS, DM-LET), we introduce a new let binding in the current constraint, which, by Definition 1.6, expands to a new def binding; where it looks up the current typing environment (DM-VAR), we employ an instantiation constraint. (The reader may wish to have a look ahead at Figure 1-10 on page 80.) The point is that it is then up to a constraint solver to choose a strategy for reducing explicit substitutions—for instance, one might wish to *simplify* σ before substituting it for x within C—whereas the use of environments in standard type systems such as DM and HM(X) imposes an eager substitution strategy, which is inefficient and thus never literally implemented. The use of type scheme introduction and instantiation constraints allows separating constraint generation and constraint solving *without compromising efficiency*, or, in other words, without introducing a gap between the description of the type inference algorithm and its actual implementation. Although the algorithm that we plan to describe is not new (Rémy, 1992a), its description in

terms of constraints is: to the best of our knowledge, the only close relative of our def constraints is to be found in (Gustavsson and Svenningsson, 2001b). An earlier work that contains similar ideas is (Müller, 1994). Fähndrich, Rehof, and Das's instantiation constraints (2000) are also related, but may be recursive and are meant to be solved using a semi-unification procedure, as opposed to a unification algorithm extended with facilities for creating and instantiating type schemes, as in our case.

In Damas and Milner's type system, every type scheme $\mathtt{S}$ has a fixed, nonempty set of instances. In a constraint-based setting, things are more complex: given a type scheme σ and a type $\mathtt{T}$, whether $\mathtt{T}$ is an instance of σ (that is, whether the constraint $\sigma \preceq \mathtt{T}$ is satisfied) depends on the meaning assigned to the type variables in $\mathit{ftv}(\sigma, \mathtt{T})$. Similarly, given a type scheme, whether *some* type is an instance of σ (that is, whether the constraint $\exists \mathtt{Z}.\sigma \preceq \mathtt{Z}$, where $\mathtt{Z}$ is fresh for σ, is satisfied) depends on the meaning assigned to the type variables in $\mathit{ftv}(\sigma)$. Because we do not wish to allow forming type schemes that have no instances, we often use the constraint $\exists \mathtt{Z}.\sigma \preceq \mathtt{Z}$. In fact, we later prove that it is equivalent to $\exists\sigma$, as defined above (Exercise 1.3.32). We also use the constraint form $\mathsf{let}\ \mathtt{x} : \sigma\ \mathsf{in}\ \mathsf{C}$, which requires σ to have an instance and at the same time associates it with the name $\mathtt{x}$. Because the def form is more primitive, it is easier to work with at a low level, but it is no longer explicitly used after §1.3; we always use let instead.

1.3.4 DEFINITION: Environments Γ remain as in Definition 1.2.22, except DM type schemes $\mathtt{S}$ are replaced with constrained type schemes σ. The set of *free program identifiers* of an environment Γ, written $\mathit{fpi}(\Gamma)$, is defined by $\mathit{fpi}(\varnothing) = \varnothing$ and $\mathit{fpi}(\Gamma; \mathtt{x} : \sigma) = \mathit{fpi}(\Gamma) \cup \mathit{fpi}(\sigma)$. We write $\mathit{dfpi}(\Gamma)$ for $\mathit{dpi}(\Gamma) \cup \mathit{fpi}(\Gamma)$. We define $\mathsf{def}\ \varnothing\ \mathsf{in}\ \mathsf{C}$ as C and $\mathsf{def}\ \Gamma; \mathtt{x} : \sigma\ \mathsf{in}\ \mathsf{C}$ as $\mathsf{def}\ \Gamma\ \mathsf{in}\ \mathsf{def}\ \mathtt{x} : \sigma\ \mathsf{in}\ \mathsf{C}$. Similarly, we define $\mathsf{let}\ \varnothing\ \mathsf{in}\ \mathsf{C}$ as C and $\mathsf{let}\ \Gamma; \mathtt{x} : \sigma\ \mathsf{in}\ \mathsf{C}$ as $\mathsf{let}\ \Gamma\ \mathsf{in}\ \mathsf{let}\ \mathtt{x} : \sigma\ \mathsf{in}\ \mathsf{C}$. We define $\exists\varnothing$ as true and $\exists(\Gamma; \mathtt{x} : \sigma)$ as $\exists\Gamma \wedge \mathsf{def}\ \Gamma\ \mathsf{in}\ \exists\sigma$. □

These three constraint forms will be related by Lemma 1.3.37.

In order to establish or express certain laws of equivalence between constraints, we need *constraint contexts*. A constraint context is a constraint with zero, one, or several *holes*, written $[]$. The syntax of contexts is as follows:

$$\mathcal{C} ::= [] \mid \mathsf{C} \mid \mathcal{C} \wedge \mathcal{C} \mid \exists\bar{\mathtt{X}}.\mathcal{C} \mid \mathsf{def}\ \mathtt{x} : \sigma\ \mathsf{in}\ \mathcal{C} \mid \mathsf{def}\ \mathtt{x} : \forall\bar{\mathtt{X}}[\mathcal{C}].\mathtt{T}\ \mathsf{in}\ \mathsf{C}$$

The application of a constraint context $\mathcal{C}$ to a constraint C, written $\mathcal{C}[\mathsf{C}]$, is defined in the usual way. Because a constraint context may have any number of holes, C may disappear or be duplicated in the process. Because a hole may appear in the scope of a binder, some of C's free type variables and free program identifiers may become bound in $\mathcal{C}[\mathsf{C}]$. We write $\mathit{dtv}(\mathcal{C})$ and $\mathit{dpi}(\mathcal{C})$

for the sets of type variables and program identifiers, respectively, that $\mathcal{C}$ may thus capture. We write $\mathsf{let}\ x : \forall \bar{X}[\mathcal{C}].\tau\ \mathsf{in}\ C$ for $\exists \bar{X}.\mathcal{C} \wedge \mathsf{def}\ x : \forall \bar{X}[\mathcal{C}].\tau\ \mathsf{in}\ C$. (Being able to state such a definition is why we require multi-hole contexts.) We let $\mathcal{X}$ range over *existential constraint contexts*, defined by $\mathcal{X} ::= [] \mid \exists \bar{X}.\mathcal{X}$.

1.3.2 Meaning

We have defined the syntax of constraints and given an informal description of their meaning. We now give a formal definition of the interpretation of constraints. We begin with the definition of a *model*:

1.3.5 DEFINITION: For every kind κ, let $\mathcal{M}_\kappa$ be a nonempty set, whose elements are the *ground types* of kind κ. In the following, t ranges over $\mathcal{M}_\kappa$, for some κ that may be determined from the context. For every type constructor F of signature $K \Rightarrow \kappa$, let F denote a total function from $\mathcal{M}_K$ into $\mathcal{M}_\kappa$, where the indexed product $\mathcal{M}_K$ is the set of all mappings of domain $dom(K)$ that map every $d \in dom(K)$ to an element of $\mathcal{M}_{K(d)}$. For every predicate P of signature $\kappa_1 \otimes \ldots \otimes \kappa_n \Rightarrow \cdot$, let P denote a predicate on $\mathcal{M}_{\kappa_1} \times \ldots \times \mathcal{M}_{\kappa_n}$. For every kind κ, we require the predicate $=_\kappa$ to be equality on $\mathcal{M}_\kappa$ and the predicate $\leq_\kappa$ to be a partial order on $\mathcal{M}_\kappa$. □

For the sake of convenience, we abuse notation and write F for both the type constructor and its interpretation; similarly for predicates.

By varying the set of type constructors, the set of predicates, the set of ground types, and the interpretation of type constructors and predicates, one may define an entire family of related type systems. We informally refer to the collection of these choices as X. Thus, the type systems HM(X) and PCB(X), described in §1.4 and §1.5, are *parameterized* by X.

The following examples give standard ways of defining the set of ground types and the interpretation of type constructors.

1.3.6 EXAMPLE [SYNTACTIC MODELS]: For every kind κ, let $\mathcal{M}_\kappa$ consist of the *closed* types of kind κ. Then, ground types are types that do not have any free type variables, and form the so-called *Herbrand universe*. Let every type constructor F be interpreted as itself. Models that define ground types and interpret type constructors in this manner are referred to as *syntactic*. □

1.3.7 EXAMPLE [TREE MODELS]: Let a *path* π be a finite sequence of directions. The empty path is written ϵ and the concatenation of the paths π and π' is written $\pi \cdot \pi'$. Let a *tree* be a partial function t from paths to type constructors whose domain is nonempty and prefix-closed and such that, for every path π in the domain of t, if the type constructor $t(\pi)$ has signature $K \Rightarrow \kappa$, then $\pi \cdot d \in dom(t)$ is equivalent to $d \in dom(K)$ and, furthermore, for every $d \in$

dom(K), the type constructor $t(\pi \cdot d)$ has image kind $K(d)$. If π is in the domain of t, then the *subtree* of t rooted at π, written t/π, is the partial function $\pi' \mapsto t(\pi \cdot \pi')$. A tree is *finite* if and only if it has finite domain. A tree is *regular* if and only if it has a finite number of distinct subtrees. Every finite tree is thus regular. Let $\mathcal{M}_\kappa$ consist of the *finite* (resp. *regular*) trees t such that $t(\epsilon)$ has image kind κ: then, we have a *finite* (resp. *regular*) *tree model*.

If F has signature $K \Rightarrow \kappa$, one may interpret F as the function that maps $T \in \mathcal{M}_K$ to the ground type $t \in \mathcal{M}_\kappa$ defined by $t(\epsilon) = F$ and $t/d = T(d)$ for $d \in dom(T)$, that is, the unique ground type whose head symbol is F and whose subtree rooted at d is $T(d)$. Then, we have a *free* tree model. Please note that free finite tree models coincide with syntactic models, as defined in the previous example. □

Rows (§1.11) are interpreted in a tree model, albeit not a free one. The following examples suggest different ways of interpreting the subtyping predicate.

1.3.8 EXAMPLE [EQUALITY MODELS]: The simplest way of interpreting the subtyping predicate is to let $\leq$ denote equality on every $\mathcal{M}_\kappa$. Models that do so are referred to as *equality models*. When no predicate other than equality is available, we say that the model is *equality-only*. □

1.3.9 EXAMPLE [STRUCTURAL, NONSTRUCTURAL SUBTYPING]: Let a *variance* ν be a nonempty subset of $\{-, +\}$, written $-$ (*contravariant*), $+$ (*covariant*), or $\pm$ (*invariant*) for short. Define the *composition* of two variances as an associative, commutative operation with $+$ as neutral element, $\pm$ as absorbing element (that is, $\pm- = \pm+ = \pm\pm = \pm$), and such that $-- = +$. Now, consider a free (finite or regular) tree model, where every direction d comes with a fixed variance $\nu(d)$. Define the variance $\nu(\pi)$ of a path π as the composition of the variances of its elements. Let $\leqslant$ be a partial order on type constructors such that (i) if $F_1 \leqslant F_2$ holds and F_1 and F_2 have signature $K_1 \Rightarrow \kappa_1$ and $K_2 \Rightarrow \kappa_2$, respectively, then K_1 and K_2 agree on the intersection of their domains and κ_1 and κ_2 coincide; and (ii) $F_0 \leqslant F_1 \leqslant F_2$ implies $dom(F_0) \cap dom(F_2) \subseteq dom(F_1)$. Let $\leqslant^+$, $\leqslant^-$, and $\leqslant^\pm$ stand for $\leqslant$, $\geqslant$, and $=$, respectively. Then, define the interpretation of subtyping as follows: if $t_1, t_2 \in \mathcal{M}_\kappa$, let $t_1 \leq t_2$ hold if and only if, for every path $\pi \in dom(t_1) \cap dom(t_2)$, $t_1(\pi) \leqslant^{\nu(\pi)} t_2(\pi)$ holds. It is not difficult to check that $\leq$ is a partial order on every $\mathcal{M}_\kappa$. The reader is referred to (Amadio and Cardelli, 1993; Kozen, Palsberg, and Schwartzbach, 1995; Brandt and Henglein, 1997) for more details about this construction. Models that define subtyping in this manner are referred to as *nonstructural subtyping models*.

A simple nonstructural subtyping model is obtained by letting the directions *domain* and *codomain* be contra- and covariant, respectively; introduc-

ing, in addition to the type constructor $\rightarrow$, two type constructors $\bot$ and $\top$ of signature $\star$; and letting $\bot \leqslant \rightarrow \leqslant \top$. This gives rise to a model where $\bot$ is the least ground type, $\top$ is the greatest ground type, and the arrow type constructor is, as usual, contravariant in its domain and covariant in its codomain. This form of subtyping is called *nonstructural* because comparable ground types may have different shapes: consider, for instance, $\bot$ and $\bot \rightarrow \top$.

A typical use of nonstructural subtyping is in type systems for records. One may, for instance, introduce a covariant direction *content* of kind $\star$, a kind $\circ$, a type constructor abs of signature $\circ$, a type constructor pre of signature $\{\mathit{content} \mapsto \star\} \Rightarrow \circ$, and let $\mathsf{pre} \leqslant \mathsf{abs}$. This gives rise to a model where $\mathsf{pre}\ t \leq \mathsf{abs}$ holds for every $t \in \mathcal{M}_\star$. Again, comparable ground types may have different shapes: consider, for instance, $\mathsf{pre}\ \top$ and abs. §1.11 says more about typechecking operations on records.

Nonstructural subtyping has been studied, for example, in (Kozen, Palsberg, and Schwartzbach, 1995; Palsberg, Wand, and O'Keefe, 1997; Jim and Palsberg, 1999; Pottier, 2001b; Su, Aiken, Niehren, Priesnitz, and Treinen, 2002; Niehren and Priesnitz, 2003).

An important particular case arises when any two type constructors related by $\leqslant$ have the same arity (and thus also the same signatures). In that case, it is not difficult to show that *any two ground types related by subtyping must have the same shape*, that is, if $t_1 \leq t_2$ holds, then $\mathit{dom}(t_1)$ and $\mathit{dom}(t_2)$ must coincide. For this reason, such an interpretation of subtyping is usually referred to as *atomic* or *structural* subtyping. It has been studied in the finite (Mitchell, 1984, 1991b; Tiuryn, 1992; Pratt and Tiuryn, 1996; Frey, 1997; Rehof, 1997; Kuncak and Rinard, 2003; Simonet, 2003) and regular (Tiuryn and Wand, 1993) cases. Structural subtyping is often used in automated program analyses that enrich standard types with atomic annotations without altering their shape. □

Our last example suggests a predicate other than equality and subtyping.

1.3.10 EXAMPLE [CONDITIONAL CONSTRAINTS]: Consider a nonstructural subtyping model. For every type constructor F of image kind κ and for every kind κ', let $(\mathsf{F} \leqslant \cdot \Rightarrow \cdot \leq \cdot)$ be a predicate of signature $\kappa \otimes \kappa' \otimes \kappa' \Rightarrow \cdot$. Thus, if $\mathtt{T}_0$ has kind κ and $\mathtt{T}_1$, $\mathtt{T}_2$ have the same kind, then $\mathsf{F} \leqslant \mathtt{T}_0 \Rightarrow \mathtt{T}_1 \leq \mathtt{T}_2$ is a well-formed constraint, called a *conditional subtyping constraint*. Its interpretation is defined as follows: if $t_0 \in \mathcal{M}_\kappa$ and $t_1, t_2 \in \mathcal{M}_{\kappa'}$, then $\mathsf{F} \leqslant t_0 \Rightarrow t_1 \leq t_2$ holds if and only if $\mathsf{F} \leqslant t_0(\epsilon)$ implies $t_1 \leq t_2$. In other words, if t_0's head symbol exceeds F according to the ordering on type constructors, then the subtyping constraint $t_1 \leq t_2$ must hold; otherwise, the conditional constraint holds vacuously. Conditional constraints have been studied *e.g.* in (Reynolds,

$$\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ \mathsf{true} \qquad \text{(CM-TRUE)}$$

$$\frac{P(\phi(\mathtt{T}_1), \ldots, \phi(\mathtt{T}_n))}{\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ P\ \mathtt{T}_1\ \ldots\ \mathtt{T}_n} \qquad \text{(CM-PREDICATE)}$$

$$\frac{\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C_1 \quad \phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C_2}{\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ (C_1 \wedge C_2)} \qquad \text{(CM-AND)}$$

$$\frac{\phi[\vec{X} \mapsto \vec{t}] \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C \quad \bar{X}\ \#\ ftv(\Gamma)}{\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ \exists \bar{X}.C} \qquad \text{(CM-EXISTS)}$$

$$\frac{\phi \models \mathsf{def}\ \Gamma_1\ \mathsf{in}\ \sigma \preceq \mathtt{T}' \quad x \notin dpi(\Gamma_2)}{\phi \models \mathsf{def}\ \Gamma_1; x : \sigma; \Gamma_2\ \mathsf{in}\ x \preceq \mathtt{T}'} \qquad \text{(CM-INSTANCE)}$$

Figure 1-5: Meaning of constraints

1969a; Heintze, 1993; Aiken, Wimmers, and Lakshman, 1994; Pottier, 2000; Su and Aiken, 2001). □

Many other kinds of constraints exist, which we lack space to list; see (Comon, 1993) for a short survey.

Throughout this chapter, we assume (unless otherwise stated) that the set of type constructors, the set of predicates, and the model—which, together, form the parameter X—are arbitrary and fixed.

As usual, the meaning of a constraint is a function of the meaning of its free type variables, which is given by a *ground assignment*. The meaning of free program identifiers may be defined as part of the constraint, if desired, using a def prefix, so it need not be given by a separate assignment.

1.3.11 DEFINITION: A *ground assignment* ϕ is a total, kind-preserving mapping from $\mathcal{V}$ into $\mathcal{M}$. Ground assignments are extended to types by $\phi(\mathtt{F}\ \mathtt{T}_1\ \ldots\ \mathtt{T}_n) = \mathtt{F}(\phi(\mathtt{T}_1), \ldots, \phi(\mathtt{T}_n))$. Then, for every type $\mathtt{T}$ of kind κ, $\phi(\mathtt{T})$ is a ground type of kind κ. Whether a constraint C holds under a ground assignment ϕ, written $\phi \models C$ (read: ϕ *satisfies* C), is defined by the rules in Figure 1-5. A constraint C is *satisfiable* if and only if $\phi \models C$ holds for some ϕ. It is *false* if and only if $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C$ holds for *no* ground assignment ϕ and environment Γ. □

Let us now explain the rules that define constraint satisfaction (Figure 1-5). They are syntax-directed: that is, to a given constraint, at most one rule applies. It is determined by the nature of the first construct that appears under a maximal def prefix. CM-TRUE states that a constraint of the form def Γ in true is a tautology, that is, holds under every ground assignment. No rule matches constraints of the form def Γ in false, which means that such constraints do not have a solution. CM-PREDICATE states that the meaning

of a predicate application is given by the predicate's interpretation within the model. More specifically, if P's signature is $\kappa_1 \otimes \ldots \otimes \kappa_n \Rightarrow \cdot$, then, by well-formedness of the constraint, every T_i is of kind κ_i, so $\phi(T_i)$ is a ground type in $\mathcal{M}_{\kappa_i}$. By Definition 1.3.5, P denotes a predicate on $\mathcal{M}_{\kappa_1} \times \ldots \times \mathcal{M}_{\kappa_n}$, so the rule's premise is mathematically well-formed. It is independent of Γ, which is natural, since a predicate application has no free program identifiers. CM-AND requires each of the conjuncts to be valid in isolation. The information in Γ is made available to each branch. CM-EXISTS allows the type variables $\vec{X}$ to denote arbitrary ground types $\vec{t}$ within C, independently of their image through ϕ. We implicitly require $\vec{X}$ and $\vec{t}$ to have matching kinds, so that $\phi[\vec{X} \mapsto \vec{t}]$ remains a kind-preserving ground assignment. The side condition $\bar{X} \# \mathit{ftv}(\Gamma)$—which may always be satisfied by suitable α-conversion of the constraint $\exists \bar{X}.C$—prevents free occurrences of the type variables $\bar{X}$ within Γ from being unduly affected. CM-INSTANCE concerns constraints of the form $\mathsf{def}\ \Gamma\ \mathsf{in}\ x \preceq T'$. The constraint $x \preceq T'$ is turned into $\sigma \preceq T'$, where, according to the second premise, σ is $\Gamma(x)$. Please recall that constraints of such a form were introduced in Definition 1.3.3. The environment Γ is replaced with a suitable prefix of itself, namely Γ_1, so that the free program identifiers of σ retain their meaning.

It is intuitively clear that the constraints $\mathsf{def}\ x : \sigma\ \mathsf{in}\ C$ and $[x \mapsto \sigma]C$ have the same meaning, where the latter denotes the capture-avoiding substitution of σ for x throughout C. As a matter of fact, it would have been possible to use this equivalence as a *definition* of the meaning of def constraints, but the present style is pleasant as well. A related equivalence law is established and exploited below (Lemma 1.3.42). This confirms our (informal) claim that the def form is an explicit substitution form.

It is possible for a constraint to be neither satisfiable nor false. Consider, for instance, the constraint $\exists Z.x \preceq Z$. Because the identifier x is free, CM-INSTANCE is not applicable, so the constraint is not satisfiable. Furthermore, placing it within the context $\mathsf{let}\ x : \forall X.X\ \mathsf{in}\ []$ makes it satisfied by every ground assignment, so it is not false. Here, the assertions "C is satisfiable" and "C is false" are complementary when $\mathit{fpi}(C) = \varnothing$ holds, whereas in a standard first-order logic, they always are.

In a judgement of the form $\phi \models C$, the ground assignment ϕ applies to the free type variables of C. This is made precise by the two following statements. In the second one, $\circ$ is composition and $\theta(C)$ is the capture-avoiding application of the type substitution θ to C.

1.3.12 LEMMA: If $\bar{X} \# \mathit{ftv}(C)$ holds, then $\phi \models C$ and $\phi[\vec{X} \mapsto \vec{t}] \models C$ are equivalent. □

1.3.13 LEMMA: $\phi \circ \theta \models C$ and $\phi \models \theta(C)$ are equivalent. □

Because constraints lie at the heart of our treatment of ML-the-type-system, most of our proofs involve establishing logical properties of constraints. These properties are usually not stated in terms of the satisfaction predicate $\models$, which is too low-level. Instead, we reason in terms of *entailment* or *equivalence* assertions. Let us first define these notions.

1.3.14 DEFINITION: We write $C_1 \Vdash C_2$, and say that C_1 *entails* C_2, if and only if, for every ground assignment ϕ and for every environment Γ, $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C_1$ implies $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C_2$. We write $C_1 \equiv C_2$, and say that C_1 and C_2 are *equivalent*, if and only if $C_1 \Vdash C_2$ and $C_2 \Vdash C_1$ hold. □

This definition measures the strength of a constraint by the set of pairs (ϕ, Γ) that satisfy it, and considers a constraint stronger if fewer such pairs satisfy it. In other words, C_1 entails C_2 when C_1 imposes stricter requirements on its free type variables and program identifiers than C_2 does. We remark that C is false if and only if $C \equiv \mathsf{false}$ holds. It is straightforward to check that entailment is reflexive and transitive and that $\equiv$ is indeed an equivalence relation.

The fact that the predicates $=$ and $\leq$ are respectively interpreted as equality and as an ordering implies that the constraints $\mathtt{T}_1 = \mathtt{T}_2$ and $\mathtt{T}_1 \leq \mathtt{T}_2 \wedge \mathtt{T}_2 \leq \mathtt{T}_1$ are equivalent.

We immediately exploit the notion of constraint equivalence to define what it means for a type constructor to be covariant, contravariant, or invariant with respect to one of its parameters. Let $\mathtt{F}$ be a type constructor of signature $\kappa_1 \otimes \ldots \otimes \kappa_n \Rightarrow \kappa$. Let $i \in \{1, \ldots, n\}$. $\mathtt{F}$ is *covariant* (resp. *contravariant*, *invariant*) with respect to its i^{th} parameter if and only if, for all types $\mathtt{T}_1, \ldots, \mathtt{T}_n$ and $\mathtt{T}_i'$ of appropriate kinds, the constraint $\mathtt{F}\,\mathtt{T}_1 \ldots \mathtt{T}_i \ldots \mathtt{T}_n \leq \mathtt{F}\,\mathtt{T}_1 \ldots \mathtt{T}_i' \ldots \mathtt{T}_n$ is equivalent to $\mathtt{T}_i \leq \mathtt{T}_i'$ (resp. $\mathtt{T}_i' \leq \mathtt{T}_i$, $\mathtt{T}_i = \mathtt{T}_i'$). We let the reader check the following facts: (i) in an equality model, these three notions coincide; (ii) in an equality free tree model, every type constructor is invariant with respect to each of its parameters; and (iii) in a nonstructural subtyping model, if the direction d has been declared covariant (resp. contravariant, invariant), then every type constructor whose arity includes d is covariant (resp. contravariant, invariant) with respect to d. In the following, *we require the type constructor $\rightarrow$ to be contravariant with respect to its domain and covariant with respect to its codomain*—a standard requirement in type systems with subtyping (TAPL Chapter 15). This requirement is summed up by the following equivalence law:

$$\mathtt{T}_1 \rightarrow \mathtt{T}_2 \leq \mathtt{T}_1' \rightarrow \mathtt{T}_2' \equiv \mathtt{T}_1' \leq \mathtt{T}_1 \wedge \mathtt{T}_2 \leq \mathtt{T}_2' \qquad \text{(C-ARROW)}$$

Please note that this requirement bears on the interpretation of types and of the subtyping predicate. In an equality free tree model, by (i) and (ii) above, it is always satisfied. In a nonstructural subtyping model, it boils down to

requiring that the directions *domain* and *codomain* be declared contravariant and covariant, respectively. In the general case, we do not have any knowledge of the model, and cannot formulate a more precise requirement. Thus, it is up to the designer of the model to ensure that C-ARROW holds.

We also exploit the notion of constraint equivalence to define what it means for two type constructors to be incompatible. Two type constructors F_1 and F_2 with the same image kind are *incompatible* if and only if all constraints of the form $\mathsf{F}_1\ \vec{\mathsf{T}}_1 \leq \mathsf{F}_2\ \vec{\mathsf{T}}_2$ and $\mathsf{F}_2\ \vec{\mathsf{T}}_2 \leq \mathsf{F}_1\ \vec{\mathsf{T}}_1$ are false. Please note that in an equality free tree model, any two distinct type constructors are incompatible. In the following, we often indicate that a newly introduced type constructor must be *isolated*. We implicitly require that, whenever both F_1 and F_2 are isolated, F_1 and F_2 be incompatible. Thus, the notion of "isolation" provides a concise and modular way of stating a collection of incompatibility requirements. We require the type constructor $\rightarrow$ to be isolated.

1.3.3 Reasoning with constraints

In this section, we give a number of entailment and equivalence laws that are often useful and help understand the meaning of constraints. The following lemma provides a slightly weaker characterization of entailment, and is sometimes a useful tool in establishing entailment assertions.

1.3.15 LEMMA: Let $\bar{\mathsf{X}}$ be an arbitrary finite set of type variables. If, for every ground assignment ϕ and for every environment Γ such that $\bar{\mathsf{X}} \# \mathit{ftv}(\Gamma), \phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ \mathsf{C}_1$ implies $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ \mathsf{C}_2$, then C_1 entails C_2. □

Proof: Let us remark that, if the hypothesis $\bar{\mathsf{X}} \# \mathit{ftv}(\mathsf{C}_1, \mathsf{C}_2)$ was added to the statement of the present Lemma, then the result would follow immediately via a simple renaming argument. In the absence of this extra hypothesis, however, we must rely on Lemma 1.3.13 and on the fact that entailment involves a universal quantification on ϕ, as we shall now do.

Let $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ \mathsf{C}_1$ **(1)**, where $\bar{\mathsf{X}}$ and $\mathit{ftv}(\Gamma)$ are not necessarily disjoint. Let $\bar{\mathsf{Y}} = \mathit{ftv}(\Gamma)$ **(2)**. Let $[\vec{\mathsf{Y}} \mapsto \vec{\mathsf{Z}}]$ be a one-to-one, kind-preserving mapping of $\bar{\mathsf{Y}}$ onto a set of type variables $\bar{\mathsf{Z}}$, where $\bar{\mathsf{Z}} \# \bar{\mathsf{X}}$ **(3)** and $\bar{\mathsf{Z}} \# \mathit{ftv}(\mathsf{C}_1, \mathsf{C}_2)$ **(4)**. We may view $[\vec{\mathsf{Z}} \mapsto \vec{\mathsf{Y}}]$ as a type substitution of domain $\bar{\mathsf{Z}}$. (Note that it is *not* a renaming, because even though it is a bijection when viewed as a *finite* mapping, it is no longer one when viewed as a *total* mapping.) Then, thanks to (2) and (4), the constraint $\mathsf{def}\ \Gamma\ \mathsf{in}\ \mathsf{C}_1$ may be written $[\vec{\mathsf{Z}} \mapsto \vec{\mathsf{Y}}](\mathsf{def}\ [\vec{\mathsf{Y}} \mapsto \vec{\mathsf{Z}}]\Gamma\ \mathsf{in}\ \mathsf{C}_1)$. By this observation and by Lemma 1.3.13, (1) may be written $\phi \circ [\vec{\mathsf{Z}} \mapsto \vec{\mathsf{Y}}] \models \mathsf{def}\ [\vec{\mathsf{Y}} \mapsto \vec{\mathsf{Z}}]\Gamma\ \mathsf{in}\ \mathsf{C}_1$ **(5)**. Now, we have $\mathit{ftv}([\vec{\mathsf{Y}} \mapsto \vec{\mathsf{Z}}]\Gamma) = \bar{\mathsf{Z}}$, which by (3) is disjoint with $\bar{\mathsf{X}}$. Thus, the Lemma's main hypothesis applies to (5) and yields

$\phi \circ [Z \mapsto Y] \models \mathsf{def}\ [\vec{Y} \mapsto \vec{Z}]\Gamma\ \mathsf{in}\ C_2$. Performing the same steps in reverse, we find that this assertion may be written $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C_2$. □

The remainder of this section offers a constraint manipulation toolbox. First, we prove that entailment and equivalence are preserved by conjunction, existential quantification, and def forms. Then, we establish a number of entailment and equivalence laws that are heavily used in the forthcoming sections. Entailment is preserved by conjunction and by existential quantification. In the following, this property is often used implicitly.

1.3.16 LEMMA: $C_1 \Vdash C_2$ and $C_1' \Vdash C_2'$ imply $C_1 \wedge C_1' \Vdash C_2 \wedge C_2'$. □

Proof: Assume $C_1 \Vdash C_2$ **(1)** and $C_1' \Vdash C_2'$ **(2)** and $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C_1 \wedge C_1'$ **(3)**. By CM-AND, (3) implies $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C_1$ **(4)** and $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C_1'$ **(5)**. Together, (1) and (4) imply $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C_2$ **(6)**. Similarly, (2) and (5) imply $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C_2'$ **(7)**. By CM-AND, (6) and (7) yield $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C_2 \wedge C_2'$. Discharging (3) yields the goal. □

1.3.17 LEMMA: $C_1 \Vdash C_2$ implies $\exists \bar{X}.C_1 \Vdash \exists \bar{X}.C_2$. □

Proof: Assume $C_1 \Vdash C_2$ **(1)** and $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ \exists \bar{X}.C_1$ **(2)** and $\bar{X} \# \mathit{ftv}(\Gamma)$ **(3)**. By CM-EXISTS, (2) and (3) imply $\phi[\vec{X} \mapsto \vec{t}] \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C_1$ **(4)** for some $\vec{t}$. By definition of entailment, (1) and (4) imply $\phi[\vec{X} \mapsto \vec{t}] \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C_2$ which, by (3) and CM-EXISTS again, yields $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ \exists \bar{X}.C_2$. By Lemma 1.3.15, discharging (2) and (3) yields the goal. □

Conjunction is commutative and associative. Adding a new member to a conjunction strengthens it. Furthermore, a conjunct is redundant if it is entailed by another conjunct. In the following, these properties are often used implicitly.

1.3.18 LEMMA: $C_1 \wedge C_2 \equiv C_2 \wedge C_1$. $(C_1 \wedge C_2) \wedge C_3 \equiv C_1 \wedge (C_2 \wedge C_3)$. □

Proof: By CM-AND, both $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ (C_1 \wedge C_2)$ and $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ (C_2 \wedge C_1)$ are equivalent to $(\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C_1) \wedge (\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C_2)$. Similarly, both $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ ((C_1 \wedge C_2) \wedge C_3)$ and $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ (C_1 \wedge (C_2 \wedge C_3))$ are equivalent to $(\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C_1) \wedge (\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C_2) \wedge (\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C_3)$. □

1.3.19 LEMMA: $C_1 \wedge C_2 \Vdash C_1$. Furthermore, $C_1 \Vdash C_2$ implies $C_1 \Vdash C_1 \wedge C_2$. □

Proof: By CM-AND, $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ (C_1 \wedge C_2)$ implies $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C_1$, so $C_1 \wedge C_2$ entails C_1. Conversely, assume $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C_1$. Then, given the hypothesis $C_1 \Vdash C_2$, we have $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C_2$ as well. By CM-AND, the two imply $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ (C_1 \wedge C_2)$. So, C_1 entails $C_1 \wedge C_2$. □

The following laws are known as *scope extrusion* of the existential quantifier.

1.3.20 LEMMA: If $\bar{X} \# \mathit{ftv}(C_2)$ then $\exists \bar{X}.(C_1 \wedge C_2) \equiv (\exists \bar{X}.C_1) \wedge C_2$. □

Proof: Assume $\bar{X} \# \mathit{ftv}(C_2)$ **(1)** and $\bar{X} \# \mathit{ftv}(\Gamma)$ **(2)**. By (2), CM-EXISTS, and CM-AND, the assertion $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ \exists \bar{X}.(C_1 \wedge C_2)$ **(3)** is equivalent to the existence of $\vec{t}$ such that $\phi[\vec{X} \mapsto \vec{t}] \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C_1 \wedge \phi[\vec{X} \mapsto \vec{t}] \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C_2$ **(4)** holds. By (1), (2), and Lemma 1.3.12, the second conjunct of (4) is equivalent to $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C_2$. Thus, by (2), CM-EXISTS, and CM-AND, (3) is equivalent to $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ (\exists \bar{X}.C_1) \wedge C_2$. Lemma 1.3.15 allows discharging (2) and yields the goal. □

1.3.21 LEMMA: If $\bar{X} \# \mathit{ftv}(\sigma)$ then $\exists \bar{X}.\mathsf{def}\ x : \sigma\ \mathsf{in}\ C \equiv \mathsf{def}\ x : \sigma\ \mathsf{in}\ \exists \bar{X}.C$. □

Proof: Assume $\bar{X} \# \mathit{ftv}(\sigma)$ **(1)** and $\bar{X} \# \mathit{ftv}(\Gamma)$ **(2)**. By (2) and CM-EXISTS, the assertion $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ \exists \bar{X}.\mathsf{def}\ x : \sigma\ \mathsf{in}\ C$ is equivalent to the existence of $\vec{t}$ such that $\phi[\vec{X} \mapsto \vec{t}] \models \mathsf{def}\ \Gamma; x : \sigma\ \mathsf{in}\ C$ holds, which, by (1), (2), and CM-EXISTS, is equivalent to $\phi \models \mathsf{def}\ \Gamma; x : \sigma\ \mathsf{in}\ \exists \bar{X}.C$. Lemma 1.3.15 allows discharging (2) and yields the goal. □

Here are some further properties of existential quantification.

1.3.22 LEMMA: $C \Vdash \exists \bar{X}.C$. □

Proof: Assume $\bar{X} \# \mathit{ftv}(\Gamma)$ **(1)** and $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C$ **(2)**. Applying CM-EXISTS to (1) and (2) yields $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ \exists \bar{X}.C$. Lemma 1.3.15 allows discharging (1) and yields the goal. □

1.3.23 LEMMA: $\exists \bar{X}.\exists \bar{Y}.C \equiv \exists \bar{X}\bar{Y}.C$. □

Proof: Let $\bar{X}\bar{Y} \# \mathit{ftv}(\Gamma)$ **(1)**. We find, by (1) and CM-EXISTS, that the assertions $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ \exists \bar{X}.\exists \bar{Y}.C$ and $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ \exists \bar{X}\bar{Y}.C$ are both equivalent to the existence of a ground assignment ϕ' such that ϕ and ϕ' agree outside of $\bar{X}\bar{Y}$ and $\phi' \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C$ holds. Lemma 1.3.15 allows discharging (1) and yields the goal. □

The following result states that it is equivalent for a type T' to be an instance of σ or to be a supertype of some instance of σ.

1.3.24 LEMMA: If $Z \notin \mathit{ftv}(\sigma, T')$ then $\sigma \preceq T' \equiv \exists Z.(\sigma \preceq Z \wedge Z \leq T')$. □

Proof: Assume $\mathtt{Z} \notin \mathit{ftv}(\sigma, \mathtt{T}')$ **(1)** and $\mathtt{Z} \notin \mathit{ftv}(\Gamma)$ **(2)**. Write σ as $\forall \bar{\mathtt{X}}[\mathsf{C}].\mathtt{T}$, where $\bar{\mathtt{X}} \# \mathit{ftv}(\mathtt{T}', \mathtt{Z}, \Gamma)$ **(3)**. Then, the following statements are equivalent:

$$
\begin{array}{rl}
 & \phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ \forall \bar{\mathtt{X}}[\mathsf{C}].\mathtt{T} \preceq \mathtt{T}' \\
 & \phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ \exists \bar{\mathtt{X}}.(\mathsf{C} \wedge \mathtt{T} \leq \mathtt{T}') \qquad \textbf{(4)} \\
\exists \vec{\mathtt{t}} & \phi[\vec{\mathtt{X}} \mapsto \vec{\mathtt{t}}] \models \mathsf{def}\ \Gamma\ \mathsf{in}\ \mathsf{C} \wedge \phi[\vec{\mathtt{X}} \mapsto \vec{\mathtt{t}}](\mathtt{T}) \leq \phi(\mathtt{T}') \qquad \textbf{(5)} \\
\exists \mathtt{t} \exists \vec{\mathtt{t}} & \phi[\vec{\mathtt{X}} \mapsto \vec{\mathtt{t}}] \models \mathsf{def}\ \Gamma\ \mathsf{in}\ \mathsf{C} \wedge \phi[\vec{\mathtt{X}} \mapsto \vec{\mathtt{t}}](\mathtt{T}) \leq \mathtt{t} \wedge \mathtt{t} \leq \phi(\mathtt{T}') \qquad \textbf{(6)} \\
\exists \mathtt{t} \exists \vec{\mathtt{t}} & \phi'[\vec{\mathtt{X}} \mapsto \vec{\mathtt{t}}] \models \mathsf{def}\ \Gamma\ \mathsf{in}\ \mathsf{C} \wedge \phi'[\vec{\mathtt{X}} \mapsto \vec{\mathtt{t}}](\mathtt{T}) \leq \phi'(\mathtt{Z}) \wedge \phi'(\mathtt{Z}) \leq \phi'(\mathtt{T}') \qquad \textbf{(7)} \\
\exists \mathtt{t} & \phi' \models \mathsf{def}\ \Gamma\ \mathsf{in}\ \forall \bar{\mathtt{X}}[\mathsf{C}].\mathtt{T} \preceq \mathtt{Z} \wedge \phi'(\mathtt{Z}) \leq \phi'(\mathtt{T}') \qquad \textbf{(8)} \\
 & \phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ \exists \mathtt{Z}.(\forall \bar{\mathtt{X}}[\mathsf{C}].\mathtt{T} \preceq \mathtt{Z} \wedge \mathtt{Z} \leq \mathtt{T}') \qquad \textbf{(9)}
\end{array}
$$

Indeed, (4) follows from (3) and Definition 1.3.3; (5) is by (3), CM-EXISTS, CM-AND, and CM-PREDICATE; (6) follows from the fact that subtyping in the model is reflexive and transitive; in (7), ϕ' stands for $\phi[\mathtt{Z} \mapsto \mathtt{t}]$, and the equivalence follows from Lemma 1.3.12 because (1), (2), and (3) imply $\mathtt{Z} \notin \mathit{ftv}(\Gamma, \mathsf{C}, \mathtt{T}, \mathtt{T}')$; (8) is by (3), CM-PREDICATE, CM-AND, CM-EXISTS, and by Definition 1.3.3; (9) is by CM-PREDICATE, CM-AND, (2), and CM-EXISTS. Lemma 1.3.15 allows discharging (2) and yields the goal. □

The following lemma states that any supertype of an instance of σ is also an instance of σ.

1.3.25 LEMMA: $\sigma \preceq \mathtt{T} \wedge \mathtt{T} \leq \mathtt{T}' \Vdash \sigma \preceq \mathtt{T}'$. □

Proof: Let $\mathtt{Z} \notin \mathit{ftv}(\sigma, \mathtt{T}, \mathtt{T}')$ **(1)**. By (1) and Lemma 1.3.24, the constraint $\sigma \preceq \mathtt{T} \wedge \mathtt{T} \leq \mathtt{T}'$ **(2)** is equivalent to $\exists \mathtt{Z}.(\sigma \preceq \mathtt{Z} \wedge \mathtt{Z} \leq \mathtt{T}) \wedge \mathtt{T} \leq \mathtt{T}'$ **(3)**. By (1) and Lemma 1.3.20, (3) is equivalent to $\exists \mathtt{Z}.(\sigma \preceq \mathtt{Z} \wedge \mathtt{Z} \leq \mathtt{T} \wedge \mathtt{T} \leq \mathtt{T}')$ **(4)**. Because $\mathtt{Z} \leq \mathtt{T} \wedge \mathtt{T} \leq \mathtt{T}' \Vdash \mathtt{Z} \leq \mathtt{T}'$ holds, (4) entails $\exists \mathtt{Z}.(\sigma \preceq \mathtt{Z} \wedge \mathtt{Z} \leq \mathtt{T}')$, which by (1) and Lemma 1.3.24 is equivalent to $\sigma \preceq \mathtt{T}'$. □

The next lemma gives another interesting simplification law.

1.3.26 LEMMA: $\mathtt{X} \notin \mathit{ftv}(\mathtt{T})$ implies $\exists \mathtt{X}.(\mathtt{X} = \mathtt{T}) \equiv \mathsf{true}$. □

Proof: Let $\mathtt{X} \notin \mathit{ftv}(\mathtt{T})$ **(1)**. The constraint $\exists \mathtt{X}.(\mathtt{X} = \mathtt{T})$ may be written $\exists \mathtt{X}.(\mathtt{T} \leq \mathtt{X} \wedge \mathtt{X} \leq \mathtt{T})$. By (1) and Lemma 1.3.24, it is equivalent to $\mathtt{T} \leq \mathtt{T}$, which by CM-PREDICATE is equivalent to true. □

The following lemma states that, provided D is satisfied, the type $\mathtt{T}$ is an instance of the constrained type scheme $\forall \bar{\mathtt{X}}[\mathsf{D}].\mathtt{T}$.

1.3.27 LEMMA: $\mathsf{D} \Vdash \forall \bar{\mathtt{X}}[\mathsf{D}].\mathtt{T} \preceq \mathtt{T}$. □

Proof: Let $\mathtt{Z} \# \mathit{ftv}(\bar{\mathtt{X}}, \mathtt{D}, \mathtt{T})$ **(1)**. By Lemma 1.3.24, the constraint $\forall\bar{\mathtt{X}}[\mathtt{D}].\mathtt{T} \preceq \mathtt{T}$ is equivalent to $\exists\mathtt{Z}.(\forall\bar{\mathtt{X}}[\mathtt{D}].\mathtt{T} \preceq \mathtt{Z} \wedge \mathtt{Z} \leq \mathtt{T})$, which by (1) and Definition 1.3.3 is $\exists\mathtt{Z}.(\exists\bar{\mathtt{X}}.(\mathtt{D} \wedge \mathtt{T} \leq \mathtt{Z}) \wedge \mathtt{Z} \leq \mathtt{T})$ **(2)**. By Lemma 1.3.22 and by congruence of entailment, (2) is entailed by $\exists\mathtt{Z}.(\mathtt{D} \wedge \mathtt{T} \leq \mathtt{Z} \wedge \mathtt{Z} \leq \mathtt{T})$, that is, $\exists\mathtt{Z}.(\mathtt{D} \wedge \mathtt{Z} = \mathtt{T})$ **(3)**. By (1), C-EXAND, and Lemma 1.3.26, (3) is equivalent to D. □

This technical lemma helps justify Definition 1.3.29 below.

1.3.28 LEMMA: Let $\mathtt{Z} \notin \mathit{ftv}(\mathsf{C}, \sigma, \mathtt{T})$. Then, $\mathsf{C} \Vdash \sigma \preceq \mathtt{T}$ holds if and only if $\mathsf{C} \wedge \mathtt{T} \leq \mathtt{Z} \Vdash \sigma \preceq \mathtt{Z}$ holds. □

Proof: Assume C entails $\sigma \preceq \mathtt{T}$. Then, $\mathsf{C} \wedge \mathtt{T} \leq \mathtt{Z}$ entails $\sigma \preceq \mathtt{T} \wedge \mathtt{T} \leq \mathtt{Z}$, which, by Lemma 1.3.25, entails $\sigma \preceq \mathtt{Z}$.

Conversely, let $\mathtt{Z} \notin \mathit{ftv}(\mathsf{C}, \sigma, \mathtt{T})$ **(1)** and $\mathsf{C} \wedge \mathtt{T} \leq \mathtt{Z} \Vdash \sigma \preceq \mathtt{Z}$ **(2)**. By Lemmas 1.3.16 and 1.3.17, we have $\exists\mathtt{Z}.(\mathsf{C} \wedge \mathtt{T} \leq \mathtt{Z} \wedge \mathtt{Z} \leq \mathtt{T}) \Vdash \exists\mathtt{Z}.(\sigma \preceq \mathtt{Z} \wedge \mathtt{Z} \leq \mathtt{T})$ **(3)**. By Lemma 1.3.20, (1), and Lemma 1.3.26, the left-hand side of (3) is equivalent to C. By (1) and Lemma 1.3.24, the right-hand side of (3) is equivalent to $\sigma \preceq \mathtt{T}$. So, $\mathsf{C} \Vdash \sigma \preceq \mathtt{T}$ holds. □

It is useful to define what it means for a type scheme σ_1 to be *more general* than a type scheme σ_2. Our informal intent is for $\sigma_1 \preceq \sigma_2$ to mean: *every instance of* σ_2 *is an instance of* σ_1. In Definition 1.3.3, we have introduced the constraint form $\sigma \preceq \mathtt{T}$ as syntactic sugar. Similarly, one might wish to make $\sigma_1 \preceq \sigma_2$ a derived constraint form; however, this is impossible, because neither universal quantification nor implication are available in the constraint language. We can, however, exploit the fact that these logical connectives are implicit in entailment assertions by defining a judgement of the form $\mathsf{C} \Vdash \sigma_1 \preceq \sigma_2$, whose meaning is: *under the constraint* C, σ_1 *is more general than* σ_2.

1.3.29 DEFINITION: We write $\mathsf{C} \Vdash \sigma_1 \preceq \sigma_2$ if and only if $\mathtt{Z} \notin \mathit{ftv}(\mathsf{C}, \sigma_1, \sigma_2)$ implies $\mathsf{C} \wedge \sigma_2 \preceq \mathtt{Z} \Vdash \sigma_1 \preceq \mathtt{Z}$. We write $\mathsf{C} \Vdash \sigma_1 \equiv \sigma_2$ when both $\mathsf{C} \Vdash \sigma_1 \preceq \sigma_2$ and $\mathsf{C} \Vdash \sigma_2 \preceq \sigma_1$ hold. □

This notation is not ambiguous because the assertion $\mathsf{C} \Vdash \sigma \preceq \mathtt{T}$, whose meaning was initially given by Definitions 1.3.3 and 1.3.14, retains the same meaning under the new definition—this is shown by Lemma 1.3.28 above.

The next lemma shows that quantifying over a fresh Z in Definition 1.3.29 amounts to quantifying over an arbitrary type T.

1.3.30 LEMMA: $\mathsf{C} \Vdash \sigma_1 \preceq \sigma_2$ implies $\mathsf{C} \wedge \sigma_2 \preceq \mathtt{T} \Vdash \sigma_1 \preceq \mathtt{T}$. □

Proof: Assume $\mathtt{Z} \notin \mathit{ftv}(\mathsf{C}, \sigma_1, \sigma_2, \mathtt{T})$ **(1)** and $\mathsf{C} \wedge \sigma_2 \preceq \mathtt{Z} \Vdash \sigma_1 \preceq \mathtt{Z}$ **(2)**. Then,

we have

$$\begin{array}{lll} & C \wedge \sigma_2 \preceq \tau & \\ \equiv & C \wedge \exists Z.(\sigma_2 \preceq Z \wedge Z \leq \tau) & \textbf{(3)} \\ \equiv & \exists Z.(C \wedge \sigma_2 \preceq Z \wedge Z \leq \tau) & \textbf{(4)} \\ \Vdash & \exists Z.(\sigma_1 \preceq Z \wedge Z \leq \tau) & \textbf{(5)} \\ \equiv & \sigma_1 \preceq \tau & \textbf{(6)} \end{array}$$

where (3) is by (1) and Lemma 1.3.24; (4) is by (1) and Lemma 1.3.20; (5) follows from (2); and (6) is by (1) and Lemma 1.3.24 again. □

The next lemma provides a way of exploiting the ordering between type schemes introduced by Definition 1.3.29. It states that a type scheme occurs in *contravariant* position when it is within a def prefix. In other words, the more general the type scheme, the weaker the entire constraint. This is a key stepping stone for several forthcoming proofs.

1.3.31 LEMMA: $C \Vdash \sigma_1 \preceq \sigma_2$ implies $C \wedge \mathsf{def}\ x : \sigma_2\ \mathsf{in}\ D \Vdash \mathsf{def}\ x : \sigma_1\ \mathsf{in}\ D$. □

Proof: Assume $C \Vdash \sigma_1 \preceq \sigma_2$ **(1)**. Let us write D as $\mathsf{def}\ \Gamma'\ \mathsf{in}\ C'$, where C' is not a def form. By CM-AND, the goal is to prove that $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C$ **(2)** and $\phi \models \mathsf{def}\ \Gamma; x : \sigma_2; \Gamma'\ \mathsf{in}\ C'$ **(3)** imply $\phi \models \mathsf{def}\ \Gamma; x : \sigma_1; \Gamma'\ \mathsf{in}\ C'$. The proof is by induction on the derivation of (3).

∘ *Case* CM-TRUE, CM-PREDICATE. Immediate.

∘ *Case* CM-AND. C' is $C'_1 \wedge C'_2$. The rule's premises are $\phi \models \mathsf{def}\ \Gamma; x : \sigma_2; \Gamma'\ \mathsf{in}\ C'_1$ and $\phi \models \mathsf{def}\ \Gamma; x : \sigma_2; \Gamma'\ \mathsf{in}\ C'_2$. The induction hypothesis yields $\phi \models \mathsf{def}\ \Gamma; x : \sigma_1; \Gamma'\ \mathsf{in}\ C'_1$ and $\phi \models \mathsf{def}\ \Gamma; x : \sigma_1; \Gamma'\ \mathsf{in}\ C'_2$. The result follows by CM-AND.

∘ *Case* CM-EXISTS. C' is $\exists \bar{X}.C''$. The rule's premises are $\phi[\vec{X} \mapsto \vec{t}] \models \mathsf{def}\ \Gamma; x : \sigma_2; \Gamma'\ \mathsf{in}\ C''$ **(4)** and $\bar{X} \# \mathit{ftv}(\Gamma, \sigma_2, \Gamma')$ **(5)**. We may further require, *w.l.o.g.*, $\bar{X} \# \mathit{ftv}(\sigma_1, C)$ **(6)**. By (5), (6), and Lemma 1.3.12, (2) implies $\phi[\vec{X} \mapsto \vec{t}] \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C$. This allows applying the induction hypothesis to (4), yielding $\phi[\vec{X} \mapsto \vec{t}] \models \mathsf{def}\ \Gamma; x : \sigma_1; \Gamma'\ \mathsf{in}\ C''$. The result follows by (5), (6), and CM-EXISTS.

∘ *Case* CM-INSTANCE. C' is $y \preceq \tau$. We distinguish three subcases.

Subcase $y \in \mathit{dpi}(\Gamma')$. The premise is of the form $\phi \models \mathsf{def}\ \Gamma; x : \sigma_2; \Gamma'_1\ \mathsf{in}\ C''$. The result follows by the induction hypothesis and by CM-INSTANCE.

Subcase $y \notin \mathit{dpi}(\Gamma')$ and $y = x$. The premise is $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ \sigma_2 \preceq \tau$. By (2) and CM-AND, this implies $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ (C \wedge \sigma_2 \preceq \tau)$. By (1) and Lemma 1.3.30, this implies $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ \sigma_1 \preceq \tau$. The result follows by CM-INSTANCE.

Subcase $y \notin \mathit{dpi}(\Gamma')$ and $y \neq x$. Then, both (3) and the goal are equivalent to $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C'$. □

The following exercise generalizes this result to let forms.

1.3.32 EXERCISE [★★, ↛]: Prove that $\mathtt{Z} \notin \mathit{ftv}(\sigma)$ implies $\exists\sigma \equiv \exists\mathtt{Z}.\sigma \preceq \mathtt{Z}$. Explain why, as a result, $C \Vdash \sigma_1 \preceq \sigma_2$ implies $C \wedge \exists\sigma_2 \Vdash \exists\sigma_1$. Use this fact to prove that $C \Vdash \sigma_1 \preceq \sigma_2$ implies $C \wedge \mathsf{let}\ \mathtt{x} : \sigma_2\ \mathsf{in}\ D \Vdash \mathsf{let}\ \mathtt{x} : \sigma_1\ \mathsf{in}\ D$. □

The following lemma tells how to compare type schemes that differ only in their constraints.

1.3.33 LEMMA: Let σ_1 and σ_2 stand for $\forall\bar{\mathtt{X}}[C_1].\mathtt{T}$ and $\forall\bar{\mathtt{X}}[C_2].\mathtt{T}$, respectively. Then, $\bar{\mathtt{X}} \# \mathit{ftv}(C)$ and $C \wedge C_2 \Vdash C_1$ imply $C \Vdash \sigma_1 \preceq \sigma_2$. □

Proof: Assume $\bar{\mathtt{X}} \# \mathit{ftv}(C)$ **(1)** and $C \wedge C_2 \Vdash C_1$ **(2)**. Let $\mathtt{Z} \notin \bar{\mathtt{X}}$ **(3)**. Then, we have

$$\begin{array}{lll} & C \wedge \sigma_2 \preceq \mathtt{Z} & \\ \equiv & C \wedge \exists\bar{\mathtt{X}}.(C_2 \wedge \mathtt{T} \leq \mathtt{Z}) & \textbf{(4)} \\ \equiv & \exists\bar{\mathtt{X}}.(C \wedge C_2 \wedge \mathtt{T} \leq \mathtt{Z}) & \textbf{(5)} \\ \Vdash & \exists\bar{\mathtt{X}}.(C_1 \wedge \mathtt{T} \leq \mathtt{Z}) & \textbf{(6)} \\ \equiv & \sigma_1 \preceq \mathtt{Z} & \textbf{(7)} \end{array}$$

where (4) is by (3) and Definition 1.3.3; (5) is by (1) and Lemma 1.3.20; (6) follows from (2); (7) is by (3) and Definition 1.3.3 again. By Definition 1.3.29, we have established $C \Vdash \sigma_1 \preceq \sigma_2$. □

We now establish that entailment is preserved by arbitrary constraint contexts. As a result, constraint equivalence is a congruence. In what follows, these facts are often used implicitly.

1.3.34 THEOREM [CONGRUENCE]: $C_1 \Vdash C_2$ implies $\mathcal{C}[C_1] \Vdash \mathcal{C}[C_2]$. □

Proof: By induction on the structure of $\mathcal{C}$.

- *Case* []. The goal is the hypothesis $C_1 \Vdash C_2$.
- *Case* C. The goal is $C \Vdash C$, a tautology.
- *Case* $\mathcal{C}_1 \wedge \mathcal{C}_2$. By the induction hypothesis and lemma 1.3.16.
- *Case* $\exists\bar{\mathtt{X}}.\mathcal{C}_1$. By the induction hypothesis and lemma 1.3.17.
- *Case* $\mathsf{def}\ \mathtt{x} : \sigma\ \mathsf{in}\ \mathcal{C}_1$. By the induction hypothesis and using the fact that—by definition—entailment is preserved by all def contexts.
- *Case* $\mathsf{def}\ \forall\bar{\mathtt{X}}[\mathcal{C}_1].\mathtt{T}\ \mathsf{in}\ C$. By the induction hypothesis and by Lemmas 1.3.33 and 1.3.31. □

We proceed with a number of simple properties of def constraints.

1.3.35 LEMMA: $\mathit{dpi}(\Gamma) \# \mathit{fpi}(C)$ implies $\mathsf{def}\ \Gamma\ \mathsf{in}\ C \equiv C$. □

1.3.36 LEMMA: $\mathsf{def}\ \Gamma\ \mathsf{in}\ (C_1 \wedge C_2) \equiv (\mathsf{def}\ \Gamma\ \mathsf{in}\ C_1) \wedge (\mathsf{def}\ \Gamma\ \mathsf{in}\ C_2)$. □

Proof: By CM-AND, the assertion $\phi \models \mathsf{def}\ \Gamma'; \Gamma\ \mathsf{in}\ (C_1 \wedge C_2)$ is equivalent to the conjunction of $\phi \models \mathsf{def}\ \Gamma'; \Gamma\ \mathsf{in}\ C_1$ and $\phi \models \mathsf{def}\ \Gamma'; \Gamma\ \mathsf{in}\ C_2$. By CM-AND again, so is the assertion $\phi \models \mathsf{def}\ \Gamma'\ \mathsf{in}\ ((\mathsf{def}\ \Gamma\ \mathsf{in}\ C_1) \wedge (\mathsf{def}\ \Gamma\ \mathsf{in}\ C_2))$. □

1.3.37 LEMMA: $\mathsf{let}\ \Gamma\ \mathsf{in}\ C \equiv \exists\Gamma \wedge \mathsf{def}\ \Gamma\ \mathsf{in}\ C$. □

Proof: The proof is by induction on the structure of Γ.

- *Case* $\varnothing$. The goal is $C \equiv \mathsf{true} \wedge C$, a consequence of Lemma 1.3.19.
- *Case* $\Gamma; x : \sigma$. The goal is to show that $\mathsf{let}\ \Gamma; x : \sigma\ \mathsf{in}\ C$ **(1)** and $\exists(\Gamma; x : \sigma) \wedge \mathsf{def}\ \Gamma; x : \sigma\ \mathsf{in}\ C$ **(2)** are equivalent. By definition, (1) is $\mathsf{let}\ \Gamma\ \mathsf{in}\ (\exists\sigma \wedge \mathsf{def}\ x : \sigma\ \mathsf{in}\ C)$, which, by the induction hypothesis, is equivalent to $\exists\Gamma \wedge \mathsf{def}\ \Gamma\ \mathsf{in}\ (\exists\sigma \wedge \mathsf{def}\ x : \sigma\ \mathsf{in}\ C)$ **(3)**. By Lemma 1.3.36, (3) is $\exists\Gamma \wedge \mathsf{def}\ \Gamma\ \mathsf{in}\ \exists\sigma \wedge \mathsf{def}\ \Gamma; x : \sigma\ \mathsf{in}\ C$, which by definition is (2). □

The next lemma states that, modulo equivalence, the only constraint that constrains x without explicitly referring to it is false.

1.3.38 LEMMA: $C \Vdash x \preceq \tau$ and $x \notin \mathit{fpi}(C)$ imply $C \equiv \mathsf{false}$. □

Proof: Let $C \Vdash x \preceq \tau$ **(1)** and $x \notin \mathit{fpi}(C)$ **(2)**. Let σ stand for $\forall X[\mathsf{false}].X$. By (1) and by congruence of entailment, we have $\mathsf{def}\ x : \sigma\ \mathsf{in}\ C \Vdash \mathsf{def}\ x : \sigma\ \mathsf{in}\ x \preceq \tau$ **(3)**. By (2) and Lemma 1.3.35, the left-hand side is equivalent to C. By CM-INSTANCE, the right-hand side is equivalent to $\sigma \preceq \tau$, which by definition of σ is equivalent to false. Thus, (3) may be read $C \Vdash \mathsf{false}$. □

The following definition and lemmas exploit the fact that the def form is an explicit substitution form and show how to evaluate it, that is, how to push it down through a context. In other words, they show that, modulo a few side-conditions, a context of the form $\mathsf{def}\ \Gamma\ \mathsf{in}\ []$ commutes with an arbitrary context $\mathcal{C}$.

1.3.39 DEFINITION: Given an environment Γ and a type scheme $\sigma = \forall\bar{X}[C].\tau$ such that $\bar{X} \# \mathit{ftv}(\Gamma)$, let $\Gamma \bullet \sigma$ stand for $\forall\bar{X}[\mathsf{def}\ \Gamma\ \mathsf{in}\ C].\tau$. Given an environment Γ and a context $\mathcal{C}$ such that $\mathit{dpi}(\mathcal{C}) \# \mathit{dfpi}(\Gamma)$ and $\mathit{dtv}(\mathcal{C}) \# \mathit{ftv}(\Gamma)$, define the context $\Gamma \bullet \mathcal{C}$ as follows:

$$
\begin{array}{rcl}
\Gamma \bullet [] & = & [] \\
\Gamma \bullet C & = & \mathsf{def}\ \Gamma\ \mathsf{in}\ C \\
\Gamma \bullet (\mathcal{C}_1 \wedge \mathcal{C}_2) & = & (\Gamma \bullet \mathcal{C}_1) \wedge (\Gamma \bullet \mathcal{C}_2) \\
\Gamma \bullet (\exists\bar{X}.\mathcal{C}) & = & \exists\bar{X}.(\Gamma \bullet \mathcal{C}) \\
\Gamma \bullet (\mathsf{def}\ x : \sigma\ \mathsf{in}\ \mathcal{C}) & = & \mathsf{def}\ x : (\Gamma \bullet \sigma)\ \mathsf{in}\ (\Gamma \bullet \mathcal{C}) \\
\Gamma \bullet (\mathsf{def}\ x : \forall\bar{X}[\mathcal{C}].\tau\ \mathsf{in}\ C) & = & \mathsf{def}\ x : \forall\bar{X}[\Gamma \bullet \mathcal{C}].\tau; \Gamma\ \mathsf{in}\ C
\end{array}
$$

□

1.3.40 LEMMA: $\mathsf{def}\ \Gamma\ \mathsf{in}\ \sigma \preceq \mathtt{T}' \equiv (\Gamma \bullet \sigma) \preceq \mathtt{T}'$. □

Proof: Let $\sigma = \forall \bar{\mathtt{X}}[\mathtt{C}].\mathtt{T}$, where $\bar{\mathtt{X}} \# \mathit{ftv}(\Gamma, \mathtt{T}')$. By Definitions 1.3.3 and 1.3.39, the goal is $\mathsf{def}\ \Gamma\ \mathsf{in}\ \exists \bar{\mathtt{X}}.(\mathtt{C} \wedge \mathtt{T} \leq \mathtt{T}') \equiv \exists \bar{\mathtt{X}}.((\mathsf{def}\ \Gamma\ \mathsf{in}\ \mathtt{C}) \wedge \mathtt{T} \leq \mathtt{T}')$, whose validity may be viewed as a consequence of Lemmas 1.3.21, 1.3.36, and 1.3.35. □

1.3.41 LEMMA: $\mathtt{x} \notin \mathit{dfpi}(\Gamma)$ implies $\mathsf{def}\ \Gamma; \mathtt{x} : \sigma\ \mathsf{in}\ \mathtt{C} \equiv \mathsf{def}\ \mathtt{x} : (\Gamma \bullet \sigma); \Gamma\ \mathsf{in}\ \mathtt{C}$. □

1.3.42 LEMMA: $\mathit{dpi}(\mathcal{C}) \# \mathit{dfpi}(\Gamma)$ and $\mathit{dtv}(\mathcal{C}) \# \mathit{ftv}(\Gamma)$ imply $\mathsf{def}\ \Gamma\ \mathsf{in}\ \mathcal{C}[\mathtt{C}] \equiv (\Gamma \bullet \mathcal{C})[\mathsf{def}\ \Gamma\ \mathsf{in}\ \mathtt{C}]$. □

Proof: By induction on the structure of $\mathcal{C}$.

- *Cases* $\mathcal{C} = []$, $\mathcal{C} = \mathtt{C}_1$. Immediate.
- *Case* $\mathcal{C} = \mathcal{C}_1 \wedge \mathcal{C}_2$. By the induction hypothesis and by Lemma 1.3.36.
- *Case* $\mathcal{C} = \exists \bar{\mathtt{X}}.\mathcal{C}_1$. By the induction hypothesis and by Lemma 1.3.21.
- *Cases* $\mathcal{C} = \mathsf{def}\ \mathtt{x} : \sigma\ \mathsf{in}\ \mathcal{C}_1$ and $\mathcal{C} = \mathsf{def}\ \mathtt{x} : \forall \bar{\mathtt{X}}[\mathcal{C}_1].\mathtt{T}\ \mathsf{in}\ \mathtt{C}'$. By the induction hypothesis and by Lemma 1.3.41. □

The following lemma states that the more universal quantifiers are present, the more general the type scheme.

1.3.43 LEMMA: $\mathsf{let}\ \mathtt{x} : \forall \bar{\mathtt{X}}[\mathtt{C}_1].\mathtt{T}\ \mathsf{in}\ \mathtt{C}_2 \Vdash \mathsf{let}\ \mathtt{x} : \forall \bar{\mathtt{X}}\bar{\mathtt{Y}}[\mathtt{C}_1].\mathtt{T}\ \mathsf{in}\ \mathtt{C}_2$. □

Proof: Let us first prove that $\mathsf{true} \Vdash \forall \bar{\mathtt{X}}\bar{\mathtt{Y}}[\mathtt{C}].\mathtt{T} \preceq \forall \bar{\mathtt{X}}[\mathtt{C}].\mathtt{T}$ **(1)** holds. Let $\mathtt{Z} \# \bar{\mathtt{X}}\bar{\mathtt{Y}}$ **(2)**. By (2) and Definition 1.3.3, $\forall \bar{\mathtt{X}}[\mathtt{C}].\mathtt{T} \preceq \mathtt{Z}$ is $\exists \bar{\mathtt{X}}.(\mathtt{C} \wedge \mathtt{T} \leq \mathtt{Z})$ **(3)**. By lemmas 1.3.22 and 1.3.23, (3) entails $\exists \bar{\mathtt{X}}\bar{\mathtt{Y}}.(\mathtt{C} \wedge \mathtt{T} \leq \mathtt{Z})$, which by (2) and Definition 1.3.3 is $\forall \bar{\mathtt{X}}\bar{\mathtt{Y}}[\mathtt{C}].\mathtt{T} \preceq \mathtt{Z}$. By Definition 1.3.29, this proves (1). Now, Lemmas 1.3.22 and 1.3.23 yield $\exists \bar{\mathtt{X}}.\mathtt{C} \Vdash \exists \bar{\mathtt{X}}\bar{\mathtt{Y}}.\mathtt{C}$ **(4)**. Furthermore, by (1) and Lemma 1.3.31, we obtain $\mathsf{def}\ \mathtt{x} : \forall \bar{\mathtt{X}}[\mathtt{C}_1].\mathtt{T}\ \mathsf{in}\ \mathtt{C}_2 \Vdash \mathsf{def}\ \mathtt{x} : \forall \bar{\mathtt{X}}\bar{\mathtt{Y}}[\mathtt{C}_1].\mathtt{T}\ \mathsf{in}\ \mathtt{C}_2$ **(5)**. The result follows from (4) and (5) by definition of the let form. □

Conversely, and perhaps surprisingly, it is sometimes possible to *remove* some type variables from the universal quantifier prefix of a type scheme without compromising its generality. This is the case when the value of these type variables is determined in a unique way. In short, C *determines* $\bar{\mathtt{Y}}$ if and only if, given a ground assignment for $\mathit{ftv}(\mathtt{C}) \setminus \bar{\mathtt{Y}}$ and given that C holds, it is possible to reconstruct, in a unique way, a ground assignement for $\bar{\mathtt{Y}}$. Determinacy appears in the equivalence law C-LETALL on page 45 and is exploited by the constraint solver in §1.8.

1.3.44 DEFINITION: C *determines* $\bar{\mathtt{Y}}$ if and only if, for every environment Γ, two ground assignments that satisfy $\mathsf{def}\ \Gamma\ \mathsf{in}\ \mathtt{C}$ and that coincide outside $\bar{\mathtt{Y}}$ must coincide on $\bar{\mathtt{Y}}$ as well. □

The following lemma gives an abstract consequence of determinacy.

1.3.45 LEMMA: If C determines $\bar{Y}$, then $D \Vdash C$ implies $C \wedge \exists \bar{Y}.D \Vdash D$. □

Proof: Assume C determines $\bar{Y}$ **(1)** and $D \Vdash C$ **(2)**. Let $\bar{Y} \# \mathit{ftv}(\Gamma)$ **(3)** and $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ (C \wedge \exists \bar{Y}.D)$ **(4)**. By CM-AND, (3), and CM-EXISTS, (4) implies $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C$ **(5)** and $\phi[\vec{Y} \mapsto \vec{t}] \models \mathsf{def}\ \Gamma\ \mathsf{in}\ D$ **(6)**, for some ground types $\vec{t}$. By definition of entailment, (6) and (2) imply $\phi[\vec{Y} \mapsto \vec{t}] \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C$ **(7)**. By definition of determinacy, (1), (5), and (7) imply $\phi = \phi[\vec{Y} \mapsto \vec{t}]$. Thus, (6) may be read $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ D$ **(8)**. We have shown that, assuming (3), (4) implies (8); by Lemma 1.3.15, this proves that $C \wedge \exists \bar{Y}.D$ entails D. □

Exploiting the notion of determinacy, the next lemma states when it is safe to take type variables out of a universal quantifier prefix.

1.3.46 LEMMA: $\exists \bar{X}.C_1$ determines $\bar{Y}$ and $\bar{Y} \# \mathit{ftv}(C_2)$ imply $\mathsf{let}\ x : \forall \bar{X}\bar{Y}[C_1].\tau\ \mathsf{in}\ C_2 \Vdash \exists \bar{Y}.\mathsf{let}\ x : \forall \bar{X}[C_1].\tau\ \mathsf{in}\ C_2$. □

Proof: Assume $\exists \bar{X}.C_1$ determines $\bar{Y}$ **(1)** and $\bar{Y} \# \mathit{ftv}(C_2)$ **(2)**. Let $z \notin \bar{X}\bar{Y}$ **(3)**. By (3) and Definition 1.3.3, the constraint $\exists \bar{X}.C_1 \wedge \forall \bar{X}\bar{Y}[C_1].\tau \preceq z$ **(4)** is $\exists \bar{X}.C_1 \wedge \exists \bar{X}\bar{Y}.(C_1 \wedge \tau \leq z)$ **(5)**. By lemma 1.3.23, (5) may be written $\exists \bar{X}.C_1 \wedge \exists \bar{Y}.\exists \bar{X}.(C_1 \wedge \tau \leq z)$ **(6)**. Now, because $\exists \bar{X}.(C_1 \wedge \tau \leq z)$ entails $\exists \bar{X}.C_1$, (1) and Lemma 1.3.45 imply that (6) entails $\exists \bar{X}.(C_1 \wedge \tau \leq z)$, which by (3) and Definition 1.3.3 means $\forall \bar{X}[C_1].\tau \preceq z$ **(7)**. Thus, (4) entails (7): by Definition 1.3.29, this establishes $\exists \bar{X}.C_1 \Vdash \forall \bar{X}[C_1].\tau \preceq \forall \bar{X}\bar{Y}[C_1].\tau$ **(8)**. By Lemma 1.3.31, (8) implies $\exists \bar{X}.C_1 \wedge \mathsf{def}\ x : \forall \bar{X}\bar{Y}[C_1].\tau\ \mathsf{in}\ C_2 \Vdash \mathsf{def}\ x : \forall \bar{X}[C_1].\tau\ \mathsf{in}\ C_2$ **(9)**. Let us now place (9) within the context $\exists \bar{Y}.(\exists \bar{X}.C_1 \wedge [])$. On the left-hand side, where the conjunct $\exists \bar{X}.C_1$ is redundant, we obtain $\exists \bar{Y}.(\exists \bar{X}.C_1 \wedge \mathsf{def}\ x : \forall \bar{X}\bar{Y}[C_1].\tau\ \mathsf{in}\ C_2)$, which by (2), Lemma 1.3.20, and Lemma 1.3.23, is equivalent to $\exists \bar{X}\bar{Y}.C_1 \wedge \mathsf{def}\ x : \forall \bar{X}\bar{Y}[C_1].\tau\ \mathsf{in}\ C_2$, that is, $\mathsf{let}\ x : \forall \bar{X}\bar{Y}[C_1].\tau\ \mathsf{in}\ C_2$ **(10)**. On the right-hand side, we obtain $\exists \bar{Y}.\mathsf{let}\ x : \forall \bar{X}[C_1].\tau\ \mathsf{in}\ C_2$ **(11)**. Thus, (10) entails (11). □

We now give a toolbox of constraint equivalence laws. It is worth noting that they do not form a complete axiomatization of constraint equivalence—in fact, they cannot, since the syntax and meaning of constraints is partly unspecified.

1.3.47 THEOREM: All equivalence laws in Figure 1-6 hold. □

Proof: By examination of every law.

- *Cases* C-AND, C-ANDAND. By Lemma 1.3.18.
- *Case* C-DUP. By Lemma 1.3.19.
- *Case* C-EXEX. By Lemma 1.3.23.

$$C_1 \wedge C_2 \equiv C_2 \wedge C_1 \qquad \text{(C-And)}$$

$$(C_1 \wedge C_2) \wedge C_3 \equiv C_1 \wedge (C_2 \wedge C_3) \qquad \text{(C-AndAnd)}$$

$$C_1 \wedge C_2 \equiv C_1 \quad \text{if } C_1 \Vdash C_2 \qquad \text{(C-Dup)}$$

$$\exists \bar{X}.\exists \bar{Y}.C \equiv \exists \bar{X}\bar{Y}.C \qquad \text{(C-ExEx)}$$

$$\exists \bar{X}.C \equiv C \quad \text{if } \bar{X} \# \mathit{ftv}(C) \qquad \text{(C-Ex*)}$$

$$(\exists \bar{X}.C_1) \wedge C_2 \equiv \exists \bar{X}.(C_1 \wedge C_2) \quad \text{if } \bar{X} \# \mathit{ftv}(C_2) \qquad \text{(C-ExAnd)}$$

$$\exists Z.(\sigma \preceq Z \wedge Z \leq T) \equiv \sigma \preceq T \quad \text{if } Z \notin \mathit{ftv}(\sigma, T) \qquad \text{(C-ExTrans)}$$

$$\text{let } x : \sigma \text{ in } \mathcal{C}[x \preceq T] \equiv \text{let } x : \sigma \text{ in } \mathcal{C}[\sigma \preceq T] \qquad \text{(C-InId)}$$
$$\text{if } x \notin \mathit{dpi}(\mathcal{C}) \text{ and } \mathit{dtv}(\mathcal{C}) \# \mathit{ftv}(\sigma) \text{ and } \{x\} \cup \mathit{dpi}(\mathcal{C}) \# \mathit{fpi}(\sigma)$$

$$\text{let } \Gamma \text{ in } C \equiv \exists \Gamma \wedge C \quad \text{if } \mathit{dpi}(\Gamma) \# \mathit{fpi}(C) \qquad \text{(C-In*)}$$

$$\text{let } \Gamma \text{ in } (C_1 \wedge C_2) \equiv (\text{let } \Gamma \text{ in } C_1) \wedge (\text{let } \Gamma \text{ in } C_2) \qquad \text{(C-InAnd)}$$

$$\text{let } \Gamma \text{ in } (C_1 \wedge C_2) \equiv (\text{let } \Gamma \text{ in } C_1) \wedge C_2 \quad \text{if } \mathit{dpi}(\Gamma) \# \mathit{fpi}(C_2) \qquad \text{(C-InAnd*)}$$

$$\text{let } \Gamma \text{ in } \exists \bar{X}.C \equiv \exists \bar{X}.\text{let } \Gamma \text{ in } C \quad \text{if } \bar{X} \# \mathit{ftv}(\Gamma) \qquad \text{(C-InEx)}$$

$$\text{let } \Gamma_1; \Gamma_2 \text{ in } C \equiv \text{let } \Gamma_2; \Gamma_1 \text{ in } C \qquad \text{(C-LetLet)}$$
$$\text{if } \mathit{dpi}(\Gamma_1) \# \mathit{dpi}(\Gamma_2) \text{ and } \mathit{dpi}(\Gamma_2) \# \mathit{fpi}(\Gamma_1) \text{ and } \mathit{dpi}(\Gamma_1) \# \mathit{fpi}(\Gamma_2)$$

$$\text{let } x : \forall \bar{X}[C_1 \wedge C_2].T \text{ in } C_3 \equiv C_1 \wedge \text{let } x : \forall \bar{X}[C_2].T \text{ in } C_3 \quad \text{if } \bar{X} \# \mathit{ftv}(C_1) \qquad \text{(C-LetAnd)}$$

$$\text{let } \Gamma; x : \forall \bar{X}[C_1].T \text{ in } C_2 \equiv \text{let } \Gamma; x : \forall \bar{X}[\text{let } \Gamma \text{ in } C_1].T \text{ in } C_2 \qquad \text{(C-LetDup)}$$
$$\text{if } \bar{X} \# \mathit{ftv}(\Gamma) \text{ and } \mathit{dpi}(\Gamma) \# \mathit{fpi}(\Gamma)$$

$$\text{let } x : \forall \bar{X}[\exists \bar{Y}.C_1].T \text{ in } C_2 \equiv \text{let } x : \forall \bar{X}\bar{Y}[C_1].T \text{ in } C_2 \quad \text{if } \bar{Y} \# \mathit{ftv}(T) \qquad \text{(C-LetEx)}$$

$$\text{let } x : \forall \bar{X}\bar{Y}[C_1].T \text{ in } C_2 \equiv \exists \bar{Y}.\text{let } x : \forall \bar{X}[C_1].T \text{ in } C_2 \qquad \text{(C-LetAll)}$$
$$\text{if } \bar{Y} \# \mathit{ftv}(C_2) \text{ and } \exists \bar{X}.C_1 \text{ determines } \bar{Y}$$

$$\exists X.(T \leq X \wedge \text{let } x : X \text{ in } C) \equiv \text{let } x : T \text{ in } C \quad \text{if } X \notin \mathit{ftv}(T, C) \qquad \text{(C-LetSub)}$$

$$\vec{X} = \vec{T} \wedge [\vec{X} \mapsto \vec{T}]C \equiv \vec{X} = \vec{T} \wedge C \qquad \text{(C-Eq)}$$

$$\text{true} \equiv \exists \bar{X}.(\vec{X} = \vec{T}) \quad \text{if } \bar{X} \# \mathit{ftv}(\vec{T}) \qquad \text{(C-Name)}$$

$$[\vec{X} \mapsto \vec{T}]C \equiv \exists \bar{X}.(\vec{X} = \vec{T} \wedge C) \quad \text{if } \bar{X} \# \mathit{ftv}(\vec{T}) \qquad \text{(C-NameEq)}$$

Figure 1-6: Constraint equivalence laws

∘ *Case* C-Ex*. Let $\bar{X} \# \mathit{ftv}(C)$ **(1)** and $\bar{X} \# \mathit{ftv}(\Gamma)$ **(2)**. By CM-Exists, $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ \exists\bar{X}.C$ **(3)** is equivalent to $\exists\vec{t}\quad \phi[\vec{X} \mapsto \vec{t}] \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C$, which, by (1) and Lemma 1.3.12, is equivalent to $\exists\vec{t}\quad \phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C$ **(4)**. Because every $\mathcal{M}_\kappa$ is nonempty (Definition 1.3.5), (4) may be written $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C$ **(5)**. By Lemma 1.3.15, the equivalence of (3) and (5) under (2) establishes $\exists\bar{X}.C \equiv C$.

∘ *Case* C-ExAnd. By Lemma 1.3.20.

∘ *Case* C-ExTrans. By Lemma 1.3.24.

∘ *Case* C-InId. Assume $\mathit{dtv}(\mathcal{C}) \# \mathit{ftv}(\sigma)$ **(1)** and $\mathit{dpi}(\mathcal{C}) \# \{x\} \cup \mathit{fpi}(\sigma)$ **(2)** and $x \notin \mathit{fpi}(\sigma)$ **(3)**. By (1), (2), and Lemma 1.3.42, $\mathsf{def}\ x : \sigma\ \mathsf{in}\ \mathcal{C}[x \preceq \tau']$ **(4)** is equivalent to $((x : \sigma) \bullet \mathcal{C})[\mathsf{def}\ x : \sigma\ \mathsf{in}\ x \preceq \tau']$ **(5)**. Now, by CM-Instance, $\mathsf{def}\ x : \sigma\ \mathsf{in}\ x \preceq \tau'$ is equivalent to $\sigma \preceq \tau'$, which, by (3) and Lemma 1.3.35, is equivalent to $\mathsf{def}\ x : \sigma\ \mathsf{in}\ \sigma \preceq \tau'$. Thus, (5) may be written $((x : \sigma) \bullet \mathcal{C})[\mathsf{def}\ x : \sigma\ \mathsf{in}\ \sigma \preceq \tau']$, which, by Lemma 1.3.42 again, is $\mathsf{def}\ x : \sigma\ \mathsf{in}\ \mathcal{C}[\sigma \preceq \tau']$ **(6)**. Placing the assertion (4) ≡ (6) within the context $\exists\sigma \wedge []$ yields the result.

∘ *Case* C-In*. By Lemmas 1.3.37 and 1.3.35.

∘ *Case* C-InAnd. By Lemma 1.3.36, we have $\mathsf{def}\ x : \sigma\ \mathsf{in}\ (C_1 \wedge C_2) \equiv (\mathsf{def}\ x : \sigma\ \mathsf{in}\ C_1) \wedge (\mathsf{def}\ x : \sigma\ \mathsf{in}\ C_2)$. Placing this assertion within the context $\exists\sigma \wedge []$ and applying C-Dup to the right-hand side yields $\mathsf{let}\ x : \sigma\ \mathsf{in}\ (C_1 \wedge C_2) \equiv (\mathsf{let}\ x : \sigma\ \mathsf{in}\ C_1) \wedge (\mathsf{let}\ x : \sigma\ \mathsf{in}\ C_2)$. The result follows by structural induction on Γ.

∘ *Case* C-InAnd*. By C-InAnd, C-In* and C-Dup.

∘ *Case* C-InEx. Let $\bar{X} \# \mathit{ftv}(\sigma)$ **(1)**. The constraint $\mathsf{let}\ x : \sigma\ \mathsf{in}\ \exists\bar{X}.C$ is short for $\exists\sigma \wedge \mathsf{def}\ x : \sigma\ \mathsf{in}\ \exists\bar{X}.C$. By (1), Lemma 1.3.21, and C-ExAnd, this is $\exists\bar{X}.(\exists\sigma \wedge \mathsf{def}\ x : \sigma\ \mathsf{in}\ C)$, that is, $\exists\bar{X}.\mathsf{let}\ x : \sigma\ \mathsf{in}\ C$. The result follows by induction on the structure of Γ.

∘ *Case* C-LetLet. Let $x_1 \neq x_2$ **(1)** and $x_1 \notin \mathit{fpi}(\sigma_2)$ **(2)** and $x_2 \notin \mathit{fpi}(\sigma_1)$ **(3)**. By (1), (3), and Lemma 1.3.42, $\mathsf{def}\ x_1 : \sigma_1; x_2 : \sigma_2\ \mathsf{in}\ C$ **(4)** is equivalent to $\mathsf{def}\ x_2 : (x_1 : \sigma_1) \bullet \sigma_2; x_1 : \sigma_1\ \mathsf{in}\ C$, which by (2) and Lemma 1.3.35 is $\mathsf{def}\ x_2 : \sigma_2; x_1 : \sigma_1\ \mathsf{in}\ C$ **(5)**. Furthermore, (2), (3), and Lemma 1.3.35 yield $\mathsf{def}\ x_1 : \sigma_1\ \mathsf{in}\ \exists\sigma_2 \equiv \exists\sigma_2$ **(6)** and $\mathsf{def}\ x_2 : \sigma_2\ \mathsf{in}\ \exists\sigma_1 \equiv \exists\sigma_1$ **(7)**. Placing the assertion (4) ≡ (5) within the context $\exists\sigma_1 \wedge \exists\sigma_2 \wedge []$ and using (6), (7), and Lemma 1.3.37, we obtain $\mathsf{let}\ x_1 : \sigma_1; x_2 : \sigma_2\ \mathsf{in}\ C \equiv \mathsf{let}\ x_2 : \sigma_2; x_1 : \sigma_1\ \mathsf{in}\ C$. The result follows by induction on the structure of Γ_1 and Γ_2.

∘ *Case* C-LetAnd. Let $\bar{X} \# \mathit{ftv}(C_1)$ **(1)**. By (1) and Lemma 1.3.33, we have $C_1 \Vdash \forall\bar{X}[C_1 \wedge C_2].\tau \equiv \forall\bar{X}[C_2].\tau$. By Lemma 1.3.31, this implies $C_1 \wedge \mathsf{def}\ x : \forall\bar{X}[C_1 \wedge C_2].\tau\ \mathsf{in}\ C_3 \equiv C_1 \wedge \mathsf{def}\ x : \forall\bar{X}[C_2].\tau\ \mathsf{in}\ C_3$. Placing this assertion within the context $\exists\bar{X}.C_2 \wedge []$ and applying C-ExAnd to the left-hand side yields the result.

∘ *Case* C-LetDup. Let $\bar{X} \# \mathit{ftv}(\Gamma)$ **(1)** and $\mathit{dpi}(\Gamma) \# \mathit{fpi}(\Gamma)$ **(2)**. By (1) and

Lemma 1.3.42, $\mathsf{def}\ \Gamma\ \mathsf{in}\ \mathsf{let}\ x : \forall \bar{X}[C_1].T\ \mathsf{in}\ C_2$ **(3)** is equivalent to $\mathsf{let}\ x : \forall \bar{X}[\mathsf{def}\ \Gamma\ \mathsf{in}\ C_1].T\ \mathsf{in}\ \mathsf{def}\ \Gamma\ \mathsf{in}\ C_2$ **(4)**. Now, by (2) and Lemma 1.3.35, we have $\mathsf{def}\ \Gamma\ \mathsf{in}\ C_1 \equiv \mathsf{def}\ \Gamma;\Gamma\ \mathsf{in}\ C_1$ **(5)**. Using (5), then applying Lemma 1.3.42 in the reverse direction, we find that (4) is equivalent to $\mathsf{def}\ \Gamma\ \mathsf{in}\ \mathsf{let}\ x : \forall \bar{X}[\mathsf{def}\ \Gamma\ \mathsf{in}\ C_1].T\ \mathsf{in}\ C_2$ **(6)**. Placing the assertion (3) $\equiv$ (6) within the context $\exists\Gamma \wedge []$ and applying Lemma 1.3.37, we find that $\mathsf{let}\ \Gamma\ \mathsf{in}\ \mathsf{let}\ x : \forall \bar{X}[C_1].T\ \mathsf{in}\ C_2$ is equivalent to $\mathsf{let}\ \Gamma\ \mathsf{in}\ \mathsf{let}\ x : \forall \bar{X}[\mathsf{def}\ \Gamma\ \mathsf{in}\ C_1].T\ \mathsf{in}\ C_2$ **(7)**. Last, by C-DUP, (7) is equivalent to $\exists\Gamma \wedge$ (7), which by (2) and C-INAND*, then (1) and C-LETAND, is $\mathsf{let}\ \Gamma\ \mathsf{in}\ \mathsf{let}\ x : \forall \bar{X}[\exists\Gamma \wedge \mathsf{def}\ \Gamma\ \mathsf{in}\ C_1].T\ \mathsf{in}\ C_2$, that is, $\mathsf{let}\ \Gamma\ \mathsf{in}\ \mathsf{let}\ x : \forall \bar{X}[\mathsf{let}\ \Gamma\ \mathsf{in}\ C_1].T\ \mathsf{in}\ C_2$.

◦ *Case* C-LETEX. Let $\bar{Y} \# \mathit{ftv}(T)$ **(1)**. Let $Z \notin \bar{X}\bar{Y}$ **(2)**. By (2) and Definition 1.3.3, $\forall \bar{X}[\exists \bar{Y}.C].T \preceq Z$ is $\exists \bar{X}.(\exists \bar{Y}.C \wedge T \leq Z)$. Similarly, $\forall \bar{X}\bar{Y}[C].T \preceq Z$ is $\exists \bar{X}\bar{Y}.(C \wedge T \leq Z)$. By (1), (2), Lemma 1.3.20, and Lemma 1.3.23, these constraints are equivalent. By Definition 1.3.29, this establishes $\mathsf{true} \Vdash \forall \bar{X}[\exists \bar{Y}.C].T \equiv \forall \bar{X}\bar{Y}[C].T$. The result follows by Lemmas 1.3.23 and 1.3.31.

◦ *Case* C-LETALL. Assume $\bar{Y} \# \mathit{ftv}(C_2)$ **(3)** and $\exists \bar{X}.C_1$ determines $\bar{Y}$ **(4)**. By Lemma 1.3.43, $\mathsf{let}\ x : \forall \bar{X}[C_1].T\ \mathsf{in}\ C_2$ entails $\mathsf{let}\ x : \forall \bar{X}\bar{Y}[C_1].T\ \mathsf{in}\ C_2$. Placing this assertion within the context $\exists \bar{Y}.[]$, we find that $\exists \bar{Y}.\mathsf{let}\ x : \forall \bar{X}[C_1].T\ \mathsf{in}\ C_2$ **(5)** entails $\exists \bar{Y}.\mathsf{let}\ x : \forall \bar{X}\bar{Y}[C_1].T\ \mathsf{in}\ C_2$, which, by (3) and C-EX*, is equivalent to $\mathsf{let}\ x : \forall \bar{X}\bar{Y}[C_1].T\ \mathsf{in}\ C_2$ **(6)**. Conversely, by (3), (4), and Lemma 1.3.46, (6) entails (5).

◦ *Case* C-LETSUB. Let $X \notin \mathit{ftv}(T, C)$ **(1)**. As an immediate consequence of Lemma 1.3.31, we find that $T \leq X \wedge \mathsf{def}\ x : X\ \mathsf{in}\ C$ entails $\mathsf{def}\ x : T\ \mathsf{in}\ C$. By congruence of entailment, $\exists X.(T \leq X \wedge \mathsf{def}\ x : X\ \mathsf{in}\ C)$ entails $\exists X.\mathsf{def}\ x : T\ \mathsf{in}\ C$, which, by (1) and C-EX*, is $\mathsf{def}\ x : T\ \mathsf{in}\ C$. Conversely, by (1), Lemma 1.3.26 and C-EXAND, $\mathsf{def}\ x : T\ \mathsf{in}\ C$ is equivalent to $\exists X.(X = T \wedge \mathsf{def}\ x : T\ \mathsf{in}\ C)$, that is, $\exists X.(T \leq X \wedge X \leq T \wedge \mathsf{def}\ x : T\ \mathsf{in}\ C)$, which, by Lemma 1.3.31, entails $\exists X.(T \leq X \wedge \mathsf{def}\ x : X\ \mathsf{in}\ C)$.

◦ *Case* C-EQ. Let $\bar{X} \# \mathit{ftv}(\Gamma)$ **(1)**. By CM-PREDICATE, the assertion $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ \vec{X} = \vec{T}$ **(2)** is equivalent to $\phi(\vec{X}) = \phi(\vec{T})$, which in turn is equivalent to $\phi \circ [\vec{X} \mapsto \vec{T}] = \phi$ **(3)**. Assuming (3) and applying Lemma 1.3.13, we find that the assertion $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C$ **(4)** may be written $\phi \models \mathsf{def}\ [\vec{X} \mapsto \vec{T}]\Gamma\ \mathsf{in}\ [\vec{X} \mapsto \vec{T}]C$, which by (1) is $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ [\vec{X} \mapsto \vec{T}]C$ **(5)**. Thus, (2) implies that (4) and (5) are equivalent. By CM-AND, this proves that $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ (\vec{X} = \vec{T} \wedge C)$ and $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ (\vec{X} = \vec{T} \wedge [\vec{X} \mapsto \vec{T}]C)$ are equivalent. Lemma 1.3.15 allows discharging (1) and yields the goal.

◦ *Case* C-NAME. By Lemma 1.3.26 and by induction on the length of the vectors $\vec{X}$ and $\vec{T}$.

◦ *Case* C-NAMEEQ. By C-NAME, C-DUP, C-EXAND, and C-EQ. □

Let us explain. C-AND and C-ANDAND state that conjunction is commutative and associative. C-DUP states that redundant conjuncts may be freely added or removed, where a conjunct is redundant if and only if it is entailed by another conjunct. Throughout this chapter, these three laws are often used implicitly. C-EXEX and C-EX* allow grouping consecutive existential quantifiers and suppressing redundant ones, where a quantifier is redundant if and only if it does not occur free within its scope. C-EXAND allows conjunction and existential quantification to commute, provided no capture occurs; it is known as a *scope extrusion* law. When the rule is oriented from left to right, its side-condition may always be satisfied by suitable α-conversion. C-EXTRANS states that it is equivalent for a type $\mathtt{T}$ to be an instance of σ or to be a supertype of some instance of σ. We remark that the instances of a monotype are its supertypes, that is, by Definition 1.3.3, $\mathtt{T}' \preceq \mathtt{T}$ and $\mathtt{T}' \leq \mathtt{T}$ are equivalent. As a result, specializing C-EXTRANS to the case where σ is a monotype, we find that $\mathtt{T}' \leq \mathtt{T}$ is equivalent to $\exists \mathtt{Z}.(\mathtt{T}' \leq \mathtt{Z} \wedge \mathtt{Z} \leq \mathtt{T})$, for fresh $\mathtt{Z}$, a standard equivalence law. When oriented from left to right, it becomes an interesting *simplification* law: in a chain of subtyping constraints, an intermediate variable such as $\mathtt{Z}$ may be suppressed, provided it is *local*, as witnessed by the existential quantifier $\exists \mathtt{Z}$. C-INID states that, within the scope of the binding $\mathtt{x} : \sigma$, every *free* occurrence of $\mathtt{x}$ may be safely replaced with σ. The restriction to free occurrences stems from the side-condition $\mathtt{x} \notin \mathit{dpi}(\mathcal{C})$. When the rule is oriented from left to right, its other side-conditions, which require the context $\mathsf{let}\ \mathtt{x} : \sigma\ \mathsf{in}\ \mathcal{C}$ not to capture σ's free type variables or free program identifiers, may always be satisfied by suitable α-conversion. C-IN* complements the previous rule by allowing redundant let bindings to be simplified. We remark that C-INID and C-IN* provide a simple procedure for eliminating let forms. C-INAND states that the let form commutes with conjunction; C-INAND* spells out a common particular case. C-INEX states that it commutes with existential quantification. When the rule is oriented from left to right, its side-condition may always be satisfied by suitable α-conversion. C-LETLET states that let forms may commute, provided they bind distinct program identifiers and provided no free program identifiers are captured in the process. C-LETAND allows the conjunct C_1 to be moved outside of the constrained type scheme $\forall \bar{\mathtt{X}}[\mathsf{C}_1 \wedge \mathsf{C}_2].\mathtt{T}$, provided it does not involve any of the universally quantified type variables $\bar{\mathtt{X}}$. When oriented from left to right, the rule yields an important simplification law: indeed, taking an instance of $\forall \bar{\mathtt{X}}[\mathsf{C}_2].\mathtt{T}$ is less expensive than taking an instance of $\forall \bar{\mathtt{X}}[\mathsf{C}_1 \wedge \mathsf{C}_2].\mathtt{T}$, since the latter involves creating a copy of C_1, while the former does not. C-LETDUP allows pushing a series of let bindings into a constrained type scheme, provided no capture occurs in the process. It is not used as a simplification law but as a tool in some proofs. C-LETEX states

that it does not make any difference for a set of type variables $\bar{Y}$ to be existentially quantified inside a constrained type scheme or part of the type scheme's universal quantifiers. Indeed, in either case, taking an instance of the type scheme means producing a constraint where $\bar{Y}$ is existentially quantified. C-LETALL provides a restricted converse of Lemma 1.3.43. Together, C-LETEX and C-LETALL allow—in some situations only—to hoist existential quantifiers out of the *left*-hand side of a let form.

1.3.48 EXAMPLE: C-LETALL would be invalid without the condition that $\exists\bar{X}.C_1$ determines $\bar{Y}$. Consider, for instance, the constraint $\mathsf{let}\ x : \forall Y.Y \to Y\ \mathsf{in}\ (x \preceq \mathsf{int} \to \mathsf{int} \wedge x \preceq \mathsf{bool} \to \mathsf{bool})$ **(1)**, where int and bool are incompatible nullary type constructors. By C-INID and C-IN*, it is equivalent to $\forall Y.Y \to Y \leq \mathsf{int} \to \mathsf{int} \wedge \forall Y.Y \to Y \leq \mathsf{bool} \to \mathsf{bool}$ which, by Definition 1.3.3, mean $\exists Y.(Y \to Y \leq \mathsf{int} \to \mathsf{int}) \wedge \exists Y.(Y \to Y \leq \mathsf{bool} \to \mathsf{bool})$, that is, true. Now, if C-LETALL was valid without its side-condition, then (1) would also be equivalent to $\exists Y.\mathsf{let}\ x : Y \to Y\ \mathsf{in}\ (x \preceq \mathsf{int} \to \mathsf{int} \wedge x \preceq \mathsf{bool} \to \mathsf{bool})$, which by C-INID and C-IN* is $\exists Y.(Y \to Y \leq \mathsf{int} \to \mathsf{int} \wedge Y \to Y \leq \mathsf{bool} \to \mathsf{bool})$. By C-ARROW and C-EXTRANS, this is $\mathsf{int} = \mathsf{bool}$, that is, false. Thus, the law is invalid in this case. It is easy to see why: when the type scheme σ contains a $\forall Y$ quantifier, every instance of σ receives its own $\exists Y$ quantifier, making Y a distinct (local) type variable; when Y is not universally quantified, however, all instances of σ *share* references to a single (global) type variable Y. This corresponds to the intuition that, in the former case, σ is *polymorphic* in Y, while in the latter case, it is *monomorphic* in Y. Lemma 1.3.43 states that, when deprived of its side-condition, C-LETALL is only an entailment law, that is, its right-hand side entails its left-hand side. Similarly, it is in general invalid to hoist an existential quantifier out of the left-hand side of a let form. To see this, one may study the (equivalent) constraint $\mathsf{let}\ x : \forall X[\exists Y.X = Y \to Y].X\ \mathsf{in}\ (x \preceq \mathsf{int} \to \mathsf{int} \wedge x \preceq \mathsf{bool} \to \mathsf{bool})$.

Naturally, in the above examples, the side-condition "true determines Y" does *not* hold: by Definition 1.3.44, it is equivalent to "two ground assignments that coincide outside Y must coincide on Y as well", which is false as soon as $\mathcal{M}_\star$ contains two distinct elements, such as int and bool here. There are cases, however, where the side-condition does hold. For instance, we later prove that $\exists X.Y = \mathsf{int}$ determines Y; see Lemma 1.8.7. As a result, C-LETALL states that $\mathsf{let}\ x : \forall XY[Y = \mathsf{int}].Y \to X\ \mathsf{in}\ C$ **(1)** is equivalent to $\exists Y.\mathsf{let}\ x : \forall X[Y = \mathsf{int}].Y \to X\ \mathsf{in}\ C$ **(2)**, provided $Y \notin \mathit{ftv}(C)$. The intuition is simple: because Y is forced to assume the value int by the equation $Y = \mathsf{int}$, it makes no difference whether Y is or isn't universally quantified. We remark that, by C-LETAND, (2) is equivalent to $\exists Y.(Y = \mathsf{int} \wedge \mathsf{let}\ x : \forall X.Y \to X\ \mathsf{in}\ C)$ **(3)**. In an efficient constraint solver, simplifying (1) into (3) *before* using C-INID to eliminate the

let form is worthwhile, since doing so obviates the need for copying the type variable Y and the equation $Y = \mathsf{int}$ at every free occurrence of x inside C. □

C-LETSUB is the analogue of an environment strengthening lemma: roughly speaking, it states that, if a constraint holds under the assumption that x has type X, where X is some supertype of T, then it also holds under the assumption that x has type T. The last three rules deal with the equality predicate. C-EQ states that it is valid to replace equals with equals; note the absence of a side-condition. When oriented from left to right, C-NAME allows introducing fresh names $\vec{X}$ for the types $\vec{T}$. As always, $\vec{X}$ stands for a vector of *distinct* type variables; $\vec{T}$ stands for a vector of the same length of types of appropriate kind. Of course, this makes sense only if the definition is not circular, that is, if the type variables $\bar{X}$ do not occur free within the terms $\bar{T}$. When oriented from right to left, C-NAME may be viewed as a simplification law: it allows eliminating type variables whose value has been determined. C-NAMEEQ is a combination of C-EQ and C-NAME. It shows that applying an idempotent substitution to a constraint C amounts to placing C within a certain context. This immediately yields a proof of the following fact:

1.3.49 LEMMA: $C \Vdash D$ implies $\theta(C) \Vdash \theta(D)$. □

Proof: Because every substitution is the composition of a renaming and of an idempotent substitution, and because entailment assertions are stable under renamings, we may assume, *w.l.o.g.*, that θ is idempotent. Thus, θ may be written $[\vec{X} \mapsto \vec{T}]$, where $\bar{X} \# \mathit{ftv}(\bar{T})$ **(1)**. By congruence of entailment, $C \Vdash D$ implies $\exists \bar{X}.(\vec{X} = \vec{T} \wedge C) \Vdash \exists \bar{X}.(\vec{X} = \vec{T} \wedge D)$ **(2)**. By (1) and C-NAMEEQ, (2) may be read $[\vec{X} \mapsto \vec{T}]C \Vdash [\vec{X} \mapsto \vec{T}]D$. □

It is important to stress that, because the effect of a type substitution may be emulated using equations, conjunction, and existential quantification, there is no need ever to employ type substitutions in the definition of a constraint-based type system—it is possible, instead, to express every concept in terms of constraints. In this chapter, we follow this route, and use type substitutions only when dealing with the type system DM, whose historical formulation is substitution-based.

So far, we have considered def a primitive constraint form and defined the let form in terms of def, conjunction, and existential quantification. The motivation for this approach was to simplify the proofs of several constraint equivalence laws. However, in the remainder of this chapter, *we work with* let *forms exclusively and never employ the* def *construct*. This offers us a few extra properties, stated in the next two lemmas. First, every constraint that contains a false subconstraint must be false. Second, no satisfiable constraint has a free program identifier.

1.3.50 LEMMA: $\mathcal{C}[\mathsf{false}] \equiv \mathsf{false}$. □

Proof: It is straightforward to check that the constraints $\mathsf{false} \wedge C$, $\exists \bar{X}.\mathsf{false}$, and $\mathsf{def}\ x : \sigma\ \mathsf{in}\ \mathsf{false}$ are false. The constraint $\mathsf{let}\ x : \forall \bar{X}[\mathsf{false}].T\ \mathsf{in}\ C$ entails $\exists \bar{X}.\mathsf{false}$, so it is false as well. The result follows by induction on the structure of $\mathcal{C}$. Note that the law is invalid in the presence of def forms: for instance, $\mathsf{def}\ x : \forall X[\mathsf{false}].X\ \mathsf{in}\ \mathsf{true}$ is equivalent to true. □

1.3.51 LEMMA: If C is satisfiable, then $fpi(C) = \varnothing$. □

1.3.4 Reasoning with constraints in an equality-only syntactic model

We have given a number of equivalence laws that are valid with respect to any interpretation of constraints, that is, within any model. However, an important special case is that of *equality-only syntactic models*. Indeed, in that specific setting, our constraint-based type systems are in close correspondence with DM. In short, we aim to prove that every satisfiable constraint admits a *canonical solved form* and to show that this notion corresponds to the standard concept of a *most general unifier*. We also establish a few technical properties of most general unifiers. These results are exploited in §1.4.3.

Thus, let us now assume that constraints are interpreted in an equality-only syntactic model. Let us further assume that, for every kind κ, (i) there are at least *two* type constructors of image kind κ and (ii) for every type constructor F of image kind κ, there exists $t \in \mathcal{M}_\kappa$ such that $t(\epsilon) = F$. We refer to models that violate (i) or (ii) as *degenerate*; one may argue that such models are of little interest. The assumption that the model is nondegenerate is used in the proof of Lemmas 1.3.52 and 1.3.59.

Under these new assumptions, the interpretation of equality coincides with its syntax: every equation that holds in the model is in fact a syntactic truism. The converse, of course, holds in every model.

1.3.52 LEMMA: If $\mathsf{true} \Vdash T = T'$ holds, then T and T′ coincide. □

Proof: Let $\mathsf{true} \Vdash T = T'$ **(1)**. The proof is by induction on the structure of T and T′. First, assume neither of T and T′ is a type variable. In a free tree model, (1) implies that T and T′ have the same head symbol, that is, they must be of the form $F\vec{T}$ and $F\vec{T}'$, respectively, and implies $\mathsf{true} \Vdash \vec{T} = \vec{T}'$. By the induction hypothesis, this implies that $\vec{T}$ and $\vec{T}'$ coincide. Hence, T and T′ coincide as well. Next, assume T is a type variable and T′ is not. By requirements (i) and (ii) above, there exists a ground assignment that maps T to a ground type whose head symbol is *not* that of T′. This contradicts (1), so this case cannot arise. The case where T′ is a type variable and T is not is

symmetric. Last, assume both $\mathtt{T}$ and $\mathtt{T}'$ are type variables. If they are distinct, then, by requirements (i) and (ii) above, there exists a ground assignment that maps each of them to a distinct ground type, which contradicts (1). So, $\mathtt{T}$ and $\mathtt{T}'$ must coincide. □

In a syntactic model, ground types are finite trees. As a result, cyclic equations, such as $\mathtt{X} = \mathsf{int} \to \mathtt{X}$, are false.

1.3.53 LEMMA: $\mathtt{X} \in \mathit{ftv}(\mathtt{T})$ and $\mathtt{T} \notin \mathcal{V}$ imply $(\mathtt{X} = \mathtt{T}) \equiv \mathsf{false}$. □

Proof: Let $\mathtt{X} \in \mathit{ftv}(\mathtt{T})$ **(1)** and $\mathtt{T} \notin \mathcal{V}$ **(2)**. By way of contradiction, assume ϕ satisfies $\mathtt{X} = \mathtt{T}$, that is, $\phi(\mathtt{X}) = \phi(\mathtt{T})$ **(3)**. In a free tree model, (1) and (2) imply that $\phi(\mathtt{X})$ is a *strict* subtree of $\phi(\mathtt{T})$. Together with (3), this implies that $\phi(\mathtt{X})$ is a strict subtree of itself. In a syntactic model, where trees are inductively defined, this is impossible. Thus, $\mathtt{X} = \mathtt{T}$ is false. □

A *solved form* is a conjunction of equations, where the left-hand sides are *distinct* type variables that do not appear in the right-hand sides, possibly surrounded by a number of existential quantifiers. Our definition is identical to Lassez, Maher and Marriott's solved forms (1988) and to Jouannaud and Kirchner's *tree* solved forms (1991), except we allow for prenex existential quantifiers, which are made necessary by our richer constraint language. Jouannaud and Kirchner also define *dag* solved forms, which may be exponentially smaller. Because we define solved forms only for proof purposes, we need not take performance into account at this point. The efficient constraint solver presented in §1.8 does manipulate graphs, rather than trees. Type scheme introduction and instantiation constructs cannot appear within solved forms; indeed, provided the constraint at hand has no free program identifiers, they can be expanded away. For this reason, their presence in the constraint language has no impact on the results contained in this section.

1.3.54 DEFINITION: A *solved form* is of the form $\exists \bar{\mathtt{Y}}.(\vec{\mathtt{X}} = \vec{\mathtt{T}})$, where $\bar{\mathtt{X}} \# \mathit{ftv}(\bar{\mathtt{T}})$. □

Solved forms offer a convenient way of reasoning about constraints because every satisfiable constraint is equivalent to one. This property is established by the following lemma.

1.3.55 LEMMA: Let $\mathit{fpi}(\mathtt{C}) = \varnothing$. Then, C is equivalent to either a solved form or false. □

Proof: We first establish that every conjunction of equations is equivalent to either a solved form or false. To do so, we present Robinson's unification algorithm (1971) as a rewriting system. The system's invariant is to operate on constraints of the form either $\vec{\mathtt{X}} = \vec{\mathtt{T}}; \mathtt{C}$, where $\bar{\mathtt{X}} \# \mathit{ftv}(\bar{\mathtt{T}}, \mathtt{C})$ and the semicolon

is interpreted as a distinguished conjunction, or false. We identify equations in C up to commutativity. The system is defined as follows:

$$
\begin{array}{lrcl}
\vec{X} = \vec{T}; & X = X \wedge C & \to & \vec{X} = \vec{T}; C \\
\vec{X} = \vec{T}; & F\,\vec{T}_1 = F\,\vec{T}_2 \wedge C & \to & \vec{X} = \vec{T}; \vec{T}_1 = \vec{T}_2 \wedge C \\
\vec{X} = \vec{T}; & F_1\,\vec{T}_1 = F_2\,\vec{T}_2 \wedge C & \to & \mathsf{false} \\
 & & & \text{if } F_1 \neq F_2 \\
\vec{X} = \vec{T}; & X = T \wedge C & \to & \vec{X} = [X \mapsto T]\vec{T} \wedge X = T; [X \mapsto T]C \\
 & & & \text{if } X \notin \mathit{ftv}(T) \\
\vec{X} = \vec{T}; & X = T \wedge C & \to & \mathsf{false} \\
 & & & \text{if } X \in \mathit{ftv}(T) \text{ and } T \notin \mathcal{V}
\end{array}
$$

It is straightforward to check that the above invariant is indeed preserved by the rewriting system. Let us check that constraint equivalence is also preserved. For the first rule, this is immediate. For the second and third rules, it follows from the fact that we have assumed a free tree model; for the fourth rule, a consequence of C-EQ; for the last rule, a consequence of Lemma 1.3.53. Furthermore, the system is terminating; this is witnessed by an ordering where false is the least element and where constraints of the form $\vec{X} = \vec{T}; C$ are ordered lexicographically, first by the number of type variables that appear free within C, second by the size of C. Last, a normal form for this rewriting system must be of the form either $\vec{X} = \vec{T}; \mathsf{true}$, where (by the invariant) $\vec{X} \# \mathit{ftv}(\vec{T})$—that is, a solved form, or false.

Next, we show that the present lemma holds when C is built out of equations, conjunction, and existential quantification. Orienting C-EXAND from left to right yields a terminating rewriting system that preserves constraint equivalence. The normal form of C must be $\exists \vec{Y}.C'$, where C' is a conjunction of equations. By the previous result, C' is equivalent to either a solved form or false. Because solved forms are preserved by existential quantification and because $\exists \vec{Y}.\mathsf{false}$ is false, the same holds of C.

Last, we establish the result in the general case. We assume $\mathit{fpi}(C) = \varnothing$ **(1)**. Orienting C-INID and C-IN* from left to right yields a terminating rewriting system that preserves constraint equivalence. The normal form C' of C cannot contain any type scheme introduction forms; given (1), it cannot contain any instantiation forms either. Thus, C' is built out of equations, conjunction, and existential quantification only. By the previous result, it is equivalent to either a solved form or false, which implies that the same holds of C. □

It is possible to impose further restrictions on solved forms. A solved form $\exists \vec{Y}.(\vec{X} = \vec{T})$ is *canonical* if and only if its free type variables are exactly $\vec{X}$. This is stated, in an equivalent way, by the following definition.

1.3.56 DEFINITION: A *canonical solved form* is a constraint of the form $\exists \bar{Y}.(\vec{X} = \vec{T})$, where $ftv(\bar{T}) \subseteq \bar{Y}$ and $\bar{X} \# \bar{Y}$. □

1.3.57 LEMMA: Every solved form is equivalent to a canonical solved form. □

Proof: Consider the rewriting rule $\exists \bar{Y} X.(X = T \wedge \vec{X} = \vec{T}) \to \exists \bar{Y}.(\vec{X} = \vec{T})$. Assume its left-hand side is a solved form. Then, its right-hand side is a solved form as well. Furthermore, we have $X \notin ftv(T, \vec{X}, \vec{T})$, which by C-NAMEEQ implies that the rule preserves constraint equivalence. The rule is terminating; its normal forms are solved forms $\exists \bar{Y}.(\vec{X} = \vec{T})$ such that $\bar{X} \# \bar{Y}$.

Next, consider the rewriting rule $\exists \bar{Y}.(\vec{X} = \vec{T}) \to \exists \bar{Y} Y.(\vec{X} = [X \mapsto Y]\vec{T} \wedge X = Y)$, with the side-conditions $X \in ftv(\bar{T}) \setminus \bar{Y}$ **(1)** and $Y \notin ftv(\bar{Y}, \bar{X}, \bar{T})$ **(2)**. Assume its left-hand side is a solved form. By (1) and (2), we have $X \notin \bar{X}$, $X \neq Y$, and $Y \notin \bar{X}$, which imply that the right-hand side is a solved form as well. By (1), (2), and C-NAMEEQ, the rule preserves constraint equivalence. The rule is terminating; its normal forms are solved forms $\exists \bar{Y}.(\vec{X} = \vec{T})$ such that $ftv(\bar{T}) \subseteq \bar{Y}$. □

It is easy to describe the solutions of a canonical solved form: they are the ground refinements of the substitution $[\vec{X} \mapsto \vec{T}]$.

1.3.58 LEMMA: A ground assignment ϕ satisfies a canonical solved form $\exists \bar{Y}.(\vec{X} = \vec{T})$ if and only if there exists a ground assignment ϕ' such that $\phi(\vec{X}) = \phi'(\vec{T})$. As a result, every canonical solved form is satisfiable. □

Proof: Let $\exists \bar{Y}.(\vec{X} = \vec{T})$ be a canonical solved form. By CM-EXISTS and CM-PREDICATE, ϕ satisfies $\exists \bar{Y}.(\vec{X} = \vec{T})$ if and only if there exists $\vec{t}$ such that $\phi[\vec{Y} \mapsto \vec{t}](\vec{X}) = \phi[\vec{Y} \mapsto \vec{t}](\vec{T})$. Thanks to the hypotheses $\bar{X} \# \bar{Y}$ and $ftv(\bar{T}) \subseteq \bar{Y}$, this is equivalent to the existence of a ground assignment ϕ' such that $\phi(\vec{X}) = \phi'(\vec{T})$. Thus, for every ground assignment ϕ', $\phi'[\vec{X} \mapsto \phi'(\vec{T})]$ satisfies $\exists \bar{Y}.(\vec{X} = \vec{T})$, which proves that this constraint is satisfiable. □

Together, Lemmas 1.3.57 and 1.3.58 imply that every solved form is satisfiable. Our interest in canonical solved forms stems from the following fundamental property, which provides a *syntactic* characterization of entailment between canonical solved forms: if $\exists \bar{Y}_1.(\vec{X} = \vec{T}_1)$ is more specific than $\exists \bar{Y}_2.(\vec{X} = \vec{T}_2)$, in a logical sense, then $\vec{T}_1$ refines $\vec{T}_2$, in a syntactic sense. The converse also holds (can you prove it?), but is not needed here.

1.3.59 LEMMA: If $\exists \bar{Y}_1.(\vec{X} = \vec{T}_1) \Vdash \exists \bar{Y}_2.(\vec{X} = \vec{T}_2)$, where both sides are canonical solved forms, then there exists a type substitution φ such that $\vec{T}_1 = \varphi(\vec{T}_2)$. □

Proof: Let $\exists \bar{Y}_1.(\vec{X} = \vec{T}_1) \Vdash \exists \bar{Y}_2.(\vec{X} = \vec{T}_2)$ **(1)**. By Lemma 1.3.58, (1) may be read $\forall \phi \quad (\exists \phi_1 \quad \phi(\vec{X}) = \phi_1(\vec{T}_1)) \Rightarrow (\exists \phi_2 \quad \phi(\vec{X}) = \phi_2(\vec{T}_1))$, that is,

$\forall\phi_1\exists\phi_2 \quad \phi_1(\vec{T}_1) = \phi_2(\vec{T}_2)$ **(2)**. If all elements of $\vec{T}_2$ are variables, then $[\vec{T}_2 \mapsto \vec{T}_1]$ determines a type substitution, even though $\vec{T}_2$ may have repeated elements. Indeed, using (2) and the fact that the model is nondegenerate, it is not difficult to check that if two elements of $\vec{T}_2$ coincide, then the corresponding elements of $\vec{T}_1$ must coincide as well. The general case, where $\vec{T}_2$ contains arbitrary terms, follows by induction on the size of $\vec{T}_2$, again using the fact that the model is nondegenerate. □

As a corollary, we find that canonical solved forms are *unique* up to α-conversion and up to C-Ex*, *provided* the set $\bar{X}$ of their free type variables is fixed.

1.3.60 Lemma: If the canonical solved forms $\exists\bar{Y}_1.(\vec{X} = \vec{T}_1)$ and $\exists\bar{Y}_2.(\vec{X} = \vec{T}_2)$ are equivalent, then there exists a renaming ρ such that $\vec{T}_1 = \rho(\vec{T}_2)$. □

Please note that the fact that the canonical solved forms $\exists\bar{Y}_1.(\vec{X}_1 = \vec{T}_1)$ and $\exists\bar{Y}_2.(\vec{X}_2 = \vec{T}_2)$ are equivalent does *not* imply that $\bar{X}_1$ and $\bar{X}_2$ coincide. Consider, for example, the canonical solved forms true and $\exists Y.(X = Y)$, which by C-NameEq are equivalent. One might wish to further restrict canonical solved forms by requiring $\bar{X}$ to be the set of *essential* type variables of the constraint $\exists\bar{Y}.(\vec{X} = \vec{T})$, that is, the set of the type variables that appear free in *all* equivalent constraints. However, as far our technical development is concerned, it seems more convenient not to do so. Instead, we show that it is possible to explicitly restrict or extend $\bar{X}$ when needed (Lemma 1.3.63).

The following definition allows entertaining a dual view of canonical solved forms, either as constraints or as idempotent type substitutions. The latter view is commonly found in standard treatments of unification (Lassez, Maher, and Marriott, 1988; Jouannaud and Kirchner, 1991) and in classic presentations of ML-the-type-system.

1.3.61 Definition: If $[\vec{X} \mapsto \vec{T}]$ is an idempotent substitution of domain $\bar{X}$, let $\exists[\vec{X} \mapsto \vec{T}]$ denote the canonical solved form $\exists\bar{Y}.(\vec{X} = \vec{T})$, where $\bar{Y} = \mathit{ftv}(\bar{T})$. An idempotent substitution θ is a *most general unifier* of the constraint C if and only if $\exists\theta$ and C are equivalent. □

By definition, equivalent constraints admit the same most general unifiers. Many properties of canonical solved forms may be reformulated in terms of most general unifiers. By Lemmas 1.3.51, 1.3.55, and 1.3.57, every satisfiable constraint admits a most general unifier. By Lemma 1.3.60, if $[\vec{X} \mapsto \vec{T}_1]$ and $[\vec{X} \mapsto \vec{T}_2]$ are most general unifiers of C, then $\vec{T}_1$ and $\vec{T}_2$ coincide up to a renaming. Conversely, if $[\vec{X} \mapsto \vec{T}]$ is a most general unifier of C and if $\bar{X} \# \rho$ holds, then $[\vec{X} \mapsto \rho\vec{T}]$ is also a most general unifier of C; indeed, these two substitutions correspond to α-equivalent canonical solved forms.

The following result relates the substitution θ to the canonical solved form $\exists\theta$, stating that every ground refinement of the former satisfies the latter.

1.3.62 LEMMA: $\theta(\exists\theta) \equiv \mathsf{true}$. □

Proof: Let $\theta = [\vec{X} \mapsto \vec{T}]$ **(1)** be an idempotent substitution of domain $\bar{X}$. Then, $\exists\theta$ is the canonical solved form $\exists\bar{Y}.(\vec{X} = \vec{T})$, where $\bar{Y} = \mathit{ftv}(\bar{T})$. By Lemma 1.3.13, $\phi \models \theta(\exists\theta)$ **(2)** is equivalent to $\phi \circ \theta \models \exists\theta$, which, according to Lemma 1.3.58, is equivalent to $\exists\phi' \quad (\phi \circ \theta)(\vec{X}) = \phi'(\vec{T})$. By (1), this may be simplified to $\exists\phi' \quad \phi(\vec{T}) = \phi'(\vec{T})$, which is true. We have proved that (2) holds for every ground assignment ϕ, which means $\theta(\exists\theta) \equiv \mathsf{true}$. □

The following lemma offers two technical results: the domain of a most general unifier of C may be restricted so as to become a subset of $\mathit{ftv}(C)$; it may also be extended to include arbitrary fresh variables. The next lemma is a simple corollary.

1.3.63 LEMMA: Let θ be a most general unifier of C. If $\bar{Z} \# \mathit{ftv}(C)$, then $\theta \setminus \bar{Z}$ is also a most general unifier of C. If $\bar{Z} \# \theta$, then there exists a most general unifier of C that extends θ and whose domain is $\mathit{dom}(\theta) \cup \bar{Z}$. □

Proof: Let θ be a most general unifier of C. θ is of the form $[\vec{X} \mapsto \vec{T}]$, where $\bar{X} = \mathit{dom}(\theta)$, and C is equivalent to the canonical solved form $\exists\bar{Y}.(\vec{X} = \vec{T})$, where $\bar{Y} = \mathit{ftv}(\bar{T})$.

First, let $\bar{Z} \# \mathit{ftv}(C)$ **(1)**. Write $\vec{X} = \vec{T}$ as $\vec{X}_1 = \vec{T}_1 \wedge \vec{X}_2 = \vec{T}_2$, where $\bar{X}_1 = \bar{X} \setminus \bar{Z}$ and $\bar{X}_2 = \bar{X} \cap \bar{Z}$. By (1), we have $\bar{X}_2 \# \mathit{ftv}(C)$ **(2)**. By Lemma 1.3.58, the solutions of C are the ground assignments ϕ such that $\phi(\vec{X}_1) = \phi'(\vec{T}_1)$ and $\phi(\vec{X}_2) = \phi'(\vec{T}_2)$ hold for some ϕ'. Then, by (2) and Lemma 1.3.12, the solutions of C are the ground assignments ϕ such that $\phi(\vec{X}_1) = \phi'(\vec{T}_1)$ holds for some ϕ'. In other words, C is equivalent to the canonical solved form $\exists\bar{Y}_1.(\vec{X}_1 = \vec{T}_1)$, where $\bar{Y}_1 = \mathit{ftv}(\vec{T}_1)$. Furthermore, $[\vec{X}_1 \mapsto \vec{T}_1]$ is precisely $\theta \setminus \bar{Z}$.

Now, let $\bar{Z} \# \theta$, that is, $\bar{Z} \# \bar{X}\bar{Y}$ **(1)**. Let ρ be a renaming such that $\rho\bar{Z} \# \bar{X}\bar{Y}\bar{Z}$ **(2)**. By (2) and C-NAMEEQ, $\exists\bar{Y}.(\vec{X} = \vec{T})$ is equivalent to $\exists\bar{Y}\rho\bar{Z}.(\vec{X} = \vec{T} \wedge \vec{Z} = \rho\vec{Z})$ **(3)**. By (1) and (2), (3) is a canonical solved form. Thus, the extension of θ with $[\vec{Z} \mapsto \rho\vec{Z}]$ is a most general unifier of C. □

1.3.64 LEMMA: Let θ_1 and θ_2 be most general unifiers of C. Let $\bar{X} = \mathit{dom}(\theta_1) \cap \mathit{dom}(\theta_2)$. Then, $\theta_1(\bar{X})$ and $\theta_2(\bar{X})$ coincide up to a renaming. □

Proof: θ_1 is a most general unifier of $\exists\theta_2$, whose free type variables are $\mathit{dom}(\theta_2)$. By Lemma 1.3.63, its restriction to $\mathit{dom}(\theta_1) \cap \mathit{dom}(\theta_2)$ is also a most general unifier of $\exists\theta_2$; in other words, the restriction of θ_1 to $\bar{X}$ is a most general unifier of C. Similarly, the restriction of θ_2 to $\bar{X}$ is a most general unifier of C. The result follows by Lemma 1.3.60. □

Our last technical result relates the most general unifiers of C with the most general unifiers of $\exists X.C$. It states that the former are extensions of the latter.

Furthermore, under a few freshness conditions, *every* most general unifier of $\exists X.C$ may be extended to yield a most general unifier of C.

1.3.65 LEMMA: If θ is a most general unifier of C, then $\theta \setminus X$ is a most general unifier of $\exists X.C$. Conversely, if θ is a most general unifier of $\exists X.C$ and $X \# \theta$ and $ftv(\exists X.C) \subseteq dom(\theta)$, then there exists a type substitution θ' such that θ' extends θ, θ' is a most general unifier of C, and $dom(\theta') = dom(\theta) \cup X$. □

Proof: Let θ be a most general unifier of C. Then, C must be equivalent to a canonical solved form $\exists \bar{Y}.(\vec{X} = \vec{T})$, where θ is $[\vec{X} \mapsto \vec{T}]$. We now distinguish two cases. First, if $X \notin \bar{X}$, then C and $\exists X.C$ are equivalent, so θ is a most general unifier for $\exists X.C$. Furthermore, given $dom(\theta) = \bar{X} \not\ni X$, $\theta \setminus X$ is θ itself. Second, if $X \in \bar{X}$, then $\exists X.C$ may be written $\exists \bar{Y}.\exists X.(X = T \wedge \vec{X}' = \vec{T}')$, which by C-NAMEEQ is equivalent to $\exists \bar{Y}.(\vec{X}' = \vec{T}')$. As a result, the substitution $[\vec{X}' \mapsto \vec{T}']$—which is precisely $\theta \setminus X$—is a most general unifier for $\exists X.C$.

Conversely, let θ be a most general unifier of $\exists X.C$. Let $X \# \theta$ **(1)** and $ftv(\exists X.C) \subseteq dom(\theta)$ **(2)**. Because θ is idempotent, we have $dom(\theta) \# range(\theta)$ **(3)**. Together, (2) and (3) imply $ftv(\exists X.C) \# range(\theta)$ **(4)**. Furthermore, (1) and (4) imply $ftv(C) \# range(\theta)$ **(5)**. Because $\exists X.C$ is satisfiable, C is satisfiable as well, so it admits a most general unifier θ'. By Lemma 1.3.63 and by α-conversion, it is possible to require $dom(\theta) \subseteq dom(\theta')$ **(6)** and $range(\theta') \# range(\theta)$ **(7)**. Furthermore, by (3), (5), and Lemma 1.3.63, it is possible to further require $dom(\theta') \# range(\theta)$ **(8)** without compromising (6) or (7). By the previous result, $\theta' \setminus X$ must be a most general unifier of $\exists X.C$. By Lemma 1.3.64, θ and $\theta' \setminus X$ must coincide up to a renaming on the intersection of their domains, which by (6) and (1) is $dom(\theta)$. Thus, if θ is $[\vec{X} \mapsto \vec{T}]$, where $\bar{X} = dom(\theta)$, then $\theta'(\vec{X})$ is $[\vec{Y} \mapsto \vec{Y}'_0]\vec{T}$, where $[\vec{Y} \mapsto \vec{Y}'_0]$ is a bijection between $\bar{Y} = range(\theta)$ and $\bar{Y}'_0$, a subset of $\bar{Y}' = range(\theta')$. Thus, the constraint $C \equiv \exists \theta'$ may be written as the canonical solved form $\exists \bar{Y}'.(\vec{X} = [\vec{Y} \mapsto \vec{Y}'_0]\vec{T} \wedge \vec{X}' = \vec{T}')$ **(9)**. By (7), the substitution $[\vec{Y}'_0 \mapsto \vec{Y}]$ defines a bijection on $\bar{Y}'$. Furthermore, by (8), its image—a subset of $\bar{Y}\bar{Y}'$—is fresh for the free type variables of (9)—that is, $\bar{X}\bar{X}'$. These arguments show that an α-equivalent form of (9) is $\exists [\vec{Y}'_0 \mapsto \vec{Y}]\bar{Y}'.(\vec{X} = \vec{T} \wedge \vec{X}' = [\vec{Y}'_0 \mapsto \vec{Y}]\vec{T}')$. This yields a most general unifier θ' of C that extends θ.

By (1) and (3), we have $dom(\theta) \cup X \# range(\theta)$. This allows requiring, modulo a renaming of the image of θ', $dom(\theta) \cup X \# range(\theta')$ **(10)**. Furthermore, by (2), we have $ftv(C) \subseteq dom(\theta) \cup X$ **(11)**. By (10), (11), and Lemma 1.3.63, we may further require $dom(\theta') = dom(\theta) \cup X$. □

1.4 HM(X)

Constraint-based type systems appeared during the 1980s (Mitchell, 1984; Fuh and Mishra, 1988) and were widely studied during the following decade (Curtis, 1990; Aiken and Wimmers, 1993; Jones, 1994; Smith, 1994; Palsberg, 1995; Trifonov and Smith, 1996; Fähndrich, 1999; Pottier, 2001b). We now present one such system, baptized HM(X) because it is a *parameterized* extension of Hindley and Milner's type discipline; the meaning of the parameter X was explained on page 29. Its original description is due to Odersky, Sulzmann, and Wehr (1999). Since then, it has been completed in a number of works (Müller, 1998; Sulzmann, Müller, and Zenger, 1999; Sulzmann, 2000; Pottier, 2001a; Skalka and Pottier, 2002). Each of these presentations introduces minor variations. Here, we follow (Pottier, 2001a), which is itself inspired by (Sulzmann, Müller, and Zenger, 1999).

1.4.1 Definition

Our presentation of HM(X) relies on the constraint language introduced in §1.3. Technically, our approach to constraints is less abstract than that of (Odersky, Sulzmann, and Wehr, 1999). We interpret constraints within a model, give conjunction and existential quantification their standard meaning, and derive a number of equivalence laws (§1.3). Odersky *et al.*, on the other hand, do not explicitly rely on a logical interpretation; instead, they axiomatize constraint equivalence, that is, they consider a number of equivalence laws as axioms. Thus, they ensure that their high-level proofs, such as type soundness and correctness and completeness of type inference, are independent of the low-level details of the logical interpretation of constraints. Their approach is also more general, since it allows dealing with other logical interpretations, such as "open-world" interpretations, where constraints are interpreted not within a fixed model, but within a *family* of extensions of a "current" model. In this chapter, we have avoided this extra layer of abstraction, for the sake of definiteness; however, the changes required to adopt Odersky *et al.*'s approach would not be extensive, since the forthcoming proofs do indeed rely mostly on constraint equivalence laws, rather than on low-level details of the logical interpretation of constraints.

Another slight departure from Odersky *et al.*'s work lies in the fact that we have enriched the constraint language with type scheme introduction and instantiation forms, which were absent in the original presentation of HM(X). To prevent this addition from affecting HM(X), we require the constraints that appear in HM(X) typing judgements to *have no free program identifiers*. Please note that this does not prevent them from containing let forms; we

$$\frac{\Gamma(x) = \sigma \qquad C \Vdash \exists\sigma}{C, \Gamma \vdash x : \sigma} \text{ (HMX-VAR)}$$

$$\frac{C, (\Gamma; z : T) \vdash t : T'}{C, \Gamma \vdash \lambda z.t : T \to T'} \text{ (HMX-ABS)}$$

$$\frac{C, \Gamma \vdash t_1 : T \to T' \qquad C, \Gamma \vdash t_2 : T}{C, \Gamma \vdash t_1\ t_2 : T'} \text{ (HMX-APP)}$$

$$\frac{C, \Gamma \vdash t_1 : \sigma \quad C, (\Gamma; z : \sigma) \vdash t_2 : T}{C, \Gamma \vdash \texttt{let}\ z = t_1\ \texttt{in}\ t_2 : T} \text{ (HMX-LET)}$$

$$\frac{C \wedge D, \Gamma \vdash t : T \qquad \bar{X} \# ftv(C, \Gamma)}{C \wedge \exists\bar{X}.D, \Gamma \vdash t : \forall\bar{X}[D].T} \text{ (HMX-GEN)}$$

$$\frac{C, \Gamma \vdash t : \forall\bar{X}[D].T}{C \wedge D, \Gamma \vdash t : T} \text{ (HMX-INST)}$$

$$\frac{C, \Gamma \vdash t : T \qquad C \Vdash T \leq T'}{C, \Gamma \vdash t : T'} \text{ (HMX-SUB)}$$

$$\frac{C, \Gamma \vdash t : \sigma \qquad \bar{X} \# ftv(\Gamma, \sigma)}{\exists\bar{X}.C, \Gamma \vdash t : \sigma} \text{ (HMX-EXISTS)}$$

Figure 1-7: Typing rules for HM(X)

shall in fact exploit this feature when establishing an equivalence between HM(X) and the type system presented in §1.5, where the new constraint forms are effectively used.

The type system HM(X) consists of a four-place *judgement* whose parameters are a constraint C, an environment Γ, an expression t, and a type scheme σ. A judgement is written $C, \Gamma \vdash t : \sigma$ and is read: *under the assumptions* C *and* Γ*, the expression* t *has type* σ. One may view C as an assumption about the judgement's free type variables and Γ as an assumption about t's free program identifiers. Please recall that Γ now contains *constrained* type schemes, and that σ is a *constrained* type scheme.

We would like the validity of a typing judgement to depend not on the *syntax*, but only on the *meaning* of its constraint assumption. We enforce this point of view by considering judgements equal modulo equivalence of their constraint assumptions. In other words, the typing judgements $C, \Gamma \vdash t : \sigma$ and $D, \Gamma \vdash t : \sigma$ are considered identical when $C \equiv D$ holds. A judgement is *valid*, or *holds*, if and only if it is derivable via the rules given in Figure 1-7. Please note that a valid judgement may involve an unsatisfiable constraint. A program t is *well-typed* within the (closed) environment Γ if and only if a judgement of the form $C, \Gamma \vdash t : \sigma$ holds for some *satisfiable* constraint C. One might wonder why we do not make the apparently stronger requirement that $C \wedge \exists\sigma$ be satisfiable; however, by inspection of the typing rules, the reader may check that, if the above judgement is derivable, then $C \Vdash \exists\sigma$ must hold (see Lemma 1.4.1), hence the two requirements are equivalent.

Let us now explain the rules. Like DM-VAR, HMX-VAR looks up the envi-

ronment to determine the type scheme associated with the program identifier x. Its second premise plays a minor technical role: as noted in the previous paragraph, its presence helps simplify the definition of well-typedness. HMX-ABS, HMX-APP, and HMX-LET are identical to DM-ABS, DM-APP, and DM-LET, respectively, except that the assumption C is made available to every subderivation. We recall that the type T may be viewed as the type scheme $\forall\varnothing[\mathsf{true}].\mathtt{T}$ (Definitions 1.2.21 and 1.3.2). As a result, types form a subset of type schemes, which implies that $\Gamma; \mathtt{z} : \mathtt{T}$ is a well-formed environment and $\mathsf{C}, \Gamma \vdash \mathtt{t} : \mathtt{T}$ a well-formed typing judgement. To understand HMX-GEN, it is best to first consider the particular case where C is true. This yields the following, simpler rule:

$$\frac{\mathsf{D}, \Gamma \vdash \mathtt{t} : \mathtt{T} \qquad \bar{\mathtt{X}} \# \mathit{ftv}(\Gamma)}{\exists\bar{\mathtt{X}}.\mathsf{D}, \Gamma \vdash \mathtt{t} : \forall\bar{\mathtt{X}}[\mathsf{D}].\mathtt{T}} \qquad \text{(HMX-GEN')}$$

The second premise is identical to that of DM-GEN: the type variables that are generalized must not occur free within the environment. The conclusion forms the type scheme $\forall\bar{\mathtt{X}}[\mathsf{D}].\mathtt{T}$, where the type variables $\bar{\mathtt{X}}$ have become universally quantified, but are still subject to the constraint D. Please note that the type variables that occur free in D may include not only $\bar{\mathtt{X}}$, but also other type variables, typically free in Γ. HMX-GEN may be viewed as a more liberal version of HMX-GEN', whereby part of the current constraint, namely C, need not be copied if it does not concern the type variables that are being generalized, namely $\bar{\mathtt{X}}$. This optimization is important in practice, because C may be very large. An intuitive explanation for its correctness is given by the constraint equivalence law C-LETAND, which expresses the same optimization in terms of let constraints. Because HM(X) does not use let constraints, the optimization is hard-wired into the typing rule. As a last technical remark, let us point out that replacing $\mathsf{C} \wedge \exists\bar{\mathtt{X}}.\mathsf{D}$ with $\mathsf{C} \wedge \mathsf{D}$ in HMX-GEN's conclusion would not affect the set of derivable judgements; this fact may be established using HMX-EXISTS and Lemma 1.4.2. HMX-INST allows taking an instance of a type scheme. The reader may be surprised to find that, contrary to DM-INST, it does not involve a type substitution. Instead, the rule merely drops the universal quantifier, which amounts to applying the identity substitution $\vec{\mathtt{X}} \mapsto \vec{\mathtt{X}}$. One should recall, however, that type schemes are considered equal modulo α-conversion, so it is possible to *rename* the type scheme's universal quantifiers prior to using HMX-INST. The reason why this provides sufficient expressive power appears in Exercise 1.4.3 and in the proof of Theorem 1.4.10 below. The constraint D carried by the type scheme is recorded as part of the current constraint in HMX-INST's conclusion. The *subsumption* rule HMX-SUB allows a type T to be replaced at any time with an arbitrary supertype $\mathtt{T}'$. Because both T and $\mathtt{T}'$ may have free type vari-

ables, whether $\mathtt{T} \leq \mathtt{T}'$ holds depends on the current assumption C, which is why the rule's second premise is an *entailment* assertion. An operational explanation of HMX-SUB is that it requires all uses of subsumption to be explicitly recorded in the current constraint. Please note that HMX-SUB remains a useful and necessary rule even when subtyping is interpreted as equality: then, it allows exploiting the type *equations* found in C. Last, HMX-EXISTS allows the type variables that occur only within the current constraint to become existentially quantified. As a result, these type variables no longer occur free in the rule's conclusion; in other words, they have become *local* to the subderivation rooted at the premise. One may prove that the presence of HMX-EXISTS in the type system does not augment the set of well-typed programs, but does augment the set of valid typing judgements; it is a pleasant technical convenience. Indeed, because judgements are considered equal modulo constraint equivalence, constraints may be transparently *simplified* at any time. (By *simplifying* a constraint, we mean replacing it with an equivalent constraint whose syntactic representation is considered simpler.) Bearing this fact in mind, one finds that an effect of rule HMX-EXISTS is to enable *more* simplifications: because constraint equivalence is a congruence, $\mathrm{C} \equiv \mathrm{D}$ implies $\exists\bar{\mathtt{X}}.\mathrm{C} \equiv \exists\bar{\mathtt{X}}.\mathrm{D}$, but the converse does not hold in general. For instance, there is in general no way of simplifying the judgement $\mathtt{X} \leq \mathtt{Y} \leq \mathtt{Z}, \Gamma \vdash \mathtt{t} : \sigma$, but if it is known that $\mathtt{Y}$ does not appear free in Γ or σ, then HMX-EXISTS allows deriving $\exists\mathtt{Y}.(\mathtt{X} \leq \mathtt{Y} \leq \mathtt{Z}), \Gamma \vdash \mathtt{t} : \sigma$, which is the same judgement as $\mathtt{X} \leq \mathtt{Z}, \Gamma \vdash \mathtt{t} : \sigma$. Thus, an interesting simplification has been enabled. Please note that $\mathtt{X} \leq \mathtt{Y} \leq \mathtt{Z} \equiv \mathtt{X} \leq \mathtt{Z}$ does *not* hold, while, according to C-EXTRANS, $\exists\mathtt{Y}.(\mathtt{X} \leq \mathtt{Y} \leq \mathtt{Z}) \equiv \mathtt{X} \leq \mathtt{Z}$ does.

We now establish a few simple properties of the type system HM(X). Our first lemma is a minor technical property. As noted earlier, this justifies our simplified definition of well-typedness.

1.4.1 LEMMA: $\mathrm{C}, \Gamma \vdash \mathtt{t} : \sigma$ implies $\mathrm{C} \Vdash \exists\sigma$. □

Proof: The proof is by structural induction on a derivation of $\mathrm{C}, \Gamma \vdash \mathtt{t} : \sigma$. In each proof case, we adopt the notations of Figure 1-7.

◦ *Case* HMX-VAR. The goal is precisely the rule's second premise.

◦ *Case* HMX-ABS, HMX-APP, HMX-LET, HMX-INST, HMX-SUB. In these cases σ is in fact a type $\mathtt{T}$, that is, a type scheme of the form $\forall\varnothing[\mathsf{true}].\mathtt{T}$. As a result, the goal is $\mathrm{C} \Vdash \mathsf{true}$, a tautology.

◦ *Case* HMX-GEN. The rule's conclusion is $\mathrm{C} \wedge \exists\bar{\mathtt{X}}.\mathrm{D}, \Gamma \vdash \mathtt{t} : \forall\bar{\mathtt{X}}[\mathrm{D}].\mathtt{T}$. The goal is $\mathrm{C} \wedge \exists\bar{\mathtt{X}}.\mathrm{D} \Vdash \exists\bar{\mathtt{X}}.\mathrm{D}$, a tautology.

◦ *Case* HMX-EXISTS. The rule's conclusion is $\exists\bar{\mathtt{X}}.\mathrm{C}, \Gamma \vdash \mathtt{t} : \sigma$. Its premises are $\mathrm{C}, \Gamma \vdash \mathtt{t} : \sigma$ **(1)** and $\bar{\mathtt{X}} \# \mathit{ftv}(\Gamma, \sigma)$ **(2)**. Applying the induction hypothesis

to (1) yields $C \Vdash \exists\sigma$ **(3)**. By congruence of entailment, by (2) and C-Ex*, (3) implies $\exists\bar{X}.C \Vdash \exists\sigma$. □

A pleasant property of HM(X) is that *strengthening* a judgement's constraint assumption preserves its validity. In other words, *weakening* a judgement preserves its validity. It is worth noting that in traditional presentations, which rely more heavily on type substitutions, the analogue of this result is a *type substitution* lemma; see for instance (Tofte, 1988, Lemma 2.7), (Rémy, 1992a, Lemma 1), (Leroy, 1992, Proposition 1.2), (Skalka and Pottier, 2002, Lemma 3.4). Here, the lemma further states that weakening a judgement does not alter the shape of its derivation, a useful property when reasoning by induction on type derivations.

1.4.2 LEMMA [WEAKENING]: If $C' \Vdash C$, then every derivation of $C, \Gamma \vdash t : \sigma$ may be turned into a derivation of $C', \Gamma \vdash t : \sigma$ with the same shape. □

Proof: The proof is by structural induction on a derivation of $C, \Gamma \vdash t : \sigma$. In each proof case, we adopt the notations of Figure 1-7.

∘ *Case* HMX-VAR. The rule's conclusion is $C, \Gamma \vdash x : \sigma$. Its premises are $\Gamma(x) = \sigma$ **(1)** and $C \Vdash \exists\sigma$ **(2)**. By hypothesis, we have $C' \Vdash C$ **(3)**. By transitivity of entailment, (3) and (2) imply $C' \Vdash \exists\sigma$ **(4)**. By HMX-VAR, (1) and (4) yield $C', \Gamma \vdash x : \sigma$.

∘ *Cases* HMX-ABS, HMX-APP, HMX-LET. By the induction hypothesis and by HMX-ABS, HMX-APP, or HMX-LET, respectively.

∘ *Case* HMX-GEN. The rule's conclusion is $C \wedge \exists\bar{X}.D, \Gamma \vdash t : \forall\bar{X}[D].\tau$. Its premises are $C \wedge D, \Gamma \vdash t : \tau$ **(1)** and $\bar{X} \# \mathit{ftv}(C, \Gamma)$ **(2)**. By hypothesis, we have $C' \Vdash C \wedge \exists\bar{X}.D$ **(3)**. We may assume, *w.l.o.g.*, $\bar{X} \# \mathit{ftv}(C')$ **(4)**. Applying the induction hypothesis to (1) and to the entailment assertion $C' \wedge C \wedge D \Vdash C \wedge D$, we obtain $C' \wedge C \wedge D, \Gamma \vdash t : \tau$ **(5)**. By HMX-GEN, applied to (5), (2) and (4), we get $C' \wedge C \wedge \exists\bar{X}.D, \Gamma \vdash t : \forall\bar{X}[D].\tau$ **(6)**. By (3) and C-DUP, the constraints $C' \wedge C \wedge \exists\bar{X}.D$ and C' are equivalent, so (6) is the goal $C', \Gamma \vdash t : \forall\bar{X}[D].\tau$.

∘ *Case* HMX-INST. The rule's conclusion is $C \wedge D, \Gamma \vdash t : \tau$. Its premise is $C, \Gamma \vdash t : \forall\bar{X}[D].\tau$ **(1)**. By hypothesis, C' entails $C \wedge D$ **(2)**. Because (2) implies $C' \Vdash C$, the induction hypothesis may be applied to (1), yielding $C', \Gamma \vdash t : \forall\bar{X}[D].\tau$ **(3)**. By HMX-INST, we obtain $C' \wedge D, \Gamma \vdash t : \tau$ **(4)**. Because (2) implies $C' \equiv C' \wedge D$, (4) is the goal $C', \Gamma \vdash t : \tau$.

∘ *Case* HMX-SUB. The rule's conclusion is $C, \Gamma \vdash t : \tau'$. Its premises are $C, \Gamma \vdash t : \tau$ **(1)** and $C \Vdash \tau \leq \tau'$ **(2)**. By hypothesis, we have $C' \Vdash C$ **(3)**. Applying the induction hypothesis to (1) and (3) yields $C', \Gamma \vdash t : \tau$ **(4)**. By transitivity of entailment, (3) and (2) imply $C' \Vdash \tau \leq \tau'$ **(5)**. By HMX-SUB, (4) and (5) yield $C', \Gamma \vdash t : \tau'$.

∘ *Case* HMX-EXISTS. The rule's conclusion is $\exists\bar{X}.C, \Gamma \vdash t : \sigma$. Its premises are $C, \Gamma \vdash t : \sigma$ **(1)** and $\bar{X} \# ftv(\Gamma, \sigma)$ **(2)**. By hypothesis, we have $C' \Vdash \exists\bar{X}.C$ **(3)**. We may assume, *w.l.o.g.*, $\bar{X} \# ftv(C')$ **(4)**. Applying the induction hypothesis to (1) and to the entailment assertion $C' \wedge C \Vdash C$, we obtain $C' \wedge C, \Gamma \vdash t : \sigma$ **(5)**. By HMX-EXISTS, (5) and (2) yield $\exists\bar{X}.(C' \wedge C), \Gamma \vdash t : \sigma$ **(6)**. By (4) and C-EXAND, the constraint $\exists\bar{X}.(C' \wedge C)$ is equivalent to $C' \wedge \exists\bar{X}.C$, which, by (3) and C-DUP, is equivalent to C'. Thus, (6) is the goal $C', \Gamma \vdash t : \sigma$. □

1.4.3 EXERCISE [RECOMMENDED, ★★]: In some presentations of HM(X), HMX-INST is replaced with the following variant:

$$\frac{C, \Gamma \vdash t : \forall\bar{X}[D].\tau \qquad C \Vdash [\vec{X} \mapsto \vec{T}]D}{C, \Gamma \vdash t : [\vec{X} \mapsto \vec{T}]\tau} \qquad \text{(HMX-INST')}$$

Show that HMX-INST′ is admissible in our presentation of HM(X)—that is, if its premise is derivable according to the rules of Figure 1-7, then so is its conclusion. □

1.4.4 EXERCISE [★]: Give a derivation of $\mathsf{true}, \varnothing \vdash \lambda z.z : \mathsf{int} \to \mathsf{int}$. Give a derivation of $\mathsf{true}, \varnothing \vdash \lambda z.z : \forall X.X \to X$. Check that the former judgement also follows from the latter via HMX-INST′ (Exercise 1.4.3), and determine which derivation of $\mathsf{true}, \varnothing \vdash \lambda z.z : \mathsf{int} \to \mathsf{int}$ this path gives rise to. □

We do not give a direct type soundness proof for HM(X). Instead, in the forthcoming sections, we prove that well-typedness in HM(X) is equivalent to the satisfiability of a certain constraint, and use that characterization as a basis for our type soundness proof. A direct type soundness result, based on a denotational semantics, may be found in (Odersky, Sulzmann, and Wehr, 1999). Another type soundness proof, which follows Wright and Felleisen's syntactic approach (1994b), appears in (Skalka and Pottier, 2002). Last, a hybrid approach, which combines some of the advantages of the previous two, is given in (Pottier, 2001a).

1.4.2 An alternate presentation of HM(X)

The presentation of HM(X) given in Figure 1-7 has only four syntax-directed rules out of eight. It is a good specification of the type system, but it is far from an algorithmic description. As a first step towards such a description, we provide an alternate presentation of HM(X), where generalization is performed only at `let` expressions and instantiation takes place only at references to program identifiers (Figure 1-8). It has the property that all judgements are of the form $C, \Gamma \vdash t : \tau$, rather than $C, \Gamma \vdash t : \sigma$. The following theorems state that the two presentations are indeed equivalent.

$$\frac{\Gamma(x) = \forall \bar{X}[D].T}{C \wedge D, \Gamma \vdash x : T} \quad \text{(HMD-VarInst)}$$

$$\frac{C, (\Gamma; z : T) \vdash t : T'}{C, \Gamma \vdash \lambda z.t : T \to T'} \quad \text{(HMD-Abs)}$$

$$\frac{C, \Gamma \vdash t_1 : T \to T' \qquad C, \Gamma \vdash t_2 : T}{C, \Gamma \vdash t_1\ t_2 : T'} \quad \text{(HMD-App)}$$

$$\frac{C \wedge D, \Gamma \vdash t_1 : T_1 \qquad \bar{X} \# \mathit{ftv}(C, \Gamma) \qquad C \wedge \exists \bar{X}.D, (\Gamma; z : \forall \bar{X}[D].T_1) \vdash t_2 : T_2}{C \wedge \exists \bar{X}.D, \Gamma \vdash \mathtt{let}\ z = t_1\ \mathtt{in}\ t_2 : T_2} \quad \text{(HMD-LetGen)}$$

$$\frac{C, \Gamma \vdash t : T \qquad C \Vdash T \leq T'}{C, \Gamma \vdash t : T'} \quad \text{(HMD-Sub)}$$

$$\frac{C, \Gamma \vdash t : T \qquad \bar{X} \# \mathit{ftv}(\Gamma, T)}{\exists \bar{X}.C, \Gamma \vdash t : T} \quad \text{(HMD-Exists)}$$

Figure 1-8: An alternate presentation of HM(X)

1.4.5 THEOREM: If $C, \Gamma \vdash t : T$ is derivable via the rules of Figure 1-8, then it is a valid HM(X) judgement. □

Proof: It suffices to check that every rule in Figure 1-8 is admissible in HM(X). This is immediate in all cases but the following.

∘ *Case* HMD-VarInst. Assume $\Gamma(x) = \forall \bar{X}[D].T$ **(1)**. By Lemma 1.3.22, $C \wedge D$ entails $\exists \bar{X}.D$ **(2)**. By HMX-Var, (1) and (2) imply $C \wedge D, \Gamma \vdash x : \forall \bar{X}[D].T$ **(3)**. By HMX-Inst, (3) implies $C \wedge D \wedge D, \Gamma \vdash x : T$, which is the goal, because $C \wedge D \wedge D$ is equivalent to $C \wedge D$.

∘ *Case* HMD-LetGen. Assume $C \wedge D, \Gamma \vdash t_1 : T_1$ **(1)** and $\bar{X} \# \mathit{ftv}(C, \Gamma)$ **(2)** and $C \wedge \exists \bar{X}.D, (\Gamma; z : \forall \bar{X}[D].T_1) \vdash t_2 : T_2$ **(3)**. By HMX-Gen, (1) and (2) imply $C \wedge \exists \bar{X}.D, \Gamma \vdash t_1 : \forall \bar{X}[D].T_1$ **(4)**. By HMX-Let, (4) and (3) yield $C \wedge \exists \bar{X}.D, \Gamma \vdash \mathtt{let}\ z = t_1\ \mathtt{in}\ t_2 : T_2$. □

1.4.6 THEOREM: If $C, \Gamma \vdash t : T$ is a valid HM(X) judgement, then it is derivable via the rules of Figure 1-8. □

Proof: The proof is by induction on the *weight* of a derivation of $C, \Gamma \vdash t : T$, where the weight of a derivation is the sum of the weights of the rule instances that it involves, and the weight of a rule instance is 2 if it is an instance of HMX-Gen or HMX-Inst and 1 otherwise. The result is immediate in all cases but the following. For the sake of clarity, we write $C, \Gamma \vdash_\bullet t : T$ for judgements which are derivable via the rules of Figure 1-8.

∘ *Case* HMX-Var. The rule's conclusion is $C, \Gamma \vdash x : T$. Its first premise is $\Gamma(x) = T$ **(1)**. (Its second premise is the tautology $C \Vdash \mathsf{true}$.) By HMD-VarInst, (1) implies $C, \Gamma \vdash_\bullet x : T$, which is the goal.

∘ *Case* HMX-LET. The rule's conclusion is $C, \Gamma \vdash \texttt{let z} = t_1 \texttt{ in } t_2 : \texttt{T}$ **(1)**. Its premises are $C, \Gamma \vdash t_1 : \sigma$ **(2)** and $C, (\Gamma; \texttt{z} : \sigma) \vdash t_2 : \texttt{T}$ **(3)**. Applying the induction hypothesis to (3) yields $C, (\Gamma; \texttt{z} : \sigma) \vdash_\bullet t_2 : \texttt{T}$ **(4)**. If σ is a monotype, then applying the induction hypothesis to (2) yields $C, \Gamma \vdash_\bullet t_1 : \sigma$ **(5)**. By HMD-LETGEN, instantiated with $\bar{X} = \varnothing$ and $D = \mathsf{true}$, (5) and (4) yield $C, \Gamma \vdash_\bullet \texttt{let z} = t_1 \texttt{ in } t_2 : \texttt{T}$, which is the goal. If σ is not a monotype, then let us analyze the last rule in the derivation of (2). It must be one of the following.

Subcase HMX-VAR. Then, t_1 is a program identifier x. Let us write $\sigma = \forall \bar{X}[D].\texttt{T}_1$, where $\bar{X} \# \mathit{ftv}(C, \Gamma)$ **(6)**. HMX-VAR's premises are $\Gamma(\texttt{x}) = \forall \bar{X}[D].\texttt{T}_1$ **(7)** and $C \Vdash \exists \bar{X}.D$ **(8)**. By (7) and HMD-VARINST, we have $C \wedge D, \Gamma \vdash_\bullet \texttt{x} : \texttt{T}_1$ **(9)**. By (8), C and $C \wedge \exists \bar{X}.D$ are equivalent. Thus, we may apply HMD-LETGEN to (9), (6) and (4) and obtain $C, \Gamma \vdash_\bullet \texttt{let z} = t_1 \texttt{ in } t_2 : \texttt{T}$.

Subcase HMX-GEN. Then, we have $C \equiv C_0 \wedge \exists \bar{X}.D$ and $\sigma = \forall \bar{X}[D].\texttt{T}_1$. HMX-GEN's premises are $C_0 \wedge D, \Gamma \vdash t_1 : \texttt{T}_1$ **(10)** and $\bar{X} \# \mathit{ftv}(C_0, \Gamma)$ **(11)**. Applying the induction hypothesis to (10) yields $C_0 \wedge D, \Gamma \vdash_\bullet t_1 : \texttt{T}_1$ **(12)**. Applying HMD-LETGEN to (12), (11) and (4) yields $C, \Gamma \vdash_\bullet \texttt{let z} = t_1 \texttt{ in } t_2 : \texttt{T}$.

Subcase HMX-EXISTS. Then, we have $C \equiv \exists \bar{Y}.C_0$. HMX-EXISTS's premises are $C_0, \Gamma \vdash t_1 : \sigma$ **(13)** and $\bar{Y} \# \mathit{ftv}(\Gamma, \sigma)$ **(14)**. We may assume, *w.l.o.g.*, $\bar{Y} \# \mathit{ftv}(\texttt{T})$ **(15)**. By Lemmas 1.3.22 and 1.4.2, (3) implies $C_0, (\Gamma; \texttt{z} : \sigma) \vdash t_2 : \texttt{T}$ **(16)**. By applying HMX-LET to (13) and (16), we obtain a derivation of $C_0, \Gamma \vdash \texttt{let z} = t_1 \texttt{ in } t_2 : \texttt{T}$ **(17)** that is lighter than the derivation of (1). Thus, by the induction hypothesis, we have $C_0, \Gamma \vdash_\bullet \texttt{let z} = t_1 \texttt{ in } t_2 : \texttt{T}$ **(18)**. By (14), (15), and HMD-EXISTS, (18) implies $C, \Gamma \vdash_\bullet \texttt{let z} = t_1 \texttt{ in } t_2 : \texttt{T}$.

∘ *Case* HMX-GEN. Because $\forall \bar{X}[D].\texttt{T}$ is a monotype, $\bar{X}$ must be empty and D must be true. As a result, the rule's premise coincides with its conclusion. The result follows by the induction hypothesis.

∘ *Case* HMX-INST. The rule's conclusion is $C \wedge D, \Gamma \vdash t : \texttt{T}$ **(1)**. Its premise is $C, \Gamma \vdash t : \forall \bar{X}[D].\texttt{T}$ **(2)**. If $\forall \bar{X}[D].\texttt{T}$ is a monotype, then (1) and (2) coincide and the result is immediate. If $\forall \bar{X}[D].\texttt{T}$ is not a monotype, then let us analyze the last rule in the derivation of (2). It must be one of the following.

Subcase HMX-VAR. Then, t is a program identifier x. The rule's first premise is $\Gamma(\texttt{x}) = \forall \bar{X}[D].\texttt{T}$ **(3)**. By HMD-VARINST, (3) implies $C \wedge D, \Gamma \vdash_\bullet t : \texttt{T}$.

Subcase HMX-GEN. Then, we have $C \equiv C_0 \wedge \exists \bar{X}.D$ **(4)**. HMX-GEN's premises are $C_0 \wedge \rho D, \Gamma \vdash t : \rho\texttt{T}$ **(5)** and $\rho\bar{X} \# \mathit{ftv}(C_0, \Gamma)$ **(6)**, where ρ is a renaming that is fresh for $\mathit{ftv}(\forall \bar{X}[D].\texttt{T})$. We may assume, *w.l.o.g.*, $\rho\bar{X} \# \mathit{ftv}(D, \texttt{T})$ **(7)**. By Lemma 1.4.2, (5) implies $C_0 \wedge \rho D \wedge \vec{X} = \rho\vec{X}, \Gamma \vdash t : \rho\texttt{T}$ **(8)**. By C-EQ and HMX-SUB, (8) implies $C_0 \wedge D \wedge \vec{X} = \rho\vec{X}, \Gamma \vdash t : \texttt{T}$ **(9)**. Applying HMX-EXISTS to (9), (6), and (7) yields $C_0 \wedge D \wedge \exists \rho\bar{X}.(\vec{X} = \rho\vec{X}), \Gamma \vdash t : \texttt{T}$, that is, $C_0 \wedge D, \Gamma \vdash t : \texttt{T}$ **(10)**. Furthermore, the derivation of (10) that we have just built is lighter

than that of (1). So, by the induction hypothesis, we have $C_0 \wedge D, \Gamma \vdash_\bullet t : \tau$ **(11)**. Because (4) implies $C \wedge D \equiv C_0 \wedge D$, (11) is the goal $C \wedge D, \Gamma \vdash_\bullet t : \tau$.

Subcase HMX-EXISTS. Then, we have $C \equiv \exists \bar{Y}.C_0$. HMX-EXISTS's premises are $C_0, \Gamma \vdash t : \forall \bar{X}[D].\tau$ **(12)** and $\bar{Y} \# ftv(\Gamma, \forall \bar{X}[D].\tau)$ **(13)**. We may assume, *w.l.o.g.*, $\bar{Y} \# \bar{X}$ **(14)**. Then, by (13) and (14), we have $\bar{Y} \# ftv(\Gamma, D, \tau)$ **(15)**. By applying HMX-INST to (12), we obtain a derivation of $C_0 \wedge D, \Gamma \vdash t : \tau$ **(16)** that is lighter than the derivation of (1). Thus, by the induction hypothesis, we have $C_0 \wedge D, \Gamma \vdash_\bullet t : \tau$ **(17)**. By (15), HMD-EXISTS, and C-EXAND, (17) implies $C \wedge D, \Gamma \vdash_\bullet t : \tau$. □

Theorems 1.4.5 and 1.4.6 show that the rule sets of Figures 1-7 and 1-8 derive the same monomorphic judgements, that is, the same judgements of the form $C, \Gamma \vdash t : \tau$. The fact that judgements of the form $C, \Gamma \vdash t : \sigma$, where σ is a not a monotype, cannot be derived using the new rule set is a technical simplification, without deep significance. The first two exercises below shed some light on this issue.

1.4.7 EXERCISE [★★]: Show that both rule sets lead to the same set of *well-typed* programs. □

1.4.8 EXERCISE [★★]: Show that, if HMX-GEN is added to the rule set of Figure 1-8, then both rule sets derive exactly the same judgements. □

1.4.9 EXERCISE [★★★, ↛]: Show that it is possible to simplify the presentation of Damas and Milner's type system in an analogous manner. That is, define an alternate set of typing rules for DM, which allows deriving judgements of the form $\Gamma \vdash t : \tau$; then, show that this new rule set is equivalent to the previous one, in the same sense as above. Which auxiliary properties of DM does your proof require? A solution is given in (Clément, Despeyroux, Despeyroux, and Kahn, 1986). □

1.4.3 Relating HM(X) with Damas and Milner's type system

In order to explain our interest in HM(X), we wish to show that it is more general than Damas and Milner's type system. Since HM(X) really is a *family* of type systems, we must make this statement more precise. First, every member of the HM(X) family contains DM. Conversely, DM contains HM(=), the constraint-based type system obtained by specializing HM(X) to the setting of an equality-only syntactic model.

The first of these assertions is easy to prove, because the mapping from DM judgements to HM(X) judgements is essentially the identity: every valid DM judgement may be viewed as a valid HM(X) judgement under the trivial

assumption true. This statement relies on the fact that the DM type scheme $\forall \bar{X}.T$ is identified with the constrained type scheme $\forall \bar{X}[\mathsf{true}].T$, so DM type schemes (resp. environments) form a subset of HM(X) type schemes (resp. environments). Its proof is routine.

1.4.10 THEOREM: If $\Gamma \vdash t : S$ holds in DM, then $\mathsf{true}, \Gamma \vdash t : S$ holds in HM(X). □

Proof: The proof is by structural induction on a derivation of $\Gamma \vdash t : S$. In each proof case, we adopt the notations of Figure 1-3.

∘ *Case* DM-VAR. The rule's conclusion is $\Gamma \vdash x : S$. Its premise is $\Gamma(x) = S$ **(1)**. By definition and by C-EX*, the constraint $\exists S$ is equivalent to true. By applying HMX-VAR to (1) and to the assertion $\mathsf{true} \Vdash \mathsf{true}$, we obtain $\mathsf{true}, \Gamma \vdash x : S$.

∘ *Cases* DM-ABS, DM-APP, DM-LET. By the induction hypothesis and by HMX-ABS, HMX-APP or HMX-LET, respectively.

∘ *Case* DM-GEN. The rule's conclusion is $\Gamma \vdash t : \forall \bar{X}.T$. Its premises are $\Gamma \vdash t : T$ **(1)** and $\bar{X} \# \mathit{ftv}(\Gamma)$ **(2)**. Applying the induction hypothesis to (1) yields $\mathsf{true}, \Gamma \vdash t : T$ **(3)**. Furthermore, (2) implies $\bar{X} \# \mathit{ftv}(\mathsf{true}, \Gamma)$ **(4)**. By HMX-GEN, (3) and (4) yield $\mathsf{true}, \Gamma \vdash t : \forall \bar{X}[\mathsf{true}].T$.

∘ *Case* DM-INST. The rule's conclusion is $\Gamma \vdash t : [\vec{X} \mapsto \vec{T}]T$. Its premise is $\Gamma \vdash t : \forall \bar{X}.T$. Applying the induction hypothesis to it yields $\mathsf{true}, \Gamma \vdash t : \forall \bar{X}[\mathsf{true}].T$, which by Exercise 1.4.3 implies $\mathsf{true}, \Gamma \vdash t : [\vec{X} \mapsto \vec{T}]T$. □

We are now interested in proving that HM(=), as defined above, is contained within DM. To this end, we must translate every HM(=) judgement to a DM judgement. It turns out that this is possible if the original judgement's constraint assumption is *satisfiable*. The translation relies on the fact that the definition of HM(=) assumes an equality-only syntactic model. Indeed, in that setting, every satisfiable constraint admits a most general unifier (Definition 1.3.61), whose properties we make essential use of.

We begin by explaining how an HM(=) is translated into a DM type scheme. In fact, we must not only translate a type scheme, but also apply a type substitution to it. Instead of separating these steps, we perform both at once, and parameterize the translation by a type substitution θ. (It does not appear that separating them would help.) The definition of $[\![\sigma]\!]_\theta$ is somewhat involved: it is given in the statement of the following lemma, whose proof establishes that the definition is indeed well-formed.

1.4.11 LEMMA: Consider a type scheme σ and an idempotent type substitution θ such that $\mathit{ftv}(\sigma) \subseteq \mathit{dom}(\theta)$ **(1)** and $\exists\theta \Vdash \exists\sigma$ **(2)**. Write $\sigma = \forall \bar{X}[D].T$, where $\bar{X} \# \theta$ **(3)**. Then, there exists a type substitution θ' such that θ' extends

θ, $dom(\theta')$ is $dom(\theta) \cup \bar{X}$, and θ' is a most general unifier of $\exists\theta \wedge D$. Let $\bar{Y} = ftv(\theta'(\bar{X})) \setminus range(\theta)$. Then, the *translation* of σ under θ, written $[\![\sigma]\!]_\theta$, is the DM type scheme $\forall\bar{Y}.\theta'(T)$. This is a well-formed definition. Furthermore, $ftv([\![\sigma]\!]_\theta) \subseteq range(\theta)$ holds. □

Proof: By (2), $\exists\theta$ is equivalent to $\exists\theta \wedge \exists\sigma$, which may be written $\exists\theta \wedge \exists\bar{X}.D$. By (3) and C-EXAND, this is $\exists\bar{X}.(\exists\theta \wedge D)$. Thus, because θ is a most general unifier of $\exists\theta$, θ is also a most general unifier of $\exists\bar{X}.(\exists\theta \wedge D)$ **(4)**. Furthermore, $ftv(\exists\bar{X}.(\exists\theta \wedge D))$ is $ftv(\exists\theta \wedge \exists\sigma)$, which by definition of $\exists\theta$ and by (1) is a subset of $dom(\theta)$ **(5)**. By (4), (3), (5), and Lemma 1.3.65, there exists a type substitution θ' such that θ' extends θ **(6)** and θ' is a most general unifier of $\exists\theta \wedge D$ **(7)** and $dom(\theta') = dom(\theta) \cup \bar{X}$ **(8)**.

Let us now define $\bar{Y} = ftv(\theta'(\bar{X})) \setminus range(\theta)$ and $[\![\sigma]\!]_\theta = \forall\bar{Y}.\theta'(T)$. By (1), we have $ftv(T) \subseteq \bar{X} \cup dom(\theta)$. Applying θ' and exploiting (6), we find $ftv(\theta'(T)) \subseteq ftv(\theta'(\bar{X})) \cup range(\theta)$, which by definition of $\bar{Y}$ may be written $ftv(\theta'(T)) \subseteq \bar{Y} \cup range(\theta)$. Subtracting $\bar{Y}$ on each side, we find $ftv([\![\sigma]\!]_\theta) \subseteq range(\theta)$ **(9)**.

To show that the definition of $[\![\sigma]\!]_\theta$ is valid, there remains to show that it does not depend on the choice of $\bar{X}$ or θ'. To prove the former, it suffices to establish $\bar{X} \# ftv([\![\sigma]\!]_\theta)$, which indeed follows from (3) and (9). As for the latter, because of the constraints imposed by (6), (7), and (8), and by Lemma 1.3.64, distinct choices of θ' may differ only by a renaming of $ftv(\theta'(\bar{X})) \setminus range(\theta)$, that is, $\bar{Y}$. So, we must check $\bar{Y} \# ftv([\![\sigma]\!]_\theta)$, which holds by definition. □

Please note that if σ is in fact a type T, where $ftv(T) \subseteq dom(\theta)$, then $\bar{X}$ is empty, so θ' is θ, $\bar{Y}$ is empty, and $[\![T]\!]_\theta = \theta(T)$. In other words, the translation of a type under θ is its image through θ. More generally, the translation of an unconstrained type scheme (that is, a type scheme whose constraint is true) is its image through θ, as stated by the following exercise.

1.4.12 EXERCISE [★★, ↛]: Prove that $[\![\forall\bar{X}.T]\!]_\theta$, when defined, is $\theta(\forall\bar{X}.T)$. □

The translation becomes more than a mere type substitution when applied to a nontrivial constrained type scheme. Some examples of this situation are given below.

1.4.13 EXAMPLE: Let $\sigma = \forall XY[X = Y \to Y].X$. Let θ be the identity substitution. The type scheme σ is closed and the constraint $\exists\sigma$ is equivalent to true, so $[\![\sigma]\!]_\theta$ is defined. We must find a type substitution θ' whose domain is XY and that is a most general unifier of $X = Y \to Y$. All such substitutions are of the form $[X \mapsto (Z \to Z), Y \mapsto Z]$, where Z is fresh. We have $ftv(\theta'(XY)) = Z$, whence $[\![\sigma]\!]_\theta = \forall Z.Z \to Z$. Note that the choice of Z does not matter, since it is bound in $[\![\sigma]\!]_\theta$. Roughly speaking, the effect of the translation was to replace the

body X of the constrained type scheme with its most general solution under the constraint $X = Y \to Y$.

Let $\sigma = \forall XY_1[X = Y_1 \to Y_2].X$. Let $\theta = [Y_2 \mapsto Z_2]$. We have $\mathit{ftv}(\sigma) = Y_2 \subseteq \mathit{dom}(\theta)$. The constraint $\exists\sigma$ is equivalent to true, so $[\![\sigma]\!]_\theta$ is defined. We must find a type substitution θ' whose domain is XY_1Y_2 that extends θ and that is a most general unifier of $X = Y_1 \to Y_2$. All such substitutions are of the form $[X \mapsto (Z_1 \to Z_2), Y_1 \mapsto Z_1, Y_2 \mapsto Z_2]$, where Z_1 is fresh. We have $\mathit{ftv}(\theta'(XY_1)) \setminus \mathit{range}(\theta) = Z_1Z_2 \setminus Z_2 = Z_1$, whence $[\![\sigma]\!]_\theta = \forall Z_1.Z_1 \to Z_2$. The type variable Z_2 is *not* universally quantified—even though it appears in the image of X, which *was* universally quantified in σ—because Z_2 is the image of Y_2, which was free in σ. □

Before attacking the main theorem, let us establish a couple of technical properties of the translation. First, $[\![\sigma]\!]_\theta$ is insensitive to the behavior of θ outside $\mathit{ftv}(\sigma)$, a natural property, since our informal intent is for θ to be applied to σ.

1.4.14 LEMMA: If θ_1 and θ_2 coincide on $\mathit{ftv}(\sigma)$, then $[\![\sigma]\!]_{\theta_1}$ and $[\![\sigma]\!]_{\theta_2}$ are either both undefined, or both defined and identical. □

Proof: Let θ_1 and θ_2 coincide on $\mathit{ftv}(\sigma)$. Then, there exists a type substitution θ such that $\mathit{dom}(\theta) = \mathit{ftv}(\sigma)$ and $\theta_i = \theta \cup [\vec{X}_i \mapsto \vec{T}_i]$, for $i \in \{1, 2\}$. Then, $\exists\theta$ is equivalent to $\exists\bar{X}_i.\exists\theta_i$. It is not difficult to prove that, as a result, $\exists\theta \Vdash \exists\sigma$ is equivalent to $\exists\theta_i \Vdash \exists\sigma$. Thus, $[\![\sigma]\!]_{\theta_1}$ and $[\![\sigma]\!]_{\theta_2}$ are either both undefined, or both defined.

Let us assume that $[\![\sigma]\!]_{\theta_i}$ is defined, for $i \in \{1, 2\}$. Let $\sigma = \forall\bar{X}[D].T$, where $\bar{X} \# \theta_i$. Then, $[\![\sigma]\!]_\theta$ is defined as well; let θ' be the auxiliary substitution involved in its definition. θ' is a most general unifier of $\exists\theta \wedge D$, that is, $\exists\bar{X}_i.\exists\theta\wedge D$, which by C-EXAND is $\exists\bar{X}_i.(\exists\theta\wedge D)$ **(1)**. We have $\bar{X}_i \# \mathit{dom}(\theta)\cup\bar{X} = \mathit{dom}(\theta')$ **(2)**. Furthermore, we have $\bar{X}_i \# \mathit{range}(\theta)$, which allows requiring, *w.l.o.g.*, $\bar{X}_i \# \mathit{range}(\theta')$ **(3)**. We have $\mathit{ftv}(\exists\bar{X}_i.(\exists\theta_i \wedge D)) \subseteq \mathit{dom}(\theta) \cup \bar{X} = \mathit{dom}(\theta')$ **(4)**. By (1), (2), (3), (4), and Lemma 1.3.65, there exists a substitution θ_i' such that θ_i' extends θ' and θ_i' is a most general unifier of $\exists\theta_i \wedge D$ and $\mathit{dom}(\theta_i') = \mathit{dom}(\theta')\cup\bar{X}_i$. Thus, θ_i' is a suitable auxiliary substitution for use in the definition of $[\![\sigma]\!]_{\theta_i}$. Furthermore, by α-conversion of θ_i', we may require $\mathit{range}(\theta_i') \setminus \mathit{range}(\theta') \# \mathit{ftv}(\theta_i'(\bar{X}))$, which may be written $\mathit{range}(\theta_i) \setminus \mathit{range}(\theta) \# \mathit{ftv}(\theta_i'(\bar{X}))$ **(5)**. Besides, θ_1' and θ_2' coincide on $\mathit{dom}(\theta') = \mathit{dom}(\theta) \cup \bar{X}$ **(6)**. Using (5) and (6), it is easy to check that $[\![\sigma]\!]_{\theta_1}$ and $[\![\sigma]\!]_{\theta_2}$ coincide. □

Second, if $C \Vdash \sigma \preceq T'$ holds, then the translations of σ and T' under a most general unifier of C are in Damas and Milner's instance relation. One might say, roughly speaking, that the instance relation is preserved by the translation.

1.4.15 LEMMA: Let $ftv(\sigma, \mathtt{T}') \subseteq dom(\theta)$ **(1)** and $\exists\theta \Vdash \exists\sigma$ **(2)**. Let $\exists\theta \Vdash \sigma \preceq \mathtt{T}'$ **(3)**. Then, $\theta(\mathtt{T}')$ is an instance of the DM type scheme $[\![\sigma]\!]_\theta$. □

Proof: Write $\sigma = \forall\bar{\mathtt{X}}[\mathrm{D}].\mathtt{T}$, where $\bar{\mathtt{X}} \# \theta$ **(4)** and $\bar{\mathtt{X}} \# ftv(\mathtt{T}')$ **(5)**. By (1), (2), and (4), one may define θ', $\bar{\mathtt{Y}}$, and $[\![\sigma]\!]_\theta$ exactly as in the statement of Lemma 1.4.11. By (5) and Definition 1.3.3, (3) is synonymous with $\exists\theta \Vdash \exists\bar{\mathtt{X}}.(\mathrm{D} \wedge \mathtt{T} = \mathtt{T}')$. Reasoning in the same manner as in the first paragraph of the proof of Lemma 1.4.11, we find that there exists a type substitution θ'' such that θ'' extends θ, $dom(\theta'')$ is $dom(\theta) \cup \bar{\mathtt{X}}$, and θ'' is a most general unifier of $\exists\theta \wedge \mathrm{D} \wedge \mathtt{T} = \mathtt{T}'$.

We have $dom(\theta') = dom(\theta'')$ **(6)**. Furthermore, θ' is a most general unifier of $\exists\theta \wedge \mathrm{D}$, while θ'' is a most general unifier of $\exists\theta \wedge \mathrm{D} \wedge \mathtt{T} = \mathtt{T}'$, which implies $\exists\theta'' \Vdash \exists\theta'$ **(7)**. By Lemma 1.3.59, θ'' refines θ'. That is, there exists a type substitution φ such that θ'' is the restriction of $\varphi \circ \theta'$ to $dom(\theta) \cup \bar{\mathtt{X}}$ **(8)**. We may require $dom(\varphi) \subseteq range(\theta) \cup ftv(\theta'(\bar{\mathtt{X}}))$ **(9)** without compromising (8).

Consider $\mathtt{X} \in dom(\theta)$. Because θ'' extends θ, we have $\theta''(\mathtt{X}) = \theta(\mathtt{X})$ **(10)**. Furthermore, by (8), we have $\theta''(\mathtt{X}) = (\varphi \circ \theta')(\mathtt{X}) = (\varphi \circ \theta)(\mathtt{X})$ **(11)**. Using (10) and (11), we find $\theta(\mathtt{X}) = \varphi(\theta(\mathtt{X}))$. Because this holds for every $\mathtt{X} \in dom(\theta)$, φ must be the identity over $range(\theta)$; that is, $dom(\varphi) \# range(\theta)$ **(12)** holds. Combining (9) and (12), we find $dom(\varphi) \subseteq ftv(\theta'(\bar{\mathtt{X}})) \setminus range(\theta)$, that is, $dom(\varphi) \subseteq \bar{\mathtt{Y}}$ **(13)**.

By construction of θ'', we have $\exists\theta'' \Vdash \mathtt{T} = \mathtt{T}'$. By Lemma 1.3.49, this implies $\theta''(\exists\theta'') \Vdash \theta''(\mathtt{T}) = \theta''(\mathtt{T}')$, which by Lemma 1.3.62 may be read $\mathsf{true} \Vdash \theta''(\mathtt{T}) = \theta''(\mathtt{T}')$. By Lemma 1.3.52, $\theta''(\mathtt{T})$ and $\theta''(\mathtt{T}')$ coincide. Because by (1) $ftv(\mathtt{T})$ is a subset of $dom(\theta) \cup \bar{\mathtt{X}}$ and by (8), the former may be written $\varphi(\theta'(\mathtt{T}))$. By (1) and because θ'' extends θ, the latter is $\theta(\mathtt{T}')$. Thus, we have $\varphi(\theta'(\mathtt{T})) = \theta(\mathtt{T}')$. Together with (13), this establishes that $\theta(\mathtt{T}')$ is an instance of $\forall\bar{\mathtt{Y}}.\theta'(\mathtt{T})$, that is, $[\![\sigma]\!]_\theta$. □

We extend the translation to environments as follows. $[\![\varnothing]\!]_\theta$ is $\varnothing$. If $\exists\theta \Vdash \exists\sigma$ holds, then $[\![\Gamma; \mathtt{x} : \sigma]\!]_\theta$ is $[\![\Gamma]\!]_\theta; \mathtt{x} : [\![\sigma]\!]_\theta$, otherwise it is $[\![\Gamma]\!]_\theta$. Notice that $[\![\Gamma]\!]_\theta$ contains fewer bindings than Γ, which ensures that bindings $\mathtt{x} : \sigma$ for which $\exists\theta \Vdash \exists\sigma$ does not hold will not be used in the translation. Please note that $[\![\Gamma]\!]_\theta$ is defined when $ftv(\Gamma) \subseteq dom(\theta)$ holds. We are now ready to state the main theorem.

1.4.16 THEOREM: Let $\mathrm{C}, \Gamma \vdash \mathtt{t} : \sigma$ hold in HM(=). Let θ be a most general unifier of C such that $ftv(\Gamma, \sigma) \subseteq dom(\theta)$. Then, $[\![\Gamma]\!]_\theta \vdash \mathtt{t} : [\![\sigma]\!]_\theta$ holds in DM. □

Please note that, by requiring θ to be a most general unifier of C, we also require C to be satisfiable. Judgements that carry an unsatisfiable constraint cannot be translated.

Proof: Let us first remark that, by Lemma 1.4.1, we have $C \Vdash \exists\sigma$. This may be written $\exists\theta \Vdash \exists\sigma$, which guarantees that $[\![\sigma]\!]_\theta$ is defined. The proof is by structural induction on an HM(=) typing derivation. We assume that the derivation is expressed in terms of the rules of Figure 1-8, but split HMD-LetGen into HMX-Let and HMX-Gen for the sake of readability.

◦ *Case* HMD-VarInst. The rule's conclusion is $C \wedge D, \Gamma \vdash x : \tau$. By hypothesis, θ is a most general unifier of $C \wedge D$ **(1)**, and $ftv(\tau) \subseteq dom(\theta)$ **(2)** holds. The rule's premise is $\Gamma(x) = \sigma$ **(3)**, where σ stands for $\forall\bar{X}[D].\tau$. By (1), we have $\exists\theta \equiv C \wedge D \Vdash D \Vdash \exists\bar{X}.D \equiv \exists\sigma$ **(4)**. Furthermore, we have $ftv(\sigma) \subseteq ftv(\Gamma) \subseteq dom(\theta)$ **(5)**. These facts show that $[\![\sigma]\!]_\theta$ is defined. Together with (3), this implies $[\![\Gamma]\!]_\theta(x) = [\![\sigma]\!]_\theta$. By DM-Var, $[\![\Gamma]\!]_\theta \vdash x : [\![\sigma]\!]_\theta$ **(6)** follows. Now, by Lemma 1.3.27, we have $D \Vdash \sigma \preceq \tau$, which, combined with $\exists\theta \Vdash D$, yields $\exists\theta \Vdash \sigma \preceq \tau$ **(7)**. By (7), (4), (5), (2), and Lemma 1.4.15, we find that $\theta(\tau)$ is an instance of $[\![\sigma]\!]_\theta$. Thus, applying DM-Inst to (6) yields $[\![\Gamma]\!]_\theta \vdash t : \theta(\tau)$.

◦ *Case* HMD-Abs. The rule's conclusion is $C, \Gamma \vdash \lambda z.t : \tau \to \tau'$. Its premise is $C, (\Gamma; z : \tau) \vdash t : \tau'$. Applying the induction hypothesis to it yields $[\![\Gamma]\!]_\theta; z : \theta(\tau) \vdash t : \theta(\tau')$. By DM-Abs, this implies $[\![\Gamma]\!]_\theta \vdash \lambda z.t : \theta(\tau) \to \theta(\tau')$, that is, $[\![\Gamma]\!]_\theta \vdash \lambda z.t : \theta(\tau \to \tau')$.

◦ *Case* HMD-App. By an extension of $dom(\theta)$ to include $ftv(\tau)$, by the induction hypothesis, and by DM-App.

◦ *Case* HMX-Let. By an extension of $dom(\theta)$ to include $ftv(\sigma)$, by the induction hypothesis, and by DM-Let.

◦ *Case* HMX-Gen. The rule's conclusion is $C \wedge \exists\sigma, \Gamma \vdash t : \sigma$, where σ stands for $\forall\bar{X}[D].\tau$. By hypothesis, θ is a most general unifier of $C \wedge \exists\sigma$ **(1)**, and $ftv(\Gamma, \sigma) \subseteq dom(\theta)$ **(2)** holds. The rule's premises are $C \wedge D, \Gamma \vdash t : \tau$ **(3)** and $\bar{X} \# ftv(C, \Gamma)$ **(4)**. We may further assume, *w.l.o.g.*, $\bar{X} \# \theta$ **(5)**. Given (1), (2), and (5), we may define θ' and $\bar{Y}$ exactly as in Lemma 1.4.11. Then, θ' is a most general unifier of $\exists\theta \wedge D$, that is, $C \wedge D$. Furthermore, $dom(\theta')$ is $dom(\theta) \cup \bar{X}$, which by (2) is a superset of $ftv(\Gamma, \tau)$. Thus, the induction hypothesis applies to θ' and to (3), yielding $[\![\Gamma]\!]_{\theta'} \vdash t : \theta'(\tau)$. Because θ' extends θ, by (2) and by Lemma 1.4.14, this may be read $[\![\Gamma]\!]_\theta \vdash t : \theta'(\tau)$ **(6)**. According to Lemma 1.4.11, we have $ftv([\![\Gamma]\!]_\theta) \subseteq range(\theta)$, which by construction of $\bar{Y}$ implies $\bar{Y} \# ftv([\![\Gamma]\!]_\theta)$ **(7)**. By DM-Gen, (6) and (7) yield $[\![\Gamma]\!]_\theta \vdash t : \forall\bar{Y}.\theta'(\tau)$, that is, $[\![\Gamma]\!]_\theta \vdash t : [\![\sigma]\!]_\theta$.

◦ *Case* HMD-Sub. The rule's conclusion is $C, \Gamma \vdash t : \tau'$. By hypothesis, θ is a most general unifier of C **(1)**, and $ftv(\Gamma, \tau') \subseteq dom(\theta)$ **(2)** holds. The goal is $[\![\Gamma]\!]_\theta \vdash t : \theta(\tau')$ **(3)**. The rule's premises are $C, \Gamma \vdash t : \tau$ **(4)** and $C \Vdash \tau = \tau'$ **(5)**. We may assume, *w.l.o.g.*, $ftv(\tau) \# range(\theta)$ **(6)**. Then, by (6) and Lemma 1.3.63, we may extend the domain of θ, so as to achieve $ftv(\tau) \subseteq dom(\theta)$ **(7)**, without compromising (1) or (2) or affecting the goal (3). By (1), (2), and (7), the in-

duction hypothesis applies to (4), yielding $[\![\Gamma]\!]_\theta \vdash t : \theta(\tau)$ **(8)**. Now, thanks to (1), (5) may be read $\exists\theta \Vdash \tau = \tau'$, which by Lemmas 1.3.49 and 1.3.62 implies $\mathsf{true} \Vdash \theta(\tau) = \theta(\tau')$. Then, Lemma 1.3.52 shows that $\theta(\tau)$ and $\theta(\tau')$ coincide. As a result, (8) is the goal (3).

∘ *Case* HMD-EXISTS. The rule's conclusion is $\exists\bar{x}.C, \Gamma \vdash t : \tau$. By hypothesis, θ is a most general unifier of $\exists\bar{x}.C$ **(1)**, and $\mathit{ftv}(\Gamma, \tau) \subseteq \mathit{dom}(\theta)$ **(2)** holds. The rule's premises are $C, \Gamma \vdash t : \tau$ **(3)** and $\bar{x} \# \mathit{ftv}(\Gamma, \tau)$. We may assume, *w.l.o.g.*, $\bar{x} \# \theta$ **(4)**. As in the previous case, we may extend the domain of θ to guarantee $\mathit{ftv}(\exists\bar{x}.C) \subseteq \mathit{dom}(\theta)$ **(5)**. By (1), (4), (5), and Lemma 1.3.65, there exists a type substitution θ' such that θ' extends θ **(6)** and θ' is a most general unifier of C. Applying the induction hypothesis to θ' and to (3) yields $[\![\Gamma]\!]_{\theta'} \vdash t : \theta'(\tau)$. By (2), (6), and Lemma 1.4.14, this may be read $[\![\Gamma]\!]_\theta \vdash t : \theta(\tau)$. □

Together, Theorems 1.4.10 and 1.4.16 yield a precise correspondence between DM and HM(=): there exists a compositional translation from each to the other. In other words, they may be viewed as two equivalent formulations of a single type system. One might also say that HM(=) is a constraint-based formulation of DM. Furthermore, Theorem 1.4.10 states that every member of the HM(X) family is an extension of DM. This explains our double interest in HM(X), as an alternate formulation of DM, which we believe is more pleasant, for reasons already discussed, and as a more expressive framework.

1.5 A purely constraint-based type system: PCB(X)

In the previous section, we have presented HM(X), an elegant constraint-based extension of Damas and Milner's type system. However, HM(X), as described there, suffers from a drawback. A typing judgement involves both a constraint, which represents an assumption about its free type variables, and an environment, which represents an assumption about its free program identifiers. At a let node, HMD-LETGEN turns a part of the current constraint, namely D, into a type scheme, namely $\forall\bar{x}[D].\tau$, and stores it into the environment. Then, at every occurrence of the `let`-bound variable, HMD-VARINST retrieves this type scheme from the environment and adds a copy of D back to the current constraint. If such an approach is adopted, it is important to *simplify* the type scheme $\forall\bar{x}[D].\tau$ *before* it is stored in the environment, because it would be inefficient to copy an unsimplified constraint. In other words, it appears that, in order to preserve efficiency, constraint generation and constraint simplification cannot be separated.

Of course, in practice, it is not difficult to intermix these phases, so the

$$\frac{C \Vdash x \preceq \tau}{C \vdash x : \tau} \quad (\textsc{Var}) \qquad \frac{C_1 \vdash t_1 : \tau_1 \qquad C_2 \vdash t_2 : \tau_2}{\begin{array}{c}\mathsf{let}\ z : \forall \mathcal{V}[C_1].\tau_1\ \mathsf{in}\ C_2 \\ \vdash \mathtt{let}\ z = t_1\ \mathtt{in}\ t_2 : \tau_2\end{array}} \quad (\textsc{Let})$$

$$\frac{C \vdash t : \tau'}{\mathsf{let}\ z : \tau\ \mathsf{in}\ C \vdash \lambda z.t : \tau \to \tau'} \quad (\textsc{Abs}) \qquad \frac{C \vdash t : \tau}{C \wedge \tau \leq \tau' \vdash t : \tau'} \quad (\textsc{Sub})$$

$$\frac{C_1 \vdash t_1 : \tau \to \tau' \qquad C_2 \vdash t_2 : \tau}{C_1 \wedge C_2 \vdash t_1\ t_2 : \tau'} \quad (\textsc{App}) \qquad \frac{C \vdash t : \tau \qquad \bar{X} \# \mathit{ftv}(\tau)}{\exists \bar{X}.C \vdash t : \tau} \quad (\textsc{Exists})$$

Figure 1-9: Typing rules for PCB(X)

problem is not technical, but pedagogical. Indeed, we argued earlier that it is natural and desirable to separate them. *Type scheme introduction and elimination constraints*, which we introduced in §1.3 but did not use in the specification of HM(X), are intended as a means of solving this problem. In the present section, we exploit them to give a novel formulation of HM(X), which no longer requires copying constraints back and forth between the environment and the constraint assumption. In fact, the environment is suppressed altogether: taking advantage of the new constraint forms, we encode information about program identifiers within the constraint assumption.

1.5.1 Presentation

We now employ the full constraint language (§1.3). Typing judgements take the form $C \vdash t : \tau$, where C may have free type variables *and* free program identifiers. The rules that allow deriving such judgements appear in Figure 1-9. As before, we identify judgements up to constraint equivalence.

Let us review the rules. VAR states that x has type τ under any constraint that entails $x \preceq \tau$. Note that we no longer consult the type scheme associated with x in the environment—indeed, there is no environment. Instead, we let the constraint assumption record the fact that the type scheme should admit τ as one of its instances. Thus, in a judgement $C \vdash t : \tau$, any program identifier that occurs free within t typically also occurs free within C. ABS requires the body t of a λ-abstraction to have type τ' under assumption C. Although no explicit assumption about z appears in the premise, C typically contains a number of instantiation constraints bearing on z, of the form $z \preceq \tau_i$. In the rule's conclusion, C is wrapped within the context $\mathsf{let}\ z : \tau\ \mathsf{in}\ []$, where τ is the type assigned to z. This effectively requires every τ_i to denote a super-

type of $\mathtt{T}$, as desired. Please note that $\mathtt{z}$ does not occur free in the constraint $\mathsf{let}\ \mathtt{z} : \mathtt{T}\ \mathsf{in}\ \mathsf{C}$, which is necessary for well-formedness of the definition, since it does not occur free in $\lambda\mathtt{z}.\mathtt{t}$. APP exhibits a minor stylistic difference with respect to HMX-APP: its constraint assumption is split between its premises. It is not difficult to prove that, when weakening holds (see Lemma 1.5.2 below), this choice does not affect the set of valid judgements. This new presentation encourages reading the rules in Figure 1-9 as the specification of an algorithm, which, given $\mathtt{t}$ and $\mathtt{T}$, produces C such that $\mathsf{C} \vdash \mathtt{t} : \mathtt{T}$ holds. In the case of APP, the algorithm invokes itself recursively for each of the two subexpressions, yielding the constraints C_1 and C_2, then constructs their conjunction. LET is analogous to ABS: by wrapping C_2 within a let prefix, it gives meaning to the instantiation constraints bearing on $\mathtt{z}$ within C_2. The difference is that $\mathtt{z}$ may now be assigned a type scheme, as opposed to a monotype. An appropriate type scheme is built in the most straightforward manner from the constraint C_1 and the type $\mathtt{T}_1$ that describe $\mathtt{t}_1$. All of the type variables that appear free in the left-hand premise are generalized, hence the notation $\forall\mathcal{V}[\mathsf{C}_1].\mathtt{T}_1$, which is a convenient shorthand for $\forall \mathit{ftv}(\mathsf{C}_1, \mathtt{T}_1)[\mathsf{C}_1].\mathtt{T}_1$. The side-condition that "type variables that occur free in the environment must not be generalized", which was present in DM and HM(X), naturally disappears, since judgements no longer involve an environment. SUB again exhibits a minor stylistic difference with respect to HMX-SUB: the comments made about APP above apply here as well. EXISTS is essentially identical to HMX-EXISTS.

In the standard specification of HM(X), HMD-ABS and HMD-LETGEN accumulate information in the environment. Through the environment, this information is made available to HMD-VARINST, which retrieves and copies it. Here, instead, no information is explicitly transmitted. Where a program identifier is bound, a type scheme introduction constraint is built; where a program identifier is used, a type scheme instantiation constraint is produced. The two are related only by our definition of the meaning of constraints.

The reader may be puzzled by the fact that LET allows *all* type variables that occur free in its left-hand premise to be generalized. The following exercise sheds some light on this issue.

1.5.1 EXERCISE [RECOMMENDED, ★]: Build a type derivation for the expression $\lambda\mathtt{z}_1.\mathtt{let}\ \mathtt{z}_2 = \mathtt{z}_1\ \mathtt{in}\ \mathtt{z}_2$ within PCB(X). Draw a comparison with the solution of Exercise 1.2.24. □

The following lemma is an analogue of Lemma 1.4.2.

1.5.2 LEMMA [WEAKENING]: If $\mathsf{C}' \Vdash \mathsf{C}$, then every derivation of $\mathsf{C} \vdash \mathtt{t} : \mathtt{T}$ may be turned into a derivation of $\mathsf{C}' \vdash \mathtt{t} : \mathtt{T}$ with the same shape. □

Proof: The proof is by structural induction on a derivation of $C \vdash t : T$. In each proof case, we adopt the notations of Figure 1-9.

◦ *Case* VAR. By transitivity of entailment.

◦ *Case* ABS. The rule's conclusion is $\mathsf{let}\ z : T\ \mathsf{in}\ C \vdash \lambda z.t : T \to T'$ **(1)**. By hypothesis, we have $C' \Vdash \mathsf{let}\ z : T\ \mathsf{in}\ C$ **(2)**. We may assume, *w.l.o.g.*, $z \notin \mathit{fpi}(C')$ **(3)**. The rule's premise is $C \vdash t : T'$ **(4)**. Applying the induction hypothesis to (4) yields $C \wedge C' \vdash t : T'$, which by ABS implies $\mathsf{let}\ z : T\ \mathsf{in}\ (C \wedge C') \vdash \lambda z.t : T \to T'$ **(5)**. By (3) and C-INAND*, $\mathsf{let}\ z : T\ \mathsf{in}\ (C \wedge C')$ is equivalent to $(\mathsf{let}\ z : T\ \mathsf{in}\ C) \wedge C'$, which by (2) and C-DUP is equivalent to C'. Thus, (5) is the goal $C' \vdash \lambda z.t : T \to T'$.

◦ *Case* APP. By applying the induction hypothesis to each premise, using the fact that $C' \Vdash C_1 \wedge C_2$ implies $C' \Vdash C_1$ and $C' \Vdash C_2$.

◦ *Case* LET. Analogous to the case of ABS. The induction hypothesis is applied to the second premise only.

◦ *Case* SUB. Analogous to the case of APP.

◦ *Case* EXISTS. See the corresponding case in the proof of Lemma 1.4.2. □

1.5.2 Relating PCB(X) with HM(X)

Let us now provide evidence for our claim that PCB(X) is an alternate presentation of HM(X). The next two theorems define an effective translation from HM(X) to PCB(X) and back.

The first theorem states that if, within HM(X), t has type T under assumptions C and Γ, then, within PCB(X), t also has type T, under some assumption C'. The relationship $C \Vdash \mathsf{let}\ \Gamma\ \mathsf{in}\ C'$ states that C entails the residual constraint obtained by confronting Γ, which provides information about the free program identifiers in t, with C', which contains instantiation constraints bearing on these program identifiers. The statement requires C and Γ to have no free program identifiers, which is natural, since they are part of an HM(X) judgement. The hypothesis $C \Vdash \exists\Gamma$ excludes the somewhat pathological situation where Γ contains constraints not apparent in C. This hypothesis vanishes when Γ is the initial environment; see Definition 1.7.3.

1.5.3 THEOREM: Let $C \Vdash \exists\Gamma$. Assume $\mathit{fpi}(C, \Gamma) = \varnothing$. If $C, \Gamma \vdash t : T$ holds in HM(X), then there exists a constraint C' such that $C' \vdash t : T$ holds in PCB(X) and C entails $\mathsf{let}\ \Gamma\ \mathsf{in}\ C'$. □

Proof: The proof is by structural induction on a derivation of $C, \Gamma \vdash t : T$. In each proof case, we adopt the notations of Figure 1-8.

◦ *Case* HMD-VARINST. The rule's conclusion is $C \wedge D, \Gamma \vdash x : T$. By hypothesis, we have $C \wedge D \Vdash \exists\Gamma$ **(1)** and $\mathit{fpi}(C, D, \Gamma) = \varnothing$ **(2)**. The rule's premise

is $\Gamma(x) = \forall \bar{X}[D].T$ **(3)**. By VAR, we have $x \preceq T \vdash x : T$, so there remains to establish $C \wedge D \Vdash \mathsf{let}\ \Gamma\ \mathsf{in}\ x \preceq T$ **(4)**. By (3), (2), and C-INID, the constraint $\mathsf{let}\ \Gamma\ \mathsf{in}\ x \preceq T$ is equivalent to $\mathsf{let}\ \Gamma\ \mathsf{in}\ \forall \bar{X}[D].T \preceq T$, which, by (2) and C-IN*, is itself equivalent to $\exists \Gamma \wedge \forall \bar{X}[D].T \preceq T$ **(5)**. By (1) and Lemma 1.3.27, $C \wedge D$ entails (5). We have established (4).

∘ *Case* HMD-ABS. The rule's conclusion is $C, \Gamma \vdash \lambda z.t : T \to T'$. Its premise is $C, (\Gamma; z : T) \vdash t : T'$ **(1)**. The constraints $\exists \Gamma$ and $\exists(\Gamma; z : T)$ are equivalent, so the induction hypothesis applies to (1) and yields a constraint C' such that $C' \vdash t : T'$ **(2)** and $C \Vdash \mathsf{let}\ \Gamma; z : T\ \mathsf{in}\ C'$ **(3)**. Applying ABS to (2) yields $\mathsf{let}\ z : T\ \mathsf{in}\ C' \vdash \lambda z.t : T \to T'$. There remains to check that C entails $\mathsf{let}\ \Gamma\ \mathsf{in}\ \mathsf{let}\ z : T\ \mathsf{in}\ C'$—but that is precisely (3).

∘ *Case* HMD-APP. The rule's conclusion is $C, \Gamma \vdash t_1\ t_2 : T'$. Its premises are $C, \Gamma \vdash t_1 : T \to T'$ **(1)** and $C, \Gamma \vdash t_2 : T$ **(2)**. Applying the induction hypothesis to (1) and (2), we obtain constraints C_1' and C_2' such that $C_1' \vdash t_1 : T \to T'$ **(3)** and $C_2' \vdash t_2 : T$ **(4)** and $C \Vdash \mathsf{let}\ \Gamma\ \mathsf{in}\ C_1'$ **(5)** and $C \Vdash \mathsf{let}\ \Gamma\ \mathsf{in}\ C_2'$ **(6)**. By APP, (3) and (4) imply $C_1' \wedge C_2' \vdash t_1\ t_2 : T'$. Furthermore, by C-INAND, (5) and (6) yield $C \Vdash \mathsf{let}\ \Gamma\ \mathsf{in}\ C_1' \wedge C_2'$.

∘ *Case* HMD-LETGEN. The rule's conclusion is $C \wedge \exists \bar{X}.D, \Gamma \vdash \mathtt{let}\ z = t_1\ \mathtt{in}\ t_2 : T_2$. By hypothesis, we have $C \wedge \exists \bar{X}.D \Vdash \exists \Gamma$ **(1)** and $\mathit{fpi}(C, D, \Gamma) = \varnothing$ **(2)**. The rule's premises are $C \wedge D, \Gamma \vdash t_1 : T_1$ **(3)** and $\bar{X} \# \mathit{ftv}(C, \Gamma)$ **(4)** and $C \wedge \exists \bar{X}.D, \Gamma' \vdash t_2 : T_2$ **(5)**, where Γ' is $\Gamma; z : \forall \bar{X}[D].T_1$. Applying the induction hypothesis to (3) yields a constraint C_1' such that $C_1' \vdash t_1 : T_1$ **(6)** and $C \wedge D \Vdash \mathsf{let}\ \Gamma\ \mathsf{in}\ C_1'$ **(7)**. By (1), (2), and C-IN*, we have $C \wedge \exists \bar{X}.D \Vdash \exists \Gamma'$. Thus, the induction hypothesis applies to (5) and yields a constraint C_2' such that $C_2' \vdash t_2 : T_2$ **(8)** and $C \wedge \exists \bar{X}.D \Vdash \mathsf{let}\ \Gamma'\ \mathsf{in}\ C_2'$ **(9)**. By LET, (6) and (8) imply $\mathsf{let}\ z : \forall \mathcal{V}[C_1'].T_1\ \mathsf{in}\ C_2' \vdash \mathtt{let}\ z = t_1\ \mathtt{in}\ t_2 : T_2$ **(10)**. By Lemmas 1.3.43 and 1.5.2, (10) yields $\mathsf{let}\ z : \forall \bar{X}[C_1'].T_1\ \mathsf{in}\ C_2' \vdash \mathtt{let}\ z = t_1\ \mathtt{in}\ t_2 : T_2$ **(11)**, where the universal quantification is over $\bar{X}$ only. There remains to establish that $C \wedge \exists \bar{X}.D$ entails $\mathsf{let}\ \Gamma; z : \forall \bar{X}[C_1'].T_1\ \mathsf{in}\ C_2'$ **(12)**. By (4), (2), and C-LETDUP, the constraint (12) is equivalent to $\mathsf{let}\ \Gamma; z : \forall \bar{X}[\mathsf{let}\ \Gamma\ \mathsf{in}\ C_1'].T_1\ \mathsf{in}\ C_2'$. By (7), this constraint is entailed by $\mathsf{let}\ \Gamma; z : \forall \bar{X}[C \wedge D].T_1\ \mathsf{in}\ C_2'$, which by (4) and C-LETAND, is equivalent to $C \wedge \mathsf{let}\ \Gamma; z : \forall \bar{X}[D].T_1\ \mathsf{in}\ C_2'$, that is, $C \wedge \mathsf{let}\ \Gamma'\ \mathsf{in}\ C_2'$. By (9), this constraint is entailed by $C \wedge \exists \bar{X}.D$.

∘ *Case* HMD-SUB. The rule's conclusion is $C, \Gamma \vdash t : T'$. Its premises are $C, \Gamma \vdash t : T$ **(1)** and $C \Vdash T \leq T'$ **(2)**. Applying the induction hypothesis to (1) yields a constraint C' such that $C' \vdash t : T$ **(3)** and $C \Vdash \mathsf{let}\ \Gamma\ \mathsf{in}\ C'$ **(4)**. By SUB, (3) implies $C' \wedge T \leq T' \vdash t : T'$. There remains to establish $C \Vdash \mathsf{let}\ \Gamma\ \mathsf{in}\ (C' \wedge T \leq T')$, which follows from (4) and (2) by C-INAND*.

∘ *Case* HMD-EXISTS. The rule's conclusion is $\exists \bar{X}.C, \Gamma \vdash t : T$. Its premises are $C, \Gamma \vdash t : T$ **(1)** and $\bar{X} \# \mathit{ftv}(\Gamma, T)$ **(2)**. By hypothesis, we have $\exists \bar{X}.C \Vdash$

$\exists\Gamma$, which by Lemma 1.3.22 implies $C \Vdash \exists\Gamma$. Thus, the induction hypothesis applies to (1) and yields a constraint C' such that $C' \vdash t : T$ **(3)** and $C \Vdash \mathsf{let}\ \Gamma\ \mathsf{in}\ C'$ **(4)**. By EXISTS, (3) and (2) imply $\exists\bar{X}.C' \vdash t : T$. There remains to establish $\exists\bar{X}.C \Vdash \mathsf{let}\ \Gamma\ \mathsf{in}\ \exists\bar{X}.C'$. By congruence of entailment, (4) implies $\exists\bar{X}.C \Vdash \exists\bar{X}.\mathsf{let}\ \Gamma\ \mathsf{in}\ C'$. The result follows by (2) and C-INEX. □

The second theorem states that if, within PCB(X), t has type T under assumption C, then, within HM(X), t also has type T, under assumptions $\mathsf{let}\ \Gamma\ \mathsf{in}\ C$ and Γ. The idea is simple: the constraint C represents a combined assumption about the initial judgement's free type variables and free program identifiers. In HM(X), these two kinds of assumptions must be maintained separately. So, we split them into a pair of an environment Γ, which may be chosen arbitrarily, provided it satisfies $fpi(C) \subseteq dpi(\Gamma)$—that is, provided it defines all program variables of interest, and the residual constraint $\mathsf{let}\ \Gamma\ \mathsf{in}\ C$, which has no free program identifiers, thus represents an assumption about the new judgement's type variables only. Distinct choices of Γ give rise to distinct HM(X) judgements, which may be incomparable; this is related to the fact that ML-the-type-system does not have principal typings (Jim, 1995). Again, the hypothesis $fpi(\Gamma) = fpi(\mathsf{let}\ \Gamma\ \mathsf{in}\ C) = \varnothing$ is natural, since we wish Γ and $\mathsf{let}\ \Gamma\ \mathsf{in}\ C$ to appear in an HM(X) judgement.

1.5.4 THEOREM: Assume $fpi(\Gamma) = fpi(\mathsf{let}\ \Gamma\ \mathsf{in}\ C) = \varnothing$ and $C \not\equiv \mathsf{false}$. If $C \vdash t : T$ holds in PCB(X), then $\mathsf{let}\ \Gamma\ \mathsf{in}\ C, \Gamma \vdash t : T$ holds in HM(X). □

Proof: The proof is by structural induction on a derivation of $C \vdash t : T$. In each proof case, we adopt the notations of Figure 1-9.

By Lemma 1.3.50, the hypothesis $C \not\equiv \mathsf{false}$ is preserved whenever the induction hypothesis is invoked. It is explicitly used only in case VAR, where it guarantees that the identifier at hand is bound in Γ.

∘ *Case* VAR. The rule's conclusion is $C \vdash x : T$. Its premise is $C \Vdash x \preceq T$ **(1)**. By Lemma 1.3.38, (1) and the hypothesis $C \not\equiv \mathsf{false}$ imply $x \in fpi(C)$. Because $\mathsf{let}\ \Gamma\ \mathsf{in}\ C$ has no free program identifiers, this implies $x \in dpi(\Gamma)$, that is, the environment Γ must define x. Let $\Gamma(x) = \forall\bar{X}[D].T'$ **(2)**, where $\bar{X} \# ftv(\Gamma, T)$ **(3)**. By (2), HMD-VARINST, and HMD-SUB, we have $D \wedge T' \leq T, \Gamma \vdash x : T$. By (3) and HMD-EXISTS, this implies $\exists\bar{X}.(D \wedge T' \leq T), \Gamma \vdash x : T$ **(4)**. Now, by (3), the constraint $\exists\bar{X}.(D \wedge T' \leq T)$ may be written $\forall\bar{X}[D].T' \preceq T$ **(5)**. The hypothesis $fpi(\Gamma) = \varnothing$ implies $fpi(D) = \varnothing$ **(6)**. By (6), C-INID and C-IN*, $\mathsf{let}\ \Gamma\ \mathsf{in}\ x \preceq T$ entails (5). By (1) and by congruence of entailment, $\mathsf{let}\ \Gamma\ \mathsf{in}\ C$ entails (5) as well. Thus, by Lemma 1.4.2, the judgement (4) implies $\mathsf{let}\ \Gamma\ \mathsf{in}\ C, \Gamma \vdash x : T$.

∘ *Case* ABS. The rule's conclusion is $\mathsf{let}\ z : T\ \mathsf{in}\ C \vdash \lambda z.t : T \to T'$. Its premise is $C \vdash t : T'$ **(1)**. Let Γ' stand for $\Gamma; z : T$. Applying the induction

hypothesis to (1) yields $\mathsf{let}\ \Gamma'\ \mathsf{in}\ C, \Gamma' \vdash t : T'$. By HMD-ABS, this implies $\mathsf{let}\ \Gamma'\ \mathsf{in}\ C, \Gamma \vdash \lambda z.t : T \to T'$.

∘ *Case* APP. The rule's conclusion is $C_1 \wedge C_2 \vdash t_1\ t_2 : T'$. Its premises are $C_1 \vdash t_1 : T \to T'$ and $C_2 \vdash t_2 : T$. Applying the induction hypothesis yields respectively $\mathsf{let}\ \Gamma\ \mathsf{in}\ C_1, \Gamma \vdash t_1 : T \to T'$ and $\mathsf{let}\ \Gamma\ \mathsf{in}\ C_2, \Gamma \vdash t_2 : T$, which by Lemma 1.4.2 and HMD-APP imply $\mathsf{let}\ \Gamma\ \mathsf{in}\ (C_1 \wedge C_2), \Gamma \vdash t_1\ t_2 : T'$.

∘ *Case* LET. The rule's conclusion is $\mathsf{let}\ z : \forall\mathcal{V}[C_1].T_1\ \mathsf{in}\ C_2 \vdash \mathtt{let}\ z = t_1\ \mathtt{in}\ t_2 : T_2$. Its premises are $C_1 \vdash t_1 : T_1$ **(1)** and $C_2 \vdash t_2 : T_2$ **(2)**. Let $\bar{X}$ stand for $ftv(C_1, T_1)$. We may require, *w.l.o.g.*, $\bar{X} \# ftv(\Gamma, C_2)$ **(3)**. By hypothesis, we have $fpi(\Gamma) = \varnothing$ **(4)**. We also have $fpi(\mathsf{let}\ \Gamma; z : \forall\mathcal{V}[C_1].T_1\ \mathsf{in}\ C_2) = \varnothing$, which implies $fpi(\mathsf{let}\ \Gamma\ \mathsf{in}\ C_1) = \varnothing$. Thus, the induction hypothesis applies to (1) and yields $\mathsf{let}\ \Gamma\ \mathsf{in}\ C_1, \Gamma \vdash t_1 : T_1$ **(5)**. Now, let σ stand for $\forall\bar{X}[\mathsf{let}\ \Gamma\ \mathsf{in}\ C_1].T_1$ and Γ' stand for $\Gamma; z : \sigma$. We have $fpi(\Gamma') = fpi(\mathsf{let}\ \Gamma'\ \mathsf{in}\ C_2) = \varnothing$. Thus, the induction hypothesis applies to (2) and yields $\mathsf{let}\ \Gamma'\ \mathsf{in}\ C_2, \Gamma' \vdash t_2 : T_2$ **(6)**. Let us now weaken (5) and (6) so as to make them suitable premises for HMD-LETGEN. Applying Lemma 1.4.2 to (5) yields $(\mathsf{let}\ \Gamma'\ \mathsf{in}\ C_2) \wedge (\mathsf{let}\ \Gamma\ \mathsf{in}\ C_1), \Gamma \vdash t_1 : T_1$ **(7)**. Applying Lemma 1.4.2 to (6) yields $(\mathsf{let}\ \Gamma'\ \mathsf{in}\ C_2) \wedge \exists\bar{X}.(\mathsf{let}\ \Gamma\ \mathsf{in}\ C_1), \Gamma' \vdash t_2 : T_2$ **(8)**. Last, (3) implies $\bar{X} \# ftv(\Gamma, \mathsf{let}\ \Gamma'\ \mathsf{in}\ C_2)$ **(9)**. Applying HMD-LETGEN to (7), (9) and (8), we obtain $(\mathsf{let}\ \Gamma'\ \mathsf{in}\ C_2) \wedge \exists\bar{X}.(\mathsf{let}\ \Gamma\ \mathsf{in}\ C_1), \Gamma \vdash \mathtt{let}\ z = t_1\ \mathtt{in}\ t_2 : T_2$ **(10)**. Now, by (4), (3), and C-LETDUP, $\mathsf{let}\ \Gamma'\ \mathsf{in}\ C_2$ is equivalent to $\mathsf{let}\ \Gamma; z : \forall\bar{X}[C_1].T_1\ \mathsf{in}\ C_2$. Using this fact, as well as (3), C-INEX, and C-INAND, we find that the constraint $(\mathsf{let}\ \Gamma'\ \mathsf{in}\ C_2) \wedge \exists\bar{X}.(\mathsf{let}\ \Gamma\ \mathsf{in}\ C_1)$ is equivalent to $\mathsf{let}\ \Gamma\ \mathsf{in}\ (\mathsf{let}\ z : \forall\bar{X}[C_1].T_1\ \mathsf{in}\ C_2 \wedge \exists\bar{X}.C_1)$, which by definition of the let form, is itself equivalent to $\mathsf{let}\ \Gamma; z : \forall\bar{X}[C_1].T_1\ \mathsf{in}\ C_2$. Last, by definition of $\bar{X}$, this constraint is $\mathsf{let}\ \Gamma; z : \forall\mathcal{V}[C_1].T_1\ \mathsf{in}\ C_2$. Thus, (10) is the goal.

∘ *Case* SUB. The rule's conclusion is $C \wedge T \leq T' \vdash t : T'$. Its premise is $C \vdash t : T$ **(1)**. Applying the induction hypothesis to (1) yields $\mathsf{let}\ \Gamma\ \mathsf{in}\ C, \Gamma \vdash t : T$ **(2)**. By Lemma 1.4.2 and HMD-SUB, (2) implies $(\mathsf{let}\ \Gamma\ \mathsf{in}\ C) \wedge T \leq T', \Gamma \vdash t : T'$, which by C-INAND* may be written $\mathsf{let}\ \Gamma\ \mathsf{in}\ (C \wedge T \leq T'), \Gamma \vdash t : T'$.

∘ *Case* EXISTS. The rule's conclusion is $\exists\bar{X}.C \vdash t : T$. Its premises are $C \vdash t : T$ **(1)** and $\bar{X} \# ftv(T)$ **(2)**. We may further require, *w.l.o.g.*, $\bar{X} \# ftv(\Gamma)$ **(3)**. Applying the induction hypothesis to (1) yields $\mathsf{let}\ \Gamma\ \mathsf{in}\ C, \Gamma \vdash t : T$ **(4)**. Applying HMD-EXISTS to (2), (3), and (4), we find $\exists\bar{X}.\mathsf{let}\ \Gamma\ \mathsf{in}\ C, \Gamma \vdash t : T$, which, by (3) and C-INEX, may be written $\mathsf{let}\ \Gamma\ \mathsf{in}\ \exists\bar{X}.C, \Gamma \vdash t : T$. □

As a corollary, we find that, for closed programs, the type systems HM(X) and PCB(X) coincide. In particular, a program is well-typed with respect to one if and only if it is well-typed with respect to the other. This supports the view that PCB(X) is an alternate formulation of HM(X).

1.5.5 THEOREM: Assume $fpi(C) = \varnothing$ and $C \not\equiv \mathsf{false}$. Then, $C, \varnothing \vdash t : T$ holds in HM(X) if and only if $C \vdash t : T$ holds in PCB(X). □

Proof: First, let $C, \varnothing \vdash t : T$ hold in HM(X). The assertion $C \Vdash \exists\varnothing$ is a tautology, so, by Theorem 1.5.3, there exists a constraint C' such that $C' \vdash t : T$ and $C \Vdash \mathsf{let}\ \varnothing\ \mathsf{in}\ C'$, that is, $C \Vdash C'$. By Lemma 1.5.2, this implies $C \vdash t : T$. The converse implication is a special case of Theorem 1.5.4 where Γ is $\varnothing$. □

1.6 Constraint generation

We now explain how to reduce type inference problems for PCB(X) to constraint solving problems. A type inference problem consists of an expression t and a type T of kind $\star$. The problem is to determine whether t is well-typed with type T, that is, whether there exists a satisfiable constraint C such that $C \vdash t : T$ holds. A constraint solving problem consists of a constraint C. The problem is to determine whether C is satisfiable. To reduce a type inference problem (t, T) to a constraint solving problem, we must produce a constraint C that is both *sufficient* and *necessary* for $C \vdash t : T$ to hold. Below, we explain how to compute such a constraint, which we write $[\![t : T]\!]$. We check that it is indeed *sufficient* by proving $[\![t : T]\!] \vdash t : T$. That is, the constraint $[\![t : T]\!]$ is specific enough to guarantee that t has type T. We say that constraint generation is *sound*. We check that it is indeed *necessary* by proving that, for every constraint C, the validity of $C \vdash t : T$ implies $C \Vdash [\![t : T]\!]$. That is, every constraint that guarantees that t has type T is at least as specific as $[\![t : T]\!]$. We say that constraint generation is *complete*. Together, these properties mean that $[\![t : T]\!]$ is the *least specific* constraint that guarantees that t has type T.

We now see how to reduce a type inference problem to a constraint solving problem. Indeed, if there exists a satisfiable constraint C such that $C \vdash t : T$ holds, then, by the completeness property, $C \Vdash [\![t : T]\!]$ holds, so $[\![t : T]\!]$ is satisfiable. Conversely, by the soundness property, if $[\![t : T]\!]$ is satisfiable, then we have a satisfiable constraint C such that $C \vdash t : T$ holds. In other words, t is well-typed with type T if and only if $[\![t : T]\!]$ is satisfiable.

The reader may be somewhat puzzled by the fact that our formulation of the type inference problem requires an appropriate type T to be known in advance, whereas the very purpose of type inference seems to consist in *discovering* the type of t! In other words, we have made T an *input* of the constraint generation algorithm, instead of an *output*. Fortunately, this causes no loss of generality, because it is possible to let T be a type variable X. Then, the constraint produced by the algorithm will contain information about X. This is the point of the following exercise.

1.6.1 EXERCISE [RECOMMENDED, ★]: Let X be an arbitrary type variable. Show

$$
\begin{array}{rcl}
[\![\mathtt{x} : \mathtt{T}]\!] & = & \mathtt{x} \preceq \mathtt{T} \\
[\![\lambda \mathtt{z}.\mathtt{t} : \mathtt{T}]\!] & = & \exists \mathtt{X}_1 \mathtt{X}_2.(\mathsf{let}\ \mathtt{z} : \mathtt{X}_1\ \mathsf{in}\ [\![\mathtt{t} : \mathtt{X}_2]\!] \wedge \mathtt{X}_1 \to \mathtt{X}_2 \leq \mathtt{T}) \\
[\![\mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}]\!] & = & \exists \mathtt{X}_2.([\![\mathtt{t}_1 : \mathtt{X}_2 \to \mathtt{T}]\!] \wedge [\![\mathtt{t}_2 : \mathtt{X}_2]\!]) \\
[\![\mathtt{let}\ \mathtt{z} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}]\!] & = & \mathsf{let}\ \mathtt{z} : \forall \mathtt{X}[[\![\mathtt{t}_1 : \mathtt{X}]\!]].\mathtt{X}\ \mathsf{in}\ [\![\mathtt{t}_2 : \mathtt{T}]\!]
\end{array}
$$

Figure 1-10: Constraint generation

that, if there exist a satisfiable constraint C and a type $\mathtt{T}$ such that $\mathrm{C} \vdash \mathtt{t} : \mathtt{T}$ holds, then there exists a satisfiable constraint C' such that $\mathrm{C}' \vdash \mathtt{t} : \mathtt{X}$ holds. Conclude that a closed term $\mathtt{t}$ is well-typed if and only if $\exists \mathtt{X}.[\![\mathtt{t} : \mathtt{X}]\!] \equiv \mathsf{true}$ holds. □

This shows that providing $\mathtt{T}$ as an input to the constraint generation procedure is not essential. We adopt this style because it is convenient. A somewhat naïve alternative would be to provide $\mathtt{t}$ only, and to have the procedure return both a constraint C and a type $\mathtt{T}$ (Sulzmann, Müller, and Zenger, 1999). It turns out that this does not quite work, because C and $\mathtt{T}$ may mention "fresh" variables, which we must be able to quantify over, if we are to avoid an informal treatment of "freshness". Thus, the true alternative is to provide $\mathtt{t}$ only and to have the procedure return a *type scheme* σ (Bonniot, 2002).

The existence of a sound and complete constraint generation procedure is the analogue of the existence of *principal type schemes* in classic presentations of ML-the-type-system (Damas and Milner, 1982). Indeed, a principal type scheme is least specific in the sense that all valid types are substitution instances of it. Here, the constraint $[\![\mathtt{t} : \mathtt{T}]\!]$ is least specific in the sense that all valid constraints entail it. Earlier, we have established a connection between constraint entailment and refinement of type substitutions, in the specific case of equality constraints interpreted over a free algebra of finite types; see Lemma 1.3.59.

The constraint $[\![\mathtt{t} : \mathtt{T}]\!]$ is defined in Figure 1-10 by induction on the structure of the expression $\mathtt{t}$. We refer to these defining equations as the *constraint generation rules*. The definition is quite terse. It is perhaps even simpler than the declarative specification of PCB(X) given in Figure 1-9; yet, we prove below that the two are equivalent.

Before explaining the definition, we state the requirements that bear on the type variables $\mathtt{X}_1$, $\mathtt{X}_2$, and $\mathtt{X}$, which appear bound in the right-hand sides of the second, third, and fourth equations. These type variables must have kind $\star$. They must be chosen distinct (that is, $\mathtt{X}_1 \neq \mathtt{X}_2$ in the second equation) and fresh for the objects that appear on the left-hand side—that is, *the*

type variables that appear bound in an equation's right-hand side must not occur free in the term and type that appear in the equation's left-hand side. Provided this restriction is obeyed, different choices of $\mathtt{X}_1$, $\mathtt{X}_2$, and $\mathtt{X}$ lead to α-equivalent constraints—that is, to the same constraint, since we identify objects up to α-conversion—which guarantees that the above equations make sense. We remark that, since expressions do not have free type variables, the freshness requirement may be simplified to: type variables that appear bound in an equation's right-hand side must not appear free in $\mathtt{T}$. However, this simplification is rendered invalid by the introduction of open type annotations within expressions (page 135). Please note that we are able to state a *precise* (as opposed to informal) freshness requirement. This is made possible by the fact that $[\![\mathtt{t} : \mathtt{T}]\!]$ has no free type variables other than those of $\mathtt{T}$, which in turn depends on our explicit use of quantification.

Let us now review the four equations. The first one simply mirrors VAR. The second one requires $\mathtt{t}$ to have type $\mathtt{X}_2$ under the hypothesis that $\mathtt{z}$ has type $\mathtt{X}_1$, and forms the arrow type $\mathtt{X}_1 \to \mathtt{X}_2$; this corresponds to ABS. Here, $\mathtt{X}_1$ and $\mathtt{X}_2$ must be fresh type variables, because we cannot in general guess the expected types of $\mathtt{z}$ and $\mathtt{t}$. The expected type $\mathtt{T}$ is required to be a supertype of $\mathtt{X}_1 \to \mathtt{X}_2$; this corresponds to SUB. We must bind the fresh type variables $\mathtt{X}_1$ and $\mathtt{X}_2$, so as to guarantee that the generated constraint is unique up to α-conversion. Furthermore, we must bind them *existentially*, because we intend the constraint solver to choose some appropriate value for them. This is justified by EXISTS. The third equation uses the fresh type variable $\mathtt{X}_2$ to stand for the unknown type of $\mathtt{t}_2$. The subexpression $\mathtt{t}_1$ is expected to have type $\mathtt{X}_2 \to \mathtt{T}$. This corresponds to APP. The fourth equation, which corresponds to LET, is most interesting. It summons a fresh type variable $\mathtt{X}$ and produces $[\![\mathtt{t}_1 : \mathtt{X}]\!]$. This constraint, whose sole free type variable is $\mathtt{X}$, is the *least specific* constraint that must be imposed on $\mathtt{X}$ so as to make it a valid type for $\mathtt{t}_1$. As a result, the type scheme $\forall \mathtt{X}[[\![\mathtt{t}_1 : \mathtt{X}]\!]].\mathtt{X}$, abbreviated σ in the following, is a *principal* type scheme for $\mathtt{t}_1$. There remains to place $[\![\mathtt{t}_2 : \mathtt{T}]\!]$ inside the context $\mathsf{let}\ \mathtt{z} : \sigma\ \mathsf{in}\ []$. Indeed, when placed inside this context, an instantiation constraint of the form $\mathtt{z} \preceq \mathtt{T}'$ acquires the meaning $\sigma \preceq \mathtt{T}'$, which by definition of σ and by Lemma 1.6.5 (see below) is equivalent to $[\![\mathtt{t}_1 : \mathtt{T}']\!]$. Thus, the constraint produced by the fourth equation simulates a textual expansion of the `let` construct, where every occurrence of $\mathtt{z}$ would be replaced with $\mathtt{t}_1$. Thanks to type scheme introduction and instantiation constraints, however, this effect is achieved without duplication of source code or constraints. In other words, constraint generation has linear time and space complexity.

1.6.2 EXERCISE [★, ↛]: Define the *size* of an expression, of a type, and of a constraint, viewed as abstract syntax trees. Check that the size of $[\![\mathtt{t} : \mathtt{T}]\!]$ is linear

in the sum of the sizes of t and T. □

We now establish several properties of constraint generation. We begin with soundness, whose statement was explained above, and whose proof is straightforward.

1.6.3 THEOREM [SOUNDNESS]: $[\![t : T]\!] \vdash t : T$. □

Proof: By induction on the structure of t.

∘ *Case* x. The goal $x \preceq T \vdash x : T$ follows from VAR.

∘ *Case* $\lambda z.t$. By the induction hypothesis, we have $[\![t : X_2]\!] \vdash t : X_2$. By ABS, this implies $\mathsf{let}\ z : X_1\ \mathsf{in}\ [\![t : X_2]\!] \vdash \lambda z.t : X_1 \to X_2$. By SUB, this implies $\mathsf{let}\ z : X_1\ \mathsf{in}\ [\![t : X_2]\!] \wedge X_1 \to X_2 \leq T \vdash \lambda z.t : T$. Lastly, because $X_1 X_2 \# \mathit{ftv}(T)$ holds, EXISTS applies and yields $[\![\lambda z.t : T]\!] \vdash \lambda z.t : T$.

∘ *Case* $t_1\ t_2$. By the induction hypothesis, we have $[\![t_1 : X_2 \to T]\!] \vdash t_1 : X_2 \to T$ and $[\![t_2 : X_2]\!] \vdash t_2 : X_2$. By APP, this implies $[\![t_1 : X_2 \to T]\!] \wedge [\![t_2 : X_2]\!] \vdash t_1\ t_2 : T$. Because $X_2 \notin \mathit{ftv}(T)$ holds, EXISTS applies and yields $[\![t_1\ t_2 : T]\!] \vdash t_1\ t_2 : T$.

∘ *Case* $\mathtt{let}\ z = t_1\ \mathtt{in}\ t_2$. By the induction hypothesis, we have $[\![t_1 : X]\!] \vdash t_1 : X$ and $[\![t_2 : T]\!] \vdash t_2 : T$. By LET, these imply $\mathsf{let}\ z : \forall \mathcal{V}[[\![t_1 : X]\!]].X\ \mathsf{in}\ [\![t_2 : T]\!] \vdash \mathtt{let}\ z = t_1\ \mathtt{in}\ t_2 : T$. Because $\mathit{ftv}([\![t_1 : X]\!])$ is X, the universal quantification on $\mathcal{V}$ really bears on X alone. We have proved $[\![\mathtt{let}\ z = t_1\ \mathtt{in}\ t_2 : T]\!] \vdash \mathtt{let}\ z = t_1\ \mathtt{in}\ t_2 : T$. □

The following lemmas are used in the proof of the completeness property and in a number of other occasions. The first two state that $[\![t : T]\!]$ is *covariant* with respect to T. Roughly speaking, this means that enough subtyping constraints are generated to achieve completeness with respect to SUB.

1.6.4 LEMMA: $[\![t : T]\!] \wedge T \leq T' \Vdash [\![t : T']\!]$. □

Proof: By induction on the structure of t.

∘ *Case* x. By Lemma 1.3.25, for every type scheme σ, $\sigma \preceq T \wedge T \leq T'$ entails $\sigma \preceq T'$. The goal $x \preceq T \wedge T \leq T' \Vdash x \preceq T'$ follows from Lemma 1.3.25 by definition of entailment and by CM-INSTANCE.

∘ *Case* $\lambda z.t$. Let $X_1 X_2 \# \mathit{ftv}(T, T')$ **(1)**. Then, we have

$$\begin{array}{rll} & \exists X_1 X_2.(\mathsf{let}\ z : X_1\ \mathsf{in}\ [\![t : X_2]\!] \wedge X_1 \to X_2 \leq T) \wedge T \leq T' & \\ \equiv & \exists X_1 X_2.(\mathsf{let}\ z : X_1\ \mathsf{in}\ [\![t : X_2]\!] \wedge X_1 \to X_2 \leq T \wedge T \leq T') & \textbf{(2)} \\ \Vdash & \exists X_1 X_2.(\mathsf{let}\ z : X_1\ \mathsf{in}\ [\![t : X_2]\!] \wedge X_1 \to X_2 \leq T') & \textbf{(3)} \end{array}$$

where (2) is by (1) and C-EXAND; (3) is by transitivity of subtyping.

∘ *Case* $\mathtt{t}_1\ \mathtt{t}_2$. Let $\mathtt{X}_2 \notin \mathit{ftv}(\mathtt{T}')$ **(1)**. Then, we have

$$\begin{array}{lll} & \exists \mathtt{X}_2.([\![\mathtt{t}_1 : \mathtt{X}_2 \to \mathtt{T}]\!] \wedge [\![\mathtt{t}_2 : \mathtt{X}_2]\!]) \wedge \mathtt{T} \leq \mathtt{T}' & \\ \equiv & \exists \mathtt{X}_2.([\![\mathtt{t}_1 : \mathtt{X}_2 \to \mathtt{T}]\!] \wedge \mathtt{T} \leq \mathtt{T}' \wedge [\![\mathtt{t}_2 : \mathtt{X}_2]\!]) & \textbf{(2)} \\ \equiv & \exists \mathtt{X}_2.([\![\mathtt{t}_1 : \mathtt{X}_2 \to \mathtt{T}]\!] \wedge \mathtt{X}_2 \to \mathtt{T} \leq \mathtt{X}_2 \to \mathtt{T}' \wedge [\![\mathtt{t}_2 : \mathtt{X}_2]\!]) & \textbf{(3)} \\ \Vdash & \exists \mathtt{X}_2.([\![\mathtt{t}_1 : \mathtt{X}_2 \to \mathtt{T}']\!] \wedge [\![\mathtt{t}_2 : \mathtt{X}_2]\!]) & \textbf{(4)} \end{array}$$

where (2) is by (1) and C-ExAnd; (3) is by C-Arrow; (4) is by the induction hypothesis.

∘ *Case* $\mathtt{let\ z} = \mathtt{t}_1\ \mathtt{in\ t}_2$. We have

$$\begin{array}{lll} & (\mathsf{let}\ \mathtt{z} : \forall \mathtt{X}[[\![\mathtt{t}_1 : \mathtt{X}]\!]].\mathtt{X}\ \mathsf{in}\ [\![\mathtt{t}_2 : \mathtt{T}]\!]) \wedge \mathtt{T} \leq \mathtt{T}' & \\ \equiv & \mathsf{let}\ \mathtt{z} : \forall \mathtt{X}[[\![\mathtt{t}_1 : \mathtt{X}]\!]].\mathtt{X}\ \mathsf{in}\ ([\![\mathtt{t}_2 : \mathtt{T}]\!] \wedge \mathtt{T} \leq \mathtt{T}') & \textbf{(1)} \\ \Vdash & \mathsf{let}\ \mathtt{z} : \forall \mathtt{X}[[\![\mathtt{t}_1 : \mathtt{X}]\!]].\mathtt{X}\ \mathsf{in}\ [\![\mathtt{t}_2 : \mathtt{T}]\!]' & \textbf{(2)} \end{array}$$

where (1) is by C-InAnd*; (2) is by the induction hypothesis. □

1.6.5 Lemma: $\mathtt{X} \notin \mathit{ftv}(\mathtt{T})$ implies $\exists \mathtt{X}.([\![\mathtt{t} : \mathtt{X}]\!] \wedge \mathtt{X} \leq \mathtt{T}) \equiv [\![\mathtt{t} : \mathtt{T}]\!]$. □

Proof: $\mathtt{X} \notin \mathit{ftv}(\mathtt{T})$ yields $\mathtt{X} \notin \mathit{ftv}([\![\mathtt{t} : \mathtt{T}]\!])$ **(1)**. By Lemma 1.6.4 and by congruence of entailment, $\exists \mathtt{X}.([\![\mathtt{t} : \mathtt{X}]\!] \wedge \mathtt{X} \leq \mathtt{T})$ entails $\exists \mathtt{X}.[\![\mathtt{t} : \mathtt{T}]\!]$, which by (1) and C-Ex* is equivalent to $[\![\mathtt{t} : \mathtt{T}]\!]$. Conversely, by (1) and C-NameEq, $[\![\mathtt{t} : \mathtt{T}]\!]$ is equivalent to $\exists \mathtt{X}.([\![\mathtt{t} : \mathtt{X}]\!] \wedge \mathtt{X} = \mathtt{T})$, which entails $\exists \mathtt{X}.([\![\mathtt{t} : \mathtt{X}]\!] \wedge \mathtt{X} \leq \mathtt{T})$. □

The next lemma gives a simplified version of the second constraint generation rule, in the specific case where the expected type is an arrow type. Then, fresh type variables need not be generated; one may directly use the arrow's domain and codomain instead.

1.6.6 Lemma: $[\![\lambda \mathtt{z}.\mathtt{t} : \mathtt{T}_1 \to \mathtt{T}_2]\!]$ is equivalent to $\mathsf{let}\ \mathtt{z} : \mathtt{T}_1\ \mathsf{in}\ [\![\mathtt{t} : \mathtt{T}_2]\!]$. □

Proof: Let $\mathtt{X}_1 \neq \mathtt{X}_2$ and $\mathtt{X}_1\mathtt{X}_2 \# \mathit{ftv}(\mathtt{T}_1, \mathtt{T}_2)$. We have

$$\begin{array}{lll} & [\![\lambda \mathtt{z}.\mathtt{t} : \mathtt{T}_1 \to \mathtt{T}_2]\!] & \\ \equiv & \exists \mathtt{X}_1\mathtt{X}_2.(\mathsf{let}\ \mathtt{z} : \mathtt{X}_1\ \mathsf{in}\ [\![\mathtt{t} : \mathtt{X}_2]\!] \wedge \mathtt{X}_1 \to \mathtt{X}_2 \leq \mathtt{T}_1 \to \mathtt{T}_2) & \textbf{(1)} \\ \equiv & \exists \mathtt{X}_1.(\mathtt{T}_1 \leq \mathtt{X}_1 \wedge \mathsf{let}\ \mathtt{z} : \mathtt{X}_1\ \mathsf{in}\ \exists \mathtt{X}_2.([\![\mathtt{t} : \mathtt{X}_2]\!] \wedge \mathtt{X}_2 \leq \mathtt{T}_2)) & \textbf{(2)} \\ \equiv & \exists \mathtt{X}_1.(\mathtt{T}_1 \leq \mathtt{X}_1 \wedge \mathsf{let}\ \mathtt{z} : \mathtt{X}_1\ \mathsf{in}\ [\![\mathtt{t} : \mathtt{T}_2]\!]) & \textbf{(3)} \\ \equiv & \mathsf{let}\ \mathtt{z} : \mathtt{T}_1\ \mathsf{in}\ [\![\mathtt{t} : \mathtt{T}_2]\!] & \textbf{(4)} \end{array}$$

where (1) is by definition of constraint generation; (2) follows from C-Arrow, C-InAnd*, C-ExAnd and C-InEx; (3) is by Lemma 1.6.5; (4) is by C-LetSub. □

We conclude with the completeness property.

1.6.7 THEOREM [COMPLETENESS]: if $C \vdash t : T$, then $C \Vdash [\![t : T]\!]$. □

Proof: By induction on the derivation of $C \vdash t : T$.

∘ *Case* VAR. The rule's conclusion is $C \vdash x : T$. Its premise is $C \Vdash x \preceq T$, which is also the goal.

∘ *Case* ABS. The rule's conclusion is $\mathsf{let}\ z : T\ \mathsf{in}\ C \vdash \lambda z.t : T \to T'$. Its premise is $C \vdash t : T'$. By the induction hypothesis, we have $C \Vdash [\![t : T']\!]$. By congruence of entailment, this implies $\mathsf{let}\ z : T\ \mathsf{in}\ C \Vdash \mathsf{let}\ z : T\ \mathsf{in}\ [\![t : T']\!]$, which, by Lemma 1.6.6, may be written $\mathsf{let}\ z : T\ \mathsf{in}\ C \Vdash [\![\lambda z.t : T \to T']\!]$.

∘ *Case* APP. The rule's conclusion is $C_1 \wedge C_2 \vdash t_1\ t_2 : T'$. Its premises are $C_1 \vdash t_1 : T \to T'$ and $C_2 \vdash t_2 : T$. By the induction hypothesis, we have $C_1 \Vdash [\![t_1 : T \to T']\!]$ and $C_2 \Vdash [\![t_2 : T]\!]$. Thus, $C_1 \wedge C_2$ entails $[\![t_1 : T \to T']\!] \wedge [\![t_2 : T]\!]$, which, by C-NAMEEQ, may be written $\exists X_2.(X_2 = T \wedge [\![t_1 : X_2 \to T']\!] \wedge [\![t_2 : X_2]\!])$, where $X_2 \notin ftv(T, T')$. Forgetting about the equation $X_2 = T$, we find that $C_1 \wedge C_2$ entails $\exists X_2.([\![t_1 : X_2 \to T']\!] \wedge [\![t_2 : X_2]\!])$, which is precisely $[\![t_1\ t_2 : T']\!]$.

∘ *Case* LET. The rule's conclusion is $\mathsf{let}\ z : \forall\mathcal{V}[C_1].T_1\ \mathsf{in}\ C_2 \vdash \mathtt{let}\ z = t_1\ \mathtt{in}\ t_2 : T_2$. Its premises are $C_1 \vdash t_1 : T_1$ and $C_2 \vdash t_2 : T_2$. By the induction hypothesis, we have $C_1 \Vdash [\![t_1 : T_1]\!]$ and $C_2 \Vdash [\![t_2 : T_2]\!]$, which implies $\mathsf{let}\ z : \forall\mathcal{V}[C_1].T_1\ \mathsf{in}\ C_2 \Vdash \mathsf{let}\ z : \forall\mathcal{V}[[\![t_1 : T_1]\!]].T_1\ \mathsf{in}\ [\![t_2 : T_2]\!]$ **(1)**.

Now, let us establish $\mathsf{true} \Vdash \forall X[[\![t_1 : X]\!]].X \preceq \forall\mathcal{V}[[\![t_1 : T_1]\!]].T_1$ **(2)**. By definition, this requires proving $\exists\bar{X}_1.([\![t_1 : T_1]\!] \wedge T_1 \leq Z) \Vdash \exists X.([\![t_1 : X]\!] \wedge X \leq Z)$ **(3)**, where $\bar{X}_1 = ftv(T_1)$ and $Z \notin X\bar{X}_1$ **(4)**. By Lemma 1.6.4, (4), and C-EX*, the left-hand side of (3) entails $[\![t_1 : Z]\!]$. By (4) and Lemma 1.6.5, the right-hand side of (3) is $[\![t_1 : Z]\!]$. Thus, (3) holds, and so does (2).

By (2) and Lemma 1.3.31, we have $\mathsf{let}\ z : \forall\mathcal{V}[[\![t_1 : T_1]\!]].T_1\ \mathsf{in}\ [\![t_2 : T_2]\!] \Vdash \mathsf{let}\ z : \forall X[[\![t_1 : X]\!]].X\ \mathsf{in}\ [\![t_2 : T_2]\!]$ **(5)**. By transitivity of entailment, (1) and (5) yield $\mathsf{let}\ z : \forall\mathcal{V}[C_1].T_1\ \mathsf{in}\ C_2 \Vdash [\![\mathtt{let}\ z = t_1\ \mathtt{in}\ t_2 : T_2]\!]$.

∘ *Case* SUB. The rule's conclusion is $C \wedge T \leq T' \vdash t : T'$. Its premise is $C \vdash t : T$. By the induction hypothesis, we have $C \Vdash [\![t : T]\!]$, which implies $C \wedge T \leq T' \Vdash [\![t : T]\!] \wedge T \leq T'$. By lemma 1.6.4 and by transitivity of entailment, we obtain $C \wedge T \leq T' \Vdash [\![t : T']\!]$.

∘ *Case* EXISTS. The rule's conclusion is $\exists\bar{X}.C \vdash t : T$. Its premises are $C \vdash t : T$ and $\bar{X}\ \#\ ftv(T)$ **(1)**. By the induction hypothesis, we have $C \Vdash [\![t : T]\!]$. By congruence of entailment, this implies $\exists\bar{X}.C \Vdash \exists\bar{X}.[\![t : T]\!]$ **(2)**. Furthermore, (1) implies $\bar{X}\ \#\ ftv([\![t : T]\!])$ **(3)**. By (3) and C-EX*, (2) may be written $\exists\bar{X}.C \Vdash [\![t : T]\!]$. □

1.7 Type soundness

We are now ready to establish type soundness for our type system. The statement that we wish to prove is sometimes known as *Milner's slogan*: *well-typed programs do not go wrong* (Milner, 1978). Below, we define well-typedness in terms of our constraint generation rules, for the sake of convenience, and establish type soundness with respect to that particular definition. Theorems 1.4.10, 1.5.4, and 1.6.7 imply that type soundness also holds when well-typedness is defined with respect to the typing judgements of DM, HM(X), or PCB(X). We establish type soundness by following Wright and Felleisen's so-called *syntactic approach* (1994b). The approach consists in isolating two independent properties. *Subject reduction*, whose exact statement will be given below, implies that well-typedness is preserved by reduction. *Progress* states that no stuck configuration is well-typed. It is immediate to check that, if both properties hold, then no well-typed program can reduce to a stuck configuration. Subject reduction itself depends on a key lemma, usually known as a (term) *substitution lemma*. Here are two versions of this lemma: the former is stated in terms of PCB(X) judgements, while the latter is stated in terms of the constraint generation rules.

1.7.1 LEMMA [SUBSTITUTION]: $C \vdash \mathtt{t} : \mathtt{T}$ and $C_0 \vdash \mathtt{t}_0 : \mathtt{T}_0$ imply $\mathsf{let}\ \mathtt{z}_0 : \forall \bar{X}_0[C_0].\mathtt{T}_0\ \mathsf{in}\ C \vdash [\mathtt{z}_0 \mapsto \mathtt{t}_0]\mathtt{t} : \mathtt{T}$. □

Proof: The proof is by structural induction on the derivation of $C \vdash \mathtt{t} : \mathtt{T}$. In each proof case, we adopt the notations of Figure 1-9. We write σ_0 for $\forall \bar{X}_0[C_0].\mathtt{T}_0$. We refer to the hypothesis $C_0 \vdash \mathtt{t}_0 : \mathtt{T}_0$ as **(1)**. We assume, *w.l.o.g.*, $\bar{X}_0 \# \mathit{ftv}(C, \mathtt{T})$ **(2)** and $\mathtt{z}_0 \notin \mathit{fpi}(\sigma_0)$ **(3)**.

∘ *Case* VAR. The rule's conclusion is $C \vdash \mathtt{x} : \mathtt{T}$ **(4)**. Its premise is $C \Vdash \mathtt{x} \preceq \mathtt{T}$ **(5)**. Two subcases arise.

Subcase x is $\mathtt{z}_0$. Applying SUB to (1) yields $C_0 \wedge \mathtt{T}_0 \leq \mathtt{T} \vdash \mathtt{t}_0 : \mathtt{T}$. y (2) and EXISTS, this implies $\exists \bar{X}_0.(C_0 \wedge \mathtt{T}_0 \leq \mathtt{T}) \vdash \mathtt{t}_0 : \mathtt{T}$ **(6)**. Furthermore, by (2) again, the constraint $\exists \bar{X}_0.(C_0 \wedge \mathtt{T}_0 \leq \mathtt{T})$ is $\sigma_0 \preceq \mathtt{T}$, which is equivalent to $\mathsf{let}\ \mathtt{z}_0 : \sigma_0\ \mathsf{in}\ \mathtt{z}_0 \preceq \mathtt{T}$. As a result, (6) may be written $\mathsf{let}\ \mathtt{z}_0 : \sigma_0\ \mathsf{in}\ \mathtt{x} \preceq \mathtt{T} \vdash [\mathtt{z}_0 \mapsto \mathtt{t}_0]\mathtt{x} : \mathtt{T}$ **(7)**.

Subcase x isn't $\mathtt{z}_0$. Then, $[\mathtt{z}_0 \mapsto \mathtt{t}_0]\mathtt{x}$ is x. Thus, VAR yields $\exists \sigma_0 \wedge \mathtt{x} \preceq \mathtt{T} \vdash [\mathtt{z}_0 \mapsto \mathtt{t}_0]\mathtt{x} : \mathtt{T}$. By C-IN*, this may be read $\mathsf{let}\ \mathtt{z}_0 : \sigma_0\ \mathsf{in}\ \mathtt{x} \preceq \mathtt{T} \vdash [\mathtt{z}_0 \mapsto \mathtt{t}_0]\mathtt{x} : \mathtt{T}$, that is, again (7).

In either subcase, by (5), by congruence of entailment, and by Lemma 1.5.2, (7) implies $\mathsf{let}\ \mathtt{z}_0 : \sigma_0\ \mathsf{in}\ C \vdash [\mathtt{z}_0 \mapsto \mathtt{t}_0]\mathtt{t} : \mathtt{T}$.

∘ *Case* ABS. The rule's conclusion is $\mathsf{let}\ \mathtt{z} : \mathtt{T}\ \mathsf{in}\ C \vdash \lambda\mathtt{z}.\mathtt{t} : \mathtt{T} \to \mathtt{T}'$. Its premise is $C \vdash \mathtt{t} : \mathtt{T}'$ **(8)**. We may assume, *w.l.o.g.*, that z is distinct from $\mathtt{z}_0$

and does not occur free within t_0 or σ_0 **(9)**. Applying the induction hypothesis to (8) yields $\mathsf{let}\ z_0 : \sigma_0\ \mathsf{in}\ C \vdash [z_0 \mapsto t_0]t : T'$, which, by ABS, implies $\mathsf{let}\ z : T\ \mathsf{in}\ (\mathsf{let}\ z_0 : \sigma_0\ \mathsf{in}\ C) \vdash \lambda z.[z_0 \mapsto t_0]t : T \to T'$. By (9) and C-LETLET, this may be written $\mathsf{let}\ z_0 : \sigma_0\ \mathsf{in}\ (\mathsf{let}\ z : T\ \mathsf{in}\ C) \vdash [z_0 \mapsto t_0](\lambda z.t) : T \to T'$.

∘ *Case* APP. By the induction hypothesis, by APP, and by C-INAND.

∘ *Case* LET. The rule's conclusion is $\mathsf{let}\ z : \forall \bar{X}_1[C_1].T_1\ \mathsf{in}\ C_2 \vdash \mathtt{let}\ z = t_1\ \mathtt{in}\ t_2 : T_2$, where $\bar{X}_1 = \mathit{ftv}(C_1, T_1)$. Its premises are $C_1 \vdash t_1 : T_1$ **(10)** and $C_2 \vdash t_2 : T_2$ **(11)**. We may assume, *w.l.o.g.*, that z is distinct from z_0 and does not occur free within t_0 or σ_0 **(12)**. We may also assume, *w.l.o.g.*, $\bar{X}_1 \# \mathit{ftv}(\sigma_0)$ **(13)**. Applying the induction hypothesis to (10) and (11) respectively yields $\mathsf{let}\ z_0 : \sigma_0\ \mathsf{in}\ C_1 \vdash [z_0 \mapsto t_0]t_1 : T_1$ **(14)** and $\mathsf{let}\ z_0 : \sigma_0\ \mathsf{in}\ C_2 \vdash [z_0 \mapsto t_0]t_2 : T_2$ **(15)**. Applying LET to (14) and (15) produces $\mathsf{let}\ z : \forall \mathcal{V}[\mathsf{let}\ z_0 : \sigma_0\ \mathsf{in}\ C_1].T_1\ \mathsf{in}\ \mathsf{let}\ z_0 : \sigma_0\ \mathsf{in}\ C_2 \vdash [z_0 \mapsto t_0](\mathtt{let}\ z = t_1\ \mathtt{in}\ t_2) : T_2$ **(16)**. Now, we have

$$
\begin{array}{rll}
 & \mathsf{let}\ z_0 : \sigma_0; z : \forall \bar{X}_1[C_1].T_1\ \mathsf{in}\ C_2 & \\
\equiv & \mathsf{let}\ z_0 : \sigma_0; z : \forall \bar{X}_1[\mathsf{let}\ z_0 : \sigma_0\ \mathsf{in}\ C_1].T_1\ \mathsf{in}\ C_2 & \textbf{(17)} \\
\equiv & \mathsf{let}\ z : \forall \bar{X}_1[\mathsf{let}\ z_0 : \sigma_0\ \mathsf{in}\ C_1].T_1; z_0 : \sigma_0\ \mathsf{in}\ C_2 & \textbf{(18)} \\
\Vdash & \mathsf{let}\ z : \forall \mathcal{V}\ [\mathsf{let}\ z_0 : \sigma_0\ \mathsf{in}\ C_1].T_1; z_0 : \sigma_0\ \mathsf{in}\ C_2 & \textbf{(19)}
\end{array}
$$

where (17) follows from (13), (3), and C-LETDUP; (18) follows from (12) and C-LETLET; and (19) is by Lemma 1.3.43. Thus, applying Lemma 1.5.2 to (16) yields $\mathsf{let}\ z_0 : \sigma_0; z : \forall \bar{X}_1[C_1].T_1\ \mathsf{in}\ C_2 \vdash [z_0 \mapsto t_0](\mathtt{let}\ z = t_1\ \mathtt{in}\ t_2) : T_2$.

∘ *Case* SUB. By the induction hypothesis, by SUB, and by C-INAND*.

∘ *Case* EXISTS. The rule's conclusion is $\exists \bar{X}.C \vdash t : T$. Its premises are $C \vdash t : T$ **(20)** and $\bar{X} \# \mathit{ftv}(T)$ **(21)**. We may assume, *w.l.o.g.*, $\bar{X} \# \mathit{ftv}(\sigma_0)$ **(22)**. Applying the induction hypothesis to (20) yields $\mathsf{let}\ z_0 : \sigma_0\ \mathsf{in}\ C \vdash [z_0 \mapsto t_0]t : T$, which, by (21) and EXISTS, implies $\exists \bar{X}.\mathsf{let}\ z_0 : \sigma_0\ \mathsf{in}\ C \vdash [z_0 \mapsto t_0]t : T$ **(23)**. By (22) and C-INEX, (23) is $\mathsf{let}\ z_0 : \sigma_0\ \mathsf{in}\ \exists \bar{X}.C \vdash [z_0 \mapsto t_0]t : T$. □

1.7.2 LEMMA: $\mathsf{let}\ z : \forall \bar{X}[[\![t_2 : T_2]\!]].T_2\ \mathsf{in}\ [\![t_1 : T_1]\!]$ entails $[\![[z \mapsto t_2]t_1 : T_1]\!]$. □

Proof: This is an immediate consequence of Theorem 1.6.3, Lemma 1.7.1, and Theorem 1.6.7. □

Before going on, let us give a few definitions and formulate several requirements. First, we must define an *initial environment* Γ_0, which assigns a type scheme to every constant. A couple of requirements must be made to ensure that Γ_0 is consistent with the semantics of constants, as specified by $\xrightarrow{\delta}$. Second, we must extend constraint generation and well-typedness to *configurations*, as opposed to programs, since reduction operates on configurations. Last, we must formulate a *restriction* to tame the interaction between side effects and `let`-polymorphism, which is unsound if unrestricted.

1.7.3 DEFINITION: Let Γ_0 be an environment whose domain is the set of constants $\mathcal{Q}$. We require $\mathit{ftv}(\Gamma_0) = \varnothing$, $\mathit{fpi}(\Gamma_0) = \varnothing$, and $\exists\Gamma_0 \equiv \mathsf{true}$. We refer to Γ_0 as the *initial* typing environment. □

Since we require Γ_0 to have no free type or program variables, the constraint $\exists\Gamma_0$ must be equivalent to either true or false. If it is false, then there exists a constant c that no well-typed program may use. We consider this a degenerate case, and avoid it by requiring $\exists\Gamma_0 \equiv \mathsf{true}$.

1.7.4 DEFINITION: Let ref be an isolated, invariant type constructor of signature $\star \Rightarrow \star$. A *store type* M is a finite mapping from memory locations to types. We write $\mathsf{ref}\ M$ for the environment that maps every $m \in \mathit{dom}(M)$ to $\mathsf{ref}\ M(m)$. Assuming $\mathit{dom}(\mu)$ and $\mathit{dom}(M)$ coincide, the constraint $[\![\mu : M]\!]$ is defined as the conjunction of the constraints $[\![\mu(m) : M(m)]\!]$, where m ranges over $\mathit{dom}(\mu)$. Under the same assumption, the constraint $[\![t/\mu : T/M]\!]$ is defined as $[\![t : T]\!] \wedge [\![\mu : M]\!]$. A configuration t/μ is *well-typed* if and only if there exist a type T and a store type M such that $\mathit{dom}(\mu) = \mathit{dom}(M)$ and the constraint $\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![t/\mu : T/M]\!]$ is satisfiable. □

The type $\mathsf{ref}\ T$ is the type of references (that is, memory locations) that store data of type T. It must be *invariant* in its parameter, reflecting the fact that references may be *read* and *written*.

A store is a complex object: it may contain values that indirectly refer to each other via memory locations. In fact, it is a representation of the graph formed by objects and pointers in memory, which may contain cycles. We rely on store types to deal with such cycles. In the definition of well-typedness, the store type M imposes a constraint on the contents of the store—the value $\mu(m)$ must have type $M(m)$—but also plays the role of a hypothesis: by placing the constraint $[\![t/\mu : T/M]\!]$ within the context $\mathsf{let}\ \mathsf{ref}\ M\ \mathsf{in}\ []$, we give meaning to free occurrences of memory locations within $[\![t/\mu : T/M]\!]$, and stipulate that it is valid to assume that m has type $M(m)$. In other words, we essentially view the store as a large, mutually recursive binding of locations to values. Since no satisfiable constraint may have a free program identifier (Lemma 1.3.51), every well-typed configuration must be closed. The context $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ []$ gives meaning to occurrences of constants within $[\![t/\mu : T/M]\!]$.

1.7.5 REMARK: A reference of type $\mathsf{ref}\ T$ may be viewed as an object with *set* and *get* methods, that is, as an object of type $(T \to \mathsf{unit}) \times (\mathsf{unit} \to T)$, where unit is the type of the unit constant $()$. Notice that, in this encoding, T appears both as the domain and as the codomain of an arrow type: this informally explains why ref must be invariant.

Please note that it is also possible to replace ref with a *binary* type constructor biref. The type $\mathsf{biref}\ T_i\ T_o$ is the type of references to which one may write

data of type $\mathtt{T}_i$ and out of which one may read data of type $\mathtt{T}_o$. A reference of type $\mathsf{biref}\ \mathtt{T}_i\ \mathtt{T}_o$ may be viewed as an object of type $(\mathtt{T}_i \to \mathsf{unit}) \times (\mathsf{unit} \to \mathtt{T}_o)$. Notice that, in this encoding, $\mathtt{T}_i$ appears only as the domain of an arrow type, while $\mathtt{T}_o$ appears only as the codomain of an arrow type. This informally justifies why it is sound for biref to be contravariant with respect to its first parameter and covariant in its second parameter, just like the arrow type constructor. Using biref instead of ref makes the type system strictly more expressive. Furthermore, in type systems equipped with subtyping, the absence of type constructors with invariant parameters may simplify the design of constraint solvers and simplifiers. However, this approach has a pragmatic drawback: it leads to larger and more complex types. The idea, which seems due to Reynolds (1988), has been studied in other settings, such as process and object calculi (Pierce and Sangiorgi, 1993; Bugliesi and Pericás-Geertsen, 2002). □

We now define a relation between configurations that plays a key role in the statement of the subject reduction property. The point of subject reduction is to guarantee that well-typedness is preserved by reduction. However, such a simple statement is too weak to be amenable to inductive proof. Thus, for the purposes of the proof, we must be more specific. To begin, let us consider the simpler case of a pure semantics, that is, a semantics without stores. Then, we must state that if an expression $\mathtt{t}$ has type $\mathtt{T}$ under a certain constraint, then its reduct $\mathtt{t}'$ has type $\mathtt{T}$ under the same constraint. In terms of generated constraints, this statement becomes: $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathtt{t} : \mathtt{T}]\!]$ entails $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathtt{t}' : \mathtt{T}]\!]$. Let us now return to the general case, where a store is present. Then, the statement of well-typedness for a configuration $\mathtt{t}/\mu$ involves a store type M whose domain is that of μ. So, the statement of well-typedness for its reduct $\mathtt{t}'/\mu'$ must involve a store type M' whose domain is that of μ'—which is larger if allocation occurred. The types of existing memory locations must not change: we must request that M and M' agree on $dom(M)$, that is, M' must extend M. Furthermore, the types assigned to new memory locations in $dom(M') \setminus dom(M)$ might involve new type variables, that is, variables that do not appear free in M or $\mathtt{T}$. We must allow these variables to be hidden—that is, existentially quantified—otherwise the entailment assertion cannot hold. These considerations lead us to the following definition:

1.7.6 DEFINITION: $\mathtt{t}/\mu \sqsubseteq \mathtt{t}'/\mu'$ holds if and only if, for every type $\mathtt{T}$ and for every store type M such that $dom(\mu) = dom(M)$, there exist a set of type variables $\bar{\mathtt{Y}}$ and a store type M' such that $\bar{\mathtt{Y}} \# ftv(\mathtt{T}, M)$ and $ftv(M') \subseteq \bar{\mathtt{Y}} \cup ftv(M)$ and

$dom(M') = dom(\mu')$ and M' extends M and

$$\begin{array}{r} \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \ \mathsf{in}\ [\![\mathtt{t}\ /\mu\ : \mathtt{T}/M\ ]\!] \\ \Vdash\ \exists \mathtt{Y}.\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M'\ \mathsf{in}\ [\![\mathtt{t}'/\mu' : \mathtt{T}/M']\!]. \end{array}$$

The relation $\sqsubseteq$ is intended to express a connection between a configuration and its reduct. Thus, subject reduction may be stated as: $(\longrightarrow\!\!\!\!\triangleright) \subseteq (\sqsubseteq)$, that is, $\sqsubseteq$ is indeed a conservative description of reduction. □

We have introduced an initial environment Γ_0 and used it in the definition of well-typedness, but we haven't yet ensured that the type schemes assigned to constants are an adequate description of their semantics. We now formulate two requirements that relate Γ_0 with $\xrightarrow{\delta}$. They are specializations of the subject reduction and progress properties to configurations that involve an application of a constant. They represent proof obligations that must be discharged when concrete definitions of $\mathcal{Q}$, $\xrightarrow{\delta}$, and Γ_0 are given.

1.7.7 DEFINITION: We require (i) $(\xrightarrow{\delta}) \subseteq (\sqsubseteq)$; and (ii) if the configuration $\mathtt{c}\ \mathtt{v}_1\ \ldots\ \mathtt{v}_k/\mu$ (where $k \geq 0$) is well-typed, then either it is reducible, or $\mathtt{c}\ \mathtt{v}_1\ \ldots\ \mathtt{v}_k$ is a value. □

The last point that remains to be settled before proving type soundness is the interaction between side effects and `let`-polymorphism. The following example illustrates the problem:

$$\mathtt{let\ r} = \mathtt{ref}\ \lambda \mathtt{z.z\ in\ let\ \_} = (\mathtt{r} := \lambda \mathtt{z}.(\mathtt{z} \hat{+} \hat{1}))\ \mathtt{in\ !r\ true}$$

This expression reduces to $\mathtt{true} \hat{+} \hat{1}$, so it must not be well-typed. Yet, if natural type schemes are assigned to `ref`, !, and := (see Example 1.9.6), then it *is* well-typed with respect to the rules given so far, because `r` receives the polymorphic type scheme $\forall \mathtt{X}.\mathsf{ref}\ (\mathtt{X} \to \mathtt{X})$, which allows writing a function of type $\mathsf{int} \to \mathsf{int}$ into `r` and reading it back with type $\mathsf{bool} \to \mathsf{bool}$. The problem is that `let`-polymorphism simulates a textual duplication of the `let`-bound expression $\mathtt{ref}\ \lambda \mathtt{z.z}$, while the semantics first reduces it to a value m, causing a new binding $m \mapsto \lambda \mathtt{z.z}$ to appear in the store, then duplicates the address m. The new store binding is not duplicated: both copies of m refer to the same memory cell. For this reason, generalization is unsound in this case, and must be restricted. Many authors have attempted to come up with a sound type system that accepts *all* pure programs and remains flexible enough in the presence of side effects (Tofte, 1988; Leroy, 1992). These proposals are often complex, which is why they have been abandoned in favor of an extremely simple *syntactic* restriction, known as the *value restriction* (Wright, 1995).

1.7.8 DEFINITION: A program satisfies the *value restriction* if and only if all subexpressions of the form `let z =` $\mathtt{t}_1$ `in` $\mathtt{t}_2$ are in fact of the form `let z =` $\mathtt{v}_1$ `in` $\mathtt{t}_2$. In the following, we assume that either all constants have pure semantics, or all programs satisfy the value restriction. □

Put slightly differently, the value restriction states that only values may be generalized. This eliminates the problem altogether, since duplicating values does not affect a program's semantics. Note that any program that does not satisfy the value restriction can be turned into one that does and has the same semantics: it suffices to change `let z =` $\mathtt{t}_1$ `in` $\mathtt{t}_2$ into $(\lambda \mathtt{z}.\mathtt{t}_2)\ \mathtt{t}_1$ when $\mathtt{t}_1$ is not a value. Of course, such a transformation may cause the program to become ill-typed. In other words, the value restriction causes some perfectly safe programs to be rejected. In particular, in its above form, it prevents generalizing applications of the form `c` $\mathtt{v}_1$ ... $\mathtt{v}_k$, where `c` is a destructor of arity k. This is excessive, because many destructors have pure semantics; only a few, such as `ref`, allocate new mutable storage. Furthermore, we use pure destructors to encode numerous language features (§1.9). Fortunately, it is easy to relax the restriction to allow generalizing not only values, but also a more general class of *nonexpansive* expressions, whose syntax guarantees that such expressions cannot allocate new mutable storage (that is, *expand* the domain of the store). The term *nonexpansive* was coined by Tofte (1988). Nonexpansive expressions may include applications of the form `c` $\mathtt{t}_1$ ... $\mathtt{t}_k$, where `c` is a pure destructor of arity k and $\mathtt{t}_1, \ldots, \mathtt{t}_k$ are nonexpansive. Experience shows that this slightly relaxed restriction is acceptable in practice. Some limitations remain: for instance, "constructor functions" (that is, functions that do not allocate mutable storage and build a value) are regarded as ordinary functions, so their applications are considered potentially expansive, even though a naked constructor application would be a value and thus considered nonexpansive. For instance, in the expression `let f = c v in let z = f w in t`, where `c` is a constructor of arity 2, the partial application `c v`, to which the name `f` is bound, is a constructor function (of arity 1). The program variable `z` cannot receive a polymorphic type scheme, because `f w` is not a value, even though it has the same semantic meaning as `c v w`, which is a value. A recent improvement to the value restriction (Garrigue, 2003) provides a partial remedy. Technically, the effect of the value restriction (as stated in Definition 1.7.8) is summarized by the following result.

1.7.9 LEMMA: Under the value restriction, the production $\mathcal{E}$ ::= `let z =` $\mathcal{E}$ `in t` may be suppressed from the grammar of evaluation contexts (Figure 1-1) without altering the operational semantics. □

Proof: First, the value restriction is preserved by reduction. Indeed, none of the reduction rules creates new `let` expressions. Furthermore, applying a

substitution of the form $[z \mapsto v]$ to a value must yield a value, so existing `let` expressions—which, by assumption, satisfy the restriction—still satisfy it when some of their free program variables are substituted away.

Second, by Lemma 1.2.14, values are irreducible. Thus, under the value restriction, the left-hand side of every `let` form is irreducible. As a result, suppressing the production $\mathcal{E} ::= \texttt{let z} = \mathcal{E} \texttt{ in t}$ does not alter the semantics. □

We are done with definitions and requirements. We now come to the type soundness results.

1.7.10 THEOREM [SUBJECT REDUCTION]: $(\longrightarrow) \subseteq (\sqsubseteq)$. □

Proof: Because $\longrightarrow$ and $\longrightarrow\!\!\!\!\triangleright$ are the smallest relations that satisfy the rules of Figure 1-2, it suffices to prove that $\sqsubseteq$ satisfies these rules as well. We remark that if, for every type $\texttt{T}$, $[\![\texttt{t} : \texttt{T}]\!] \Vdash [\![\texttt{t}' : \texttt{T}]\!]$ holds, then $\texttt{t}/\mu \sqsubseteq \texttt{t}'/\mu$ holds. (Take $\bar{Y} = \varnothing$ and $M' = M$ and use the fact that entailment is a congruence to check that the conditions of Definition 1.7.6 are met.) We make use of this fact in cases R-BETA and R-LET below.

◦ *Case* R-BETA. We have

$$\begin{array}{lll} & [\![(\lambda \texttt{z.t})\ \texttt{v} : \texttt{T}]\!] & \\ \equiv & \exists \texttt{X}.([\![\lambda \texttt{z.t} : \texttt{X} \to \texttt{T}]\!] \wedge [\![\texttt{v} : \texttt{X}]\!]) & \textbf{(1)} \\ \equiv & \exists \texttt{X}.(\mathsf{let}\ \texttt{z} : \texttt{X}\ \mathsf{in}\ [\![\texttt{t} : \texttt{T}]\!] \wedge [\![\texttt{v} : \texttt{X}]\!]) & \textbf{(2)} \\ \equiv & \exists \texttt{X}.\mathsf{let}\ \texttt{z} : \forall \varnothing[[\![\texttt{v} : \texttt{X}]\!]].\texttt{X}\ \mathsf{in}\ [\![\texttt{t} : \texttt{T}]\!] & \textbf{(3)} \\ \Vdash & [\![[\texttt{z} \mapsto \texttt{v}]\texttt{t} : \texttt{T}]\!] & \textbf{(4)} \end{array}$$

where (1) is by definition of constraint generation; (2) is by Lemma 1.6.6; (3) is by C-LETAND; (4) is by Lemma 1.7.2 and C-EX*.

◦ *Case* R-LET. We have

$$\begin{array}{lll} & [\![\texttt{let z} = \texttt{v in t} : \texttt{T}]\!] & \\ = & \mathsf{let}\ \texttt{z} : \forall \texttt{X}[[\![\texttt{v} : \texttt{X}]\!]].\texttt{X}\ \mathsf{in}\ [\![\texttt{t} : \texttt{T}]\!] & \textbf{(1)} \\ \Vdash & [\![[\texttt{z} \mapsto \texttt{v}]\texttt{t} : \texttt{T}]\!] & \textbf{(2)} \end{array}$$

where (1) is by definition of constraint generation and (2) is by Lemma 1.7.2.

◦ *Case* R-DELTA. This case is exactly requirement (i) in Definition 1.7.7.

◦ *Case* R-EXTEND. Our hypotheses are $\texttt{t}/\mu \sqsubseteq \texttt{t}'/\mu'$ **(1)** and $dom(\mu'') \# dom(\mu')$ **(2)** and $range(\mu'') \# dom(\mu' \setminus \mu)$ **(3)**. Because $dom(\mu)$ must be a subset of $dom(\mu')$, it is also disjoint with $dom(\mu'')$. Our goal is $\texttt{t}/\mu\mu'' \sqsubseteq \texttt{t}'/\mu'\mu''$ **(4)**. Thus, let us introduce a type $\texttt{T}$ and a store type of domain $dom(\mu\mu'')$, or (equivalently) two store types M and M'' whose domains are respectively $dom(\mu)$ and $dom(\mu'')$. By (1), there exist type variables $\bar{Y}$ and a store type M'

such that $\bar{Y} \# \mathit{ftv}(T, M)$ **(5)** and $\mathit{ftv}(M') \subseteq \bar{Y} \cup \mathit{ftv}(M)$ and $\mathit{dom}(M') = \mathit{dom}(\mu')$ and M' extends M **(6)** and $\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![t/\mu : T/M]\!] \Vdash \exists \bar{Y}.\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M'\ \mathsf{in}\ [\![t'/\mu' : T/M']\!]$. We may further require, *w.l.o.g.*, $\bar{Y} \# \mathit{ftv}(M'')$ **(7)**. Let us now add the conjunct $\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![\mu'' : M'']\!]$ to each side of this entailment assertion. On the left-hand side, by C-InAnd and by Definition 1.7.4, we obtain $\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![t/\mu\mu'' : T/MM'']\!]$ **(8)**. On the right-hand side, by (5), (7), C-ExAnd, and C-InAnd, we obtain $\exists \bar{Y}.\mathsf{let}\ \Gamma_0\ \mathsf{in}\ (\mathsf{let}\ \mathsf{ref}\ M'\ \mathsf{in}\ [\![t'/\mu' : T/M']\!] \wedge \mathsf{let}\ \mathsf{ref}\ M\ \mathsf{in}\ [\![\mu'' : M'']\!])$ **(9)**. Now, recall that M' extends M (6) and, furthermore, (3) implies $\mathit{fpi}([\![\mu'' : M'']\!]) \# \mathit{dpi}(M' \setminus M)$ **(10)**. By (10), C-InAnd*, and C-InAnd, (9) is equivalent to $\exists \bar{Y}.\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M'\ \mathsf{in}\ ([\![t'/\mu' : T/M']\!] \wedge [\![\mu'' : M'']\!])$, that is, $\exists \bar{Y}.\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M'\ \mathsf{in}\ [\![t'/\mu'\mu'' : T/M'M'']\!]$ **(11)**. Thus, we have established that (8) entails (11). Let us now place this entailment assertion within the constraint context $\mathsf{let}\ \mathsf{ref}\ M''\ \mathsf{in}\ []$. On the left-hand side, because $\mathit{fpi}(\Gamma_0, M, M'') = \varnothing$ and $\mathit{dpi}(M'') \cap \mathit{dpi}(\Gamma_0, M) \subseteq \mathit{dom}(\mu'') \cap (\mathcal{Q} \cup \mathit{dom}(\mu)) = \varnothing$, C-LetLet applies, yielding $\mathsf{let}\ \Gamma_0; \mathsf{ref}\ MM''\ \mathsf{in}\ [\![t/\mu\mu'' : T/MM'']\!]$ **(12)**. On the right-hand side, by (7), C-InEx, and by analogous reasoning, we obtain $\exists \bar{Y}.\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M'M''\ \mathsf{in}\ [\![t'/\mu'\mu'' : T/M'M'']\!]$ **(13)**. Thus, (12) entails (13). Given (5), (7), given $\mathit{ftv}(M'M'') \subseteq \bar{Y} \cup \mathit{ftv}(MM'')$, and given that $M'M''$ extends MM'', this establishes the goal (4).

∘ *Case* R-Context. The hypothesis is $t/\mu \sqsubseteq t'/\mu'$. The goal is $\mathcal{E}[t]/\mu \sqsubseteq \mathcal{E}[t']/\mu'$. Because $\longrightarrow$ relates closed configurations only, we may assume that the configuration $\mathcal{E}[t]/\mu$ is closed, so the memory locations that appear free within $\mathcal{E}$ are members of $\mathit{dom}(\mu)$. Let us now reason by induction on the structure of $\mathcal{E}$.

Subcase $\mathcal{E} = []$. The hypothesis and the goal coincide.

Subcase $\mathcal{E} = \mathcal{E}_1\ t_1$. The induction hypothesis is $\mathcal{E}_1[t]/\mu \sqsubseteq \mathcal{E}_1[t']/\mu'$ **(1)**. Let us introduce a type T and a store type M such that $\mathit{dom}(M) = \mathit{dom}(\mu)$. Consider the constraint $\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![\mathcal{E}[t]/\mu : T/M]\!]$ **(2)**. By definition of constraint generation, C-ExAnd, C-InEx, and C-InAnd, it is equivalent to

$$\exists X.(\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![\mathcal{E}_1[t]/\mu : X \to T/M]\!] \wedge \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![t_1 : X]\!]) \textbf{ (3)}$$

where $X \notin \mathit{ftv}(T, M)$ **(4)**. By (1), there exist type variables $\bar{Y}$ and a store type M' such that $\bar{Y} \# \mathit{ftv}(X, T, M)$ **(5)** and $\mathit{ftv}(M') \subseteq \bar{Y} \cup \mathit{ftv}(M)$ **(6)** and $\mathit{dom}(M') = \mathit{dom}(\mu')$ and M' extends M and (3) entails

$$\exists X.(\exists \bar{Y}.\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M'\ \mathsf{in}\ [\![\mathcal{E}_1[t']/\mu' : X \to T/M']\!] \wedge \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![t_1 : X]\!]) \textbf{ (7)}.$$

We pointed out earlier that the memory locations that appear free in t_1 are members of $\mathit{dom}(M)$, which implies $\mathsf{let}\ \mathsf{ref}\ M\ \mathsf{in}\ [\![t_1 : X]\!] \equiv \mathsf{let}\ \mathsf{ref}\ M'\ \mathsf{in}\ [\![t_1 : X]\!]$ **(8)**. By (5), C-ExAnd, (8), C-InAnd, and by definition of constraint generation, we find that (7) is equivalent to

$$\exists X \bar{Y}.\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M'\ \mathsf{in}\ ([\![\mathcal{E}_1[t'] : X \to T]\!] \wedge [\![t_1 : X]\!] \wedge [\![\mu' : M']\!]) \textbf{ (9)}.$$

(4), (5) and (6) imply $\mathtt{X} \notin \mathit{ftv}(\mathtt{M}')$. Thus, by C-InEx and C-ExAnd, (9) may be written

$$\exists \bar{\mathtt{Y}}.\mathsf{let}\ \Gamma_0; \mathsf{ref}\ \mathtt{M}'\ \mathsf{in}\ (\exists \mathtt{X}.([\![\mathcal{E}_1[\mathtt{t}'] : \mathtt{X} \to \mathtt{T}]\!] \wedge [\![\mathtt{t}_1 : \mathtt{X}]\!]) \wedge [\![\mu' : \mathtt{M}']\!]),$$

which, by definition of constraint generation, is

$$\exists \bar{\mathtt{Y}}.\mathsf{let}\ \Gamma_0; \mathsf{ref}\ \mathtt{M}'\ \mathsf{in}\ [\![\mathcal{E}[\mathtt{t}']/\mu' : \mathtt{T}/\mathtt{M}']\!]\ \textbf{(10)}.$$

Thus, we have proved that (2) entails (10). By Definition 1.7.6, this establishes $\mathcal{E}[\mathtt{t}]/\mu \sqsubseteq \mathcal{E}[\mathtt{t}']/\mu'$.

Subcase $\mathcal{E} = \mathtt{v}\ \mathcal{E}_1$. Analogous to the previous subcase.

Subcase $\mathcal{E} = \mathtt{let\ z} = \mathcal{E}_1\ \mathtt{in\ t}_1$. The induction hypothesis is $\mathcal{E}_1[\mathtt{t}]/\mu \sqsubseteq \mathcal{E}_1[\mathtt{t}']/\mu'$ **(1)**. This subcase is particularly interesting, because it is where `let`-polymorphism and side effects interact. In the previous two subcases, we relied on the fact that the $\exists \bar{\mathtt{Y}}$ quantifier, which hides the types of the memory cells created by the reduction step, *commutes* with the connectives $\exists$ and $\wedge$ introduced by application contexts. However, it does not in general (left-)commute with the let connective (Example 1.3.48). Fortunately, under the value restriction, this subcase *never arises* (Lemma 1.7.9). By Definition 1.7.8, this subcase may arise only if all constants have pure semantics, which implies $\mu = \mu' = \varnothing$. Then, we have

$$\begin{array}{lll}
 & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathcal{E}[\mathtt{t}] : \mathtt{T}]\!] & \\
= & \mathsf{let}\ \Gamma_0; \mathtt{z} : \forall \mathtt{X}[[\![\mathcal{E}_1[\mathtt{t}] : \mathtt{X}]\!]].\mathtt{X}\ \mathsf{in}\ [\![\mathtt{t}_1 : \mathtt{T}]\!] & \textbf{(2)} \\
\equiv & \mathsf{let}\ \Gamma_0; \mathtt{z} : \forall \mathtt{X}[\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathcal{E}_1[\mathtt{t}] : \mathtt{X}]\!]].\mathtt{X}\ \mathsf{in}\ [\![\mathtt{t}_1 : \mathtt{T}]\!] & \textbf{(3)} \\
\Vdash & \mathsf{let}\ \Gamma_0; \mathtt{z} : \forall \mathtt{X}[\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathcal{E}_1[\mathtt{t}'] : \mathtt{X}]\!]].\mathtt{X}\ \mathsf{in}\ [\![\mathtt{t}_1 : \mathtt{T}]\!] & \textbf{(4)} \\
\equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathcal{E}[\mathtt{t}'] : \mathtt{T}]\!] & \textbf{(5)}
\end{array}$$

where (2) is by definition of constraint generation; (3) follows from $\mathit{ftv}(\Gamma_0) = \mathit{fpi}(\Gamma_0) = \varnothing$ and C-LetDup; (4) follows from (1), specialized to the case of a pure semantics; and (5) is obtained by performing these steps in reverse. □

1.7.11 Exercise [Recommended, ★★★]: Try to carry out the last subcase of the above proof in the case of an impure semantics and in the absence of the value restriction. Find out why it fails. Show that it succeeds if $\bar{\mathtt{Y}}$ is assumed to be empty. Use this fact to prove that generalization is still safe when restricted to *nonexpansive* expressions, provided (i) evaluating a nonexpansive expression cannot cause new memory cells to be allocated, (ii) nonexpansive expressions are stable by substitution of values for variables, and (iii) nonexpansive expressions are preserved by reduction. □

Subject reduction ensures that well-typedness is preserved by reduction.

1.7.12 COROLLARY: Let $\mathtt{t}/\mu \longrightarrow \mathtt{t}'/\mu'$. If $\mathtt{t}/\mu$ is well-typed, then so is $\mathtt{t}'/\mu'$. □

Proof: Assume $\mathtt{t}/\mu \longrightarrow \mathtt{t}'/\mu'$ **(1)** and $\mathtt{t}/\mu$ is well-typed **(2)**. By (2) and Definition 1.7.4, there exist a type $\mathtt{T}$ and a store type M such that $dom(\mu) = dom(M)$ and the constraint $\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![\mathtt{t}/\mu : \mathtt{T}/M]\!]$ **(3)** is satisfiable. By Theorem 1.7.10 and Definition 1.7.6, (1) implies that there exist a set of type variables $\bar{Y}$ and a store type M' such that $dom(M') = dom(\mu')$ **(4)** and the constraint (3) entails $\exists \bar{Y}.\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M'\ \mathsf{in}\ [\![\mathtt{t}'/\mu' : \mathtt{T}/M']\!]$ **(5)**. Because (3) is satisfiable, so is (5), which implies that $\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M'\ \mathsf{in}\ [\![\mathtt{t}'/\mu' : \mathtt{T}/M']\!]$ is satisfiable **(6)**. By (4) and (6) and Definition 1.7.4, $\mathtt{t}'/\mu'$ is well-typed. □

Let us now establish the progress property.

1.7.13 LEMMA: If $\mathtt{t}_1\ \mathtt{t}_2$ is well-typed, then $\mathtt{t}_1/\mu$ and $\mathtt{t}_2/\mu$ are well-typed. If $\mathtt{let\ z} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2/\mu$ is well-typed, then $\mathtt{t}_1/\mu$ is well-typed. □

Proof: Let us prove the second statement; the proof of the first one is analogous. Because $\mathtt{let\ z} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2/\mu$ is well-typed, there exist a type $\mathtt{T}$ and a store type M such that $dom(\mu) = dom(M)$ and the constraint $\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![\mathtt{let\ z} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2/\mu : \mathtt{T}/M]\!]$ **(1)** is satisfiable. By definition of constraint generation, (1) is $\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ ((\mathsf{let}\ \mathtt{z} : \forall \mathtt{X}[[\![\mathtt{t}_1 : \mathtt{X}]\!]].\mathtt{X}\ \mathsf{in}\ [\![\mathtt{t}_2 : \mathtt{T}]\!]) \wedge [\![\mu : M]\!])$ **(2)**, where $\mathtt{X} \notin ftv(\mathtt{T}, M)$ **(3)**. By definition of let, (2) entails $\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ ((\exists \mathtt{X}.[\![\mathtt{t}_1 : \mathtt{X}]\!]) \wedge [\![\mu : M]\!])$, which, by (3), C-EXAND and C-INEX, is equivalent to $\exists \mathtt{X}.\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![\mathtt{t}_1/\mu : \mathtt{X}/M]\!]$ **(4)**. Thus, (4) is satisfiable, so $\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![\mathtt{t}_1/\mu : \mathtt{X}/M]\!]$ is satisfiable as well, and $\mathtt{t}_1/\mu$ is well-typed. □

1.7.14 THEOREM [PROGRESS]: If $\mathtt{t}/\mu$ is well-typed, then either it is reducible, or $\mathtt{t}$ is a value. □

Proof: The proof is by induction on the structure of $\mathtt{t}$.

- *Case* $\mathtt{t} = \mathtt{z}$. Well-typed configurations are closed: this case cannot occur.
- *Case* $\mathtt{t} = \mathtt{m}$. $\mathtt{t}$ is a value.
- *Case* $\mathtt{t} = \mathtt{c}$. By requirement (ii) of Definition 1.7.7.
- *Case* $\mathtt{t} = \lambda \mathtt{z}.\mathtt{t}_1$. $\mathtt{t}$ is a value.
- *Case* $\mathtt{t} = \mathtt{t}_1\ \mathtt{t}_2$. By Lemma 1.7.13, $\mathtt{t}_1/\mu$ is well-typed. By the induction hypothesis, either it is reducible, or $\mathtt{t}_1$ is a value. If the former, by R-CONTEXT and because every context of the form $\mathcal{E}\ \mathtt{t}_2$ is an evaluation context, the configuration $\mathtt{t}/\mu$ is reducible as well. Thus, let us assume $\mathtt{t}_1$ is a value. By Lemma 1.7.13, $\mathtt{t}_2/\mu$ is well-typed. By the induction hypothesis, either it is reducible, or $\mathtt{t}_2$ is a value. If the former, by R-CONTEXT and because every context of the form $\mathtt{t}_1\ \mathcal{E}$—where $\mathtt{t}_1$ is a value—is an evaluation context, the configuration $\mathtt{t}/\mu$ is reducible as well. Thus, let us assume $\mathtt{t}_2$ is a value. Let us now reason by cases on the structure of $\mathtt{t}_1$.

Subcase $t_1 = z$. Again, this subcase cannot occur.

Subcase $t_1 = m$. Because t/μ is well-typed, a constraint of the form $\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ (\exists X.(m \preceq X \to T \wedge [\![t_2 : X]\!]) \wedge [\![\mu : M]\!])$ must be satisfiable. This implies that m is a member of $dom(M)$ and that the constraint $\mathsf{ref}\ M(m) \leq X \to T$ is satisfiable. Because the type constructors ref and $\to$ are incompatible, this is a contradiction. So, this subcase cannot occur.

Subcase $t_1 = \lambda z.t_1'$. By R-BETA, t/μ is reducible.

Subcase $t_1 = c\ v_1\ \ldots\ v_k$. Then, t is of the form $c\ v_1\ \ldots\ v_{k+1}$. The result follows by requirement (ii) of Definition 1.7.7.

∘ *Case* $t = \mathtt{let}\ z = t_1\ \mathtt{in}\ t_2$. By Lemma 1.7.13, t_1/μ is well-typed. By the induction hypothesis, either t_1/μ is reducible, or t_1 is a value. If the former, by R-CONTEXT and because every context of the form $\mathtt{let}\ z = \mathcal{E}\ \mathtt{in}\ t_2$ is an evaluation context, the configuration t/μ is reducible as well. If the latter, then t/μ is reducible by R-LET. □

We may now conclude:

1.7.15 THEOREM [TYPE SOUNDNESS]: Well-typed source programs do not go wrong. □

Proof: We say that a source program t is well-typed if and only if the configuration $t/\varnothing$ is well-typed, that is, if and only if $\exists X.\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![t : X]\!] \equiv \mathsf{true}$ holds. By Lemma 1.7.12, all reducts of $t/\varnothing$ are well-typed. By Theorem 1.7.14, none is stuck. □

Let us recall that this result holds only if the requirements of Definition 1.7.7 are met. In other words, some proof obligations remain to be discharged when concrete definitions of $\mathcal{Q}$, $\xrightarrow{\delta}$, and Γ_0 are given. This is illustrated by several examples in §1.9 and §1.11.

Of course, the type system is, in general, incomplete: that is, there exist source programs that do not go wrong, yet are (unfortunately) ill-typed. More precisely, let us assume that there exists at least one ill-typed source program t_0. Then, it is easy to reduce the problem of determining whether a program terminates to the problem of determining whether a program goes wrong: indeed, t terminates if and only if $t; t_0$ goes wrong. Because the λ-calculus is Turing complete, its termination problem is undecidable, so determining whether a program goes wrong must be undecidable as well. However, assuming that constraint satisfiability is decidable, well-typedness in HM(X) is decidable. Thus, the type system must be incomplete. Please note that this argument is valid not only for HM(X), but for any Turing complete programming language equipped with a decidable type system. Of course, the hypothesis that some programs *do* go wrong is important: otherwise, a

trivial type system, where every program is well-typed, is both sound and complete. This is illustrated by the next exercise.

1.7.16 EXERCISE [★]: Show that, when the set of constants $\mathcal{Q}$ is empty (that is, when the programming language is the pure λ-calculus) and when types are interpreted in a regular tree model (that is, when recursive types exist), every closed source program is well-typed. □

1.8 Constraint solving

We have introduced a parameterized constraint language, given equivalence laws that describe the interaction between its logical connectives, and exploited them to prove theorems about type inference and type soundness, which are valid independently of the nature of primitive constraints—the so-called predicate applications. However, there would be little point in proposing a parameterized constraint solver, because much of the difficulty of designing an efficient constraint solver precisely lies in the treatment of primitive constraints and in its interaction with `let`-polymorphism. For this reason, in this section, we focus on constraint solving in the setting of an *equality-only free tree model*. Thus, the constraint solver developed here allows performing type inference for HM(=) (that is, for Damas and Milner's type system) and for its extension with recursive types. Of course, some of its mechanisms may be useful in other settings. Other constraint solvers used in program analysis or type inference are described *e.g.* in (Aiken and Wimmers, 1992; Niehren, Müller, and Podelski, 1997; Fähndrich, 1999; Melski and Reps, 2000; Müller, Niehren, and Treinen, 2001; Pottier, 2001b; Nielson, Nielson, and Seidl, 2002; McAllester, 2002, 2003; Simonet, 2003).

We begin with a rule-based presentation of a standard, efficient first-order unification algorithm. This yields a constraint solver for a subset of the constraint language, deprived of type scheme introduction and instantiation forms. On top of it, we build a full constraint solver, which corresponds to the code that accompanies this chapter. Then, we discuss another constraint solver, which uses a different strategy to eliminate type scheme introduction and instantiation constraints. We conclude with a brief discussion of type errors.

1.8.1 Unification

Unification is the process of solving equations between terms. We present a unification algorithm due to Huet (1976), whose time complexity is quasi-linear. The specification, a (nondeterministic) system of constraint rewriting

rules, is almost the same for *finite* and *regular* tree models: only one rule, which implements the *occurs check*, must be removed in the latter case. In other words, the algorithm works with *possibly cyclic* terms, and does not rely in an essential way on the occurs check. In order to more closely reflect the behavior of the actual algorithm, which relies on a *union-find* data structure (Tarjan, 1975), we modify the syntax of constraints by replacing equations with *multi-equations*. A multi-equation is an equation that involves an arbitrary number of types, as opposed to exactly two.

1.8.1 DEFINITION: Let there be, for every kind κ and for every $n \geq 1$, a predicate $=^n_\kappa$, of signature $\kappa^n \Rightarrow \cdot$, whose interpretation is (n-ary) equality. The predicate constraint $=^n_\kappa\ T_1 \ldots T_n$ is written $T_1 = \ldots = T_n$, and called a *multi-equation*. We consider the constraint true as a multi-equation of length 0 and let ϵ range over all multi-equations. In the following, we identify multi-equations up to permutations of their members, so a multi-equation ϵ of kind κ may be viewed as a finite *multiset* of types of kind κ. We write $\epsilon = \epsilon'$ for the multi-equation obtained by concatenating ϵ and ϵ'. □

Thus, we are interested in the following subset of the constraint language:

$$U ::= \mathsf{true} \mid \mathsf{false} \mid \epsilon \mid U \wedge U \mid \exists \bar{X}.U$$

Equations are replaced with multi-equations; no other predicates are available. Type scheme introduction and instantiation forms are absent.

1.8.2 DEFINITION: A multi-equation is *standard* if and only if its variable members are distinct and it has at most one nonvariable member. A constraint U is *standard* if and only if every multi-equation inside U is standard and every variable that occurs (free or bound) in U is a member of at most one multi-equation inside U. □

A union-find algorithm maintains equivalence classes (that is, disjoint sets) of variables, and associates, with each class, a *descriptor*, which in our case is either absent or a nonvariable term. Thus, a *standard* constraint represents a state of the union-find algorithm. A constraint that is *not* standard may be viewed as a superposition of a state of the union-find algorithm, on the one hand, and of control information, on the other hand. For instance, a multi-equation of the form $\epsilon = T_1 = T_2$, where ϵ is made up of distinct variables and T_1 and T_2 are nonvariable terms, may be viewed, roughly speaking, as the equivalence class $\epsilon = T_1$, together with a pending request to solve $T_1 = T_2$ and to update the class's descriptor accordingly. Because multi-equations encode both state and control, our specification of the unification algorithm remains rather abstract. It would be possible to give a lower-level

$$(\exists \bar{X}.U_1) \wedge U_2 \rightarrow \exists \bar{X}.(U_1 \wedge U_2) \quad \text{if } \bar{X} \# \mathit{ftv}(U_2) \qquad \text{(S-EXAND)}$$

$$X = \epsilon \wedge X = \epsilon' \rightarrow X = \epsilon = \epsilon' \qquad \text{(S-FUSE)}$$

$$X = X = \epsilon \rightarrow X = \epsilon \qquad \text{(S-STUTTER)}$$

$$F\,\vec{X} = F\,\vec{T} = \epsilon \rightarrow \vec{X} = \vec{T} \wedge F\,\vec{X} = \epsilon \qquad \text{(S-DECOMPOSE)}$$

$$F\,T_1 \ldots T_i \ldots T_n = \epsilon \rightarrow \exists X.(X = T_i \wedge F\,T_1 \ldots X \ldots T_n = \epsilon) \quad \text{if } T_i \notin \mathcal{V} \wedge X \notin \mathit{ftv}(T_1, \ldots, T_n, \epsilon) \qquad \text{(S-NAME-1)}$$

$$F\,\vec{T} = F'\,\vec{T}' = \epsilon \rightarrow \mathsf{false} \quad \text{if } F \neq F' \qquad \text{(S-CLASH)}$$

$$T \rightarrow \mathsf{true} \quad \text{if } T \notin \mathcal{V} \qquad \text{(S-SINGLE)}$$

$$U \wedge \mathsf{true} \rightarrow U \qquad \text{(S-TRUE)}$$

$$U \rightarrow \mathsf{false} \quad \text{if the model is syntactic and } U \text{ is cyclic} \qquad \text{(S-CYCLE)}$$

$$\mathcal{U}[\mathsf{false}] \rightarrow \mathsf{false} \quad \text{if } \mathcal{U} \neq [] \qquad \text{(S-FAIL)}$$

Figure 1-11: Unification

description, where state (standard conjunctions of multi-equations) and control (pending binary equations) are distinguished.

1.8.3 DEFINITION: Let U be a conjunction of multi-equations. Y is *dominated* by X with respect to U (written: $Y \prec_U X$) if and only if U contains a conjunct of the form $X = F\,\vec{T} = \epsilon$, where $Y \in \mathit{ftv}(\vec{T})$. U is *cyclic* if and only if the graph of $\prec_U$ exhibits a cycle. □

The specification of the unification algorithm consists of a set of constraint rewriting rules, given in Figure 1-11. Rewriting is performed modulo α-conversion, modulo permutations of the members of a multi-equation, modulo commutativity and associativity of conjunction, and under an arbitrary context. The specification is nondeterministic: several rule instances may be simultaneously applicable.

S-EXAND is a directed version of C-EXAND, whose effect is to float up all existential quantifiers. In the process, all multi-equations become part of a

single conjunction, possibly causing rules whose left-hand side is a conjunction of multi-equations, namely S-FUSE and S-CYCLE, to become applicable. S-FUSE identifies two multi-equations that share a common variable x, and fuses them. The new multi-equation is not necessarily standard, even if the two original multi-equations were. Indeed, it may have repeated variables or contain two nonvariable terms. The purpose of the next few rules, whose left-hand side consists of a single multi-equation, is to deal with these situations. S-STUTTER eliminates redundant variables. It only deals with variables, as opposed to terms of arbitrary size, so as to have constant time cost. The comparison of nonvariable terms is implemented by S-DECOMPOSE and S-CLASH. S-DECOMPOSE decomposes an equation between two terms whose head symbols match. It produces a conjunction of equations between their subterms, namely $\vec{x} = \vec{t}$. Only one of the two terms remains in the original multi-equation, which may thus become standard. The terms $\vec{x}$ are copied—there are two occurrences of $\vec{x}$ on the right-hand side. For this reason, we require them to be type variables, as opposed to terms of arbitrary size. (We slightly abuse notation by using $\vec{x}$ to denote a vector of type variables whose elements are *not* necessarily distinct.) By doing so, we allow explicitly reasoning about *sharing*: since a variable represents a pointer to an equivalence class, we explicitly specify that only *pointers*, not whole terms, are copied. As a result of this decision, S-DECOMPOSE is not applicable when both terms at hand have a nonvariable subterm. S-NAME-1 remedies this problem by introducing a fresh variable that stands for one such subterm. When repeatedly applied, S-NAME-1 yields a unification problem composed of so-called *small terms* only—that is, where sharing has been made fully explicit. S-CLASH complements S-DECOMPOSE by dealing with the case where two terms with different head symbols are equated; in a free tree model, such an equation is false, so failure is signaled. S-SINGLE and S-TRUE suppress multi-equations of size 1 and 0, respectively, which are tautologies. S-SINGLE is restricted to nonvariable terms so as not to break the property that every variable is a member of exactly one multi-equation (Definition 1.8.2). S-CYCLE is the occurs check: that is, it signals failure if the constraint is cyclic. It is applicable only in the case of syntactic unification, that is, when ground types are finite trees. It is a global check: its left-hand side is an entire conjunction of multi-equations. S-FAIL propagates failure; $\mathcal{U}$ ranges over unification constraint contexts.

The constraint rewriting system in Figure 1-11 enjoys the following properties. First, rewriting is strongly normalizing, so the rules define a (nondeterministic) algorithm. Second, rewriting is meaning-preserving. Third, every normal form is either false or of the form $\exists\bar{x}.U$, where U is satisfiable. The latter two properties indicate that the algorithm is indeed a constraint solver.

1.8.4 LEMMA: The rewriting system $\rightarrow$ is strongly normalizing. □

Proof: Let every type constructor F have an integer *weight* $cw(\mathsf{F}) \geq 4 + 2 \times a(\mathsf{F})$ **(1)**. Define the *internal weight* of a type by $iw(\mathtt{X}) = 0$ and $iw(\mathsf{F}\ \vec{\mathtt{T}}) = cw(\mathsf{F}) + \Sigma_{\mathtt{T} \in \vec{\mathtt{T}}} iw(\mathtt{T})$. Define the *weight* of a type by $w(\mathtt{T}) = iw(\mathtt{T}) + 1$ if $\mathtt{T} \in \mathcal{V}$ and $w(\mathtt{T}) = iw(\mathtt{T}) - 2$ otherwise. Then, $iw(\mathtt{T}) + 1 \geq w(\mathtt{T})$ holds for all terms $\mathtt{T}$ **(2)**, and $iw(\mathtt{T}) > w(\mathtt{T}) + 1$ holds for all nonvariable terms $\mathtt{T}$ **(3)**. Define the weight of a multi-equation to be the sum of the weights of its members and the weight of a constraint to be the sum of the weights of its multi-equations. By (1) and (2), S-DECOMPOSE is weight decreasing. By (3), so is S-NAME-1. It is straightforward to check that S-FUSE, S-STUTTER, S-CLASH, S-SINGLE, and S-CYCLE are weight decreasing, while the remaining rules are non-weight increasing. Thus, it is sufficient to check that S-EXAND, S-TRUE, and S-FAIL form a strongly normalizing system, which is immediate. □

1.8.5 LEMMA: $\mathsf{U}_1 \rightarrow \mathsf{U}_2$ implies $\mathsf{U}_1 \equiv \mathsf{U}_2$. □

1.8.6 LEMMA: Every normal form is either false or of the form $\mathcal{X}[\mathsf{U}]$, where $\mathcal{X}$ is an existential constraint context, U is a standard conjunction of multi-equations and, if the model is syntactic, U is acyclic. These conditions imply that U is satisfiable. □

Proof: Let U be a normal form. If U contains false as a subconstraint, then, because U is normal with respect to S-FAIL, U must be false. Let us now assume that U does not contain false. Because U is normal with respect to S-EXAND, it must be of the form $\mathcal{X}[\mathsf{U}']$, where U' is also a normal form and does not contain any existential quantification constructs. Thus, U' must be a (possibly empty) conjunction, whose conjuncts are either true or multi-equations. Because U' is normal with respect to S-TRUE, U' must in fact be a (possibly empty) conjunction of multi-equations. Because U' is normal with respect to S-FUSE, every type variable is a member of at most one multi-equation. Because U' is normal with respect to S-STUTTER, S-DECOMPOSE, S-NAME-1, and S-CLASH, every multi-equation in U' must be standard. Thus, U' is standard. Last, because U' is normal with respect to S-CYCLE, U' must be acyclic if the model is a finite tree model, as opposed to a regular tree model.

Now, in each multi-equation, pick an arbitrary distinguished member, while ensuring that every nondistinguished member is a variable; this is made possible by the fact that every multi-equation is standard. Then, let U'' be the conjunction of multi-equations obtained from U' by replacing every multi-equation $\mathtt{X}_1 = \ldots = \mathtt{X}_n = \mathtt{T}$, where $\mathtt{T}$ is the distinguished member, with the conjunction of binary equations $\mathtt{X}_1 = \mathtt{T} \wedge \ldots \wedge \mathtt{X}_n = \mathtt{T}$. It is not difficult to

check that $\prec_{\mathcal{U}'}$ and $\prec_{\mathcal{U}''}$ coincide; so, when $\mathcal{U}'$ is acyclic, so is $\mathcal{U}''$. Furthermore, because $\mathcal{U}'$ is standard, every equation in $\mathcal{U}''$ has a distinct left-hand side; in other words, $\mathcal{U}''$ is of the form $\vec{\mathrm{X}} = \vec{\mathrm{T}}$. Naturally, $\mathcal{U}'$ and $\mathcal{U}''$ are equivalent, so there only remains to show that $\mathcal{U}''$ is satisfiable.

We now distinguish two cases. First, if the model is a regular tree model, it is a standard result that every constraint of the form $\vec{\mathrm{X}} = \vec{\mathrm{T}}$ is satisfiable (Courcelle, 1983), so $\mathcal{U}''$ is satisfiable. Second, assume that the model is a finite tree model. Then, $\prec_{\mathcal{U}''}$ is acyclic. We have $\mathcal{U}'' \equiv \vec{\mathrm{X}}_1 = \vec{\mathrm{T}}_1 \wedge \vec{\mathrm{X}}_2 = \vec{\mathrm{T}}_2$ **(4)**, where $\mathit{ftv}(\bar{\mathrm{T}}_2) \# \bar{\mathrm{X}}_1\bar{\mathrm{X}}_2$ **(5)**, provided $\vec{\mathrm{X}}_1$ is $\vec{\mathrm{X}}$, $\vec{\mathrm{T}}_1$ is $\vec{\mathrm{T}}$, and $\vec{\mathrm{X}}_2$ and $\vec{\mathrm{T}}_2$ are empty. We now show, by induction on the length of the vectors $\vec{\mathrm{X}}_1$ and $\vec{\mathrm{T}}_1$, that the satisfiability of $\mathcal{U}''$ follows from (4) and (5). In the base case, $\vec{\mathrm{X}}_1$ and $\vec{\mathrm{T}}_1$ are empty, so $\mathcal{U}''$ is equivalent to a constraint of the form $\vec{\mathrm{X}}_2 = \vec{\mathrm{T}}_2$, where $\mathit{ftv}(\bar{\mathrm{T}}_2) \# \bar{\mathrm{X}}_2$, that is, a solved form. By Lemmas 1.3.57 and 1.3.58, $\mathcal{U}''$ is satisfiable. In the inductive case, $\vec{\mathrm{X}}_1 = \vec{\mathrm{T}}_1$ may be written $\vec{\mathrm{X}}'_1 = \vec{\mathrm{T}}'_1 \wedge \mathrm{X}_1 = \mathrm{T}_1$, where X_1 is minimal within $\bar{\mathrm{X}}_1$ with respect to $\prec_{\mathcal{U}''}$, which implies $\bar{\mathrm{X}}_1 \# \mathit{ftv}(\mathrm{T}_1)$. By C-EQ, $\mathcal{U}''$ is equivalent to $\vec{\mathrm{X}}'_1 = \vec{\mathrm{T}}'_1 \wedge (\mathrm{X}_1 = [\vec{\mathrm{X}}_2 \mapsto \vec{\mathrm{T}}_2]\mathrm{T}_1 \wedge \vec{\mathrm{X}}_2 = \vec{\mathrm{T}}_2)$. Furthermore, we have $\mathit{ftv}([\vec{\mathrm{X}}_2 \mapsto \vec{\mathrm{T}}_2]\mathrm{T}_1) \# \bar{\mathrm{X}}_1\bar{\mathrm{X}}_2$. The result then follows from the induction hypothesis. □

1.8.2 A constraint solver

On top of the unification algorithm, we now define a constraint solver. Its specification is independent of the rules and strategy employed by the unification algorithm. However, the structure of the unification algorithm's normal forms, as well as the logical properties of multi-equations, are exploited when performing generalization, that is, when creating and simplifying type schemes. Like the unification algorithm, the constraint solver is specified in terms of a *reduction system*. However, the objects that are subject to rewriting are not just constraints: they have more complex structure. Working with such richer *states* allows distinguishing the solver's external language—namely, the full constraint language, which is used to express the problem that one wishes to solve—and an internal language, introduced below, which is used to describe the solver's private data structures. In the following, C and D range over *external* constraints, that is, constraints that were part of the solver's input. External constraints are to be viewed as abstract syntax trees, subject to no implicit laws other than α-conversion. As a simplifying assumption, we require external constraints not to contain any occurrence of false—otherwise the problem at hand is clearly false. *Internal* data structures include unification constraints $\mathcal{U}$, as previously studied, and *stacks*, whose syntax is

as follows:

$$S ::= [] \mid S[[] \wedge C] \mid S[\exists \bar{X}.[]] \mid S[\mathsf{let}\ x : \forall \bar{X}[[]].T\ \mathsf{in}\ C] \mid S[\mathsf{let}\ x : \sigma\ \mathsf{in}\ []]$$

In the second and fourth productions, C is an external constraint. In the last production, we require σ to be of the form $\forall \bar{X}[U].X$, and we demand $\exists \sigma \equiv \mathsf{true}$. Every stack may be viewed as a one-hole constraint context (page 28): indeed, one may interpret $[]$ as the empty context and $\cdot[\cdot]$ as context composition, which replaces the hole of its first context argument with its second context argument. A stack may also be viewed, literally, as a list of *frames*. Frames may be added and deleted at the inner end of a stack, that is, near the hole of the constraint context that it represents. We refer to the four kinds of frames as *conjunction*, *existential*, let, and *environment* frames, respectively. A *state* of the constraint solver is a triple $S; U; C$, where S is a stack, U is a unification constraint, and C is an external constraint. The state $S; U; C$ is to be understood as a representation of the constraint $S[U \wedge C]$, that is, the constraint obtained by placing both U and C within the hole of the constraint context S. The notion of α-equivalence between states is defined accordingly. In particular, one may rename type variables in $dtv(S)$, provided U and C are renamed as well. In short, the three components of a state play the following roles. C is an external constraint that the solver intends to examine next. U is the internal state of the underlying unification algorithm: one might think of it as the knowledge that has been obtained so far. S tells where the type variables that occur free in U and C are bound, associates type schemes with the program variables that occur free in C, and records what should be done after C is solved. The solver's initial state is usually of the form $[]; \mathsf{true}; C$, where C is the external constraint that one wishes to solve—that is, whose satisfiability one wishes to determine. If the constraint to be solved is of the form $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ C$, and if the type schemes that appear within Γ_0 meet the requirements that bear on environment frames, as defined above, then it is possible to pick $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ []; \mathsf{true}; C$ as an initial state. For simplicity, we make the (unessential) assumption that states have no free type variables.

The solver consists of a (nondeterministic) state rewriting system, given in Figure 1-12. Rewriting is performed modulo α-conversion. S-UNIFY makes the unification algorithm a component of the constraint solver, and allows the current unification problem U to be solved at any time. Rules S-EX-1 to S-EX-4 float existential quantifiers out of the unification problem into the stack, and through the stack up to the nearest enclosing let frame, if there is any, or to the outermost level, otherwise. Their side-conditions prevent capture of type variables, and may always be satisfied by suitable α-conversion of the left-hand state. If $S; U; C$ is a normal form with respect to these five rules, then U must be either false or a conjunction of standard multi-equations,

$$S; U; C \rightarrow S; U'; C \quad \text{if } U \rightarrow U' \qquad \text{(S-UNIFY)}$$

$$S; \exists \bar{X}.U; C \rightarrow S[\exists \bar{X}.[]]; U; C \quad \text{if } \bar{X} \# \mathit{ftv}(C) \qquad \text{(S-EX-1)}$$

$$S[(\exists \bar{X}.[]) \wedge D]; U; C \rightarrow S[\exists \bar{X}.([] \wedge D)]; U; C \quad \text{if } \bar{X} \# \mathit{ftv}(D) \qquad \text{(S-EX-2)}$$

$$S[\text{let } x : \forall \bar{X}[\exists \bar{Y}.[]].T \text{ in } D]; U; C \rightarrow S[\text{let } x : \forall \bar{X}\bar{Y}[].T \text{ in } D]; U; C \quad \text{if } \bar{Y} \# \mathit{ftv}(T) \qquad \text{(S-EX-3)}$$

$$S[\text{let } x : \sigma \text{ in } \exists \bar{X}.[]]; U; C \rightarrow S[\exists \bar{X}.\text{let } x : \sigma \text{ in } []]; U; C \quad \text{if } \bar{X} \# \mathit{ftv}(\sigma) \qquad \text{(S-EX-4)}$$

$$S; U; T_1 = T_2 \rightarrow S; U \wedge T_1 = T_2; \text{true} \qquad \text{(S-SOLVE-EQ)}$$

$$S; U; x \preceq T \rightarrow S; U; S(x) \preceq T \qquad \text{(S-SOLVE-ID)}$$

$$S; U; C_1 \wedge C_2 \rightarrow S[[] \wedge C_2]; U; C_1 \qquad \text{(S-SOLVE-AND)}$$

$$S; U; \exists \bar{X}.C \rightarrow S[\exists \bar{X}.[]]; U; C \quad \text{if } \bar{X} \# \mathit{ftv}(U) \qquad \text{(S-SOLVE-EX)}$$

$$S; U; \text{let } x : \forall \bar{X}[D].T \text{ in } C \rightarrow S[\text{let } x : \forall \bar{X}[[]].T \text{ in } C]; U; D \quad \text{if } \bar{X} \# \mathit{ftv}(U) \qquad \text{(S-SOLVE-LET)}$$

$$S[[] \wedge C]; U; \text{true} \rightarrow S; U; C \qquad \text{(S-POP-AND)}$$

$$S[\text{let } x : \forall \bar{X}[[]].T \text{ in } C]; U; \text{true} \rightarrow S[\text{let } x : \forall \bar{X}X[[]].X \text{ in } C]; U \wedge X = T; \text{true} \quad \text{if } X \notin \mathit{ftv}(U, T) \wedge T \notin \mathcal{V} \qquad \text{(S-NAME-2)}$$

$$S[\text{let } x : \forall \bar{X}Y[[]].X \text{ in } C]; Y = Z = \epsilon \wedge U; \text{true} \rightarrow S[\text{let } x : \forall \bar{X}Y[[]].\theta(X) \text{ in } C]; Y \wedge Z = \theta(\epsilon) \wedge \theta(U); \text{true} \quad \text{if } Y \neq Z \wedge \theta = [Y \mapsto Z] \qquad \text{(S-COMPRESS)}$$

$$S[\text{let } x : \forall \bar{X}Y[[]].X \text{ in } C]; Y = \epsilon \wedge U; \text{true} \rightarrow S[\text{let } x : \forall \bar{X}[[]].X \text{ in } C]; \epsilon \wedge U; \text{true} \quad \text{if } Y \notin X \cup \mathit{ftv}(\epsilon, U) \qquad \text{(S-UNNAME)}$$

$$S[\text{let } x : \forall \bar{X}\bar{Y}[[]].X \text{ in } C]; U; \text{true} \rightarrow S[\exists \bar{Y}.\text{let } x : \forall \bar{X}[[]].X \text{ in } C]; U; \text{true} \quad \text{if } \bar{Y} \# \mathit{ftv}(C) \wedge \exists \bar{X}.U \text{ determines } \bar{Y} \qquad \text{(S-LETALL)}$$

$$S[\text{let } x : \forall \bar{X}[[]].X \text{ in } C]; U_1 \wedge U_2; \text{true} \rightarrow S[\text{let } x : \forall \bar{X}[U_2].X \text{ in } []]; U_1; C \quad \text{if } \bar{X} \# \mathit{ftv}(U_1) \wedge \exists \bar{X}.U_2 \equiv \text{true} \qquad \text{(S-POP-LET)}$$

$$S[\text{let } x : \sigma \text{ in } []]; U; \text{true} \rightarrow S; U; \text{true} \qquad \text{(S-POP-ENV)}$$

Figure 1-12: A constraint solver

and every type variable in $dtv(S)$ must be either universally quantified at a let frame, or existentially bound at the outermost level. (Recall that, by assumption, states have no free type variables.) In other words, provided these rules are applied in an eager fashion, *there is no need for existential frames to appear in the machine representation of stacks*. Instead, it suffices to maintain, at every let frame and at the outermost level, a list of the type variables that are bound at this point; and, conversely, to annotate every type variable in $dtv(S)$ with an integer *rank*, which allows telling, in constant time, where the variable is bound: type variables of rank 0 are bound at the outermost level, and type variables of rank $k \geq 1$ are bound at the k^{th} let frame down in the stack S. The code that accompanies this chapter adopts this convention. Ranks were initially described in (Rémy, 1992a), and also appear in (McAllester, 2003).

Rules S-SOLVE-EQ to S-SOLVE-LET encode an analysis of the structure of the third component of the current state. There is one rule for each possible case, except false, which by assumption cannot arise, and true, which is dealt with further on. S-SOLVE-EQ discovers an equation and makes it available to the unification algorithm. S-SOLVE-ID discovers an instantiation constraint $x \preceq T$ and replaces it with $\sigma \preceq T$, where the type scheme $\sigma = S(x)$ is the type scheme carried by the nearest environment frame that defines x in the stack S. It is defined as follows:

$$\begin{array}{rcll} S[[] \wedge C](x) & = & S(x) & \\ S[\exists \bar{X}.[]](x) & = & S(x) & \text{if } \bar{X} \# \mathit{ftv}(S(x)) \\ S[\mathsf{let}\ y : \forall \bar{X}[[]].T\ \mathsf{in}\ C](x) & = & S(x) & \text{if } \bar{X} \# \mathit{ftv}(S(x)) \\ S[\mathsf{let}\ y : \sigma\ \mathsf{in}\ []](x) & = & S(x) & \text{if } x \neq y \\ S[\mathsf{let}\ x : \sigma\ \mathsf{in}\ []](x) & = & \sigma & \end{array}$$

If $x \in dpi(S)$ does not hold, then $S(x)$ is undefined and the rule is not applicable. If it does hold, then the rule may always be made applicable by suitable α-conversion of the left-hand state. Please recall that, if σ is of the form $\forall \bar{X}[U].X$, where $\bar{X} \# \mathit{ftv}(T)$, then $\sigma \preceq T$ stands for $\exists \bar{X}.(U \wedge X = T)$. The process of constructing this constraint is informally referred to as "taking an instance of σ". In the worst case, it is just as inefficient as textually expanding the corresponding `let` construct in the program's source code, and leads to exponential time complexity (Mairson, Kanellakis, and Mitchell, 1991). In practice, however, the unification constraint U is often compact, because it was simplified before the environment frame $\mathsf{let}\ x : \sigma\ \mathsf{in}\ []$ was created, which explains why the solver usually performs well. (The creation of environment frames, performed by S-POP-LET, is discussed below.) S-SOLVE-AND discovers a conjunction. It arbitrarily chooses to explore the left branch first, and pushes a conjunction frame onto the stack, so as to record that the right branch should be explored afterwards. S-SOLVE-EX discovers an existential

quantifier and enters it, creating a new existential frame to record its existence. Similarly, S-SOLVE-LET discovers a let form and enters its left-hand side, creating a new let frame to record its existence. The choice of examining the left-hand side first is *not* arbitrary. Indeed, examining the right-hand side first would require creating an environment frame—but environment frames must contain *simplified* type schemes of the form $\forall \bar{X}[U].X$, whereas the type scheme $\forall \bar{X}[D].T$ is arbitrary. In other words, our strategy is to simplify type schemes prior to allowing them to be copied by S-SOLVE-ID, so as to avoid any duplication of effort. The side-conditions of S-SOLVE-EX and S-SOLVE-LET may always be satisfied by suitable α-conversion of the left-hand state.

Rules S-SOLVE-EQ to S-SOLVE-LET may be referred to as *forward* rules, because they "move down into" the external constraint, causing the stack to grow. This process stops when the external constraint at hand becomes true. Then, part of the work has been finished, and the solver must examine the stack in order to determine what to do next. This task is performed by the last series of rules, which may be referred to as *backward* rules, because they "move back out", causing the stack to shrink, and possibly scheduling new external constraints for examination. These rules encode an analysis of the structure of the innermost stack frame. There are three cases, corresponding to conjunction, let, and environment frames. The case of existential stack frames need not be considered, because rules S-EX-2 to S-EX-4 allow either fusing them with let frames or floating them up to the outermost level, where they shall remain inert. S-POP-AND deals with conjunction frames. The frame is popped, and the external constraint that it carries is scheduled for examination. S-POP-ENV deals with environment frames. Because the right-hand side of the let construct at hand has been solved—that is, turned into a unification constraint U—it cannot contain an occurrence of x. Furthermore, by assumption, $\exists\sigma$ is true. Thus, this environment frame is no longer useful: it is destroyed. The remaining rules deal with let frames. Roughly speaking, their purpose is to change the state $S[\mathsf{let}\ x : \forall \bar{X}[\,].T\ \mathsf{in}\ C]; U; \mathsf{true}$ into $S[\mathsf{let}\ x : \forall \bar{X}[U].T\ \mathsf{in}\ \,]; \mathsf{true}; C$, that is, to turn the current unification constraint U into a type scheme, turn the let frame into an environment frame, and schedule the right-hand side of the let construct (that is, the external constraint C) for examination. In fact, the process is more complex, because the type scheme $\forall \bar{X}[U].T$ must be *simplified* before becoming part of an environment frame. The simplification process is described by rules S-NAME-2 to S-POP-LET. In the following, we refer to type variables in $\bar{X}$ as *young* and to type variables in $dtv(S) \setminus \bar{X}$ as *old*. The former are the universal quantifiers of the type scheme that is being created; the latter contain its free type variables.

S-NAME-2 ensures that the body T of the type scheme that is being created is a type variable, as opposed to an arbitrary term. If it isn't, then it is

replaced with a fresh variable X, and the equation X = T is added so as to recall that X stands for T. Thus, the rule moves the term T into the current unification problem, where it potentially becomes subject to S-NAME-1. This ensures that sharing is made explicit everywhere. S-COMPRESS determines that the (young) type variable Y is an alias for the type variable Z. Then, every free occurrence of Y other than its defining occurrence is replaced with Z. In an actual implementation, this occurs transparently when the union-find algorithm performs *path compression* (Tarjan, 1975, 1979). We note that the rule does not allow substituting a younger type variable for an older one: indeed, that would make no sense, since the younger variable could then possibly escape its scope. In other words, in implementation terms, the union-find algorithm must be slightly modified so that, in each equivalence class, the representative element is always a type variable with minimum rank. S-UNNAME determines that the (young) type variable Y has no occurrences other than its defining occurrence in the current type scheme. (This occurs, in particular, when S-COMPRESS has just been applied.) Then, Y is suppressed altogether. In the particular case where the remaining multi-equation ϵ has cardinal 1, it may then be suppressed by S-SINGLE. In other words, the combination of S-UNNAME and S-SINGLE is able to suppress young unused type variables as well as the term that they stand for. This may, in turn, cause new type variables to become eligible for elimination by S-UNNAME. In fact, assuming the current unification constraint is acyclic, an inductive argument shows that every young type variable may be suppressed unless it is dominated either by X or by an old type variable. (In the setting of a regular tree model, it is possible to extend the rule so that young cycles that are not dominated either by X or by an old type variable are suppressed as well.) S-LETALL is a directed version of C-LETALL. It turns the young type variables $\bar{Y}$ into old variables. How to tell whether $\exists\bar{X}.U$ determines $\bar{Y}$ is discussed later (see Lemma 1.8.7). Why S-LETALL is an interesting and important rule will be explained shortly. S-POP-LET is meant to be applied when the current state has become a normal form with respect to S-UNIFY, S-NAME-2, S-COMPRESS, S-UNNAME, and S-LETALL, that is, when the type scheme that is about to be created is fully simplified. It splits the current unification constraint into two components U_1 and U_2, where U_1 is made up entirely of *old* variables—as expressed by the side-condition $\bar{X} \# \mathit{ftv}(U_1)$—and U_2 constrains *young* variables only—as expressed by the side-condition $\exists\bar{X}.U_2 \equiv \mathsf{true}$. Please note that U_2 may still contain free occurrences of old type variables, so the type scheme $\forall\bar{X}[U_2].X$ that appears on the right-hand side is not necessarily closed. It is not obvious why such a decomposition must exist; Lemma 1.8.11 proves that it does. Let us say, for now, that S-LETALL plays a role in guaranteeing its existence, whence part of its importance. Once the decomposition $U_1 \wedge U_2$

is obtained, the behavior of S-POP-LET is simple. The unification constraint U_1 concerns old variables only, that is, variables that are not quantified in the current let frame; thus, it need not become part of the new type scheme, and may instead remain part of the current unification constraint. This is justified by C-LETAND and C-INAND* (see the proof of Lemma 1.8.10) and corresponds to the difference between HMX-GEN′ and HMX-GEN discussed in §1.4. The unification constraint U_2, on the other hand, becomes part of the newly built type scheme $\forall \bar{X}[U_2].X$. The property $\exists \bar{X}.U_2 \equiv \mathsf{true}$ guarantees that the newly created environment frame meets the requirements imposed on such frames. Please note that, the more type variables are considered old, the larger U_1 may become, and the smaller U_2. This is another reason why S-LETALL is interesting: by allowing more variables to be considered old, it decreases the size of the type scheme $\forall \bar{X}[U_2].X$, making it cheaper to instantiate.

To complete our description of the constraint solver, there remains to explain how to decide when $\exists \bar{X}.U$ determines $\bar{Y}$, since this predicate occurs in the side-condition of S-LETALL. The following lemma describes two important situations where, by examining the structure of an equation, it is possible to discover that a constraint C *determines* some of its free type variables $\bar{Y}$ (Definition 1.3.44). In the first situation, the type variables $\bar{Y}$ are *equated* with or *dominated* by a distinct type variable X that occurs *free* in C. In that case, because the model is a free tree model, the values of the type variables $\bar{Y}$ are determined by the value of X—they are subtrees of it at specific positions. For instance, $X = Y_1 \to Y_2$ determines $Y_1 Y_2$, while $\exists Y_1.(X = Y_1 \to Y_2)$ determines Y_2. In the second situation, the type variables $\bar{Y}$ are equated with a term T, *all* of whose type variables are *free* in C. Again, the value of the type variables $\bar{Y}$ is then determined by the values of the type variables $ftv(T)$. For instance, $X = Y_1 \to Y_2$ determines X, while $\exists Y_1.(X = Y_1 \to Y_2)$ does not. In the second situation, no assumption is in fact made about the model. Please note that $X = Y_1 \to Y_2$ determines $Y_1 Y_2$ and determines X, but does *not* simultaneously determine $X Y_1 Y_2$.

1.8.7 LEMMA: Let $\bar{X} \# \bar{Y}$. Assume either ϵ is $X = \epsilon'$, where $X \notin \bar{X}\bar{Y}$ and $\bar{Y} \subseteq ftv(\epsilon')$, or ϵ is $\bar{Y} = T = \epsilon'$, where $ftv(T) \# \bar{X}\bar{Y}$. Then, $\exists \bar{X}.(C \wedge \epsilon)$ determines $\bar{Y}$. □

Proof: Let $\bar{X} \# \bar{Y}$ **(1)**. Let $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ \exists \bar{X}.(C \wedge \epsilon)$ **(2)** and $\phi' \models \mathsf{def}\ \Gamma\ \mathsf{in}\ \exists \bar{X}.(C \wedge \epsilon)$ **(3)**, where ϕ and ϕ' coincide outside of $\bar{Y}$. We may assume, *w.l.o.g.*, $\bar{X} \# ftv(\Gamma)$ **(4)**. By (2), (4), CM-EXISTS, and CM-AND, we obtain $\phi_1 \models \mathsf{def}\ \Gamma\ \mathsf{in}\ \epsilon$ **(5)**, where ϕ and ϕ_1 coincide outside $\bar{X}$. By CM-PREDICATE, (5) implies that all members of ϵ have the same image through ϕ_1. Similarly, exploiting (3) and (4), we find that all members of ϵ have the same image through ϕ'_1, where ϕ' and ϕ'_1 coincide outside $\bar{X}$. Now, we claim that ϕ_1 and ϕ'_1 coincide on $\bar{Y}$.

Once the claim is established, by (1), there follows that ϕ and ϕ' must coincide on $\bar{Y}$ as well, which is the goal. So, there only remains to establish the claim; we distinguish two subcases.

Subcase ϵ is $X = \epsilon'$ and $X \notin \bar{X}\bar{Y}$ **(6)** and $\bar{Y} \subseteq ftv(\epsilon')$ **(7)**. Because ϕ_1 and ϕ'_1 coincide outside $\bar{X}\bar{Y}$ and by (6), we have $\phi_1(X) = \phi'_1(X)$. As a result, all members of ϵ' have the same image through ϕ_1 and ϕ'_1. In a free tree model, where decomposition is valid, a simple inductive argument shows that ϕ_1 and ϕ'_1 must coincide on $ftv(\epsilon')$, hence—by (7)—also on $\bar{Y}$.

Subcase ϵ is $\bar{Y} = T = \epsilon'$ and $ftv(T) \# \bar{X}\bar{Y}$ **(8)**. Because ϕ_1 and ϕ'_1 coincide outside $\bar{X}\bar{Y}$ and by (8), we have $\phi_1(T) = \phi'_1(T)$. Thus, for every $Y \in \bar{Y}$, we have $\phi_1(Y) = \phi_1(T) = \phi'_1(T) = \phi'_1(Y)$. That is, ϕ_1 and ϕ'_1 coincide on $\bar{Y}$. □

Thanks to Lemma 1.8.7, an efficient implementation of S-LETALL comes to mind. The problem is, given a constraint $\exists\bar{X}.U$, where U is a standard conjunction of multi-equations, to determine the greatest subset $\bar{Y}$ of $\bar{X}$ such that $\exists(\bar{X} \setminus \bar{Y}).U$ determines $\bar{Y}$. By the first part of the lemma, it is safe for $\bar{Y}$ to include all members of $\bar{X}$ that are directly or indirectly dominated (with respect to U) by some free variable of $\exists\bar{X}.U$. Those can be found, in time linear in the size of U, by a top-down traversal of the graph of $\prec_U$. By the second part of the lemma, it is safe to close $\bar{Y}$ under the closure law $X \in \bar{X} \wedge (\forall Y \quad Y \prec_U X \Rightarrow Y \in \bar{Y}) \Rightarrow X \in \bar{Y}$. That is, it is safe to also include all members of $\bar{X}$ whose descendants (with respect to U) have already been found to be members of $\bar{Y}$. This closure computation may be performed, again in linear time, by a bottom-up traversal of the graph of $\prec_U$. When U is acyclic, it is possible to show that this procedure is complete, that is, does compute the greatest subset $\bar{Y}$ that meets our requirement. This is the topic of the following exercise.

1.8.8 EXERCISE [★★★, ↛]: Assuming U is acyclic, prove that the above procedure computes the greatest subset $\bar{Y}$ of $\bar{X}$ such that $\exists(\bar{X} \setminus \bar{Y}).U$ determines $\bar{Y}$. In the setting of a regular tree model, exhibit a satisfiable constraint U such that the above procedure is incomplete. Can you define a complete procedure in that setting? □

The above discussion has shown that when Y and Z are equated, if Y is young and Z is old, then S-LETALL allows making Y old as well. If binding information is encoded in terms of integer ranks, as suggested earlier, then this remark may be formulated as follows: when Y and Z are equated, if the rank of Y exceeds that of Z, then it may be decreased so that both ranks match. As a result, it is possible to attach ranks with *multi-equations*, rather than with variables. When two multi-equations are fused, the smaller rank is kept. This treatment of ranks is inspired by (Rémy, 1992a): see the resolution rule FUSE, as well as the simplification rules PROPAGATE and REALIZE, in that paper.

S-SOLVE-LET and S-NAME-2 to S-POP-LET are unnecessarily complex when x is assigned a *monotype* T, rather than an arbitrary type scheme $\forall\bar{X}[D].T$. In that case, the combined effect of these rules may be obtained directly via the following two new rules, which may be implemented in a more efficient way:

$$S; U; \mathsf{let}\ x : T\ \mathsf{in}\ C \quad \to \quad S[\exists X.[]]; U \wedge X = T; \mathsf{let}\ x : X\ \mathsf{in}\ C \qquad \text{(S-NAME-2-MONO)}$$
$$\text{if } X \notin \mathit{ftv}(U, T, C) \wedge T \notin \mathcal{V}$$

$$S; U; \mathsf{let}\ x : X\ \mathsf{in}\ C \quad \to \quad S[\mathsf{let}\ x : X\ \mathsf{in}\ []]; U; C \qquad \text{(S-SOLVE-LET-MONO)}$$

If T isn't a variable, it is replaced with a fresh variable X, together with the equation $X = T$. This corresponds to the effect of S-NAME-2. Then, we directly create an environment frame for x, without bothering to create and discard a `let` frame, since there is no way the type scheme X may be further simplified.

Let us now state and establish the properties of the constraint solver. First, the reduction system is terminating, so it defines an algorithm.

1.8.9 LEMMA: The reduction system $\to$ is strongly normalizing. □

Second, every rewriting step preserves the meaning of the constraint that the current state represents. We recall that the state $S; U; C$ is meant to represent the constraint $S[U \wedge C]$.

1.8.10 LEMMA: $S; U; C \to S'; U'; C'$ implies $S[U \wedge C] \equiv S'[U' \wedge C']$. □

Proof: By examination of every rule.

- *Case* S-UNIFY. By Lemma 1.8.5.
- *Case* S-EX-1, S-EX-2, S-SOLVE-EX. By C-EXAND.
- *Case* S-EX-3. By C-LETEX.
- *Case* S-EX-4. By C-INEX.
- *Case* S-SOLVE-EQ, S-POP-AND. By C-DUP.
- *Case* S-SOLVE-ID. Because σ is of the form $\forall\bar{X}[U].X$, we have $\mathit{fpi}(\sigma) = \varnothing$. The result follows by C-INID.
- *Case* S-SOLVE-AND. By C-ANDAND.
- *Case* S-SOLVE-LET. By C-LETAND.
- *Case* S-NAME-2. By Definition 1.3.29 and C-NAMEEQ, $X \notin \mathit{ftv}(U, T)$ implies $\mathsf{true} \Vdash \forall\bar{X}[U].T \equiv \forall\bar{X}X[U \wedge X = T].X$. The result follows by Lemma 1.3.31.
- *Case* S-COMPRESS. Let $\theta = [Y \mapsto Z]$. By Definition 1.3.29 and C-NAMEEQ, $Y \neq Z$ implies $\mathsf{true} \Vdash \forall\bar{X}Y[Y = Z = \epsilon \wedge U].X \equiv \forall\bar{X}Y[Y \wedge Z = \theta(\epsilon) \wedge \theta(U)].\theta(X)$. The result follows by Lemma 1.3.31.

∘ *Case* S-UNNAME. Using Lemma 1.3.26, it is straightforward to check that $Y \notin ftv(\epsilon)$ implies $\exists Y.(Y = \epsilon) \equiv \epsilon$. The result follows by C-EXAND and C-LETEX.

∘ *Case* S-LETALL. By C-LETALL.

∘ *Case* S-POP-LET. By C-LETAND and C-INAND*.

∘ *Case* S-POP-ENV. By C-IN*, recalling that $\exists\sigma$ must be true. □

Last, we classify the normal forms of the reduction system:

1.8.11 LEMMA: A normal form for the reduction system $\rightarrow$ is one of (i) $S; U; x \preceq T$, where $x \notin dpi(S)$; (ii) S; false; true; or (iii) $\mathcal{X}; U$; true, where $\mathcal{X}$ is an existential constraint context and U a satisfiable conjunction of multi-equations. □

Proof: Because, by definition, $S; U$; false is not a valid state, a normal form for S-SOLVE-EQ, S-SOLVE-ID, S-SOLVE-AND, S-SOLVE-EX, and S-SOLVE-LET must be either an instance of the left-hand side of S-SOLVE-ID, with $x \notin dpi(S)$, which corresponds to case (i), or of the form $S; U$; true. Let us consider the latter case. Because $S; U$; true is a normal form with respect to S-UNIFY, by Lemma 1.8.6, U must be either false of the form $\mathcal{X}[U']$, where U' is a standard conjunction of multi-equations and, if the model is syntactic, U' is acyclic. The former case corresponds to (ii); thus, let us consider the latter case. Because $S; \mathcal{X}[U']$; true is a normal form with respect to S-EX-1, the context $\mathcal{X}$ must in fact be empty, and U' is U. If S is an existential constraint context, then we are in situation (iii). Otherwise, because $S; U$; true is a normal form with respect to S-EX-2, S-EX-3, and S-EX-4, the stack S does not end with an existential frame. Because $S; U$; true is a normal form with respect to S-POP-AND and S-POP-ENV, S must then be of the form S'[let $x : \forall\bar{X}[].T$ in C]. Because $S; U$; true is a normal form with respect to S-NAME-2, T must be a type variable X. Let us write U as $U_1 \wedge U_2$, where $\bar{X} \# ftv(U_1)$, and where U_1 is maximal for this criterion. Then, consider a multi-equation $\epsilon \in U$. By the first part of Lemma 1.8.7, if one variable member of ϵ is old (that is, outside $\bar{X}$), then $\exists\bar{X}.U$ determines all other variables in $ftv(\epsilon)$. Because $S; U$; true is a normal form with respect to S-LETALL, all variables in $ftv(\epsilon)$ must then be old (that is, outside $\bar{X}$). By definition of U_1, this implies $\epsilon \in U_1$. By contraposition, for every multi-equation $\epsilon \in U_2$, *all variable members of* ϵ *are in* $\bar{X}$. Furthermore, let us recall that U_2 is a standard conjunction of multi-equations and, if the model is syntactic, U_2 is acyclic. We let the reader establish that this implies $\exists\bar{X}.U_2 \equiv$ true; the proof is a slight generalization of the last part of that of Lemma 1.8.6. Then, $S; U$; true is reducible via S-POP-LET. This is a contradiction, so this last case cannot arise. □

In case (i), the constraint $S[U \wedge C]$ has a free program identifier x, so it is not satisfiable. In other words, the source program contains an unbound

program identifier. Such an error could of course be detected prior to constraint solving, if desired. In case (ii), the unification algorithm failed. By Lemma 1.3.50, the constraint $S[U \wedge C]$ is then false. In case (iii), the constraint $S[U \wedge C]$ is equivalent to $\mathcal{X}[U]$, where U is satisfiable, so it is satisfiable as well. So, each of the three classes of normal forms may be immediately identified as denoting success or failure. Thus, Lemmas 1.8.10 and 1.8.11 indeed prove that the algorithm is a constraint solver.

1.8.12 REMARK: Type inference for ML-the-calculus has been proved NP-hard (Mairson, Kanellakis, and Mitchell, 1991). Thus, our constraint solver cannot run any faster. Mairson *et al.* explain that the cost is essentially due to `let`-polymorphism, which requires a constraint to be duplicated at every occurrence of a `let`-bound variable (S-SOLVE-ID). In order to limit the amount of duplication to a bare minimum, it is important that rule S-LETALL be applied before S-POP-LET, allowing variables and constraints that need not be duplicated to be shared. We have observed that algorithms based on this strategy behave remarkably well in practice (Rémy, 1992a). In fact, McAllester (2003) has proved that they have linear time complexity, provided the size of type schemes and the (left-) nesting depth of `let` constructs are bounded. Unfortunately, many implementations of type inference for ML-the-programming-language do not behave as efficiently as the algorithm presented here. Some spend an excessive amount of time in computing the set of nongeneralizable type variables; some do not treat types as dags, thus losing precious sharing information; others perform the expensive occurs check after every unification step, rather than only once at every `let` construct, as suggested here (S-POP-LET). □

1.8.3 An alternate constraint solver

The constraint solver presented above deals with instantiation constraints in an *eager* way: as soon as one is found, it is rewritten by S-SOLVE-ID into a unification constraint. In other words, instantiation constraints may appear only within *external* constraints; they are not part of the solver's internal data structures. However, it is possible to imagine other constraint solving strategies, where instantiation constraints are longer-lived and may become part of the solver's internal state. Here, we briefly (and somewhat informally) present one such strategy. It is perhaps slightly less efficient, but offers some interesting theoretical properties. In particular, under this alternate strategy, the type schemes found inside environment frames have *no free type variables*, but may have free program variables. In the following, we say that σ is *closed* when $ftv(\sigma)$ is empty. Let us recall that the constraint solver presented above

has the converse property: environment frames contain type schemes of the form $\forall\bar{X}[U].X$, which have no free program variables, but may have free type variables.

The new strategy exploits the following observation about the structure of the constraints produced by the generation rules: in every constraint of the form $\mathsf{let}\ x : \sigma\ \mathsf{in}\ C$, either σ is a monotype T—this occurs when x was λ-bound in the source program—or σ is closed—this occurs when x is a constant or was `let`-bound in the source program. Indeed, the type scheme assigned to a `let`-bound identifier is of the form $\forall X[[\![t : X]\!].X$. Because $ftv([\![t : X]\!])$ is X, this type scheme is closed. Our observation breaks down when terms are allowed to contain type annotations (§1.9); it seems possible to come up with a workaround, which we do not describe here. In the following, we assume that we are able to distinguish between λ-bound program identifiers, on the one hand, and constants and `let`-bound program identifiers, on the other hand; we refer to them as *mono* and *poly* identifiers, respectively.

The idea behind the alternate strategy is to preserve the property that *poly* identifiers have closed type schemes throughout the constraint solving process. More precisely, the new strategy maintains the invariant that all *poly* stack frames are closed. That is, if x is *poly*, then in every state of the form $S_1[\mathsf{let}\ x : \sigma\ \mathsf{in}\ S_2]; U; C$, we have $ftv(\sigma) = \varnothing$, and in every state of the form $S_1[\mathsf{let}\ x : \forall\bar{X}[S_2].T\ \mathsf{in}\ C_1]; U; C_2$, we have $ftv(S_2[U \wedge C_2], T) \subseteq \bar{X}$. We now review how this invariant is violated by the previous constraint solver, and explain how the alternate solver addresses each of these violations.

One important culprit is S-SOLVE-ID, which replaces a program identifier x with a type scheme $S(x)$. If $S(x)$ has free type variables, then any `let` frame comprised between the environment frame that defines x and the inner end of the stack acquires free type variables through the rewriting step. If $S(x)$ is closed, however, the invariant is preserved. Thus, in the alternate constraint solver, S-SOLVE-ID is replaced with the following two rules:

$$S; U; x \preceq T \quad \rightarrow \quad S; U; S(x) \preceq T \qquad \text{(AS-SOLVE-ID-POLY)}$$
$$\text{if } x \text{ is } \textit{poly}$$

$$S; U; x \preceq T \quad \rightarrow \quad S; U \wedge x \preceq T; \mathsf{true} \qquad \text{(AS-SOLVE-ID-MONO)}$$
$$\text{if } x \text{ is } \textit{mono}$$

AS-SOLVE-ID-POLY is identical to S-SOLVE-ID, but is restricted to *poly* identifiers. In that case, according to the invariant, the type scheme $S(x)$ is closed, so the invariant is preserved. The definition of $S(x)$ must be slightly modified to prevent the type scheme's free program variables from capture—an issue that did not arise in the previous solver. In AS-SOLVE-ID-MONO, we

cannot replace x with the monotype $S(x)$, since that might break the invariant; so, the instantiation constraint is kept in symbolic form. For the rule to be well-formed, we must extend the syntax of unification constraints:

$$U ::= \mathsf{true} \mid \mathsf{false} \mid \epsilon \mid U \wedge U \mid \exists \bar{X}.U \mid x \preceq T$$

We should really introduce a letter other than U to range over this new syntactic class. However, this abuse of notation allows us to keep many of the previous constraint solving rules unchanged. Thus, in addition to a system of equations, the constraint solver now maintains a system of unresolved instantiation constraints. To ensure that each of these bears on a distinct program variable, it is possible to introduce the following rule:

$$S; U \wedge x \preceq T_1 \wedge x \preceq T_2; C \quad \rightarrow \quad S; U \wedge x \preceq T_1 \wedge T_1 = T_2; C \qquad \text{(AS-FUSE)}$$

Its logical validity is guaranteed by the fact that x must be *mono*, so $S(x)$ must be a monotype, all of whose instances are equal.

Another reason why the previous constraint solver violates the invariant is S-SOLVE-LET. The `let` frame that it creates is in general not closed, because the current unification constraint U concerns type variables that are not bound in it. The alternate constraint solver works around this issue by *not* pushing U inside the new `let` frame when x is *poly*:

$$S; U; \mathsf{let}\ x : \forall \bar{X}[D].T\ \mathsf{in}\ C \quad \rightarrow \quad S[U \wedge \mathsf{let}\ x : \forall \bar{X}[[]].T\ \mathsf{in}\ C]; \mathsf{true}; D \qquad \text{(AS-SOLVE-LET)}$$
$$\text{if } x \text{ is } \textit{poly}$$

The current unification constraint U is left behind as part of the new stack frame, and we start afresh with an empty unification constraint. There would be no point in keeping U at hand, anyway. Indeed, the type scheme $\forall \bar{X}[D].T$ is closed and will remain so throughout reduction, so the information contained in U cannot affect its reduction in any way. For AS-SOLVE-LET to be well-formed, we must extend the syntax of stacks, so as to be able to store U as part of a `let` frame; we skip the details and (again) continue to let S range over stacks. When x is *mono*, the alternate solver uses S-NAME-2-MONO and S-SOLVE-LET-MONO, as before.

The last reason why the previous constraint solver violates the invariant is S-LETALL. In the alternate solver, this rule is simply removed. The last two rules, namely S-POP-LET and S-POP-ENV, are replaced with the following

three:

$$S[U_1 \wedge \mathsf{let}\ x : \forall \bar{X}[[]].X\ \mathsf{in}\ C]; U_2; \mathsf{true} \rightarrow S[\mathsf{let}\ x : \forall \bar{X}[U_2].X\ \mathsf{in}\ []]; U_1; C \quad \text{if } x \notin \mathit{ftv}(U_1) \qquad \text{(AS-POP-LET)}$$

$$S[\mathsf{let}\ x : \forall \bar{X}[U_1].X\ \mathsf{in}\ []]; U_2; \mathsf{true} \rightarrow S; \exists \bar{X}.U_1 \wedge U_2; \mathsf{true} \quad \text{if } x \text{ is } \mathit{poly} \qquad \text{(AS-POP-ENV-POLY)}$$

$$S[\mathsf{let}\ x : X\ \mathsf{in}\ []]; U; \mathsf{true} \rightarrow S; [x \mapsto X]U; \mathsf{true} \quad \text{if } x \text{ is } \mathit{mono} \qquad \text{(AS-POP-ENV-MONO)}$$

AS-POP-LET simply creates the type scheme $\forall \bar{X}[U_2].X$, which, by the invariant, must be closed. Thus, the invariant is preserved. The unification constraint U_1, which was put away by AS-SOLVE-LET when this `let` frame was created, becomes current again. AS-POP-ENV-POLY discards a *poly* environment frame. The constraint $\exists \bar{X}.U_1$ is not, in general, equivalent to true, so we must keep track of it in order to ensure the rule's logical validity. AS-POP-ENV-MONO discards a *mono* environment frame. If unresolved instantiation constraints bearing on x remain, they are turned into subtyping constraints bearing on X; then, the frame is discarded.

1.8.13 EXERCISE [★★★★, ↛]: Write down a full specification of the alternate constraint solver. Prove that it defines a terminating rewriting system. Prove that it preserves constraint equivalence. Characterize its normal forms. □

1.8.14 EXERCISE [★★★★, ↛]: Implement the alternate constraint solver. □

What is the point of developing such an alternate constraint solver? For one thing, its invariant property—all *poly* stack frames are closed—is strong. It may help reason about the solver's implementation and simplify its internal data structures. For instance, in a state $S; U; C$, every type variable that appears free in U or C must be bound at the nearest enclosing *poly* `let` frame. Thus, keeping track of where type variables are bound becomes very simple; the mutable integer ranks used in the previous solver's implementation are no longer required here. As a related point, in the machine representation of a type scheme, all type variables now have the same status; there is no need to distinguish between free and bound variables. This may help simplify the representation of type schemes as well as the algorithms that simplify them.

It is worth making a couple of points here. First, the ideas developed here are not specific of the setting of first-order unification in a free theory. Indeed, we have been mainly concerned with the treatment of type scheme introduction and instantiation constraints; this aspect is, to some extent, independent of the nature of primitive constraints. In the presence of subtyping, for instance, our two approaches to solving instantiation constraints would still

make sense. Second, it is the expressiveness of the constraint language that makes it easy to come up with these two algorithms—and possibly yet with others—and to reason about their correctness. If our only starting point was Damas and Milner's typing rules, it would perhaps be more difficult to reason about such variations.

The idea of treating instantiation constraints differently, depending on whether the program identifier at hand is *poly* or *mono*, may be traced back to two independent sources (Mitchell, 1996a; Trifonov and Smith, 1996). Mitchell's algorithm PTL implements type inference for DM. It manipulates pairs of the form $(\Gamma \vdash \mathtt{T})$, where Γ is an environment mapping *mono* program identifiers to types and $\mathtt{T}$ is a type; we call such objects *mono* typings. All type variables that appear in a *mono* typing are to be viewed as implicitly universally quantified, so *mono* typings are closed objects. Please note that a type scheme produced by our alternate constraint solver closely corresponds to a *mono* typing: indeed, it is closed, and the conjunction of unresolved instantiation constraints that it contains is a mapping Γ from *mono* program identifiers to types. Mitchell's algorithm accepts a term $\mathtt{t}$, as well as a mapping A from *poly* program identifiers to *mono* typings, and produces a principal *mono* typing for $\mathtt{t}$ under A. The algorithm's definition bears a rather strong resemblance (at least in spirit) with that of our alternate constraint solver. Mitchell's algorithm was later given a more declarative flavor (that is, turned into a set of typing rules) by Chitil, who noticed that such a constraint solving strategy may help interactively trace the source of type errors (Chitil, 2001). Trifonov and Smith independently defined a type system whose expressive power is exactly that of HM(X), but where all types schemes are closed. Again, the idea is to include unresolved instantiation constraints inside type schemes. Their approach makes it somewhat easier to reason about the simplification of subtyping constraints, and was followed by Pottier (2001b).

1.8.4 Reporting type errors

The constraint solvers presented above do not explain type errors: they only answer the question: "is the program well-typed?" with "yes" or "no". The answer is "no" when the underlying unification algorithm produces the constraint false, or when an unbound program identifier is found. The latter kind of error is easy to explain, so we disregard it here. How does one produce a good error message when the constraint at hand reduces to false?

In many compilers, constraint generation and constraint solving are not distinguished. No constraints are explicitly generated: the type inference algorithm analyzes the program's abstract syntax tree and drives the unification algorithm directly. The unification algorithm, in turn, solves equations

in an eager fashion and reports a failure as soon as possible. As a result, when failure is detected, the type inference algorithm is able to pinpoint a program location, namely the current abstract syntax node, and to display an unsatisfiable type equation.

This basic approach has the merit of simplicity, and yields tolerable error messages. Yet, it suffers from several weaknesses. One is that the program location that is current when an inconsistency is first detected may not be that of the actual programming mistake; it may, in fact, be quite far from it. Indeed, an expression that contains a mistake might still be well-typed, albeit usually with a different type than expected. Because ML-the-programming-language programs usually contain very little explicit type information, it might take a great deal of context to determine that the expression's type is incompatible with that of its context. The compiler may then pinpoint an arbitrary location within the context, instead of the erroneous expression itself. Another weakness is that the program location and type equation that form the error message may vary greatly according to the strategy that was chosen to generate and solve constraints. For instance, generating $[\![\mathtt{t}_1 : \mathtt{T}_1]\!] \wedge [\![\mathtt{t}_2 : \mathtt{T}_2]\!]$ instead of $[\![\mathtt{t}_2 : \mathtt{T}_2]\!] \wedge [\![\mathtt{t}_1 : \mathtt{T}_1]\!]$, or $[\![\mathtt{t} : \mathtt{T}]\!]$ instead of $\exists \mathtt{X}.([\![\mathtt{t} : \mathtt{X}]\!] \wedge \mathtt{X} = \mathtt{T})$, may lead to dramatically different error messages. Such phenomena have been studied *e.g.* in (McAdam, 1998; Lee and Yi, 1998).

In the literature, the issue has been approached from two somewhat different angles. One is, given a failed constraint solving run, to try and produce an explanation of the failure (Wand, 1986; Beaven and Stansifer, 1993; Duggan and Bent, 1996), or a set of program points involved in it (Flanagan, Flatt, Krishnamurthi, Weirich, and Felleisen, 1996; Choppella, 2002; Haack and Wells, 2003). This usually involves annotating constraints with information about their origin. Some authors attempt to determine which, among the program points that participate in the type error, are more likely to contain the actual programming mistake (Johnson and Walz, 1986). The second angle is allow the user to understand the program's type derivation, by interactively exploring it. Tools that allow determining the type of a program fragment *in isolation* are useful for this purpose (Bernstein and Stark, 1995; Chitil, 2001). The development of such tools is made somewhat difficult by the fact that ML-the-type-system does not have *principal typings* (Jim, 1995).

We lack space to give a more in-depth treatment of this issue. Let us simply propose two remarks. First, it is possible to develop *several* constraint solvers, one geared towards efficiency, the other towards good error reports. These two solvers may employ different strategies and different data structures. Correct programs may be accepted quickly by first running the former. If it fails, then time is less of the essence, and the latter may be run to obtain a good error message. Second, there is no reason why a constraint solver

should stop as soon as failure is detected. On the contrary, we have pointed out above that this design choice makes the error report dependent on the strategy that was followed. We suggest that, in order to make the error report independent of the strategy, the unification algorithm should report not only the *first* cause of failure, but *all* of them. Let us remove the rules S-CLASH and S-CYCLE, because, by replacing the cause of an error with false, they discard crucial information. Then, a type error manifests itself either as an inconsistent multi-equation (one that contains two incompatible nonvariable terms) or as a cycle in the type structure. It is possible to show that, up to a few simple structural equivalence rules, namely commutativity and associativity of conjunction and exchange of existential quantifiers, the algorithm is *confluent*. In other words, all constraint solving strategies lead to the same normal form. In general, a normal form contains a number of type errors, all of which may be presented to the user. The fact that the constraint rewriting system remains confluent, even in the presence of failures, guarantees that error reports are independent of the constraint solving strategy.

1.9 From ML-the-calculus to ML-the-programming-language

In this section, we explain how to extend the framework developed so far to accommodate operations on values of base type (such as integers), pairs, sums, references, and recursive function definitions. We explain how to combine such extensions when they are independent of one another. Then, we describe more complex extensions, namely algebraic data type definitions, pattern matching, and type annotations. Last, the issues associated with recursive types are briefly discussed. Exceptions are not discussed; the reader is referred to (TAPL Chapter 14).

1.9.1 Simple extensions

Introducing new constants and extending $\xrightarrow{\delta}$ and Γ_0 appropriately allows adding many features of ML-the-programming-language to ML-the-calculus. In each case, it is necessary to check that the requirements of Definition 1.7.7 are met, that is, to ensure that the new initial environment faithfully reflects the nature of the new constants as well as the behavior of the new reduction rules. Below, we describe several such extensions in isolation. The first exercise in the series establishes a technical result that is useful in the next exercises.

1.9.1 EXERCISE [RECOMMENDED, ★]: Let Γ_0 contain the binding $\mathtt{c} : \forall \bar{\mathtt{X}}.\mathtt{T}_1 \to \ldots \to \mathtt{T}_n \to \mathtt{T}$. Prove that let Γ_0 in $[\![\mathtt{c}\ \mathtt{t}_1\ \ldots\ \mathtt{t}_n : \mathtt{T}']\!]$ is equivalent to

$\mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists\bar{X}.(\bigwedge_{i=1}^{n}[\![t_i : T_i]\!] \wedge T \leq T')$. □

1.9.2 EXERCISE [INTEGERS, RECOMMENDED, ★★]: Integer literals and integer addition have been introduced and given an operational semantics in Examples 1.2.1, 1.2.2 and 1.2.4. Let us now introduce an isolated type constructor int of signature $\star$ and extend the initial environment Γ_0 with the bindings $\hat{n} : \mathsf{int}$, for every integer n, and $\hat{+} : \mathsf{int} \to \mathsf{int} \to \mathsf{int}$. Check that these definitions meet the requirements of Definition 1.7.7. □

1.9.3 EXERCISE [BOOLEANS, RECOMMENDED, ★★, ↛]: Booleans and conditionals have been introduced and given an operational semantics in Exercise 1.2.6. Introduce an isolated type constructor bool to represent Boolean values and explain how to extend the initial environment. Check that your definitions meet the requirements of Definition 1.7.7. What is the constraint generation rule for the syntactic sugar $\mathtt{if}\ t_0\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2$? □

1.9.4 EXERCISE [PAIRS, ★★, ↛]: Pairs and pair projections have been introduced and given an operational semantics in Examples 1.2.3 and 1.2.5. Let us now introduce an isolated type constructor $\times$ of signature $\star \otimes \star \Rightarrow \star$, covariant in both of its parameters, and extend the initial environment Γ_0 with the following bindings:

$$\begin{array}{rl} (\cdot,\cdot): & \forall XY.X \to Y \to X \times Y \\ \pi_1 : & \forall XY.X \times Y \to X \\ \pi_2 : & \forall XY.X \times Y \to Y \end{array}$$

Check that these definitions meet the requirements of Definition 1.7.7. □

1.9.5 EXERCISE [SUMS, ★★, ↛]: Sums have been introduced and given an operational semantics in Example 1.2.7. Let us now introduce an isolated type constructor $+$ of signature $\star \otimes \star \Rightarrow \star$, covariant in both of its parameters, and extend the initial environment Γ_0 with the following bindings:

$$\begin{array}{rl} \mathtt{inj}_1 : & \forall XY.X \to X + Y \\ \mathtt{inj}_2 : & \forall XY.Y \to X + Y \\ \mathtt{case}: & \forall XYZ.(X + Y) \to (X \to Z) \to (Y \to Z) \to Z \end{array}$$

Check that these definitions meet the requirements of Definition 1.7.7. □

1.9.6 EXERCISE [REFERENCES, ★★★]: References have been introduced and given an operational semantics in Example 1.2.9. The type constructor ref has been introduced in Definition 1.7.4. Let us now extend the initial environment Γ_0 with the following bindings:

$$\begin{array}{rl} \mathtt{ref} : & \forall X.X \to \mathsf{ref}\ X \\ ! : & \forall X.\mathsf{ref}\ X \to X \\ := : & \forall X.\mathsf{ref}\ X \to X \to X \end{array}$$

Check that these definitions meet the requirements of Definition 1.7.7. □

1.9.7 EXERCISE [RECURSION, RECOMMENDED, ★★★]: The fixpoint combinator `fix` has been introduced and given an operational semantics in Example 1.2.10. Let us now extend the initial environment Γ_0 with the following binding:

$$\texttt{fix}: \quad \forall \texttt{XY}.((\texttt{X} \to \texttt{Y}) \to (\texttt{X} \to \texttt{Y})) \to \texttt{X} \to \texttt{Y}$$

Check that these definitions meet the requirements of Definition 1.7.7. Recall how the `letrec` syntactic sugar was defined in Example 1.2.10, and check that this gives rise to the following constraint generation rule:

$$\begin{array}{ll} & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\texttt{letrec f} = \lambda\texttt{z}.\texttt{t}_1\ \texttt{in}\ \texttt{t}_2 : \texttt{T}]\!] \\ \equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \mathsf{let}\ \texttt{f} : \forall \texttt{XY}[\mathsf{let}\ \texttt{f} : \texttt{X} \to \texttt{Y}; \texttt{z} : \texttt{X}\ \mathsf{in}\ [\![\texttt{t}_1 : \texttt{Y}]\!]].\texttt{X} \to \texttt{Y}\ \mathsf{in}\ [\![\texttt{t}_2 : \texttt{T}]\!] \end{array}$$

Note the somewhat peculiar structure of this constraint: the program variable `f` is bound twice in it, with different type schemes. The constraint requires all occurrences of `f` within $\texttt{t}_1$ to be assigned the *monomorphic* type $\texttt{X} \to \texttt{Y}$. This type is generalized and turned into a type scheme before inspecting $\texttt{t}_2$, however, so every occurrence of `f` within $\texttt{t}_2$ may receive a different type, as usual with `let`-polymorphism. A more powerful way of typechecking recursive function definitions is discussed in §1.10 (page 149). □

1.9.2 Combining independent extensions

Type soundness has been proved for integers, booleans, pairs, *etc.* but when added to ML-the-calculus, independently. What happens when there are taken altogether? What happens if, for instance, integers must later be extended with an exponential primitive? Of course, some conditions should be set before this question even makes sense. A constant could certainly not belong to both extensions and be assigned incompatible types! Even two extensions with disjoint set of constants cannot be safely combined, in general. For instance, adding a new constructor to represent infinity to integers will not work magically, without redefining the all operations on integers so that they can also treat the infinity. Thus, all requirements of Definition 1.7.7 must be checked again, and it is unclear whether any part can be reused at all. In fact, even extensions that are a priori orthogonal do raise problem in principle. Consider two constants, id of arity 1 with reduction rule $\mathsf{id}\ \texttt{v} \xrightarrow{\delta} \texttt{v}$ and typing assumption $\mathsf{id} : \forall \texttt{X}[].\texttt{X} \to \texttt{X}$, and apply of arity 2 with reduction rule $\mathsf{apply}\ \texttt{v}\ \texttt{v} \xrightarrow{\delta} \texttt{v}$ and typing assumption $\mathsf{id} : \forall \texttt{XY}[].\texttt{X} \to \texttt{Y} \to \texttt{X}$. Extending ML-the-calculus with either id or apply is safe and so is the extension with both of them. However, the later cannot be formally deduced from the former.

1.9.8 EXERCISE [★, ↛]: Check that id and apply satisfy the Definition 1.7.7 both when taken independently and when taken simultaneously. □

The problem is that the two proofs taken independently will never consider the program apply id id. This program is actually stuck when combining the extensions per say (by taking the union of the reduction rules), but reduces to id in the extension that consider both constants simultaneously. The problem is that requirements of Definition 1.7.7 must interpret the meta-variable v in the current extension, while some form of open-world interpretation is needed for an extension to be compositional.

1.9.3 Algebraic data types

Exercises 1.9.4 and 1.9.5 have shown how to extend the language with binary, anonymous products and sums. These constructs are quite general, but still have several shortcomings. First, they are only binary, while we would like to have k-ary products and sums, for arbitrary $k \geq 0$. Such a generalization is of course straightforward. Second, more interestingly, their components must be referred to by numeric index (as in "please extract the *second* component of the pair"), rather than by name ("extract the component named y"). In practice, it is crucial to use names, because they make programs more readable and more robust in the face of changes. One could introduce a mechanism that allows defining names as syntactic sugar for numeric indices. That would help a little, but not much, because these names would not appear in *types*, which would still be made of anonymous products and sums. Third, in the absence of recursive types, products and sums do not have sufficient expressiveness to allow defining unbounded data structures, such as lists. Indeed, it is easy to see that every value whose type T is composed of base types (int, bool, *etc.*), products, and sums must have bounded size, where the bound $|T|$ is a function of T. More precisely, up to a constant factor, we have $|\mathsf{int}| = |\mathsf{bool}| = 1$, $|T_1 \times T_2| = 1 + |T_1| + |T_2|$, and $|T_1 + T_2| = 1 + \max(|T_1|, |T_2|)$. The following example describes another facet of the same problem.

1.9.9 EXAMPLE: A list is either empty, or a pair of an element and another list. So, it seems natural to try and encode the type of lists as a sum of some arbitrary type (say, unit), on the one hand, and of a product of some element type and of the type of lists itself, on the other hand. With this encoding in mind, we can go ahead and write code—for instance, a function that computes the length of a list:

$$\mathtt{letrec\ length} = \lambda \mathtt{l.case\ l}\ (\lambda\_.\hat{0})\ (\lambda \mathtt{z}.\hat{1} \mathbin{\hat{+}} \mathtt{length}\ (\pi_2\ \mathtt{z}))$$

We have used integers, pairs, sums, and the `letrec` construct introduced in the previous section. The code analyzes the list `l` using a `case` construct. If the left branch is taken, the list is empty, so 0 is returned. If the right branch is taken, then `z` becomes bound to a pair of some element and the tail of the list. The latter is obtained using the projection operator π_2. Its length is computed using a recursive call to `length` and incremented by 1. This code makes perfect sense. However, applying the constraint generation and constraint solving algorithms eventually leads to an equation of the form $\mathtt{X} = \mathtt{Y} + (\mathtt{Z} \times \mathtt{X})$, where `X` stands for the type of `l`. This equation accurately reflects our encoding of the type of lists. However, in a syntactic model, it has no solution, so our definition of `length` is ill-typed. It is possible to adopt a free regular tree model,thus introducing *equirecursive* types into the system (TAPL Chapter 20); however, there are good reasons not to do so (§1.9.6). □

To work around this problem, ML-the-programming-language offers *algebraic data type* definitions, whose elegance lies in the fact that, while representing only a modest theoretical extension, they do solve the three problems mentioned above. An algebraic data type may be viewed as an *abstract type* that is declared to be *isomorphic* to a (k-ary) product or sum type with named components. The type of each component is declared as well, and may refer to the algebraic data type that is being defined: thus, algebraic data types are *isorecursive* (TAPL Chapter 20). In order to allow sufficient flexibility when declaring the type of each component, algebraic data type definitions may be *parameterized* by a number of type variables. Last, in order to allow the description of complex data structures, it is necessary to allow several algebraic data types to be defined at once; the definitions may then be *mutually recursive*. In fact, in order to simplify this formal presentation, we assume that *all* algebraic data types are defined at once at the beginning of the program. This decision is of course at odds with modular programming, but will not otherwise be a problem.

In the following, `D` ranges over a set of *data types*. We assume that data types form a subset of type constructors. We require each of them to be isolated and to have a signature of the form $\vec{\kappa} \Rightarrow \star$. Furthermore, ℓ ranges over a set $\mathcal{L}$ of *labels*, which we use indifferently as *data constructors* and as *record labels*. An *algebraic data type definition* is either a *variant type* definition or a *record type* definition, whose respective forms are

$$\mathtt{D}\,\vec{\mathtt{X}} \approx \sum_{i=1}^{k} \ell_i : \mathtt{T}_i \qquad \text{and} \qquad \mathtt{D}\,\vec{\mathtt{X}} \approx \prod_{i=1}^{k} \ell_i : \mathtt{T}_i.$$

In either case, k must be nonnegative. If $\mathtt{D}$ has signature $\vec{\kappa} \Rightarrow \star$, then the type variables $\vec{X}$ must have kind $\vec{\kappa}$. Every $\mathtt{T}_i$ must have kind $\star$. We refer to $\vec{X}$ as the *parameters* and to $\vec{\mathtt{T}}$ (the vector formed by $\mathtt{T}_1, \ldots, \mathtt{T}_k$) as the *components* of the definition. The parameters are bound within the components, and the definition must be closed, that is, $ftv(\vec{\mathtt{T}}) \subseteq \vec{X}$ must hold. Last, for an algebraic data type definition to be valid, the behavior of the type constructor $\mathtt{D}$ with respect to subtyping must match its definition. This requirement is clarified below.

1.9.10 DEFINITION: Consider an algebraic data type definition whose parameters and components are respectively $\vec{X}$ and $\vec{\mathtt{T}}$. Let $\vec{X}'$ and $\vec{\mathtt{T}}'$ be their images under an arbitrary renaming. Then, $\mathtt{D}\,\vec{X} \leq \mathtt{D}\,\vec{X}' \Vdash \vec{\mathtt{T}} \leq \vec{\mathtt{T}}'$ must hold. □

Because it is stated in terms of an entailment assertion, the above requirement bears on the interpretation of subtyping. The idea is, since $\mathtt{D}\,\vec{X}$ is declared to be isomorphic to (a sum or a product of) $\vec{\mathtt{T}}$, whenever two types built with $\mathtt{D}$ are comparable, their unfoldings should be comparable as well. The reverse entailment assertion is not required for type soundness, and it is sometimes useful to declare algebraic data types that do not validate it—so-called *phantom types* (Fluet and Pucella, 2002). Note that the requirement may always be satisfied by making the type constructor $\mathtt{D}$ *invariant* in all of its parameters. Indeed, in that case, $\mathtt{D}\,\vec{X} \leq \mathtt{D}\,\vec{X}'$ entails $\vec{X} = \vec{X}'$, which must entail $\vec{\mathtt{T}} = \vec{\mathtt{T}}'$ since $\vec{\mathtt{T}}'$ is precisely $[\vec{X} \mapsto \vec{X}']\vec{\mathtt{T}}$. In an equality free tree model, every type constructor is naturally invariant, so the requirement is trivially satisfied. In other settings, however, it is often possible to satisfy the requirement of Definition 1.9.10 while assigning $\mathtt{D}$ a less restrictive variance. The following example illustrates such a case.

1.9.11 EXAMPLE: Let list be a data type of signature $\star \Rightarrow \star$. Let $\mathtt{Nil}$ and $\mathtt{Cons}$ be data constructors. Then, the following is a definition of list as a variant type:

$$\mathsf{list}\,\mathtt{X} \approx \Sigma\,(\mathtt{Nil} : \mathsf{unit}; \mathtt{Cons} : \mathtt{X} \times \mathsf{list}\,\mathtt{X})$$

Because data types form a subset of type constructors, it is valid to form the type $\mathsf{list}\,\mathtt{X}$ in the right-hand side of the definition, even though we are still in the process of defining the meaning of list. In other words, data type definitions may be recursive. However, because $\approx$ is not interpreted as equality, the type $\mathsf{list}\,\mathtt{X}$ is *not* a recursive type: it is nothing but an application of the unary type constructor list to the type variable $\mathtt{X}$. To check that the definition of list satisfies the requirement of Definition 1.9.10, we must ensure that

$$\mathsf{list}\,\mathtt{X} \leq \mathsf{list}\,\mathtt{X}' \Vdash \mathsf{unit} \leq \mathsf{unit} \wedge \mathtt{X} \times \mathsf{list}\,\mathtt{X} \leq \mathtt{X}' \times \mathsf{list}\,\mathtt{X}'$$

holds. This assertion is equivalent to $\mathsf{list}\,X \leq \mathsf{list}\,X' \Vdash X \leq X'$. To satisfy the requirement, it is sufficient to make list a *covariant* type constructor, that is, to define subtyping in the model so that $\mathsf{list}\,X \leq \mathsf{list}\,X' \equiv X \leq X'$ holds.

Let tree be a data type of signature $\star \Rightarrow \star$. Let `root` and `sons` be record labels. Then, the following is a definition of tree as a record type:

$$\mathsf{tree}\,X \approx \Pi(\texttt{root} : X; \texttt{sons} : \mathsf{list}\,(\mathsf{tree}\,X))$$

This definition is again recursive, and relies on the previous definition. Because list is covariant, it is straightforward to check that the definition of tree is valid if tree is made a covariant type constructor as well. □

1.9.12 EXERCISE [★★★, ↛]: Consider a nonrecursive algebraic data type definition, where the variance of every type constructor that appears on the right-hand side is known. Can you systematically determine, for each of the parameters, the least restrictive variance that makes the definition valid? Generalize this procedure to the case of recursive and mutually recursive algebraic data type definitions. □

A *prologue* is a set of algebraic data type definitions, where each data type is defined at most once and where each data constructor or record label appears at most once. A *program* is a pair of a prologue and an expression. The effect of a prologue is to enrich the programming language with new constants. That is, a variant type definition extends the operational semantics with several injections and a `case` construct, as in Example 1.2.7. A record type definition extends it with a record formation construct and several projections, as in Examples 1.2.3 and 1.2.5. In either case, the initial typing environment Γ_0 is extended with information about these new constants. Thus, algebraic data type definitions might be viewed as a simple configuration language that allows specifying in which instance of ML-the-calculus the expression that follows the prologue should be typechecked and interpreted. Let us now give a precise account of this phenomenon.

To begin, suppose the prologue contains the definition $\mathtt{D}\,\vec{X} \approx \sum_{i=1}^{k} \ell_i : \mathtt{T}_i$. Then, for each $i \in \{1, \ldots, k\}$, a constructor of arity 1, named ℓ_i, is introduced. Furthermore, a destructor of arity $k + 1$, named $\mathtt{case}_D$, is introduced. When $k > 0$, it is common to write $\mathtt{case}\ t\ [\ell_i : t_i]_{i=1}^{k}$ for the application $\mathtt{case}_D\ t\ t_1\ \ldots\ t_n$. The operational semantics is extended with the following reduction rules, for $i \in \{1, \ldots, k\}$:

$$\mathtt{case}\ (\ell_i\ v)\ [\ell_j : v_j]_{j=1}^{k} \xrightarrow{\delta} v_i\ v \qquad \text{(R-ALG-CASE)}$$

For each $i \in \{1, \ldots, k\}$, the initial environment is extended with the binding $\ell_i : \forall \bar{X}.\mathtt{T}_i \to \mathtt{D}\,\vec{X}$. It is further extended with the binding $\mathtt{case}_D : \forall \bar{X} Z.\mathtt{D}\,\vec{X} \to (\mathtt{T}_1 \to Z) \to \ldots (\mathtt{T}_k \to Z) \to Z$.

Now, suppose the prologue contains the definition $\mathtt{D}\,\vec{X} \approx \prod_{i=1}^{k} \ell_i : \mathtt{T}_i$. Then, for each $i \in \{1, \ldots, k\}$, a destructor of arity 1, named ℓ_i, is introduced. Furthermore, a constructor of arity k, named $\mathtt{make}_D$, is introduced. It is common to write $\mathtt{t}.\ell$ for the application $\ell\ \mathtt{t}$ and, when $k > 0$, to write $\{\ell_i = \mathtt{t}_i\}_{i=1}^{k}$ for the application $\mathtt{make}_D\ \mathtt{t}_1\ \ldots\ \mathtt{t}_k$. The operational semantics is extended with the following reduction rules, for $i \in \{1, \ldots, k\}$:

$$(\{\ell_j = v_j\}_{j=1}^{k}).\ell_i \xrightarrow{\delta} v_i \qquad \text{(R-ALG-PROJ)}$$

For each $i \in \{1, \ldots, k\}$, the initial environment is extended with the binding $\ell_i : \forall\vec{X}.D\,\vec{X} \to \mathtt{T}_i$. It is further extended with the binding $\mathtt{make}_D : \forall\vec{X}.\mathtt{T}_1 \to \ldots \to \mathtt{T}_k \to D\,\vec{X}$.

1.9.13 EXAMPLE: The effect of defining list (Example 1.9.11) is to make `Nil` and `Cons` data constructors of arity 1 and to introduce a binary destructor $\mathtt{case}_{\mathsf{list}}$. The definition also extends the initial environment as follows:

$$\begin{array}{rl} \mathtt{Nil}: & \forall X.\mathsf{unit} \to \mathsf{list}\,X \\ \mathtt{Cons}: & \forall X.X \times \mathsf{list}\,X \to \mathsf{list}\,X \\ \mathtt{case}_{\mathsf{list}}: & \forall XZ.\mathsf{list}\,X \to (\mathsf{unit} \to Z) \to (X \times \mathsf{list}\,X \to Z) \to Z \end{array}$$

Thus, the value $\mathtt{Cons}(\hat{0}, \mathtt{Nil}())$, an integer list of length 1, has type list int. A function that computes the length of a list may now be written as follows:

$$\mathtt{letrec\ length} = \lambda \mathtt{l.case\ l\ [Nil} : \lambda\_.\hat{0} \mid \mathtt{Cons} : \lambda \mathtt{z}.\hat{1} \mathbin{\hat{+}} \mathtt{length}\,(\pi_2\ \mathtt{z})\mathtt{]}$$

Recall that this notation is syntactic sugar for

$$\mathtt{letrec\ length} = \lambda \mathtt{l.case}_{\mathsf{list}}\ \mathtt{l}\ (\lambda\_.\hat{0})\ (\lambda \mathtt{z}.\hat{1} \mathbin{\hat{+}} \mathtt{length}\,(\pi_2\ \mathtt{z}))$$

The difference with the code in Example 1.9.9 appears minimal: the `case` construct is now annotated with the data type list. As a result, the type inference algorithm employs the type scheme assigned to $\mathtt{case}_{\mathsf{list}}$, which is derived from the definition of list, instead of the type scheme assigned to the anonymous `case` construct, given in Exercise 1.9.5. This is good for a couple of reasons. First, the former is more informative than the latter, because it contains the type $\mathtt{T}_i$ associated with the data constructor ℓ_i. Here, for instance, the generated constraint requires the type of `z` to be $\mathtt{X} \times \mathsf{list}\,\mathtt{X}$ for some `X`, so a good error message would be given if a mistake was made in the second branch, such as omitting the use of π_2. Second, and more fundamentally, *the code is now well-typed*, even in the absence of recursive types. In Example 1.9.9, a cyclic equation was produced because `case` required the type of `l` to be a sum type and because a sum type carries the types of its left and right branches as subterms. Here, instead, $\mathtt{case}_{\mathsf{list}}$ requires `l` to have

type $\mathsf{list}\,X$ for some X. This is an abstract type: it does not explicitly contain the types of the branches. As a result, the generated constraint no longer involves a cyclic equation. It is, in fact, satisfiable; the reader may check that `length` has type $\forall X.\mathsf{list}\,X \to \mathsf{int}$, as expected. □

Example 1.9.13 stresses the importance of using *declared, abstract* types, as opposed to *anonymous, concrete* sum or product types, in order to obviate the need for recursive types. The essence of the trick lies in the fact that the type schemes associated with operations on algebraic data types implicitly *fold* and *unfold* the data type's definition. More precisely, let us recall the type scheme assigned to the i^{th} injection in the setting of (k-ary) anonymous sums: it is $\forall X_1 \ldots X_k.X_i \to X_1 + \ldots + X_k$, or, more concisely, $\forall X_1 \ldots X_k.X_i \to \sum_{i=1}^{k} X_i$. By instantiating each X_i with T_i and generalizing again, we find that a more specific type scheme is $\forall \vec{X}.T_i \to \sum_{i=1}^{k} T_i$. Perhaps this could have been the type scheme assigned to ℓ_i? Instead, however, it is $\forall \vec{X}.T_i \to D\,\vec{X}$. We now realize that the latter type scheme not only reflects the operational behavior of the i^{th} injection, but also *folds* the definition of the algebraic data type D by turning the anonymous sum $\sum_{i=1}^{k} T_i$—which forms the definition's right-hand side—into the parameterized abstract type $D\,\vec{X}$—which is the definition's left-hand side. Conversely, the type scheme assigned to $\mathtt{case}_D$ *unfolds* the definition. The situation is identical in the case of record types: in either case, *constructors fold, destructors unfold.* In other words, occurrences of data constructors and record labels in the code may be viewed as explicit instructions for the typechecker to fold or unfold an algebraic data type definition. This mechanism is characteristic of *isorecursive* types.

1.9.14 EXERCISE [★, ↛]: For a fixed k, check that all of the machinery associated with k-ary anonymous products—that is, constructors, destructors, reduction rules, and extensions to the initial typing environment—may be viewed as the result of a single algebraic data type definition. Conduct a similar check in the case of k-ary anonymous sums. □

1.9.15 EXERCISE [★★★, ↛]: Check that the above definitions meet the requirements of Definition 1.7.7. □

1.9.16 EXERCISE [★★★, ↛]: For the sake of simplicity, we have assumed that all data constructors have arity one. If desired, it is possible to accept variant data type definitions of the form $D\,\vec{X} \approx \sum_{i=1}^{k} \ell_i : \vec{T}_i$, where the arity of the data constructor ℓ_i is the length of the vector $\vec{T}_i$, and may be an arbitrary nonnegative integer. This allows, for instance, altering the definition of list so that the data constructors `Nil` and `Cons` are respectively nullary and binary. Make the necessary changes in the above definitions and check that the requirements of Definition 1.7.7 are still met. □

In this formal presentation of algebraic data types, we have assumed that all algebraic data type definitions are known before the program is type-checked. This simplifying assumption is forced on us by the fact that we interpret constraints in a fixed model, that is, we assume a fixed universe of types. In practice, programming languages have *module systems*, which allow distinct modules to have distinct, partial views of the universe of types. Then, it becomes possible for each module to come with its own data type definitions. Interestingly, it is even possible, in principle, to split the definition of a single data type over several modules, yielding *extensible* algebraic data types. For instance, module A might declare the existence of a parameterized variant type $\mathtt{D}\,\vec{\mathtt{X}}$, without giving its components. Later on, module B might define a component $\ell : \mathtt{T}$, where $ftv(\mathtt{T}) \subseteq \vec{\mathtt{X}}$. Such a definition makes ℓ a unary constructor with type scheme $\forall\vec{\mathtt{X}}.\mathtt{T} \to \mathtt{D}\,\vec{\mathtt{X}}$, as before. It becomes impossible, however, to introduce a destructor $\mathtt{case}_\mathtt{D}$, because the definition of an extensible variant type can never be assumed to be complete—other, unknown modules might extend it further. To compensate for its absence, one may supplement every constructor ℓ with a destructor ℓ^{-1}, whose semantics is given by $\ell^{-1}\ (\ell\ \mathtt{v})\ \mathtt{v}_1\ \mathtt{v}_2 \xrightarrow{\delta} \mathtt{v}_1\ \mathtt{v}$ and $\ell^{-1}\ (\ell'\ \mathtt{v})\ \mathtt{v}_1\ \mathtt{v}_2 \xrightarrow{\delta} \mathtt{v}_2\ (\ell'\ \mathtt{v})$ when $\ell \neq \ell'$, and whose type scheme is $\forall\vec{\mathtt{X}}\mathtt{Z}.\mathtt{D}\,\vec{\mathtt{X}} \to (\mathtt{T} \to \mathtt{Z}) \to (\mathtt{D}\,\vec{\mathtt{X}} \to \mathtt{Z}) \to \mathtt{Z}$. When pattern matching is available, ℓ^{-1} may in fact be defined in the language. ML-the-programming-language does not offer extensible algebraic data types as a language feature, but does have one built-in extensible variant type, namely the type `exn` of exceptions. Thus, it is possible to define new constructors for the type `exn` within any module. The price of this extra flexibility is that no exhaustive case analysis on values of type `exn` is possible.

One significant drawback of algebraic data type definitions resides in the fact that a label ℓ cannot be *shared* by two distinct variant or record type definitions. Indeed, every algebraic data type definition extends the calculus with new constants. Strictly speaking, our presentation does not allow a single constant `c` to be associated with two distinct definitions. Even if we did allow such a collision, the initial environment would contain two bindings for `c`, one of which would then hide the other. This phenomenon arises in actual implementations of ML-the-programming-language, where a new algebraic data type definition may hide some of the data constructors or record labels introduced by a previous definition. An elegant solution to this lack of expressiveness is discussed in §1.11.

$p ::=$	*Patterns:*
$\_$	*Wildcard*
z	*Variable*
$c\ p_1 \ldots p_k$	*Data*
	$c \in \mathcal{Q}^+ \wedge k = a(c)$
$p \wedge p$	*Conjunction*
$p \vee p$	*Disjunction*

Pattern matching

$$[\_ \mapsto v] = \varnothing$$
$$[c\ p_1 \ldots p_k \mapsto c\ v_1 \ldots v_k] = [p_1 \mapsto v_1] \otimes \ldots \otimes [p_k \mapsto v_k]$$
$$[p_1 \wedge p_2 \mapsto v] = [p_1 \mapsto v] \otimes [p_2 \mapsto v]$$
$$[p_1 \vee p_2 \mapsto v] = [p_1 \mapsto v] \oplus [p_2 \mapsto v]$$

Figure 1-13: Patterns and pattern matching

1.9.4 Pattern matching

Our presentation of products, sums and algebraic data types has remained within the setting of ML-the-calculus: that is, data structures have been built out of constructors, while the case analysis and record access operations have been viewed as destructors. Some syntactic sugar has been used to recover standard notations. The language is now expressive enough to allow defining and manipulating complex data structures, such as lists and trees. Yet, experience shows that programming in such a language is still somewhat cumbersome. Indeed, case analysis and record access are low-level operations: the former allows inspecting a tag and branching, while the latter allows dereferencing a pointer. In practice, one often needs to carry out more complex tasks, such as determining whether a data structure has a certain shape or whether two data structures have comparable shapes. Currently, the only way to carry out these tasks is to program an explicit sequence of low-level operations. It would be much preferable to extend the language so that it becomes directly possible to describe shapes, called *patterns*, and so that checking whether a patterns *matches* a value becomes an elementary operation. ML-the-programming-language offers this feature, called *pattern matching*. Although pattern matching may be added to ML-the-calculus by introducing a family of destructors, we rather choose to extend the calculus with a new `match` construct, which subsumes the existing `let` construct. This approach appears somewhat simpler and more powerful. We now carry out this extension.

Let us first define the syntax of patterns (Figure 1-13) and describe (informally, for now) which values they match. To a pattern p, we associate a set of *defined program variables* $dpi(p)$, whose definition appears in the text that follows. The pattern p is well-formed if and only if $dpi(p)$ is defined. To begin, the wildcard _ is a pattern, which matches every value and binds no

variables. We let $dpi(\_) = \varnothing$. Although the wildcard may be viewed as an anonymous variable, and we have done so thus far, it is now simpler to view it as a distinct pattern. A program variable $\mathtt{z}$ is also a pattern, which matches every value and binds $\mathtt{z}$ to the matched value. We let $dpi(\mathtt{z}) = \{\mathtt{z}\}$. Next, if $\mathtt{c}$ is a constructor of arity $\mathtt{k}$, then $\mathtt{c}\ \mathtt{p}_1\ \ldots\ \mathtt{p}_\mathtt{k}$ is a pattern, which matches $\mathtt{c}\ \mathtt{v}_1\ \ldots\ \mathtt{v}_\mathtt{k}$ when $\mathtt{p}_\mathtt{i}$ matches $\mathtt{v}_\mathtt{i}$ for every $\mathtt{i} \in \{1, \ldots, \mathtt{k}\}$. We let $dpi(\mathtt{c}\ \mathtt{p}_1\ \ldots\ \mathtt{p}_\mathtt{k}) = dpi(\mathtt{p}_1) \uplus \ldots \uplus dpi(\mathtt{p}_\mathtt{k})$. That is, the pattern $\mathtt{c}\ \mathtt{p}_1\ \ldots\ \mathtt{p}_\mathtt{k}$ is well-formed when $\mathtt{p}_1, \ldots, \mathtt{p}_\mathtt{k}$ define *disjoint* sets of variables. This condition rules out *nonlinear* patterns such as $(\mathtt{z}, \mathtt{z})$. Defining the semantics of such a pattern would require a notion of equality at every type, which introduces various complications, so it is commonly considered ill-formed. The pattern $\mathtt{p}_1 \wedge \mathtt{p}_2$ matches all values that both $\mathtt{p}_1$ and $\mathtt{p}_2$ match. It is commonly used with $\mathtt{p}_2$ a program variable: then, it allows examining the shape of a value and binding a name to it at the same time. Again, we define $dpi(\mathtt{p}_1 \wedge \mathtt{p}_2) = dpi(\mathtt{p}_1) \uplus dpi(\mathtt{p}_2)$. The pattern $\mathtt{p}_1 \vee \mathtt{p}_2$ matches all values that either $\mathtt{p}_1$ or $\mathtt{p}_2$ matches. We define $dpi(\mathtt{p}_1 \vee \mathtt{p}_2) = dpi(\mathtt{p}_1) = dpi(\mathtt{p}_2)$. That is, the pattern $\mathtt{p}_1 \vee \mathtt{p}_2$ is well-formed when $\mathtt{p}_1$ and $\mathtt{p}_2$ define the *same* variables. Thus, $(\mathtt{inj}_1\ \mathtt{z}) \vee (\mathtt{inj}_2\ \mathtt{z})$ is a well-formed pattern, which binds $\mathtt{z}$ to the component of a binary sum, without regard for its tag. However, $(\mathtt{inj}_1\ \mathtt{z}_1) \vee (\mathtt{inj}_2\ \mathtt{z}_2)$ is ill-formed, because one cannot statically predict whether it defines $\mathtt{z}_1$ or $\mathtt{z}_2$.

Let us now formally define whether a pattern $\mathtt{p}$ matches a value $\mathtt{v}$ and how the variables in $dpi(\mathtt{p})$ become bound to values in the process. This is done by introducing a *generalized substitution*, written $[\mathtt{p} \mapsto \mathtt{v}]$, which is either undefined or a substitution of values for the program variables in $dpi(\mathtt{p})$. If the former, then $\mathtt{p}$ does not match $\mathtt{v}$. If the latter, then $\mathtt{p}$ matches $\mathtt{v}$ and, for every $\mathtt{z} \in dpi(\mathtt{p})$, the variable $\mathtt{z}$ becomes bound to the value $[\mathtt{p} \mapsto \mathtt{v}]\mathtt{z}$. Of course, when $\mathtt{p}$ is a variable $\mathtt{z}$, the generalized substitution $[\mathtt{z} \mapsto \mathtt{v}]$ is defined and coincides with the substitution $[\mathtt{z} \mapsto \mathtt{v}]$, which justifies our abuse of notation. To construct generalized substitutions, we use two simple combinators. First, when $dpi(\mathtt{p}_1)$ and $dpi(\mathtt{p}_2)$ are disjoint, $[\mathtt{p}_1 \mapsto \mathtt{v}_1] \otimes [\mathtt{p}_2 \mapsto \mathtt{v}_2]$ stands for the set-theoretic union of $[\mathtt{p}_1 \mapsto \mathtt{v}_1]$ and $[\mathtt{p}_2 \mapsto \mathtt{v}_2]$, if both are defined, and is undefined otherwise. We use this combinator to ensure that $\mathtt{p}_1$ matches $\mathtt{v}_1$ and $\mathtt{p}_2$ matches $\mathtt{v}_2$ and to combine the two corresponding sets of bindings. Second, when $\mathtt{o}_1$ and $\mathtt{o}_2$ are two possibly undefined mathematical objects that belong to the same space when defined, $\mathtt{o}_1 \oplus \mathtt{o}_2$ stands for $\mathtt{o}_1$, if it is defined, and for $\mathtt{o}_2$ otherwise—that is, $\oplus$ is an angelic choice operator with a left bias. In particular, when $dpi(\mathtt{p}_1)$ and $dpi(\mathtt{p}_2)$ coincide, $[\mathtt{p}_1 \mapsto \mathtt{v}_1] \oplus [\mathtt{p}_2 \mapsto \mathtt{v}_2]$ stands for $[\mathtt{p}_1 \mapsto \mathtt{v}_1]$, if it is defined, and for $[\mathtt{p}_2 \mapsto \mathtt{v}_2]$ otherwise. We use this combinator to ensure that $\mathtt{p}_1$ matches $\mathtt{v}_1$ or $\mathtt{p}_2$ matches $\mathtt{v}_2$ and to retain the corresponding set of bindings. The full definition of generalized substitutions, which relies on these combinators, appears in Figure 1-13. It reflects

$t ::= \ldots$ *Expressions:* $\quad \texttt{match}\, t\, \texttt{with}\, (p_i \,.\, t_i)_{i=1}^{k}$ $\mathcal{E} ::= \ldots$ *Evaluation Contexts:* $\quad \texttt{match}\, \mathcal{E}\, \texttt{with}\, (p_i \,.\, t_i)_{i=1}^{k}$	*Reduction rules* $\texttt{match}\, v\, \texttt{with}\, (p_i \,.\, t_i)_{i=1}^{k} \longrightarrow \bigoplus_{i=1}^{k} [p_i \mapsto v]t_i$ (R-MATCH)

Figure 1-14: Extended syntax and semantics of ML-the-calculus

the informal presentation of the previous paragraph.

Once patterns and pattern matching are defined, it is straightforward to extend the syntax and operational semantics of ML-the-calculus. We enrich the syntax of expressions with a new construct, $\texttt{match}\, t\, \texttt{with}\, (p_i \,.\, t_i)_{i=1}^{k}$, where $k \geq 1$. It consists of a term t and a nonempty, ordered list of clauses, each of which is composed of a pattern p_i and a term t_i. The syntax of evaluation contexts is extended as well, so that the term t that is being examined is first reduced to a value v. The operational semantics is extended with a new rule, R-MATCH, which states that $\texttt{match}\, v\, \texttt{with}\, (p_i \,.\, t_i)_{i=1}^{k}$ reduces to $[p_i \mapsto v]t_i$, where i is the least element of $\{1, \ldots, k\}$ such that p_i matches v_i. Technically, $\bigoplus_{i=1}^{k} [p_i \mapsto v]t_i$ stands for $[p_1 \mapsto v]t_1 \oplus \ldots \oplus [p_k \mapsto v]t_k$, so that the reduct is the first term that is defined in this sequence.

As far as semantics is concerned, the `match` construct may be viewed as a generalization of the `let` construct. Indeed, $\texttt{let}\, z = t_1\, \texttt{in}\, t_2$ may now be viewed as syntactic sugar for $\texttt{match}\, t_1\, \texttt{with}\, z \,.\, t_2$, that is, a `match` construct with a single clause and a variable pattern. Then, R-LET becomes a special case of R-MATCH.

It is pleasant to introduce some more syntactic sugar. We write $\lambda(p_i.t_i)_{i=1}^{k}$ for $\lambda z.\texttt{match}\, z\, \texttt{with}\, (p_i \,.\, t_i)_{i=1}^{k}$, where z is fresh for $(p_i.t_i)_{i=1}^{k}$. Thus, it becomes possible to define functions by cases—a common idiom in ML-the-programming-language.

1.9.17 EXAMPLE: Using pattern matching, a function that computes the length of a list (Example 1.9.13) may now be written as follows:

$$\texttt{letrec length} = \lambda(\texttt{Nil}\ \_ \,.\, \hat{0} \mid \texttt{Cons}\ (\_, z) \,.\, \hat{1} \hat{+} \texttt{length}\ z)$$

The second pattern matches a nonempty list and binds z to its tail at the same time, obviating the need for an explicit application of π_2. □

1.9.18 EXERCISE [RECOMMENDED, ★★, ↛]: Under the above definition of `length`, consider an application of `length` to the list $\texttt{Cons}(\hat{0}, \texttt{Nil}())$. After eliminat-

$$
\begin{array}{rcl}
[\![\mathtt{T} : \_]\!] & = & \mathsf{true} \\
[\![\mathtt{T} : \mathtt{z}]\!] & = & \mathtt{T} \preceq \mathtt{z} \\
[\![\mathtt{T} : \mathtt{c}\ \mathtt{p}_1\ \ldots\ \mathtt{p}_k]\!] & = & \exists \bar{\mathtt{X}}.(\vec{\mathtt{X}} \to \mathtt{T} \preceq \mathtt{c} \wedge \bigwedge_{i=1}^{k} [\![\mathtt{X}_i : \mathtt{p}_i]\!]) \\
[\![\mathtt{T} : \mathtt{p}_1 \wedge \mathtt{p}_2]\!] & = & [\![\mathtt{T} : \mathtt{p}_1]\!] \wedge [\![\mathtt{T} : \mathtt{p}_2]\!] \\
[\![\mathtt{T} : \mathtt{p}_1 \vee \mathtt{p}_2]\!] & = & [\![\mathtt{T} : \mathtt{p}_1]\!] \wedge [\![\mathtt{T} : \mathtt{p}_2]\!]
\end{array}
$$

$$
[\![\mathtt{match\ t\ with}\ (\mathtt{p}_i\ .\ \mathtt{t}_i)_{i=1}^{k} : \mathtt{T}]\!] = \bigwedge_{i=1}^{k} \mathsf{let}\ \forall \mathtt{X} \bar{\mathtt{X}}_i [\![\mathtt{t} : \mathtt{X}]\!] \wedge \mathsf{let}\ \vec{\mathtt{z}}_i : \vec{\mathtt{X}}_i\ \mathsf{in}\ [\![\mathtt{X} : \mathtt{p}_i]\!].(\vec{\mathtt{z}}_i : \vec{\mathtt{X}}_i)\ \mathsf{in}\ [\![\mathtt{t}_i : \mathtt{T}]\!]
\quad \text{where } \vec{\mathtt{z}}_i = \mathit{dpi}(\mathtt{p}_i)
$$

Figure 1-15: Constraint generation for patterns and pattern matching

ing the syntactic sugar, determine by which reduction sequence this expression reduces to a value. □

Before we can proceed and extend the type system to deal with the new `match` construct, we must make two mild extensions to the syntax and meaning of constraints. First, if σ is $\forall \bar{\mathtt{X}}[\mathtt{C}].\mathtt{T}$, where $\bar{\mathtt{X}} \# \mathit{ftv}(\mathtt{T}')$, then $\mathtt{T}' \preceq \sigma$ stands for the constraint $\exists \bar{\mathtt{X}}.(\mathtt{C} \wedge \mathtt{T}' \leq \mathtt{T})$. This relation is identical to the instance relation (Definition 1.3.3), except the direction of subtyping is reversed. We extend the syntax of constraints with instantiation constraints of the form $\mathtt{T} \preceq \mathtt{x}$ and define their meaning by adding a symmetric counterpart of CM-INSTANCE. We remark that, when subtyping is interpreted as equality, the relations $\sigma \preceq \mathtt{T}$ and $\mathtt{T} \preceq \sigma$ coincide, so this extension is unnecessary in that particular case. Second, we extend the syntax of environments so that several successive bindings may *share* a set of quantifiers and a constraint. That is, we allow writing $\forall \bar{\mathtt{X}}[\mathtt{C}].(\mathtt{x}_1 : \mathtt{T}_1; \ldots; \mathtt{x}_k : \mathtt{T}_k)$ for $\mathtt{x}_1 : \forall \bar{\mathtt{X}}[\mathtt{C}].\mathtt{T}_1; \ldots; \mathtt{x}_k : \forall \bar{\mathtt{X}}[\mathtt{C}].\mathtt{T}_k$. From a theoretical standpoint, this is little more than syntactic sugar; however, in practice, it is useful to implement this new idiom literally, since it avoids unnecessary copying of the constraint C.

Let us now extend the type system. For the sake of brevity, we extend the constraint generation rules only. Of course, it would also be possible to define corresponding extensions of the rule-based type systems shown earlier, namely DM, HM(X), and PCB(X). We begin by defining a constraint $[\![\mathtt{T} : \mathtt{p}]\!]$ that represents a necessary and sufficient condition for values of type T to be acceptable inputs for the pattern p. Its free type variables are a subset of $\mathit{ftv}(\mathtt{T})$, while its free program identifiers are either constructors or program variables bound by p. It is defined in the upper part of Figure 1-15. The first rule states that a wildcard matches values of arbitrary type. The second and third rules govern program variables and constructor applications in *patterns*. They are identical to the rules that govern these constructs in *expressions*

(page 80), except that the direction of subtyping is reversed. In the absence of subtyping, they would be entirely identical. We write $\vec{\mathtt{X}}$ for $\mathtt{X}_1 \ldots \mathtt{X}_k$ and $\vec{\mathtt{X}} \to \mathtt{T}$ for $\mathtt{X}_1 \to \ldots \to \mathtt{X}_k \to \mathtt{T}$. As usual, the type variables $\mathtt{X}_1, \ldots, \mathtt{X}_k$ must have kind $\star$ and must be distinct and fresh for the equation's left-hand side. The last two rules simply distribute the type $\mathtt{T}$ to both subpatterns. It is easy to check that $[\![\mathtt{T} : \mathtt{p}]\!]$ is *contravariant* in $\mathtt{T}$:

1.9.19 LEMMA: $\mathtt{T}' \leq \mathtt{T} \wedge [\![\mathtt{T} : \mathtt{p}]\!]$ entails $[\![\mathtt{T}' : \mathtt{p}]\!]$. □

This property reflects the fact that $\mathtt{T}$ represents the type of an *input* for the pattern $\mathtt{p}$. Compare it with Lemma 1.6.4.

1.9.20 EXAMPLE: Consider the pattern $\mathtt{Cons}\ (\_, \mathtt{z})$, which appears in Example 1.9.17. We have

$$
\begin{array}{ll}
 & [\![\mathtt{T} : \mathtt{Cons}\ (\_, \mathtt{z})]\!] \\
\equiv & \exists \mathtt{Z}_1.([\![\mathtt{Z}_1 \to \mathtt{T} : \mathtt{Cons}]\!] \wedge [\![\mathtt{Z}_1 : (\_, \mathtt{z})]\!]) \\
\equiv & \exists \mathtt{Z}_1.(\mathtt{Z}_1 \to \mathtt{T} \preceq \mathtt{Cons} \wedge \exists \mathtt{Z}_2 \mathtt{Z}_3.([\![\mathtt{Z}_2 \to \mathtt{Z}_3 \to \mathtt{Z}_1 : (\cdot, \cdot)]\!] \wedge [\![\mathtt{Z}_2 : \_]\!] \wedge [\![\mathtt{Z}_3 : \mathtt{z}]\!])) \\
\equiv & \exists \mathtt{Z}_1 \mathtt{Z}_2 \mathtt{Z}_3.(\mathtt{Z}_1 \to \mathtt{T} \preceq \mathtt{Cons} \wedge \mathtt{Z}_2 \to \mathtt{Z}_3 \to \mathtt{Z}_1 \preceq (\cdot, \cdot) \wedge \mathtt{Z}_3 \preceq \mathtt{z})
\end{array}
$$

where $\mathtt{Z}_1$, $\mathtt{Z}_2$, $\mathtt{Z}_3$ are fresh for $\mathtt{T}$. Let us now place this constraint within the scope of the initial environment, which assigns type schemes to the constructors $\mathtt{Cons}$ and $(\cdot, \cdot)$, and within the scope of a binding of $\mathtt{z}$ to some type $\mathtt{T}'$. We find

$$
\begin{array}{ll}
 & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \mathsf{let}\ \mathtt{z} : \mathtt{T}'\ \mathsf{in}\ [\![\mathtt{T} : \mathtt{Cons}\ (\_, \mathtt{z})]\!] \\
\equiv & \exists \mathtt{Z}_1 \mathtt{Z}_2 \mathtt{Z}_3.(\exists \mathtt{X}.(\mathtt{Z}_1 \to \mathtt{T} \leq \mathtt{X} \times \mathsf{list}\,\mathtt{X} \to \mathsf{list}\,\mathtt{X}) \wedge \\
 & \qquad \exists \mathtt{Y}_1 \mathtt{Y}_2.(\mathtt{Z}_2 \to \mathtt{Z}_3 \to \mathtt{Z}_1 \leq \mathtt{Y}_1 \to \mathtt{Y}_2 \to \mathtt{Y}_1 \times \mathtt{Y}_2) \wedge \mathtt{Z}_3 \leq \mathtt{T}') \\
\equiv & \exists \mathtt{X}.(\mathtt{T} \leq \mathsf{list}\,\mathtt{X} \wedge \mathsf{list}\,\mathtt{X} \leq \mathtt{T}')
\end{array}
$$

where the final simplification relies mainly on C-ARROW, on the corresponding rule for products, and on C-EXTRANS, and is left as an exercise to the reader. Thus, the constraint states that the pattern matches values that have type $\mathsf{list}\,\mathtt{X}$ (equivalently, values whose type $\mathtt{T}$ is a subtype of $\mathsf{list}\,\mathtt{X}$), for some undetermined element type $\mathtt{X}$, and binds $\mathtt{z}$ to values of type $\mathsf{list}\,\mathtt{X}$ (equivalently, values whose type $\mathtt{T}'$ is a supertype of $\mathsf{list}\,\mathtt{X}$). □

The above example seems to indicate that the constraint generation rules for patterns make some sense. Still, the careful reader may be somewhat puzzled by the third rule, which, compared to its analogue for expressions, reverses the direction of subtyping, but does not reverse the direction of instantiation. Indeed, in order for this rule to make sense, and to be sound, we must formulate a requirement concerning the type schemes assigned to constructors.

1.9.21 DEFINITION: A constructor $\mathtt{c}$ is *invertible* if and only if, when $\vec{X}$ and $\vec{X}'$ have length $\mathtt{a}(\mathtt{c})$, the constraint $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ (\vec{X}' \to \mathtt{T} \preceq \mathtt{c} \wedge \mathtt{c} \preceq \vec{X} \to \mathtt{T})$ entails $\vec{X} \leq \vec{X}'$. In the following, we assume patterns contain invertible constructors only. □

Intuitively, when $\mathtt{c}$ is invertible, it is possible to recover the type of every $\mathtt{v}_i$ from the type of $\mathtt{c}\ \mathtt{v}_1\ \ldots\ \mathtt{v}_k$, a crucial property for pattern matching to be possible. Please note that, if $\Gamma_0(\mathtt{c})$ is monomorphic, then $\mathtt{c}$ is invertible. The following lemma identifies another important class of invertible constructors.

1.9.22 LEMMA: The constructors of algebraic data types are invertible. □

Proof: Let $\mathtt{c}$ be a constructor introduced by the definition of an algebraic data type $\mathtt{D}$. Let $k = \mathtt{a}(\mathtt{c})$. Then, the type scheme $\Gamma_0(\mathtt{c})$ is of the form $\forall\vec{Y}.\vec{T} \to \mathtt{D}\,\vec{Y}$, where $\vec{Y}$ are the parameters of the definition and $\vec{T}$, a vector of length k, consists of *some of* the definition's components. (More precisely, $\vec{T}$ contains just one component in the case of variant types and contains all components in the case of record types.) Let $\vec{X}$ and $\vec{X}'$ have length k. Let $\forall\vec{Y}_1.\vec{T}_1 \to \mathtt{D}\,\vec{Y}_1$ and $\forall\vec{Y}_2.\vec{T}_2 \to \mathtt{D}\,\vec{Y}_2$ be two α-equivalent forms of the type scheme $\Gamma_0(\mathtt{c})$, with $\vec{Y}_1 \# \vec{Y}_2$ and $\vec{Y}_1\vec{Y}_2 \# \mathit{ftv}(\vec{X}, \vec{X}', \mathtt{T})$. The constraint $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ (\vec{X}' \to \mathtt{T} \preceq \mathtt{c} \wedge \mathtt{c} \preceq \vec{X} \to \mathtt{T})$ is, by definition, equivalent to $\vec{X}' \to \mathtt{T} \preceq \Gamma_0(\mathtt{c}) \wedge \Gamma_0(\mathtt{c}) \preceq \vec{X} \to \mathtt{T}$, that is, $\exists\vec{Y}_1.(\vec{X}' \to \mathtt{T} \leq \vec{T}_1 \to \mathtt{D}\,\vec{Y}_1) \wedge \exists\vec{Y}_2.(\vec{T}_2 \to \mathtt{D}\,\vec{Y}_2 \leq \vec{X} \to \mathtt{T})$. By C-EXAND and C-ARROW, this may be written $\exists\vec{Y}_1\vec{Y}_2.(\mathtt{D}\,\vec{Y}_2 \leq \mathtt{T} \leq \mathtt{D}\,\vec{Y}_1 \wedge \vec{X} \leq \vec{T}_2 \wedge \vec{T}_1 \leq \vec{X}')$. Now, by Definition 1.9.10, $\mathtt{D}\,\vec{Y}_2 \leq \mathtt{D}\,\vec{Y}_1$ entails $\vec{T}_2 \leq \vec{T}_1$, so the previous constraint entails $\exists\vec{Y}_1\vec{Y}_2.(\vec{X} \leq \vec{X}')$, that is, $\vec{X} \leq \vec{X}'$. □

An important class of *noninvertible* constructors are those associated with existential type definitions (page 153), where not all quantifiers of the type scheme $\Gamma_0(\mathtt{c})$ are parameters of the type constructor $\mathtt{D}$. For instance, under the definition $\mathtt{D} \approx \ell : \exists\mathtt{X}.\mathtt{X}$, the type scheme associated with ℓ is $\forall\mathtt{X}.\mathtt{X} \to \mathtt{D}$. Then, it is easy to check that ℓ is not invertible. This reflects the fact that it is not possible to recover the type of $\mathtt{v}$ from the type of $\ell\ \mathtt{v}$—which must be $\mathtt{D}$ in any case—and explains why existential types require special treatment.

We are now ready to associate a constraint generation rule with the `match` construct. It is given in the lower part of Figure 1-15. In the rule's right-hand side, we write $\vec{z}_i$ for the program variables bound by the pattern $\mathtt{p}_i$, and we write $\vec{X}_i$ for a vector of type variables of the same length. The type variables $\mathtt{X}\vec{X}_i$ must have kind $\star$, must be pairwise distinct and must not appear free in the rule's left-hand side. Let us now explain the rule. Its right-hand side is a conjunction, where each conjunct deals with one clause of the `match` construct, requiring $\mathtt{t}_i$ to have type $\mathtt{T}$ under certain assumptions about the program variables $\vec{z}_i$ bound by the pattern $\mathtt{p}_i$. There remains to

explain how these assumptions are built. First, as in the case of a `let` construct, we summon a fresh type variable X and produce $[\![\mathtt{t} : \mathtt{X}]\!]$, the least specific constraint that guarantees t has type X. Then, reflecting the operational semantics, which feeds (the value produced by) t into the pattern $\mathtt{p}_i$, we feed the type X into $\mathtt{p}_i$ and produce $\mathsf{let}\ \vec{\mathtt{z}}_i : \vec{\mathtt{X}}_i\ \mathsf{in}\ [\![\mathtt{X} : \mathtt{p}_i]\!]$, a constraint that guarantees that $\vec{\mathtt{X}}_i$ is a correct vector of type assumptions for the program variables $\vec{\mathtt{z}}_i$ (see Example 1.9.20). This explains why we may place $[\![\mathtt{T} : \mathtt{t}_i]\!]$ within the scope of $(\vec{\mathtt{z}}_i : \vec{\mathtt{X}}_i)$. There remains to point out that, as in the case of the `let` construct, *every* assignment of ground types to $\mathtt{X}\bar{\mathtt{X}}_i$ that satisfies the constraint $[\![\mathtt{t} : \mathtt{X}]\!] \wedge \mathsf{let}\ \vec{\mathtt{z}}_i : \vec{\mathtt{X}}_i\ \mathsf{in}\ [\![\mathtt{X} : \mathtt{p}_i]\!]$ is acceptable, so it is valid to universally quantify these type variables. This allows the program variables $\vec{\mathtt{z}}_i$ to receive polymorphic type schemes when t itself has polymorphic type.

1.9.23 EXERCISE [RECOMMENDED, ★]: We have previously suggested viewing `let z = t1 in t2` as syntactic sugar for `match t1 with z.t2`, and shown that the operational semantics validates this view. Check that it is also valid from a typing perspective. □

The `match` constraint generation rule, if implemented literally, takes k copies of the constraint $[\![\mathtt{t} : \mathtt{X}]\!]$. When k is greater than 1, this compromises the linear time and space complexity of constraint generation. To remedy this problem, one may modify the rule as follows: replace every copy of $[\![\mathtt{t} : \mathtt{X}]\!]$ with $\mathtt{z} \preceq \mathtt{X}$ and place the constraint within the context $\mathsf{let}\ \mathtt{z} : \forall \mathtt{X}[[\![\mathtt{t} : \mathtt{X}]\!].\mathtt{X}\ \mathsf{in}\ []$, where z is a fresh program variable. It is not difficult to check that the logical meaning of the constraint is not affected and that a linear behavior is recovered. In practice, solving the new constraint requires taking instances of the type scheme $\forall \mathtt{X}[[\![\mathtt{t} : \mathtt{X}]\!].\mathtt{X}$, which essentially requires copying $[\![\mathtt{t} : \mathtt{X}]\!]$ again—however, an efficient solver may now *simplify* this subconstraint before duplicating it.

The following lemma is a key to establishing subject reduction for R-MATCH. It relies on the requirement that constructors be invertible.

1.9.24 LEMMA: Assume $[\mathtt{p} \mapsto \mathtt{v}]$ is defined and maps $\vec{\mathtt{z}}$ to $\vec{\mathtt{w}}$, where $\bar{\mathtt{z}} = \mathit{dpi}(\mathtt{p})$. Let $\vec{\mathtt{z}} : \vec{\mathtt{T}}$ be an arbitrary monomorphic environment of domain $\bar{\mathtt{z}}$. Then, $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ ([\![\mathtt{v} : \mathtt{T}]\!] \wedge \mathsf{let}\ \vec{\mathtt{z}} : \vec{\mathtt{T}}\ \mathsf{in}\ [\![\mathtt{T} : \mathtt{p}]\!])$ entails $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\vec{\mathtt{w}} : \vec{\mathtt{T}}]\!]$. □

Proof: The proof is by induction on the structure of p.

∘ *Case* _. The goal is the tautology $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathtt{v} : \mathtt{T}]\!] \Vdash \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \mathsf{true}$.

∘ *Case* z. The goal is $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ ([\![\mathtt{v} : \mathtt{T}]\!] \wedge \mathsf{let}\ \mathtt{z} : \mathtt{T}'\ \mathsf{in}\ \mathtt{T} \preceq \mathtt{z}) \Vdash \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathtt{v} : \mathtt{T}']\!]$, that is, $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ ([\![\mathtt{v} : \mathtt{T}]\!] \wedge \mathtt{T} \leq \mathtt{T}') \Vdash \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathtt{v} : \mathtt{T}']\!]$, which follows from Lemma 1.6.4.

∘ *Case* $\mathtt{c}\ \mathtt{p}_1\ \ldots\ \mathtt{p}_k$. Any value matched by this pattern must be of the form $\mathtt{c}\ \mathtt{v}_1\ \ldots\ \mathtt{v}_k$, so the goal is to show that $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ ([\![\mathtt{c}\ \mathtt{v}_1\ \ldots\ \mathtt{v}_k : \mathtt{T}]\!] \wedge \mathsf{let}\ \vec{\mathtt{z}} :$

$\vec{T}$ in $[\![T : c\ p_1\ \ldots\ p_k]\!]$) **(1)** entails let Γ_0 in $[\![\vec{w} : \vec{T}]\!]$. By definition of constraint generation and by C-EXAND, C-IN*, C-INAND, C-INAND*, C-INEX, and (1) is equivalent to let Γ_0 in $\exists \bar{X}\bar{X}'.(\vec{X}' \to T \preceq c \wedge c \preceq \vec{X} \to T \wedge \bigwedge_{i=1}^{k} ([\![v_i : X_i]\!] \wedge$ let $\vec{z}_i : \vec{T}_i$ in $[\![X_i' : p_i]\!]))$, where $\bar{X}$ and $\bar{X}'$ are fresh, $\vec{X}$ (resp. $\vec{X}'$) is short for $X_1 \ldots X_k$ (resp. $X_1' \ldots X_k'$), and $\vec{z}_i : \vec{T}_i$ is the fragment of $\vec{z} : \vec{T}$ whose domain is $dpi(p_i)$. By Definition 1.9.21, this entails let Γ_0 in $\exists \bar{X}\bar{X}'. \bigwedge_{i=1}^{k} ([\![v_i : X_i]\!] \wedge X_i \leq X_i' \wedge$ let $\vec{z}_i : \vec{T}_i$ in $[\![X_i' : p_i]\!])$, which by Lemma 1.6.5 is let Γ_0 in $\exists \bar{X}'. \bigwedge_{i=1}^{k} ([\![v_i : X_i']\!] \wedge$ let $\vec{z}_i : \vec{T}_i$ in $[\![X_i' : p_i]\!])$ **(2)**. Applying the induction hypothesis for every $i \in \{1, \ldots, k\}$, we find that (2) entails let Γ_0 in $\exists \bar{X}'. \bigwedge_{i=1}^{k} [\![\vec{w}_i : \vec{T}_i]\!]$, where $\vec{w}_i$ is the fragment of $\vec{w}$ associated with $\vec{z}_i$, that is, let Γ_0 in $[\![\vec{w} : \vec{T}]\!]$.

◦ *Cases* $p_1 \wedge p_2$ and $p_1 \vee p_2$. By the induction hypothesis, C-IN*, C-INAND, and C-INAND*. □

We now prove that our extension of ML-the-calculus with pattern matching enjoys subject reduction. We only state that R-MATCH preserves types, and leave the new subcase of R-CONTEXT, where the evaluation context involves a `match` construct, to the reader. For this subcase to succeed, the value restriction (Definition 1.7.8) must be extended to require that either all constants have pure semantics or all `match` constructs are in fact of the form `match v with` $(p_i \,.\, t_i)_{i=1}^{k}$.

1.9.25 THEOREM [SUBJECT REDUCTION]: (R-MATCH) $\subseteq$ ($\sqsubseteq$). □

Proof: The goal is to prove that let Γ_0 in $[\![$`match v with` $(p_i \,.\, t_i)_{i=1}^{k} : T]\!]$ entails let Γ_0 in $[\![\bigoplus_{i=1}^{k} [p_i \mapsto v]t_i : T]\!]$. It is clear that the latter is entailed by let Γ_0 in $\bigwedge_{i=1}^{k} [\![[p_i \mapsto v]t_i : T]\!]$. Because the former is also, by definition of constraint generation, a conjunction of k conjuncts, we may decompose the goal on a per-clause basis and abandon the index i. The goal becomes, assuming that $[p \mapsto v]$ is defined, to prove that let Γ_0 in let $\forall X\bar{X}[\![v : X]\!] \wedge$ let $\vec{z} : \vec{X}$ in $[\![X : p]\!].(\vec{z} : \vec{X})$ in $[\![t : T]\!]$ **(1)** entails let Γ_0 in $[\![[p \mapsto v]t : T]\!]$ **(2)**, where $\vec{z}$ are the program variables bound by the pattern p. The substitution $[p \mapsto v]$ must be of the form $[\vec{z} \mapsto \vec{w}]$. By Lemma 1.9.24, (1) entails let Γ_0 in let $\forall \bar{X}[\![\vec{w} : \vec{X}]\!].(\vec{z} : \vec{X})$ in $[\![t : T]\!]$ **(3)**. Let us assume, *w.l.o.g.*, $\bar{z} \# fpi(v)$, which implies $\bar{z} \# fpi(\vec{w})$. By definition of shared quantifiers and constraints within environments (page 130), by C-EX*, C-$\overrightarrow{\text{LETEX}}$, C-LETAND, and C-DUP, (3) is found to be equivalent to let Γ_0 in let $z : \forall X[\![w : X]\!].X$ in $[\![t : T]\!]$ **(4)**. By repeated application of Lemma 1.7.2, (4) entails let Γ_0 in $[\![[\vec{z} \mapsto \vec{w}]t : T]\!]$, that is, (2). □

1.9.26 EXERCISE [★★★, ↛]: For the sake of simplicity, we have omitted the production `ref p` from the syntax of patterns. The pattern `ref p` matches every

memory location whose content (with respect to the current store) is matched by p. Determine how the previous definitions and proofs must be extended in order to accommodate this new production. □

The progress property does *not* hold in general: for instance, `match Nil with (Cons z . z)` is well-typed (with type $\forall X.X$) but is stuck. In actual implementations of ML-the-programming-language, such errors are dynamically detected. This may be considered a weakness of ML-the-type-system. Fortunately, however, it is often possible to statically prove that a particular `match` construct is *exhaustive* and cannot go wrong. Indeed, if `match v with` $(p_i\ .\ t_i)_{i=1}^{k}$ is well-typed, then for every $i \in \{1,\ldots,k\}$, the constraint $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ (\llbracket v : X \rrbracket \wedge \exists \bar{X}.\mathsf{let}\ \vec{z}_i : \vec{X}\ \mathsf{in}\ \llbracket X : p_i \rrbracket)$, where $\bar{z}_i$ are the program variables bound by p_i, must be satisfiable; that is, v must have some type that is an acceptable input for p_i. This fact yields information about v, from which it may be possible to derive that v must match one of the patterns p_i.

1.9.27 EXAMPLE: Let $k = 2$, $p_1 =$ `Nil _`, and $p_2 =$ `Cons` (z_1, z_2). Then, the constraints $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \bar{X}.\mathsf{let}\ \vec{z}_i : \vec{X}\ \mathsf{in}\ \llbracket X : p_i \rrbracket$, for $i \in \{1,2\}$, are both equivalent (after simplification, when $i = 2$) to $\exists Z.X \leq \mathsf{list}\ Z$. Because the type constructor list is isolated, every closed value v whose type X satisfies this constraint must be an application of `Nil` or `Cons`. If the latter, because `Cons` has type $\forall X.X \times \mathsf{list}\ X \to \mathsf{list}\ X$, and because the type constructor $\times$ is isolated, the argument to `Cons` must be a pair. We conclude that v must match either p_1 or p_2, which guarantees that this `match` construct is exhaustive and its evaluation cannot go wrong. □

It is beyond the scope of this chapter to give more details about the check for exhaustiveness. The reader is referred to (Sekar, Ramesh, and Ramakrishnan, 1995; Le Fessant and Maranget, 2001).

1.9.5 Type annotations

So far, we have been interested in a very pure, and extreme, form of type inference. Indeed, in ML-the-calculus, expressions contain no explicit type information whatsoever: it is entirely inferred. In practice, however, it is often useful to insert *type annotations* within expressions, because they provide a form of machine-checked documentation. Type annotations are also helpful when attempting to trace the cause of a type error: by supplying the type-checker with (supposedly) correct type information, one runs a better chance of finding a type inconsistency near an actual programming mistake.

When type annotations are allowed to contain type variables, one must be quite careful about *where* (at which program point) and *how* (existentially or

universally) these variables are bound. Indeed, the meaning of type annotations cannot be made precise without settling these issues. In what follows, we first explain how to introduce type annotations whose type variables are bound *locally* and existentially. We show that extending ML-the-calculus with such limited type annotations is again a simple matter of introducing new constants. Then, we turn to a more general case, where type variables may be explicitly existentially introduced *at any program point*. We defer the discussion of *universally* bound type variables to §1.10.

Let a *local existential type annotation* $\exists\bar{\mathtt{X}}.\mathtt{T}$ be a pair of a set of type variables $\bar{\mathtt{X}}$ and a type $\mathtt{T}$, where $\mathtt{T}$ has kind $\star$, $\bar{\mathtt{X}}$ is considered bound within $\mathtt{T}$, and $\bar{\mathtt{X}}$ contains $\mathit{ftv}(\mathtt{T})$. For every such annotation, we introduce a new unary destructor $(\cdot : \exists\bar{\mathtt{X}}.\mathtt{T})$. Such a definition is valid only because a type annotation must be *closed*, that is, does not have any free type variables. We write $(\mathtt{t} : \exists\bar{\mathtt{X}}.\mathtt{T})$ for the application $(\cdot : \exists\bar{\mathtt{X}}.\mathtt{T})\ \mathtt{t}$. Since a type annotation does not affect the meaning of a program, the new destructor has identity semantics:

$$(\mathtt{v} : \exists\bar{\mathtt{X}}.\mathtt{T}) \xrightarrow{\delta} \mathtt{v} \qquad \text{(R-ANNOTATION)}$$

Its type scheme, however, is not that of the identity, namely $\forall\mathtt{X}.\mathtt{X} \to \mathtt{X}$: instead, it is less general, so that annotating an expression *restricts* its type. Indeed, we extend the initial environment Γ_0 with the binding

$$(\cdot : \exists\bar{\mathtt{X}}.\mathtt{T}) : \forall\bar{\mathtt{X}}.\mathtt{T} \to \mathtt{T}$$

1.9.28 EXERCISE [★]: Check that $\forall\bar{\mathtt{X}}.\mathtt{T} \to \mathtt{T}$ is an *instance* of $\forall\mathtt{X}.\mathtt{X} \to \mathtt{X}$ in Damas and Milner's sense, that is, the former is obtained from the latter via the rule DM-INST′ given in Exercise 1.2.26. Does this allow arguing that the type scheme assigned to $(\cdot : \exists\bar{\mathtt{X}}.\mathtt{T})$ is sound? Check that the above definitions meet the requirements of Definition 1.7.7. □

Although inserting a type annotation does not change the semantics of the program, it does affect constraint generation, hence type inference. We let the reader check that, assuming $\bar{\mathtt{X}} \# \mathit{ftv}(\mathtt{t}, \mathtt{T}')$, the following derived constraint generation rule holds:

$$\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![(\mathtt{t} : \exists\bar{\mathtt{X}}.\mathtt{T}) : \mathtt{T}']\!] \equiv \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists\bar{\mathtt{X}}.([\![\mathtt{t} : \mathtt{T}]\!] \wedge \mathtt{T} \leq \mathtt{T}')$$

So far, expressions cannot have free type variables, so the hypothesis $\bar{\mathtt{X}} \# \mathit{ftv}(\mathtt{t})$ may seem superfluous. However, we shall soon allow expressions to contain type annotations with free type variables, so we prefer to make this condition explicit now. According to this rule, the effect of the type annotation is to force the expression $\mathtt{t}$ to have type $\mathtt{T}$, for *some* choice of the type variables $\bar{\mathtt{X}}$. As usual in type systems with subtyping, the expression's final type $\mathtt{T}'$ may then be an arbitrary supertype of this particular instance

of $\mathtt{T}$. When subtyping is interpreted as equality, $\mathtt{T}'$ and $\mathtt{T}$ are equated by the constraint, so this constraint generation rule may be read: *a valid type for* $(\mathtt{t} : \exists \bar{\mathtt{X}}.\mathtt{T})$ *must be of the form* $\mathtt{T}$*, for some choice of the type variables* $\bar{\mathtt{X}}$.

1.9.29 EXAMPLE: In DM extended with integers, the expression $(\lambda \mathtt{z}.\mathtt{z} : \mathsf{int} \to \mathsf{int})$ has most general type $\mathsf{int} \to \mathsf{int}$, even though the underlying identity function has most general type $\forall \mathtt{X}.\mathtt{X} \to \mathtt{X}$, so the annotation restricts its type. The expression $(\lambda \mathtt{z}.\mathtt{z} \hat{+} \hat{1} : \exists \mathtt{X}.\mathtt{X} \to \mathtt{X})$ has type $\mathsf{int} \to \mathsf{int}$, which is also the most general type of the underlying function, so the annotation acts merely as documentation in this case. Note that the type variable $\mathtt{X}$ is instantiated to int by the constraint solver. The expression $(\lambda \mathtt{z}.(\mathtt{z}\ \hat{1}) : \exists \mathtt{X}.\mathtt{X} \to \mathsf{int})$ has type $(\mathsf{int} \to \mathsf{int}) \to \mathsf{int}$ because the underlying function has type $(\mathsf{int} \to \mathtt{Y}) \to \mathtt{Y}$, which successfully unifies with $\mathtt{X} \to \mathsf{int}$ by instantiating $\mathtt{X}$ to $\mathsf{int} \to \mathsf{int}$ and $\mathtt{Y}$ to int. Last, the expression $(\lambda \mathtt{z}.(\mathtt{z}\ \hat{1}) : \exists \mathtt{X}.\mathsf{int} \to \mathtt{X})$ is ill-typed—even though the underlying expression is well-typed—because the equation $(\mathsf{int} \to \mathtt{Y}) \to \mathtt{Y} = \mathsf{int} \to \mathtt{X}$ is unsatisfiable. □

1.9.30 EXAMPLE: In DM extended with pairs, the expression $\lambda \mathtt{z}_1.\lambda \mathtt{z}_2.((\mathtt{z}_1 : \exists \mathtt{X}.\mathtt{X}), (\mathtt{z}_2 : \exists \mathtt{X}.\mathtt{X}))$ has most general type $\forall \mathtt{X}\mathtt{Y}.\mathtt{X} \to \mathtt{Y} \to \mathtt{X} \times \mathtt{Y}$. In other words, the two occurrences of $\mathtt{X}$ do not represent the same type. Indeed, one could just as well have written $\lambda \mathtt{z}_1.\lambda \mathtt{z}_2.((\mathtt{z}_1 : \exists \mathtt{X}.\mathtt{X}), (\mathtt{z}_2 : \exists \mathtt{Y}.\mathtt{Y}))$. If one wishes $\mathtt{z}_1$ and $\mathtt{z}_2$ to receive the same type, one must lift the type annotations and merge them above the pair constructor, as follows: $\lambda \mathtt{z}_1.\lambda \mathtt{z}_2.((\mathtt{z}_1, \mathtt{z}_2) : \exists \mathtt{X}.\mathtt{X} \times \mathtt{X})$. In the process, the type constructor $\times$ has appeared in the annotation, causing its size to increase. □

The above example reveals a limitation of this style of type annotations: by requiring every type annotation to be closed, we lose the ability for two separate annotations to *share* a type variable. Yet, such a feature is sometimes desirable. If the two annotations where sharing is desired are distant in the code, it may be awkward to lift and merge them into a single annotation; so, more expressive power is sometimes truly needed.

Thus, we are lead to consider more general type annotations, of the form $(\mathtt{t} : \mathtt{T})$, where $\mathtt{T}$ has kind $\star$, and where the type variables that appear within $\mathtt{T}$ are considered *free*, so that distinct type annotations may refer to shared type variables. For this idea to make sense, however, it is still necessary to specify where these type variables are bound. We do so using expressions of the form $\exists \bar{\mathtt{X}}.\mathtt{t}$. Such an expression binds the type variables $\bar{\mathtt{X}}$ within the expression $\mathtt{t}$, so that all free occurrences of $\mathtt{X}$ (where $\mathtt{X} \in \bar{\mathtt{X}}$) in type annotations inside $\mathtt{t}$ stand for the same type. Thus, we break the simple type annotation construct $(\cdot : \exists \bar{\mathtt{X}}.\mathtt{T})$ into two more elementary constituents, namely *existential type variable introduction* $\exists \bar{\mathtt{X}}.\cdot$ and *type constraint* $(\cdot : \mathtt{T})$. Note that both are new

forms of expressions; neither can be encoded by adding new constants to the calculus, because it is not possible to assign *closed* type schemes to them.

Technically, allowing expressions to contain type variables requires some care. Several constraint generation rules employ auxiliary type variables, which become bound in the generated constraint. These type variables may be chosen in an arbitrary way, provided they do not appear free in the rule's left-hand side—a side-condition intended to avoid inadvertent capture. So far, this side-condition could be read: the auxiliary type variables used to form the constraint $[\![\mathtt{t} : \mathtt{T}]\!]$ must not appear free within $\mathtt{T}$. Now, since type annotations may contain free type variables, the side-condition becomes: *the auxiliary type variables used to form* $[\![\mathtt{t} : \mathtt{T}]\!]$ *must not appear free within* $\mathtt{t}$ *or* $\mathtt{T}$.

With this extended side-condition in mind, our original constraint generation rules remain unchanged. We add two new rules to describe how the new expression forms affect constraint generation:

$$\begin{array}{rcll} [\![\exists\bar{\mathtt{X}}.\mathtt{t} : \mathtt{T}]\!] & = & \exists\bar{\mathtt{X}}.[\![\mathtt{t} : \mathtt{T}]\!] & \text{provided } \bar{\mathtt{X}} \# \mathit{ftv}(\mathtt{T}) \\ [\![(\mathtt{t} : \mathtt{T}) : \mathtt{T}']\!] & = & [\![\mathtt{t} : \mathtt{T}]\!] \wedge \mathtt{T} \leq \mathtt{T}' & \end{array}$$

The effect of these rules is simple. The construct $\exists\bar{\mathtt{X}}.\mathtt{t}$ is an indication to the constraint generator that the type variables $\bar{\mathtt{X}}$, which may occur free within type annotations inside $\mathtt{t}$, should be existentially bound at this point. The side-condition $\bar{\mathtt{X}} \# \mathit{ftv}(\mathtt{T})$ ensures that quantifying over $\bar{\mathtt{X}}$ in the generated constraint does not capture type variables in the expected type $\mathtt{T}$. It can always be satisfied by α-conversion of the expression $\exists\bar{\mathtt{X}}.\mathtt{t}$. The construct $(\mathtt{t} : \mathtt{T})$ is an indication to the constraint generator that the expression $\mathtt{t}$ should have type $\mathtt{T}$, and it is treated as such by generating the subconstraint $[\![\mathtt{t} : \mathtt{T}]\!]$. The expression's type may be an arbitrary supertype of $\mathtt{T}$, hence the auxiliary constraint $\mathtt{T} \leq \mathtt{T}'$.

1.9.31 EXAMPLE: In DM extended with pairs, the expression $\lambda\mathtt{z}_1.\lambda\mathtt{z}_2.\exists\mathtt{X}.((\mathtt{z}_1 : \mathtt{X}), (\mathtt{z}_2 : \mathtt{X}))$ has most general type $\forall\mathtt{X}.\mathtt{X} \to \mathtt{X} \to \mathtt{X} \times \mathtt{X}$. Indeed, the constraint generated for this expression contains the pattern $\exists\mathtt{X}.([\![\mathtt{z}_1 : \mathtt{X}]\!] \wedge [\![\mathtt{z}_2 : \mathtt{X}]\!] \wedge \ldots)$, which causes $\mathtt{z}_1$ and $\mathtt{z}_2$ to receive the same type. Note that this style is more flexible than that employed in Example 1.9.30, where we were forced to use a single, monolithic type annotation to express this sharing constraint. □

1.9.32 REMARK: In practice, a type variable is usually represented as a memory cell in the typechecker's heap. So, one cannot say that the source code contains type variables; rather, it contains *names* that are meant to stand for type variables. Let us write X for such a name, and T for a type made of type constructors and names, rather than of type constructors and type variables. Then, our new expression forms are really $\exists\bar{\mathtt{X}}.\mathtt{t}$ and $(\mathtt{t} : \mathsf{T})$. When the constraint generator enters the scope of an introduction form $\exists\bar{\mathtt{X}}.\mathtt{t}$, it allocates a

vector of fresh type variables $\bar{X}$, and augments an internal environment with the bindings $\bar{X} \mapsto \bar{X}$. Because the type variables are fresh, the side-condition of the first constraint generation rule above is automatically satisfied. When the constraint generator finds a type annotation $(t : T)$, it looks up the internal environment to translate the type annotation T into an internal type T—which fails if T contains a name that is not in scope—and applies the second constraint generation rule above. □

1.9.33 EXERCISE [★★, ↛]: Let $\bar{X} \supseteq ftv(T)$ and $\bar{X} \# ftv(t)$. Check that the constraints $[\![(t : \exists\bar{X}.T) : T']\!]$ and $[\![\exists\bar{X}.(t : T) : T']\!]$ are equivalent. In other words, the *local* type annotations introduced earlier may be expressed in terms of the more complex constructs described above. □

1.9.34 EXERCISE [★★, ↛]: One way of giving identity semantics to our new type annotation constructs is to *erase* them altogether prior to execution. Give an inductive definition of $\lfloor t \rfloor$, the expression obtained by removing all type annotation constructs from the expression t. Check that $[\![t : T]\!]$ entails $[\![\lfloor t \rfloor : T]\!]$ and explain why this is sufficient to ensure type soundness. □

It is interesting to study how explicit introduction of existentially quantified type variables interacts with `let`-polymorphism. The source of their interaction lies in the difference between the constraints $\mathsf{let}\ z : \forall\bar{X}[\exists X.C_1].T\ \mathsf{in}\ C_2$ and $\exists X.\mathsf{let}\ z : \forall\bar{X}[C_1].T\ \mathsf{in}\ C_2$, which was explained in Example 1.3.48. In the former constraint, every free occurrence of z inside C_2 causes a copy of $\exists X.C_1$ to be taken, thus creating its own fresh copy of X. In the latter constraint, on the other hand, every free occurrence of z inside C_2 produces a copy of C_1. All such copies *share* references to X, because its quantifier was not duplicated. In the former case, one may say that the type scheme assigned to z is *polymorphic* with respect to X, while in the latter case it is *monomorphic*. As a result, the placement of type variable introduction expressions with respect to `let` bindings in the source code is meaningful: introducing a type variable *outside* of a `let` construct *prevents* it from being generalized.

1.9.35 EXAMPLE: In DM extended with integers and Booleans, the program $\texttt{let f} = \exists X.\lambda z.(\texttt{z} : X)\ \texttt{in}\ (\texttt{f}\ 0, \texttt{f}\ \texttt{true})$ is well-typed. Indeed, the type scheme assigned to `f` is $\forall X.X \to X$. However, the program $\exists X.\texttt{let f} = \lambda z.(\texttt{z} : X)\ \texttt{in}\ (\texttt{f}\ 0, \texttt{f}\ \texttt{true})$ is ill-typed. Indeed, the type scheme assigned to `f` is $X \to X$; then, no value of X satisfies the constraints associated with the applications `f 0` and `f true`. The latter behavior is observed in Objective Caml, where type variables are *implicitly* introduced at the outermost level of expressions:

```
# let f z = (z:'a) in (f 0, f true);;
This expression has type bool but is here used with type int
```

More details about the treatment of type annotations in Standard ML, Objective Caml, and Haskell are given on page 148. □

1.9.36 EXERCISE [★, ↛]: Determine which constraints are generated for the two programs in Example 1.9.35. Check that the former is indeed well-typed, while the latter is ill-typed. □

1.9.6 Recursive types

We have shown that specializing HM(X) with an equality-only syntactic model yields HM(=), a constraint-based formulation of Damas and Milner's type system. Similarly, it is possible to specialize HM(X) with an equality-only free *regular* tree model, yielding a constraint-based type system that may be viewed as an extension of Damas and Milner's type discipline with recursive types. This flavor of recursive types is sometimes known as *equirecursive*, since cyclic *equations*, such as $\mathtt{X} = \mathtt{X} \to \mathtt{X}$, are then satisfiable. Our theorems about type inference and type soundness, which are independent of the model, remain valid. The constraint solver described in §1.8 may be used in the setting of an equality-only free regular tree model: the only difference with the syntactic case is that the occurs check is no longer performed.

Please note that, although *ground types* are regular, *types* remain finite objects: their syntax is unchanged. The μ notation commonly employed to describe recursive types may be emulated using type equations: for instance, the notation $\mu\mathtt{X}.\mathtt{X} \to \mathtt{X}$ corresponds, in our constraint-based approach, to the type scheme $\forall\mathtt{X}[\mathtt{X} = \mathtt{X} \to \mathtt{X}].\mathtt{X}$.

Although recursive types come for free, as explained above, they have not been adopted in mainstream programming languages based on ML-the-type-system. The reason is pragmatic: experience shows that many nonsensical expressions are well-typed in the presence of recursive types, whereas they are not in their absence. Thus, the gain in expressiveness is offset by the fact that many programming mistakes are detected later than otherwise possible. Consider, for instance, the following `OCaml` session:

```
ocaml -rectypes
# let rec map f = function
    | [] → []
    | x :: l → (map f x) :: (map f l);;
val map : 'a → ('b list as 'b) → ('c list as 'c) = <fun>
```

This nonsensical version of `map` is essentially useless, yet well-typed. Its principal type scheme, in our notation, is $\forall\mathtt{XYZ}[\mathtt{Y} = \mathsf{list}\,\mathtt{Y} \wedge \mathtt{Z} = \mathsf{list}\,\mathtt{Z}].\mathtt{X} \to \mathtt{Y} \to \mathtt{Z}$. In the absence of recursive types, it is ill-typed, since the constraint $\mathtt{Y} = \mathsf{list}\,\mathtt{Y} \wedge \mathtt{Z} = \mathsf{list}\,\mathtt{Z}$ is then false.

The need for equirecursive types is usually suppressed by the presence of algebraic data types, which offer *isorecursive* types, in the language. Yet, they are still necessary in some situations, such as in Objective Caml's extensions with objects (Rémy and Vouillon, 1998) or polymorphic variants (Garrigue, 1998, 2000, 2002), where recursive object or variant types are commonly inferred. In order to allow recursive object or variant types while still rejecting the above version of `map`, Objective Caml's constraint solver implements a selective occurs check, which forbids cycles unless they involve the type constructors $\langle\cdot\rangle$ or $[\cdot]$ respectively associated with objects and variants. The corresponding model is a tree model where every infinite path down a tree must encounter the type constructor $\langle\cdot\rangle$ or $[\cdot]$ infinitely often.

1.10 Universal quantification in constraints

The constraint logic studied so far allows a set of variables $\bar{\mathrm{x}}$ to be existentially quantified within a formula C. The resulting formula $\exists\bar{\mathrm{x}}.\mathrm{C}$ receives its standard meaning: it requires C to hold for *some* $\bar{\mathrm{x}}$. However, we currently have no way of requiring a formula C to hold for *all* $\bar{\mathrm{x}}$. Is it possible to extend our logic with universal quantification? If so, what are the new possibilities offered by this extension, in terms of type inference? The present section proposes some answers to these questions.

It is worth noting that, although the standard notation for type schemes involves the symbol $\forall$, type scheme introduction and instantiation constraints do not allow an encoding of universal quantification. Indeed, a universal quantifier in a type scheme is very much like an existential quantifier in a constraint: this is suggested, for instance, by Definition 1.3.3 and by C-LetEx.

1.10.1 Constraints

We extend the syntax of constraints as follows:

$$\mathrm{C} ::= \ldots \mid \forall\bar{\mathrm{x}}.\mathrm{C}$$

Universally quantified variables are often referred to as *rigid*, while existentially quantified variables are known as *flexible*. The logical interpretation of constraints (Figure 1-5) is extended as follows:

$$\frac{\forall\vec{\mathrm{t}} \quad \phi[\vec{\mathrm{x}} \mapsto \vec{\mathrm{t}}] \models \mathsf{def}\ \Gamma\ \mathsf{in}\ \mathrm{C} \qquad \bar{\mathrm{x}} \# \mathit{ftv}(\Gamma)}{\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ \forall\bar{\mathrm{x}}.\mathrm{C}} \qquad \text{(CM-ForAll)}$$

We let the reader check that none of the results established in §1.3 are affected by this addition. Furthermore, the extended constraint language enjoys the following properties.

1.10.1 LEMMA: $\forall\bar{X}.C \Vdash C$. Conversely, $\bar{X} \# \mathit{ftv}(C)$ implies $C \Vdash \forall\bar{X}.C$. □

Proof: Assume $\bar{X} \# \mathit{ftv}(\Gamma)$ **(1)** and $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ \forall\bar{X}.C$ **(2)**. By (1), (2), and CM-FORALL, we have $\forall\vec{t}\quad \phi[\vec{X} \mapsto \vec{t}] \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C$, which implies $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C$ **(3)**. Lemma 1.3.15 allows discharging (1) and shows that $\forall\bar{X}.C$ entails C. Conversely, let $\bar{X} \# \mathit{ftv}(C)$ **(4)**. Then, by Lemma 1.3.12, (1), and CM-FORALL, (3) implies (2). We conclude as above. □

1.10.2 LEMMA: $\bar{X} \# \mathit{ftv}(C_2)$ implies $\forall\bar{X}.(C_1 \wedge C_2) \equiv (\forall\bar{X}.C_1) \wedge C_2$. □

Proof: Analogous to the proof of Lemma 1.3.20. □

1.10.3 LEMMA: $\forall\bar{X}.\forall\bar{Y}.C \equiv \forall\bar{X}\bar{Y}.C$. □

Proof: Analogous to the proof of Lemma 1.3.23. □

1.10.4 LEMMA: Let $\bar{X} \# \bar{Y}$. Then, $\exists\bar{X}.\forall\bar{Y}.C$ entails $\forall\bar{Y}.\exists\bar{X}.C$. Conversely, if $\exists\bar{Y}.C$ determines $\bar{X}$, then $\forall\bar{Y}.\exists\bar{X}.C$ entails $\exists\bar{X}.\forall\bar{Y}.C$. □

Proof: Let $\bar{X} \# \bar{Y}$ **(1)** and $\bar{X}\bar{Y} \# \mathit{ftv}(\Gamma)$ **(2)**. Then, by (2), CM-EXISTS, and CM-FORALL, the assertion $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ \exists\bar{X}.\forall\bar{Y}.C$ is equivalent to $\exists\vec{t}\quad \forall\vec{t}'\quad \phi[\vec{X} \mapsto \vec{t}][\vec{Y} \mapsto \vec{t}'] \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C$ **(3)**. Similarly, the assertion $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ \forall\bar{Y}.\exists\bar{X}.C$ is equivalent to $\forall\vec{t}'\quad \exists\vec{t}\quad \phi[\vec{Y} \mapsto \vec{t}'][\vec{X} \mapsto \vec{t}] \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C$ **(4)**. By (1), (3) implies (4). Lemma 1.3.15 allows discharging (2) and shows that $\exists\bar{X}.\forall\bar{Y}.C$ entails $\forall\bar{Y}.\exists\bar{X}.C$.

Conversely, assume $\exists\bar{Y}.C$ determines $\bar{X}$ **(5)**. Assume (4) holds. By (1), (2), and CM-EXISTS, $\phi[\vec{Y} \mapsto \vec{t}'][\vec{X} \mapsto \vec{t}] \models \mathsf{def}\ \Gamma\ \mathsf{in}\ C$ implies $\phi[\vec{X} \mapsto \vec{t}] \models \mathsf{def}\ \Gamma\ \mathsf{in}\ \exists\bar{Y}.C$. Thus, when $\vec{t}'$ varies (and, presumably, $\vec{t}$ varies as well), the ground assignments $\phi[\vec{X} \mapsto \vec{t}]$ satisfy $\mathsf{def}\ \Gamma\ \mathsf{in}\ \exists\bar{Y}.C$, and coincide outside of $\bar{X}$. By (5) and Definition 1.3.44, they must coincide on $\bar{X}$ as well: that is, $\vec{t}'$ does *not* vary when $\vec{t}$ varies. Thus, (3) holds. We conclude as above. □

1.10.2 Constraint solving

We briefly explain how to extend the constraint solver described in §1.8 with support for universal quantification. (Thus, we again assume an equality-only free tree model.) Constraint solving in the presence of equations and of existential and universal quantifiers is known as *unification under a mixed prefix*. It is a particular case of the decision problem for the first-order theory of

$$S; U; \forall \bar{X}.C \rightarrow S[\forall \bar{X}.[]]; U; C \quad \text{if } \bar{X} \# \mathit{ftv}(U) \qquad \text{(S-SOLVE-ALL)}$$

$$S[\forall \bar{X}.\exists \bar{Y}\bar{Z}.[]]; U; \mathsf{true} \rightarrow S[\exists \bar{Y}.\forall \bar{X}.\exists \bar{Z}.[]]; U; \mathsf{true} \quad \text{if } \bar{X} \# \bar{Y} \wedge \exists \bar{X}\bar{Z}.U \text{ determines } \bar{Y} \qquad \text{(S-ALLEX)}$$

$$S[\forall \bar{X}X.\exists \bar{Y}.[]]; U; \mathsf{true} \rightarrow \mathsf{false} \quad \text{if } X \notin \bar{Y} \wedge X \prec^*_U Z \wedge Z \notin X\bar{Y} \qquad \text{(S-ALL-FAIL-1)}$$

$$S[\forall \bar{X}X.\exists \bar{Y}.[]]; X = T = \epsilon \wedge U; \mathsf{true} \rightarrow \mathsf{false} \quad \text{if } X \notin \bar{Y} \wedge T \notin \mathcal{V} \qquad \text{(S-ALL-FAIL-2)}$$

$$S[\forall \bar{X}.\exists \bar{Y}.[]]; U_1 \wedge U_2; \mathsf{true} \rightarrow S; U_1; \mathsf{true} \quad \text{if } \bar{X}\bar{Y} \# \mathit{ftv}(U_1) \wedge \exists \bar{Y}.U_2 \equiv \mathsf{true} \qquad \text{(S-POP-ALL)}$$

Figure 1-16: Solving universal constraints

equality on trees; see *e.g.* (Comon and Lescanne, 1989). Extending our solver is straightforward: in fact, the treatment of universal quantifiers turns out to be surprisingly analogous to that of let constraints. To begin, we extend the syntax of stacks with so-called *universal frames*:

$$S ::= \ldots \mid S[\forall \bar{X}.[]]$$

Because existential quantifiers cannot, in general, be hoisted out of universal quantifiers, rules S-EX-1 to S-EX-4 now allow floating them up to the nearest enclosing let *or universal* frame, if any, or to the outermost level, otherwise. Thus, in our machine representation of stacks, where rules S-EX-1 to S-EX-4 are applied in an eager fashion, every universal frame carries a list of the type variables that are existentially bound immediately after it, and integer ranks count not only let frames, but also universal frames.

The solver's specification is extended with the rules in Figure 1-16. S-SOLVE-ALL, a forward rule, discovers a universal constraint and enters it, creating a new universal frame to record its existence. S-ALLEX exploits Lemma 1.10.4 to hoist existential quantifiers out of the universal frame. It is analogous to S-LETALL, and its implementation may rely upon the same procedure (Exercise 1.8.8). The next two rules detect failure conditions. S-ALL-FAIL-1 states that the constraint $\forall X.\exists \bar{Y}.U$ is false if the rigid variable X is directly or indirectly *dominated* by a *free* variable Z. Indeed, the value of X is then determined by that of Z—but a universally quantified variable ranges over *all* values, so this is a contradiction. In such a case, X is com-

monly said to *escape its scope*. S-ALL-FAIL-2 states that the same constraint is false if X is equated with a *nonvariable* term. Indeed, the value of X is then partially determined, since its head constructor is known, which again contradicts its universal status. Last, S-POP-ALL splits the current unification constraint into two components U_1 and U_2, where U_1 is made up entirely of *old* variables and U_2 constrains *young* variables only. This decomposition is analogous to that performed by S-POP-LET. Then, it is not difficult to check that $\forall\mathfrak{X}.\exists\mathfrak{Y}.(\mathrm{U}_1 \wedge \mathrm{U}_2)$ is equivalent to U_1. So, the universal frame, as well as U_2, are discarded, and the solver proceeds by examining whatever remains on top of the stack S.

It is possible to further extend the treatment of universal frames with two rules analogous to S-COMPRESS and S-UNNAME. In practice, this improves the solver's efficiency, and makes it easier to share code between the treatment of let frames and that of universal frames.

It is interesting to remark that, as far as the underlying unification algorithm is concerned, there is no difference between existentially and universally quantified type variables. The algorithm solves whatever equations are presented to it, without inquiring about the status of their variables. Equations that lead to failure, because a rigid variable escapes its scope or is equated with a nonvariable term, are detected only when the universal frame is exited. A perhaps more common approach is to *mark* rigid variables as such, allowing the unification algorithm to signal failure as soon as one of the two error conditions is encountered. In this approach, a rigid variable may successfully unify only with itself or with flexible variables fresher than itself. It is often called a *Skolem constructor* in the literature (Läufer and Odersky, 1994; Shields and Peyton Jones, 2002). An interesting variant of this approach appears in Dowek, Hardin, Kirchner and Pfenning's treatment of (higher-order) unification (1995; 1998), where flexible variables are represented as ordinary variables, while rigid variables are encoded using De Bruijn indices.

The properties of our constraint solver are preserved by this extension, as shown by the following lemmas, whose statements are identical to those of Lemmas 1.8.9, 1.8.10, and 1.8.11.

1.10.5 LEMMA: The reduction system $\rightarrow$ is strongly normalizing. □

1.10.6 LEMMA: $\mathrm{S}; \mathrm{U}; \mathrm{C} \rightarrow \mathrm{S}'; \mathrm{U}'; \mathrm{C}'$ implies $\mathrm{S}[\mathrm{U} \wedge \mathrm{C}] \equiv \mathrm{S}'[\mathrm{U}' \wedge \mathrm{C}']$. □

Proof: By examination of every rule.

- *Case* S-SOLVE-ALL. By Lemma 1.10.2.
- *Case* S-ALLEX. By Lemma 1.10.4.

∘ *Case* S-ALL-FAIL-1. Let $X \notin \bar{Y}$ **(1)** and $X \prec^{\star}_{U} Z$ **(2)** and $Z \notin X\bar{Y}$ **(3)**. By (2), there exists a path π such that, for every ground assignment ϕ that satisfies U, $\phi(X)$ is $\phi(Z)/\pi$, that is, the subtree of $\phi(Z)$ rooted at π. By (1) and (3), the same holds for every ground assignment ϕ that satisfies $\exists\bar{Y}.U$. Let us now assume that ϕ satisfies $\forall X.\exists\bar{Y}.U$ **(4)**. By CM-FORALL, for every ground type t, we have $\phi[X \mapsto t] \models \exists\bar{Y}.U$. By (3) and by the above result, this implies $t = \phi(Z)/\pi$. In a nondegenerate model, this cannot possibly hold for every t, so the hypothesis (4) is inconsistent. We have shown that $\forall X.\exists\bar{Y}.U$ is false. The result follows by Lemmas 1.3.50 and 1.10.3.

∘ *Case* S-ALL-FAIL-2. Let $X \notin \bar{Y}$ **(1)** and $T \notin \mathcal{V}$ **(2)** and $\phi \models \forall X.\exists\bar{Y}.(X = T = \epsilon \wedge U)$ **(3)**. By (2), T has a head symbol F. Then, by (1), CM-FORALL, CM-EXISTS, CM-AND, and CM-PREDICATE, (3) implies that every ground type t has the head symbol F. In a nondegenerate model, this is a contradiction, so the hypothesis (3) is inconsistent. We conclude as in the previous case.

∘ *Case* S-POP-ALL. Let $\bar{X}\bar{Y}$ # $ftv(U_1)$ **(1)** and $\exists\bar{Y}.U_2 \equiv \mathsf{true}$ **(2)**. We have $\forall\bar{X}.\exists\bar{Y}.(U_1 \wedge U_2) \equiv \forall\bar{X}.(U_1 \wedge \exists\bar{Y}.U_2) \equiv \forall\bar{X}.U_1 \equiv U_1$, where the first equivalence is by (1) and C-EXAND, the second one is by (2), and the last one follows from (1) and Lemma 1.10.1. □

1.10.7 LEMMA: A normal form for the reduction system $\rightarrow$ is one of (i) $S; U; x \preceq T$, where $x \notin dpi(S)$; (ii) $S; \mathsf{false}; \mathsf{true}$; or (iii) $\mathcal{X}; U; \mathsf{true}$, where $\mathcal{X}$ is an existential constraint context and U a satisfiable conjunction of multi-equations. □

Proof: It is clear that, thanks to S-SOLVE-ALL, the analysis of the structure of the external constraint remains exhaustive. There remains to consider a state of the form $S[\forall\bar{X}.\exists\bar{Y}.[]]; U; \mathsf{true}$, where U is a standard conjunction of multi-equations and, if the model is syntactic, U is acyclic. We may assume, *w.l.o.g.*, $\bar{X}$ # $\bar{Y}$. Let us write U as $U_1 \wedge U_2$, where $\bar{X}\bar{Y}$ # $ftv(U_1)$, and where U_1 is maximal for this criterion. Then, consider a multi-equation $\epsilon \in U$, one of whose variable members is outside $\bar{X}\bar{Y}$. Because the state at hand is a normal form with respect to S-ALLEX, we may deduce, as in the proof of Lemma 1.8.11, that all variables in $ftv(\epsilon)$ must be outside $\bar{Y}$. Furthermore, because it is a normal form with respect to S-ALL-FAIL-1, all variables in $ftv(\epsilon)$ must be outside $\bar{X}$ as well. By definition of U_1, this implies $\epsilon \in U_1$. By contraposition, for every multi-equation $\epsilon \in U_2$, *all variable members of* ϵ *are in* $\bar{X}\bar{Y}$. Last, if $\epsilon \in U_2$ and X is a variable member of ϵ, where $X \in \bar{X}$, then, because the state at hand is a normal form with respect to S-ALL-FAIL-1 and S-ALL-FAIL-2, *every member of* ϵ *is either* X *itself or inside* $\bar{Y}$. Thus, every rigid variable X must be a minimal element with respect to $\prec_{U_2}$. Let us now recall that U_2 is a standard conjunction of multi-equations and, if the model is syntactic, U_2 is acyclic. We let the reader check that this implies $\exists\bar{Y}.U_2 \equiv \mathsf{true}$; the proof is

again a slight generalization of the last part of that of Lemma 1.8.6. Then, the state at hand is reducible via S-POP-ALL. This is a contradiction, so this case cannot arise. □

1.10.3 Type annotations, continued

In §1.9, we introduced the expression form $(\mathtt{t} : \exists \bar{\mathtt{X}}.\mathtt{T})$, allowing an expression $\mathtt{t}$ to be annotated with a type $\mathtt{T}$ whose free variables $\bar{\mathtt{X}}$ are *locally* and *existentially* bound. It is now natural to introduce the symmetric expression form $(\mathtt{t} : \forall \bar{\mathtt{X}}.\mathtt{T})$, where $\mathtt{T}$ has kind $\star$, $\bar{\mathtt{X}}$ is bound within $\mathtt{T}$, and $\bar{\mathtt{X}}$ contains $\mathit{ftv}(\mathtt{T})$, as before. Its constraint generation rule is as follows:

$$[\![(\mathtt{t} : \forall \bar{\mathtt{X}}.\mathtt{T}) : \mathtt{T}']\!] \quad = \quad \forall \bar{\mathtt{X}}.[\![\mathtt{t} : \mathtt{T}]\!] \wedge \exists \bar{\mathtt{X}}.(\mathtt{T} \leq \mathtt{T}') \quad \text{provided } \bar{\mathtt{X}} \# \mathit{ftv}(\mathtt{t}, \mathtt{T}')$$

The first conjunct requires $\mathtt{t}$ to have type $\mathtt{T}$ for *all* values of $\bar{\mathtt{X}}$. Here, the type variables $\bar{\mathtt{X}}$ are *universally* bound, as expected. The second conjunct requires $\mathtt{T}'$ to be *some* instance of the universal annotation $\forall \bar{\mathtt{X}}.\mathtt{T}$. Since $\mathtt{T}'$ is only a monotype, it seems difficult to think of another sensible way of constraining $\mathtt{T}'$. For this reason, the type variables $\bar{\mathtt{X}}$ are still *existentially* bound in the second conjunct. This makes the interpretation of the universal quantifier in type annotations a bit more complex than that of the existential quantifier. For instance, when subtyping is interpreted as equality, the constraint generation rule may be read: *a valid type for* $(\mathtt{t} : \forall \bar{\mathtt{X}}.\mathtt{T})$ *is of the form* $\mathtt{T}$, *for* some *choice of the type variables* $\bar{\mathtt{X}}$, *provided* $\mathtt{t}$ *has type* $\mathtt{T}$ *for* all *choices of* $\bar{\mathtt{X}}$.

We remark that $(\mathtt{t} : \forall \bar{\mathtt{X}}.\mathtt{T})$ must be a new expression form: it cannot be encoded by adding new constants to the calculus—whereas $(\mathtt{t} : \exists \bar{\mathtt{X}}.\mathtt{T})$ could—because none of the existing constraint generation rules produce universally quantified constraints. Like all type annotations, it has identity semantics.

What is the use of universal type annotations, compared with existential type annotations? When a type variable is existentially bound, the typechecker is free to assign it whatever value makes the program well-typed. As a result, the expressions $(\lambda \mathtt{z}.\mathtt{z} \mathbin{\hat{+}} \hat{1} : \exists \mathtt{X}.\mathtt{X} \to \mathtt{X})$ and $(\lambda \mathtt{z}.\mathtt{z} : \exists \mathtt{X}.\mathtt{X} \to \mathtt{X})$ are both well-typed: $\mathtt{X}$ is assigned int in the former case, and remains undetermined in the latter. However, it is sometimes useful to be able to insist that an expression should be polymorphic. This effect is naturally achieved by using a universally bound type variable. Indeed, $(\lambda \mathtt{z}.\mathtt{z} \mathbin{\hat{+}} \hat{1} : \forall \mathtt{X}.\mathtt{X} \to \mathtt{X})$ is ill-typed, because $\forall \mathtt{X}.(\mathtt{X} = \mathsf{int})$ is false, while $(\lambda \mathtt{z}.\mathtt{z} : \forall \mathtt{X}.\mathtt{X} \to \mathtt{X})$ is well-typed.

1.10.8 EXERCISE [★]: Write down the constraints $\exists \mathtt{Z}.[\![(\lambda \mathtt{z}.\mathtt{z} \mathbin{\hat{+}} \hat{1} : \forall \mathtt{X}.\mathtt{X} \to \mathtt{X}) : \mathtt{Z}]\!]$ and $\exists \mathtt{Z}.[\![(\lambda \mathtt{z}.\mathtt{z} : \forall \mathtt{X}.\mathtt{X} \to \mathtt{X}) : \mathtt{Z}]\!]$, which tell whether these expressions are well-typed. Check that the former is false, while the latter is satisfiable. □

A universal type annotation, as defined above, is nothing but a (closed)

Damas-Milner type scheme. Thus, the new construct $(\mathtt{t} : \forall\bar{\mathtt{X}}.\mathtt{T})$ gives us the ability to ensure that the expression $\mathtt{t}$ admits the type scheme $\forall\bar{\mathtt{X}}.\mathtt{T}$. This feature is exploited at the module level in ML-the-programming-language, where it is necessary to check that the inferred type for a module component $\mathtt{t}$ is more general than the type scheme $\mathtt{S}$ that appears in the module's signature. In our view, this process simply consists in ensuring that $(\mathtt{t} : \mathtt{S})$ is well-typed.

In §1.9, we have pointed out that local (that is, closed) type annotations offer limited expressiveness, because they cannot share type variables. To lift this limitation, we have introduced the expression forms $\exists\bar{\mathtt{X}}.\mathtt{t}$ and $(\mathtt{t} : \mathtt{T})$. The former binds the type variables $\bar{\mathtt{X}}$ within $\mathtt{t}$, making them available for use in type annotations, and instructs the constraint generator to existentially quantify them at this point. The latter requires $\mathtt{t}$ to have $\mathtt{T}$. It is natural to proceed in the same manner in the case of universal type annotations. We now introduce the expression form $\forall\bar{\mathtt{X}}.\mathtt{t}$, which also binds $\bar{\mathtt{X}}$ within $\mathtt{t}$, but comes with a different constraint generation rule:

$$[\![\forall\bar{\mathtt{X}}.\mathtt{t} : \mathtt{T}]\!] \quad = \quad \forall\bar{\mathtt{X}}.\exists\mathtt{Z}.[\![\mathtt{t} : \mathtt{Z}]\!] \wedge \exists\bar{\mathtt{X}}.[\![\mathtt{t} : \mathtt{T}]\!] \quad \text{provided } \bar{\mathtt{X}} \# \mathit{ftv}(\mathtt{T}) \wedge \mathtt{Z} \notin \mathit{ftv}(\mathtt{t})$$

This rule is a bit more complex than that associated with the expression form $\exists\bar{\mathtt{X}}.\mathtt{t}$. Again, this is due to the fact that we do not wish to overconstrain $\mathtt{T}$. The first exercise below shows that a more naïve version of the rule does not yield the desired behavior. The second exercise shows that this version does. The third exercise clarifies an efficiency concern.

1.10.9 EXERCISE [★]: Assume that $[\![\forall\bar{\mathtt{X}}.\mathtt{t} : \mathtt{T}]\!]$ is defined as $\forall\bar{\mathtt{X}}.[\![\mathtt{t} : \mathtt{T}]\!]$, provided $\bar{\mathtt{X}} \# \mathit{ftv}(\mathtt{T})$. Write down the constraint $[\![\forall\mathtt{X}.(\lambda\mathtt{z}.\mathtt{z} : \mathtt{X} \to \mathtt{X}) : \mathtt{Z}]\!]$. Can you describe its solutions? Does it have the intended meaning? □

1.10.10 EXERCISE [★★]: Let $\bar{\mathtt{X}} \supseteq \mathit{ftv}(\mathtt{T})$ and $\bar{\mathtt{X}} \# \mathit{ftv}(\mathtt{t})$. Check that the constraints $[\![(\mathtt{t} : \forall\bar{\mathtt{X}}.\mathtt{T}) : \mathtt{T}']\!]$ and $[\![\forall\bar{\mathtt{X}}.(\mathtt{t} : \mathtt{T}) : \mathtt{T}']\!]$ are equivalent. In other words, *local* universal type annotations may also be expressed in terms of the more complex constructs described above. □

1.10.11 EXERCISE [★★★★, ↛]: The constraint generation rule that appears above compromises the linear time and space complexity of constraint generation, because it duplicates the term $\mathtt{t}$. It is possible to avoid this problem, but this requires a slight generalization of the constraint language. Let us write $\mathsf{let}\ \mathtt{x} : \forall\underline{\bar{\mathtt{X}}}\bar{\mathtt{Y}}[\mathsf{C}_1].\mathtt{T}\ \mathsf{in}\ \mathsf{C}_2$ for $\forall\bar{\mathtt{X}}.\exists\bar{\mathtt{Y}}.\mathsf{C}_1 \wedge \mathsf{def}\ \mathtt{x} : \forall\bar{\mathtt{X}}\bar{\mathtt{Y}}[\mathsf{C}_1].\mathtt{T}\ \mathsf{in}\ \mathsf{C}_2$. In this extended let form, the underlined variables $\bar{\mathtt{X}}$ are interpreted as *rigid*, instead of *flexible*, while checking that C_1 is satisfiable. However, the type scheme associated with $\mathtt{x}$ is not affected. *Check that the above constraint generation rule may now be*

written as follows:

$$[\![\forall \bar{X}.t : T]\!] \quad = \quad \mathsf{let}\ x : \forall \bar{X}\underline{Z}[[\![t : Z]\!]].Z\ \mathsf{in}\ x \preceq T \quad \text{provided } Z \notin \mathit{ftv}(t)$$

Roughly speaking, the new rule forms a most general type scheme for t, ensures that the type variables $\bar{X}$ are unconstrained in it, and checks that T is an instance of it. Furthermore, it does not duplicate t. To complete the exercise, *extend the specification of the constraint solver (Figures 1-12 and 1-16), as well as its implementation,* to deal with this extension of the constraint language. □

To conclude, let us once again stress that, if T has free type variables, the effect of the type annotation $(t : T)$ depends on *how* and *where* they are bound. The effect of *how* stems from the fact that binding a type variable universally, rather than existentially, leads to a stricter constraint. Indeed, we let the reader check that $[\![\forall \bar{X}.t : T]\!]$ entails $[\![\exists \bar{X}.t : T]\!]$, while the converse does not hold in general. The effect of *where* has been illustrated, in the case of existentially bound type variables, in §1.9. It is due, in that case, to the fact that let and $\exists$ do not commute. In the case of universally bound type variables, it may be imputed to the fact that $\forall$ and $\exists$ do not commute. For instance, $\lambda z.\forall X.(z : X)$ is ill-typed, because *inside the* λ*-abstraction*, the program variable z cannot be said to have every type. However, $\forall X.\lambda z.(z : X)$ is well-typed, because the identity function does have type $X \to X$ for every X.

1.10.12 EXERCISE [★]: Write down the constraints $\exists Z.[\![\lambda z.\forall X.(z : X) : Z]\!]$ and $\exists Z.[\![\forall X.\lambda z.(z : X) : Z]\!]$, which tell whether these expressions are well-typed. Is the former satisfiable? Is the latter? □

In Standard ML and Objective Caml, the type variables that appear in type annotations are *implicitly* bound. That is, there is no syntax in the language for the constructs $\exists \bar{X}.t$ and $\forall \bar{X}.t$. When a type annotation $(t : T)$ contains a free type variable X, a fixed convention tells how and where X is bound. In Standard ML, X is *universally* bound at the nearest `val` binding that encloses all related occurrences of X (Milner, Tofte, and Harper, 1990). The 1997 revision of Standard ML (Milner, Tofte, Harper, and MacQueen, 1997b) slightly improves on this situation by allowing type variables to be *explicitly* introduced at `val` bindings. However, they still must be universally bound. In Objective Caml, X is *existentially* bound at the nearest enclosing toplevel `let` binding; this behavior seems to be presently undocumented. We argue that (i) allowing type variables to be implicitly introduced is confusing; and (ii) for expressiveness, both universal and existential quantifiers should be made available to programmers. Surprisingly, these language design and type inference issues seem to have received little attention in the literature,

although they have most likely been "folklore" for a long time. Peyton Jones and Shields (2003) study these issues in the context of Haskell, and concur with (i). Concerning (ii), they seem to think that the language designer must choose between existential and universal type variable introduction forms—which they refer to as "type-sharing" and "type-lambda"—whereas we point out that they may and should coexist.

1.10.4 Polymorphic recursion

Example 1.2.10 explains how the `letrec` construct found in ML-the-programming-language may be viewed as an application of the constant `fix`, wrapped inside a normal `let` construct. Exercise 1.9.7 shows that this gives rise to a somewhat restrictive constraint generation rule: generalization occurs only *after* the application of `fix` is typechecked. In other words, in `letrec f` $=$ λ`z.`$\mathtt{t}_1$ `in` $\mathtt{t}_2$, all occurrences of `f` within $\mathtt{t}_1$ must have the same (monomorphic) *type*. This restriction is sometimes a nuisance, and seems unwarranted: if the function that is being defined is polymorphic, it should be possible to use it at different types even inside its own definition. Indeed, Mycroft (1984) extended Damas and Milner's type system with a more liberal treatment of recursion, commonly known as *polymorphic recursion*. The idea is to only request occurrences of `f` within $\mathtt{t}_1$ to have the same *type scheme*. Hence, they may have different *types*, all of which are instances of a common type scheme that should also be a valid type scheme for $\mathtt{t}_1$. It was later shown that well-typedness in Mycroft's extended type system is undecidable (Henglein, 1993; Kfoury, Tiuryn, and Urzyczyn, 1993a,b). To work around this stumbling block, one solution is to use a semi-algorithm, falling back to monomorphic recursion if it does not succeed or fail in reasonable time. Although such a solution might be appealing in the setting of an automated program analysis, it is less so in the setting of a programmer-visible type system, because it may become difficult to understand why a program is ill-typed. Thus, we describe a simpler solution, which consists in requiring the programmer to explicitly supply a type scheme for `f`. This is an instance of a *mandatory* type annotation.

To begin, we must change the status of `fix`, because if `fix` remains a constant, then `f` must remain λ-bound and cannot receive a polymorphic type scheme. We turn `fix` into a language construct, which binds a program variable `f`, and annotates it with a DM type scheme. The syntax of values and expressions is thus extended as follows:

$$\mathtt{v} ::= \ldots \mid \mathtt{fix\ f : S.}\lambda\mathtt{z.t} \qquad \mathtt{t} ::= \ldots \mid \mathtt{fix\ f : S.}\lambda\mathtt{z.t}$$

Please note that `f` is bound within λ`z.t`. The operational semantics is ex-

tended as follows.

$$(\texttt{fix f : S.}\lambda\texttt{z.t})\ \texttt{v} \longrightarrow (\texttt{let f = fix f : S.}\lambda\texttt{z.t in }\lambda\texttt{z.t})\ \texttt{v} \qquad \text{(R-FIX')}$$

The type annotation S plays no essential role in the reduction; it is merely preserved. It is now possible to define $\texttt{letrec f : S} = \lambda\texttt{z.t}_1\ \texttt{in t}_2$ as syntactic sugar for $\texttt{let f} = \texttt{fix f : S.}\lambda\texttt{z.t}_1\ \texttt{in t}_2$.

We now give a constraint generation rule for fix:

$$[\![\texttt{fix f : S.}\lambda\texttt{z.t : T}]\!] \quad = \quad \mathsf{let}\ \texttt{f : S}\ \mathsf{in}\ [\![\lambda\texttt{z.t : S}]\!] \wedge \texttt{S} \preceq \texttt{T}$$

The left-hand conjunct requires the function λz.t to have type scheme S, under the assumption that f has type S. Thus, it is now possible for different occurrences of f within t to receive different types. If S is $\forall\bar{\texttt{X}}.\texttt{T}$, where $\bar{\texttt{X}} \# \mathit{ftv}(\texttt{t})$, then we write $[\![\texttt{t : S}]\!]$ for $\forall\bar{\texttt{X}}.[\![\texttt{t : T}]\!]$. Indeed, checking the validity of a polymorphic type annotation—be it mandatory, as is the case here, or optional, as was previously the case—requires a universally quantified constraint. The right-hand conjunct merely constrains T to be an instance of S.

Given the definition of $\texttt{letrec f : S} = \lambda\texttt{z.t}_1\ \texttt{in t}_2$ as syntactic sugar, the above rule leads to the following derived constraint generation rule for letrec:

$$[\![\texttt{letrec f : S} = \lambda\texttt{z.t}_1\ \texttt{in t}_2\texttt{ : T}]\!] \quad = \quad \mathsf{let}\ \texttt{f : S}\ \mathsf{in}\ ([\![\lambda\texttt{z.t}_1\texttt{ : S}]\!] \wedge [\![\texttt{t}_2\texttt{ : T}]\!])$$

This rule is arguably quite natural. The program variable f is assigned the type scheme S throughout its scope, that is, both inside and outside of the function's definition. The function $\lambda\texttt{z.t}_1$ must itself have type scheme S. Last, $\texttt{t}_2$ must have type T, as in every let construct.

1.10.13 EXERCISE [★★]: Prove that the derived constraint generation rule above is indeed valid. □

It is straightforward to prove that the extended language still enjoys subject reduction. The proof relies on the following lemma: if t has type scheme S, then every instance of S is also a valid type for t.

1.10.14 LEMMA: $[\![\texttt{t : S}]\!] \wedge \texttt{S} \preceq \texttt{T} \Vdash [\![\texttt{t : T}]\!]$. □

Proof: Let us write S as $\forall\bar{\texttt{X}}.\texttt{T}'$, where $\bar{\texttt{X}} \# \mathit{ftv}(\texttt{t}, \texttt{T})$ **(1)**. By (1) and by definition, $[\![\texttt{t : S}]\!] \wedge \texttt{S} \preceq \texttt{T}$ stands for $\forall\bar{\texttt{X}}.[\![\texttt{t : T}']\!] \wedge \exists\bar{\texttt{X}}.(\texttt{T}' \leq \texttt{T})$ **(2)**. By C-EXAND and by Lemma 1.10.1, (2) entails $\exists\bar{\texttt{X}}.([\![\texttt{t : T}']\!] \wedge \texttt{T}' \leq \texttt{T})$, which by Lemma 1.6.4 entails $\exists\bar{\texttt{X}}.[\![\texttt{t : T}]\!]$. By (1) and C-EX*, this is $[\![\texttt{t : T}]\!]$. □

1.10.15 THEOREM [SUBJECT REDUCTION]: (R-FIX') $\subseteq (\sqsubseteq)$. □

Proof: The goal is to prove that $[\![(\texttt{fix f : S}.\lambda\texttt{z.t})\ \texttt{v : T}]\!]$ entails $[\![(\texttt{let f} = \texttt{fix f : S}.\lambda\texttt{z.t in } \lambda\texttt{z.t})\ \texttt{v : T}]\!]$. By definition of constraint generation, the former is $\exists\texttt{X}.(\mathsf{let}\ \texttt{f : S}\ \mathsf{in}\ [\![\lambda\texttt{z.t : S}]\!] \wedge \texttt{S} \preceq \texttt{X} \to \texttt{T} \wedge [\![\texttt{v : X}]\!])$, where X is fresh, while the latter is $\exists\texttt{X}.(\mathsf{let}\ \texttt{f : S}\ \mathsf{in}\ ([\![\lambda\texttt{z.t : S}]\!] \wedge [\![\lambda\texttt{z.t : X} \to \texttt{T}]\!]) \wedge [\![\texttt{v : X}]\!])$. The result follows from C-InAnd* and Lemma 1.10.14. □

The programming language Haskell (Hudak, Peyton Jones, Wadler, Boutel, Fairbairn, Fasel, Guzman, Hammond, Hughes, Johnsson, Kieburtz, Nikhil, Partain, and Peterson, 1992) offers polymorphic recursion. Interesting details about its typing rules may be found in (Jones, 1999).

It is worth pointing out that some restricted instances of type inference in the presence of polymorphic recursion are decidable. This is typically the case in certain program analyses, where a type derivation for the program is already available, and the goal is only to infer extra atomic annotations, such as binding time or strictness properties. Several papers that exploit this idea are (Dussart, Henglein, and Mossin, 1995a; Jensen, 1998; Rehof and Fähndrich, 2001).

1.10.5 Universal types

ML-the-type-system enforces a strict stratification between types and type schemes, or, in other words, allows only prenex universal quantifiers inside types. We have pointed out earlier that there is good reason to do so: type inference for ML-the-type-system is decidable, while type inference for System F, which has no such restriction, is undecidable. Yet, this restriction comes at a cost in expressiveness: it prevents higher-order functions from accepting polymorphic function arguments, and forbids storing polymorphic functions inside data structures. Fortunately, it is in fact possible to circumvent the problem by requiring the programmer to supply additional type information.

The approach that we are about to describe is reminiscent of the way algebraic data type definitions allow circumventing the problems associated with equirecursive types (§1.9). Because we do not wish to extend the syntax of types with universal types of the form $\forall\bar{\texttt{Y}}.\texttt{T}$, we instead allow *universal type definitions*, of the form

$$\texttt{D}\,\vec{\texttt{X}} \approx \forall\bar{\texttt{Y}}.\texttt{T}$$

where D still ranges over data types. If D has signature $\vec{\kappa} \Rightarrow \star$, then the type variables $\vec{\texttt{X}}$ must have kind $\vec{\kappa}$. The type T must have kind $\star$. The type variables $\bar{\texttt{X}}$ and $\bar{\texttt{Y}}$ are considered bound within T, and the definition must be closed, that is, $ftv(\texttt{T}) \subseteq \bar{\texttt{X}}\bar{\texttt{Y}}$ must hold. Last, the variance of the type constructor D must match its definition—a requirement stated as follows:

1.10.16 DEFINITION: Let $\mathtt{D}\,\vec{\mathtt{X}} \approx \forall\bar{\mathtt{Y}}.\mathtt{T}$ and $\mathtt{D}\,\vec{\mathtt{X}}' \approx \forall\bar{\mathtt{Y}}'.\mathtt{T}'$ be two α-equivalent instances of a single universal type definition, such that $\bar{\mathtt{Y}} \# \mathit{ftv}(\mathtt{T}')$ and $\bar{\mathtt{Y}}' \# \mathit{ftv}(\mathtt{T})$. Then, $\mathtt{D}\,\vec{\mathtt{X}} \leq \mathtt{D}\,\vec{\mathtt{X}}' \Vdash \forall\bar{\mathtt{Y}}'.\exists\bar{\mathtt{Y}}.\mathtt{T} \leq \mathtt{T}'$ must hold. □

This requirement is analogous to that found in Definition 1.9.10. The idea is, if $\mathtt{D}\,\vec{\mathtt{X}}$ and $\mathtt{D}\,\vec{\mathtt{X}}'$ are comparable, then their unfoldings $\forall\bar{\mathtt{Y}}.\mathtt{T}$ and $\forall\bar{\mathtt{Y}}'.\mathtt{T}'$ should be comparable as well. The comparison between them is expressed by the constraint $\forall\bar{\mathtt{Y}}'.\exists\bar{\mathtt{Y}}.\mathtt{T} \leq \mathtt{T}'$, which may be read: *every instance of* $\forall\bar{\mathtt{Y}}'.\mathtt{T}'$ *is (a supertype of) an instance of* $\forall\bar{\mathtt{Y}}.\mathtt{T}$. Again, when subtyping is interpreted as equality, the requirement of Definition 1.10.16 is always satisfied; it becomes nontrivial only in the presence of true subtyping.

The effect of the universal type definition $\mathtt{D}\,\vec{\mathtt{X}} \approx \forall\bar{\mathtt{Y}}.\mathtt{T}$ is to enrich the programming language with a new construct:

$$\mathtt{v} ::= \ldots \mid \mathtt{pack}_{\mathtt{D}}\,\mathtt{v} \qquad \mathtt{t} ::= \ldots \mid \mathtt{pack}_{\mathtt{D}}\,\mathtt{t} \qquad \mathcal{E} ::= \ldots \mid \mathtt{pack}_{\mathtt{D}}\,\mathcal{E}$$

and with a new unary destructor $\mathtt{open}_{\mathtt{D}}$. Their operational semantics is as follows:

$$\mathtt{open}_{\mathtt{D}}\,(\mathtt{pack}_{\mathtt{D}}\,\mathtt{v}) \xrightarrow{\delta} \mathtt{v} \qquad \text{(R-OPEN-ALL)}$$

Intuitively, $\mathtt{pack}_{\mathtt{D}}$ and $\mathtt{open}_{\mathtt{D}}$ are the two coercions that witness the isomorphism between $\mathtt{D}\,\vec{\mathtt{X}}$ and $\forall\bar{\mathtt{Y}}.\mathtt{T}$. The value $\mathtt{pack}_{\mathtt{D}}\,\mathtt{v}$ behaves exactly like $\mathtt{v}$, except it is marked, as a hint to the typechecker. As a result, the mark must be removed using $\mathtt{open}_{\mathtt{D}}$ before the value can be used.

What are the typing rules for $\mathtt{pack}_{\mathtt{D}}$ and $\mathtt{open}_{\mathtt{D}}$? In System F, they would receive types $\forall\bar{\mathtt{X}}.(\forall\bar{\mathtt{Y}}.\mathtt{T}) \to \mathtt{D}\,\vec{\mathtt{X}}$ and $\forall\bar{\mathtt{X}}.\mathtt{D}\,\vec{\mathtt{X}} \to \forall\bar{\mathtt{Y}}.\mathtt{T}$, respectively. However, neither of these is a valid type scheme: both exhibit a universal quantifier under an arrow.

In the case of $\mathtt{pack}_{\mathtt{D}}$, which has been made a language construct rather than a constant, we work around the problem by embedding this universal quantifier in the constraint generation rule:

$$[\![\mathtt{pack}_{\mathtt{D}}\,\mathtt{t} : \mathtt{T}']\!] \quad = \quad \exists\bar{\mathtt{X}}.([\![\mathtt{t} : \forall\bar{\mathtt{Y}}.\mathtt{T}]\!] \wedge \mathtt{D}\,\vec{\mathtt{X}} \leq \mathtt{T}')$$

The rule implicitly requires that $\bar{\mathtt{X}}$ be fresh for the left-hand side and that $\mathtt{D}\,\vec{\mathtt{X}} \approx \forall\bar{\mathtt{Y}}.\mathtt{T}$ be (an α-variant of) the definition of $\mathtt{D}$. The left-hand conjunct requires $\mathtt{t}$ to have type scheme $\forall\bar{\mathtt{Y}}.\mathtt{T}$. The notation $[\![\mathtt{t} : \mathtt{S}]\!]$ was defined on page 150. The right-hand conjunct states that a valid type for $\mathtt{pack}_{\mathtt{D}}\,\mathtt{t}$ is (a supertype of) $\mathtt{D}\,\vec{\mathtt{X}}$.

We deal with $\mathtt{open}_{\mathtt{D}}$ as follows. Provided $\bar{\mathtt{X}} \# \bar{\mathtt{Y}}$, we extend the initial environment Γ_0 with the binding $\mathtt{open}_{\mathtt{D}} : \forall\bar{\mathtt{X}}\bar{\mathtt{Y}}.\mathtt{D}\,\vec{\mathtt{X}} \to \mathtt{T}$. We have simply hoisted the universal quantifier outside of the arrow—a valid isomorphism in System F.

The proof of the subject reduction theorem must be extended with the following new case:

1.10.17 THEOREM [SUBJECT REDUCTION]: (R-OPEN-ALL) $\subseteq$ ($\sqsubseteq$). □

Proof: We have

$$
\begin{array}{lll}
 & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathtt{open}_D\ (\mathtt{pack}_D\ \mathtt{v}) : \mathtt{T}_0]\!] & \\
\equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \mathtt{Z}.(\mathtt{open}_D \preceq \mathtt{Z} \to \mathtt{T}_0 \wedge [\![\mathtt{pack}_D\ \mathtt{v} : \mathtt{Z}]\!]) & \textbf{(1)} \\
\equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \mathtt{Z}.(\exists \bar{\mathtt{X}}'\bar{\mathtt{Y}}'.(\mathtt{D}\,\vec{\mathtt{X}}' \to \mathtt{T}' \leq \mathtt{Z} \to \mathtt{T}_0) \wedge \exists \bar{\mathtt{X}}.([\![\mathtt{v} : \forall \bar{\mathtt{Y}}.\mathtt{T}]\!] \wedge \mathtt{D}\,\vec{\mathtt{X}} \leq \mathtt{Z})) & \textbf{(2)} \\
\equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \bar{\mathtt{X}}\bar{\mathtt{X}}'\bar{\mathtt{Y}}'.([\![\mathtt{v} : \forall \bar{\mathtt{Y}}.\mathtt{T}]\!] \wedge \mathtt{D}\,\vec{\mathtt{X}} \leq \mathtt{D}\,\vec{\mathtt{X}}' \wedge \mathtt{T}' \leq \mathtt{T}_0) & \textbf{(3)} \\
\Vdash & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \bar{\mathtt{X}}\bar{\mathtt{Y}}\bar{\mathtt{X}}'\bar{\mathtt{Y}}'.([\![\mathtt{v} : \forall \bar{\mathtt{Y}}.\mathtt{T}]\!] \wedge \mathtt{T} \leq \mathtt{T}' \wedge \mathtt{T}' \leq \mathtt{T}_0) & \textbf{(4)} \\
\Vdash & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \bar{\mathtt{X}}\bar{\mathtt{Y}}\bar{\mathtt{X}}'\bar{\mathtt{Y}}'.[\![\mathtt{v} : \mathtt{T}_0]\!] & \textbf{(5)} \\
\equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathtt{v} : \mathtt{T}_0]\!] & \textbf{(6)}
\end{array}
$$

where (1) is by definition of constraint generation for applications and for constants; $\mathtt{Z}$ is fresh; (2) is by definition of constraint generation for $\mathtt{pack}_D$ and $\mathtt{open}_D$, where $\mathtt{D}\,\vec{\mathtt{X}} \approx \forall \bar{\mathtt{Y}}.\mathtt{T}$ and $\mathtt{D}\,\vec{\mathtt{X}}' \approx \forall \bar{\mathtt{Y}}'.\mathtt{T}'$ are two α-equivalent instances of the definition of $\mathtt{D}$; $\bar{\mathtt{X}}$, $\bar{\mathtt{Y}}$, $\bar{\mathtt{X}}'$, and $\bar{\mathtt{Y}}'$ are fresh and satisfy $\bar{\mathtt{Y}} \# \mathit{ftv}(\mathtt{T}')$ and $\bar{\mathtt{Y}}' \# \mathit{ftv}(\mathtt{T})$; (3) is by C-EXAND, C-ARROW, and C-EXTRANS, which allows eliminating $\mathtt{Z}$; (4) is by Definition 1.10.16, Lemma 1.10.1, and C-EXAND; (5) is by Lemmas 1.10.14 and 1.6.4; (6) is by C-EX*. □

The proof of (R-CONTEXT) $\subseteq$ ($\sqsubseteq$) must also be extended with a new subcase, corresponding the new production $\mathcal{E} ::= \ldots \mid \mathtt{pack}_D\ \mathcal{E}$. If the language is pure, this is straightforward. In the presence of side effects, however, this subcase fails, because universal and existential quantifiers in constraints do not commute. The problem is then avoided by restricting $\mathtt{pack}_D$ to values, as in Definition 1.7.8.

This approach to extending ML-the-type-system with universal (or existential—see below) types has been studied in (Läufer and Odersky, 1994; Rémy, 1994; Odersky and Läufer, 1996; Shields and Peyton Jones, 2002). Laüfer and Odersky have suggested combining universal or existential type declarations with algebraic data type definitions. This allows suppressing the cumbersome $\mathtt{pack}_D$ and $\mathtt{open}_D$ constructs; instead, one simply uses the standard syntax for constructing and deconstructing variants and records.

1.10.6 Existential types

Existential types (TAPL Chapter 24) are close cousins of universal types, and may be introduced into ML-the-type-system in the same manner. Actually, existential types have been introduced in ML-the-type-system before universal types. We give a brief description of this extension, insisting mainly on the differences with the case of universal types.

We now allow *existential type definitions*, of the form $\mathtt{D}\,\vec{\mathtt{X}} \approx \exists\bar{\mathtt{Y}}.\mathtt{T}$. The conditions required of a well-formed definition are unchanged, except the variance requirement, which is dual:

1.10.18 DEFINITION: Let $\mathtt{D}\,\vec{\mathtt{X}} \approx \exists\bar{\mathtt{Y}}.\mathtt{T}$ and $\mathtt{D}\,\vec{\mathtt{X}}' \approx \exists\bar{\mathtt{Y}}'.\mathtt{T}'$ be two α-equivalent instances of a single existential type definition, such that $\bar{\mathtt{Y}} \# \mathit{ftv}(\mathtt{T}')$ and $\bar{\mathtt{Y}}' \# \mathit{ftv}(\mathtt{T})$. Then, $\mathtt{D}\,\vec{\mathtt{X}} \leq \mathtt{D}\,\vec{\mathtt{X}}' \Vdash \forall\bar{\mathtt{Y}}.\exists\bar{\mathtt{Y}}'.\mathtt{T} \leq \mathtt{T}'$ must hold. □

The effect of this existential type definition is to enrich the programming language with a new unary constructor $\mathtt{pack}_\mathtt{D}$ and with a new construct: $\mathtt{t} ::= \ldots \mid \mathtt{open}_\mathtt{D}\ \mathtt{t}\ \mathtt{t}$ and $\mathcal{E} ::= \ldots \mid \mathtt{open}_\mathtt{D}\ \mathcal{E}\ \mathtt{t} \mid \mathtt{open}_\mathtt{D}\ \mathtt{v}\ \mathcal{E}$. Their operational semantics is as follows:

$$\mathtt{open}_\mathtt{D}\ (\mathtt{pack}_\mathtt{D}\mathtt{v}_1)\ \mathtt{v}_2 \longrightarrow \mathtt{v}_2\ \mathtt{v}_1 \qquad \text{(R-OPEN-EX)}$$

In the literature, the second argument of $\mathtt{open}_\mathtt{D}$ is often required to be a λ-abstraction $\lambda\mathtt{z}.\mathtt{t}$, so the construct becomes $\mathtt{open}_\mathtt{D}\ \mathtt{t}\ (\lambda\mathtt{z}.\mathtt{t})$, often written $\mathtt{open}_\mathtt{D}\ \mathtt{t}\ \mathtt{as}\ \mathtt{z}\ \mathtt{in}\ \mathtt{t}$.

Provided $\bar{\mathtt{X}} \# \bar{\mathtt{Y}}$, we extend the initial environment Γ_0 with the binding $\mathtt{pack}_\mathtt{D} : \forall\bar{\mathtt{X}}\bar{\mathtt{Y}}.\mathtt{T} \to \mathtt{D}\,\vec{\mathtt{X}}$. The constraint generation rule for $\mathtt{open}_\mathtt{D}$ is as follows:

$$[\![\mathtt{open}_\mathtt{D}\ \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}']\!] \quad = \quad \exists\bar{\mathtt{X}}.([\![\mathtt{t}_1 : \mathtt{D}\,\vec{\mathtt{X}}]\!] \wedge [\![\mathtt{t}_2 : \forall\bar{\mathtt{Y}}.\mathtt{T} \to \mathtt{T}']\!])$$

The rule implicitly requires that $\bar{\mathtt{X}}$ be fresh for the left-hand side, that $\bar{\mathtt{Y}}$ be fresh for $\mathtt{T}'$, and that $\mathtt{D}\,\vec{\mathtt{X}} \approx \forall\bar{\mathtt{Y}}.\mathtt{T}$ be (an α-variant of) the definition of $\mathtt{D}$. The left-hand conjunct simply requires $\mathtt{t}_1$ to have type $\mathtt{D}\,\vec{\mathtt{X}}$. The right-hand conjunct states that the function $\mathtt{t}_2$ must be prepared to accept an argument of type $\mathtt{T}$, for *any* $\bar{\mathtt{Y}}$, and produce a result of the expected type $\mathtt{T}'$. In other words, $\mathtt{t}_2$ must be a polymorphic function.

The type scheme of existential $\mathtt{pack}_\mathtt{D}$ resembles that of universal $\mathtt{open}_\mathtt{D}$, while the constraint generation rule for existential $\mathtt{open}_\mathtt{D}$ is a close cousin of that for universal $\mathtt{pack}_\mathtt{D}$. Thus, the duality between universal and existential types is rather strong. The main difference lies in the fact that the existential $\mathtt{open}_\mathtt{D}$ construct is *binary*, rather than unary, so as to limit the scope of the newly introduced type variables $\bar{\mathtt{Y}}$. The duality may be better understood by studying the encoding of existential types in terms of universal types (Reynolds, 1983b) TAPL §24.3.

As expected, R-OPEN-EX preserves types.

1.10.19 THEOREM [SUBJECT REDUCTION]: (R-OPEN-EX) $\subseteq$ ($\sqsubseteq$). □

1.10.20 EXERCISE [★★, $\rightarrowtail$]: Prove Theorem 1.10.19. The proof is analogous, although not identical, to that of Theorem 1.10.17. □

In the presence of side effects, the new production $\mathcal{E} ::= \ldots \mid \mathtt{open}_\mathtt{D}\ \mathtt{v}\ \mathcal{E}$ is problematic. The standard workaround is to restrict the second argument to $\mathtt{open}_\mathtt{D}$ to be a value.

1.11 Rows

In §1.9, we have shown how to extend ML-the-programming-language with algebraic data types, that is, variant and record type definitions, which we now refer to as *simple*. This mechanism has a severe limitation: two distinct definitions must define incompatible types. As a result, one cannot hope to write code that uniformly operates over variants or records of different shapes, because the type of such code is not even expressible.

For instance, it is impossible to express the type of the *polymorphic record access* operation, which retrieves the value stored at a particular field ℓ inside a record, *regardless* of which other fields are present. Indeed, if the label ℓ appears with type $\mathtt{T}$ in the definition of the simple record type $\mathtt{D}\,\vec{\mathtt{X}}$, then the associated record access operation has type $\forall\vec{\mathtt{X}}.\mathtt{D}\,\vec{\mathtt{X}} \to \mathtt{T}$. If ℓ appears with type $\mathtt{T}'$ in the definition of another simple record type, say $\mathtt{D}'\,\vec{\mathtt{X}}'$, then the associated record access operation has type $\forall\vec{\mathtt{X}}'.\mathtt{D}'\,\vec{\mathtt{X}}' \to \mathtt{T}'$; and so on. The most precise type scheme that subsumes all of these incomparable type schemes is $\forall\mathtt{XY}.\mathtt{X} \to \mathtt{Y}$. It is, however, not a sound type scheme for the record access operation. Another powerful operation whose type is currently not expressible is *polymorphic record extension*, which copies a record and stores a value at field ℓ in the copy, possibly creating the field if it did not previously exist, again *regardless* of which other fields are present. (If ℓ was known to previously exist, the operation is known as *polymorphic record update*.)

In order to assign types to polymorphic record operations, we must do away with record type definitions: we must replace *named* record types, such as $\mathtt{D}\,\vec{\mathtt{X}}$, with *structural* record types that provide a direct description of the record's domain and contents. (Following the analogy between a record and a partial function from labels to values, we use the word *domain* to refer to the set of fields that are defined in a record.) For instance, a product type is structural: the type $\mathtt{T}_1 \times \mathtt{T}_2$ is the (undeclared) type of pairs whose first component has type $\mathtt{T}_1$ and whose second component has type $\mathtt{T}_2$. Thus, we wish to design record types that behave very much like product types. In doing so, we face two orthogonal difficulties. First, as opposed to pairs, records may have different domains. Because the type system must statically ensure that no undefined field is accessed, information about a record's domain must be made part of its type. Second, because we suppress record type definitions, labels must now be predefined. However, for efficiency and modularity reasons, it is impossible to explicitly list *every* label in existence in every record type.

In what follows, we explain how to address the first difficulty in the simple setting of a finite set of labels. Then, we introduce *rows*, which allow dealing with an infinite set of labels, and address the second difficulty. We define the

syntax and logical interpretation of rows, study the new constraint equivalence laws that arise in their presence, and extend the first-order unification algorithm with support for rows. Then, we review several applications of rows, including polymorphic operations on records, variants, and objects, and discuss alternatives to rows.

1.11.1 Records with finite carrier

Let us temporarily assume that $\mathcal{L}$ is finite. In fact, for the sake of definiteness, let us assume that $\mathcal{L}$ is the three-element set $\{\ell_a, \ell_b, \ell_c\}$.

To begin, let us consider only *full* records, whose domain is exactly $\mathcal{L}$—in other words, tuples indexed by $\mathcal{L}$. To describe them, it is natural to introduce a type constructor $''$ of signature $\star\otimes\star\otimes\star \Rightarrow \star$. The type $''\ \mathtt{T}_a\ \mathtt{T}_b\ \mathtt{T}_c$ represents all records where the field ℓ_a (resp. ℓ_b, ℓ_c) contains a value of type $\mathtt{T}_a$ (resp. $\mathtt{T}_b$, $\mathtt{T}_c$). Please note that $''$ is nothing but a product type constructor of arity 3. The basic operations on records, namely *creation* of a record out of a default value, which is stored into every field, *update* of a particular field (say, ℓ_b), and *access* to a particular field (say, ℓ_b), may be assigned the following type schemes:

$$\begin{array}{rl}
\{\cdot\}: & \forall \mathtt{X}.\mathtt{X} \to {}''\ \mathtt{X}\ \mathtt{X}\ \mathtt{X} \\
\{\cdot \text{ with } \ell_b = \cdot\}: & \forall \mathtt{X}_a\mathtt{X}_b\mathtt{X}'_b\mathtt{X}_c.''\ \mathtt{X}_a\ \mathtt{X}_b\ \mathtt{X}_c \to \mathtt{X}'_b \to {}''\ \mathtt{X}_a\ \mathtt{X}'_b\ \mathtt{X}_c \\
\cdot.\{\ell_b\}: & \forall \mathtt{X}_a\mathtt{X}_b\mathtt{X}_c.''\ \mathtt{X}_a\ \mathtt{X}_b\ \mathtt{X}_c \to \mathtt{X}_b
\end{array}$$

Here, polymorphism allows updating or accessing a field without knowledge of the types of the other fields. This flexibility stems from the key property that all record types are formed using a single $''$ type constructor.

This is fine, but in general, the domain of a record is not necessarily $\mathcal{L}$: it may be a subset of $\mathcal{L}$. How may we deal with this fact, while maintaining the above key property? A naïve approach consists in encoding arbitrary records in terms of full records, using the standard algebraic data type option, whose definition is $\mathsf{option}\ \mathtt{X} \approx \mathsf{pre}\ \mathtt{X} + \mathsf{abs}$. We use pre for *present* and abs for *absent*: indeed, a field that is defined with value $\mathtt{v}$ is encoded as a field with value $\mathsf{pre}\ \mathtt{v}$, while an undefined field is encoded as a field with value abs. Thus, an arbitrary record whose fields, *if present*, have types $\mathtt{T}_a$, $\mathtt{T}_b$, and $\mathtt{T}_c$, respectively, may be encoded as a full record of type $''\ (\mathsf{option}\ \mathtt{T}_a)\ (\mathsf{option}\ \mathtt{T}_b)\ (\mathsf{option}\ \mathtt{T}_c)$. This naïve approach suffers from a serious drawback: record types still contain no domain information. As a result, field access must involve a dynamic check, so as to determine whether the desired field is present: in our encoding, this corresponds to the use of $\mathtt{case}_{\mathsf{option}}$.

To avoid this overhead and increase programming safety, we must move this check from runtime to compile time. In other words, we must make the

type system aware of the difference between pre and abs. To do so, we replace the definition of option by two separate algebraic data type definitions, namely $\mathsf{pre}\,\mathtt{X} \approx \mathsf{pre}\,\mathtt{X}$ and $\mathsf{abs} \approx \mathsf{abs}$. In other words, we introduce a unary type constructor pre, whose only associated data constructor is pre, and a nullary type constructor abs, whose only associated data constructor is abs. Record types now contain domain information: for instance, a record of type $''\,\mathsf{abs}\,(\mathsf{pre}\,\mathtt{T_b})\,(\mathsf{pre}\,\mathtt{T_c})$ must have domain $\{\ell_b, \ell_c\}$. Thus, the type of a field tells whether it is defined. Since the type pre has no data constructors other than pre, the accessor pre^{-1}, whose type is $\forall\mathtt{X}.\mathsf{pre}\,\mathtt{X} \to \mathtt{X}$, and which allows retrieving the value stored in a field, cannot fail. Thus, the dynamic check has been eliminated.

To complete the definition of our encoding, we now define operations on arbitrary records in terms of operations on full records. To distinguish between the two, we write the former with angle braces, instead of curly braces. The *empty record* $\langle\rangle$, where all fields are undefined, may be defined as $\{\mathsf{abs}\}$. *Extension* at a particular field (say, ℓ_b) $\langle\cdot\ \mathsf{with}\ \ell_b = \cdot\rangle$ is defined as $\lambda\mathtt{r}.\lambda\mathtt{z}.\{\mathtt{r}\ \mathsf{with}\ \ell_b = \mathsf{pre}\,\mathtt{z}\}$. *Access* at a particular field (say, ℓ_b) $\cdot.\langle\ell_b\rangle$ is defined as $\lambda\mathtt{z}.\mathsf{pre}^{-1}\mathtt{z}.\{\ell_b\}$. It is straightforward to check that these operations have the following principal type schemes:

$$\begin{array}{rl}
\langle\rangle: & ''\,\mathsf{abs}\,\mathsf{abs}\,\mathsf{abs} \\
\langle\cdot\ \mathsf{with}\ \ell_b = \cdot\rangle: & \forall\mathtt{X_a}\mathtt{X_b}\mathtt{X'_b}\mathtt{X_c}.''\,\mathtt{X_a}\,\mathtt{X_b}\,\mathtt{X_c} \to \mathtt{X'_b} \to ''\,\mathtt{X_a}\,(\mathsf{pre}\ \mathtt{X'_b})\,\mathtt{X_c} \\
\cdot.\langle\ell_b\rangle: & \forall\mathtt{X_a}\mathtt{X_b}\mathtt{X_c}.''\,\mathtt{X_a}\,(\mathsf{pre}\ \mathtt{X_b})\,\mathtt{X_c} \to \mathtt{X_b}
\end{array}$$

It is important to notice that the type schemes associated with extension and access at ℓ_b are polymorphic in $\mathtt{X_a}$ and $\mathtt{X_c}$, which now means that *these operations are insensitive not only to the type, but also to the presence or absence of the fields* ℓ_a *and* ℓ_c. Furthermore, extension is polymorphic in $\mathtt{X_b}$, which means that it is insensitive to the presence or absence of the field ℓ_b in its argument. The subterm $\mathsf{pre}\,\mathtt{X'_b}$ in its result type reflects the fact that ℓ_b is defined in the extended record. Conversely, the subterm $\mathsf{pre}\ \mathtt{X_b}$ in the type of the access operation reflects the requirement that ℓ_b be defined in its argument.

Our encoding of arbitrary records in terms of full records was carried out for pedagogical purposes. In practice, no such encoding is necessary: the *data* constructors pre and abs have no machine representation, and the compiler is free to lay out records in memory in an efficient manner. The encoding is interesting, however, because it provides a natural way of introducing the *type* constructors pre and abs, which play an important role in our treatment of polymorphic record operations.

We remark that, once we forget about the encoding, the arguments of the type constructor $''$ are expected to be either type variables or formed with pre or abs, while, conversely, the type constructors pre and abs are not intended

to appear anywhere else. It is possible to enforce this invariant using kinds. In addition to $\star$, let us introduce the kind $\circ$ of *field types*. Then, let us adopt the following signatures: pre: $\star \Rightarrow \circ$, abs : $\circ$, and $\tilde{}$: $\circ \otimes \circ \otimes \circ \Rightarrow \star$.

1.11.1 EXERCISE [RECOMMENDED, ★, $\twoheadrightarrow$]: Check that the three type schemes given above are well-kinded. What is the kind of each type variable? □

1.11.2 EXERCISE [RECOMMENDED, ★★, $\twoheadrightarrow$]: Our $\tilde{}$ types contain information about every field, regardless of whether it is defined: we encode definedness information within the type of each field, using the type constructors pre and abs. A perhaps more natural approach would be to introduce a family of record type constructors, indexed by the subsets of $\mathcal{L}$, so that the types of records with different domains are formed with different constructors. For instance, the empty record would have type $\{\}$; a record that defines the field ℓ_a only would have a type of the form $\{\ell_a : \mathtt{T}_a\}$; a record that defines the fields ℓ_b and ℓ_c only would have a type of the form $\{\ell_b : \mathtt{T}_b; \ell_c : \mathtt{T}_c\}$; and so on. Assuming that the type discipline is Damas and Milner's (that is, assuming an equality-only syntactic model), would it be possible to assign satisfactory type schemes to polymorphic record access and extension? Would it help to equip record types with a nontrivial subtyping relation? □

1.11.2 Records with infinite carrier

The treatment of records described in §1.11.1 is not quite satisfactory, from practical and theoretical points of view. First, in practice, the set $\mathcal{L}$ of all record labels that appear within a program could be very large. Because *every* record type is just as large as $\mathcal{L}$ itself, even if the record that it describes only has a few fields, this is unpleasant. Furthermore, in a modular setting, the set of all record labels that appear within a program cannot be determined until *link* time, so it is still unknown at *compile* time, when each compilation unit is separately typechecked. As a result, it may only be assumed to be a subset of the infinite set of all syntactically valid record labels. Resolving these issues requires coming up with a treatment of records that does not become more costly as $\mathcal{L}$ grows and that, in fact, allows $\mathcal{L}$ to be infinite. Thus, from here on, let us assume that $\mathcal{L}$ is infinite.

As in the previous section, we first concentrate on *full* records, whose domain is exactly $\mathcal{L}$. The case of arbitrary records, whose domain is a subset of $\mathcal{L}$, will then follow, in the same manner, by using the type constructors pre and abs to encode domain information.

Of course, even though we have assumed that $\mathcal{L}$ is infinite, we must ensure that every record has a finite representation. We choose to restrict our attention to records that are *almost constant*, that is, records where all fields but

a finite number contain the same value. Every such record may be defined in terms of two primitive operations, namely (i) *creation* of a constant record out of a value; for instance, $\{\texttt{false}\}$ is the record where every field contains the value `false`; and (ii) *update* of a record at a particular field; for instance, $\{\{\texttt{false}\}\ \mathsf{with}\ \ell = 1\}$ carries the value 1 at field ℓ and the value `false` at every other field. As usual, an *access* operation allows retrieving the contents of a field. Thus, the three primitive operations are the same as in §1.11.1, only in the setting of an infinite number of fields.

If we were to continue as in §1.11.1, we would now introduce a type constructor $\tilde{}$, *equipped with an infinite family of type parameters*. Because types must remain finite objects, we cannot do so. Instead, we must find a finite (and economical) representation of such an infinite family of types. This is precisely the role played by *rows*.

A row is a type that denotes a function from labels to types, or, equivalently, as a family of types, indexed by labels. Its domain is $\mathcal{L}$—the row is then *complete*—or a cofinite subset of $\mathcal{L}$—the row is then *incomplete*. (A subset of $\mathcal{L}$ is cofinite if and only if its complement is finite. Incomplete rows are used only as building blocks for complete rows.) Because rows must admit a finite representation, we build them out of two syntactic constructions, namely (i) construction of a *constant row* out of a type; for instance, the notation $\partial\mathsf{bool}$ denotes a row that maps every label in its domain to bool; and (ii) strict *extension* of an incomplete row; for instance, $(\ell : \mathsf{int}\ ;\ \partial\mathsf{bool})$ denotes a row that maps ℓ to int and every other field in its domain to bool. Formally, ∂ is a unary type constructor, while, for every label ℓ, $(\ell : \cdot\ ;\ \cdot)$ is a binary type constructor. These two constructions are reminiscent of the two operations used above to build records. There are, however, a couple of subtle but important differences. First, $\partial\,\mathtt{T}$ may be a *complete or incomplete* row. Second, $(\ell : \mathtt{T}\ ;\ \mathtt{T}')$ is defined only if ℓ is not in the domain of the row $\mathtt{T}'$, so this construction is strict extension, not update. These aspects are made clear by a *kinding discipline*, to be introduced later on (§1.11.3).

It is possible for two syntactically distinct rows to denote the same function from labels to types. For instance, according to the intuitive interpretation of rows given above, the three complete rows $(\ell : \mathsf{int}\ ;\ \partial\mathsf{bool})$, $(\ell : \mathsf{int}\ ;\ (\ell' : \mathsf{bool}\ ;\ \partial\mathsf{bool}))$, and $(\ell' : \mathsf{bool}\ ;\ (\ell : \mathsf{int}\ ;\ \partial\mathsf{bool}))$ denote the same total function from labels to types. In the following, we define the logical interpretation of types in such a way that the interpretations of these three rows in the model are indeed equal.

We may now make the record type constructor $\tilde{}$ a *unary* type constructor, *whose parameter is a row*. Then, (say) $\tilde{}\ (\ell : \mathsf{int}\ ;\ \partial\mathsf{bool})$ is a record type, and we intend it to be a valid type for the record $\{\{\texttt{false}\}\ \mathsf{with}\ \ell = 1\}$. The basic

operations on records may be assigned the following type schemes:

$$\begin{array}{rl} \{\cdot\}: & \forall \mathtt{X}.\mathtt{X} \to \text{˝}\,(\partial \mathtt{X}) \\ \{\cdot \text{ with } \ell = \cdot\}: & \forall \mathtt{X}\mathtt{X}'\mathtt{Y}.\text{˝}\,(\ell : \mathtt{X}\,;\,\mathtt{Y}) \to \mathtt{X}' \to \text{˝}\,(\ell : \mathtt{X}'\,;\,\mathtt{Y}) \\ \cdot.\{\ell\}: & \forall \mathtt{X}\mathtt{Y}.\text{˝}\,(\ell : \mathtt{X}\,;\,\mathtt{Y}) \to \mathtt{X} \end{array}$$

These type schemes are reminiscent of those given in §1.11.1. However, in the previous section, the size of the type schemes was linear in the cardinal of $\mathcal{L}$, whereas, here, it is constant, even though $\mathcal{L}$ is infinite. This is made possible by the fact that record types no longer list all labels in existence; instead, they use rows. In the type scheme assigned to record creation, the constant row $\partial\mathtt{X}$ is used to indicate that all fields have the same type in the newly created record. In the next two type schemes, the row $(\ell : \mathtt{X}_\ell\,;\,\mathtt{X})$ is used to separate the type $\mathtt{X}_\ell$, which describes the contents of the field ℓ, and the row $\mathtt{X}$, which collectively describes the contents of all other fields. Here, the type variable $\mathtt{X}$ stands for an arbitrary row; it is often referred to as a *row variable*. The ability of quantifying over row and type variables alike confers great expressiveness to the type system.

We have explained, in an informal manner, how rows allow typechecking operations on *full* records, in the setting of an infinite set of labels. We return to this issue in Example 1.11.26. To deal with the case of arbitrary records, whose domain is finite, we rely on the field type constructors pre and abs, as explained previously. We return to this point in Example 1.11.31. In the following, we give a formal exposition of rows. We begin with their syntax and logical interpretation. Then, we give some new constraint equivalence laws, which characterize rows, and allow extending our first-order unification algorithm with support for rows. We conclude with several illustrations of the use of rows and some pointers to related work.

1.11.3 Syntax

In the following, the set of labels $\mathcal{L}$ is considered denumerable. We let L range over finite subsets of $\mathcal{L}$. When $\ell \notin \mathrm{L}$ holds, we write $\ell.\mathrm{L}$ for $\{\ell\} \uplus \mathrm{L}$. Before explaining how the syntax of types is enriched with rows, we introduce *row kinds*, whose grammar is as follows:

$$\mathrm{s} ::= \mathit{Type} \mid \mathit{Row}(\mathrm{L})$$

Row kinds help distinguish between three kinds of types, namely ordinary types, complete rows, and incomplete rows. While ordinary types are used to describe expressions, complete or incomplete rows are used only as building blocks for ordinary types. For instance, the record type $\text{˝}\,(\ell{:}\mathsf{int}\,;\,\partial\mathsf{bool})$, which

was informally introduced above, is intended to be an ordinary type, that is, a type of row kind *Type*. Its subterm $(\ell{:}\mathsf{int}\ ;\ \partial\mathsf{bool})$, is a complete row, that is, a type of row kind $\mathit{Row}(\varnothing)$. Its subterm $\partial\mathsf{bool}$ is an incomplete row, whose row kind is $\mathit{Row}(\{\ell\})$. Intuitively, a row of kind $\mathit{Row}(\mathrm{L})$ denotes a family of types whose domain is $\mathcal{L} \setminus \mathrm{L}$. In other words, L is the set of labels that the row does not define. The purpose of row kinds is to outlaw meaningless types, such as $''\ (\mathsf{int})$, which makes no sense because the argument to the record type constructor $''$ should be a (complete) row, or $(\ell : \mathtt{T}_1\ ;\ \ell : \mathtt{T}_2\ ; \partial\mathsf{bool})$, which makes no sense because no label may occur twice within a row.

Let us now define the syntax of types in the presence of rows. As usual, it is given by a signature $\mathcal{S}$ (Definition 1.2.17), which lists all type constructors together with their signatures. Here, for the sake of generality, we do not wish to give a *fixed* signature $\mathcal{S}$. Instead, we give a *procedure* that builds $\mathcal{S}$ out of two simpler signatures, referred to as $\mathcal{S}_0$ and $\mathcal{S}_1$. The input signature $\mathcal{S}_0$ lists the type constructors that have nothing to do with rows, such as $\rightarrow$, $\times$, int, etc. The input signature $\mathcal{S}_1$ lists the type constructors that allow a row to be a subterm of an ordinary type, such as the record type constructor $''$. In a type system equipped with extensible variant types or with object types, there might be several such type constructors; see §1.11.9 and §1.11.10. Without loss of generality, we assume that all type constructors in $\mathcal{S}_1$ are unary. The point of parameterizing the definition of $\mathcal{S}$ over $\mathcal{S}_0$ and $\mathcal{S}_1$ is to make the construction more general: instead of defining a *fixed* type grammar featuring rows, we wish to explain how to enrich an *arbitrary* type grammar with rows.

In the following, we let G (resp. H) range over the type constructors in $\mathcal{S}_0$ (resp. $\mathcal{S}_1$). We let κ range over the kinds involved in the definition of $\mathcal{S}_0$ and $\mathcal{S}_1$, and refer to them as *basic kinds*. We let F range over the type constructors in $\mathcal{S}$. The kinds involved in the definition of $\mathcal{S}$ are *composite kinds*, that is, pairs of a basic kind κ and a row kind s, written $\kappa.s$. This allows the kind discipline enforced by $\mathcal{S}$ to reflect that enforced by $\mathcal{S}_0$ and $\mathcal{S}_1$ and to also impose restrictions on the structure and use of rows, as suggested above. For the sake of conciseness, we write K.s for the mapping $(\mathrm{d} \mapsto \mathrm{K}(\mathrm{d}).s)^{\mathrm{d} \in \mathit{dom}(\mathrm{K})}$ and $(\mathrm{K} \Rightarrow \kappa).s$ for the (composite) kind signature $\mathrm{K}.s \Rightarrow \kappa.s$. We use symmetric notations to build a composite kind signature out of a basic kind and a row kind signature.

1.11.3 DEFINITION: The signature $\mathcal{S}$ is defined as follows:

$F \in dom(\mathcal{S})$	Signature	Conditions
G^s	$(K \Rightarrow \kappa).s$	$(G : K \Rightarrow \kappa) \in \mathcal{S}_0$
H	$K.Row(\varnothing) \Rightarrow \kappa.Type$	$(H : K \Rightarrow \kappa) \in \mathcal{S}_1$
$\partial^{\kappa,L}$	$\kappa.(Type \Rightarrow Row(L))$	
$\ell^{\kappa,L}$	$\kappa.(Type \otimes Row(\ell.L) \Rightarrow Row(L))$	$\ell \notin L$

We sometimes refer to $\mathcal{S}$ as the *row extension of $\mathcal{S}_0$ with $\mathcal{S}_1$*. □

Examples 1.11.7 and 1.11.8 suggest common choices of $\mathcal{S}_0$ and $\mathcal{S}_1$, and give a perhaps more concrete-looking definition of the grammar of types that they determine. First, however, let us explain the definition. The type constructors that populate $\mathcal{S}$ come in four varieties: they may be (i) taken from $\mathcal{S}_0$, (ii) taken from $\mathcal{S}_1$, (iii) a unary row constructor ∂, or (iv) a binary row constructor $(\ell : \cdot \,; \cdot)$. Let us review and explain each case.

Let us first consider case (i) and assume, for the time being, that s is *Type*. Then, for every type constructor G in $\mathcal{S}_0$, there is a corresponding type constructor G^{Type} in $\mathcal{S}$. For instance, $\mathcal{S}_0$ must contain an arrow type constructor $\rightarrow$, whose signature is $\{domain \mapsto \star, codomain \mapsto \star\} \Rightarrow \star$. Then, $\mathcal{S}$ contains a type constructor $\rightarrow^{Type}$, whose signature is $\{domain \mapsto \star.Type, codomain \mapsto \star.Type\} \Rightarrow \star.Type$. Thus, $\rightarrow^{Type}$ is a binary type constructor whose parameters and result must have basic kind $\star$ and must have row kind *Type*; in other words, they must be ordinary types, as opposed to complete or incomplete rows. The family of all type constructors of the form G^{Type}, where G ranges over $\mathcal{S}_0$, forms a copy of $\mathcal{S}_0$ at row kind *Type*: one might say, roughly speaking, that $\mathcal{S}$ *contains* $\mathcal{S}_0$. This is not surprising, since our purpose is to *enrich* the existing signature $\mathcal{S}_0$ with syntax for rows.

Perhaps more surprising is the existence of the type constructor G^s, for every G in $\mathcal{S}_0$, *and for every row kind* s. For instance, for every L, $\mathcal{S}$ contains a type constructor $\rightarrow^{Row(L)}$, whose signature is $\{domain \mapsto \star.Row(L), codomain \mapsto \star.Row(L)\} \Rightarrow \star.Row(L)$. Thus, $\rightarrow^{Row(L)}$ is a binary type constructor whose parameters and result must have basic kind $\star$ and must have row kind $Row(L)$. In other words, this type constructor maps a pair of rows that have a common domain to a row with the same domain. Recall that a row is to be interpreted as a family of types. Our intention is that $\rightarrow^{Row(L)}$ maps two families of types to a family of arrow types. This is made precise in §1.11.4. One should point out that the type constructors G^s, with $s \neq Type$, are required only in some advanced applications of rows; Examples 1.11.29 and 1.11.47 provide illustrations. They are not used when assigning types to the usual primitive operations on records, namely creation, update, and access (Examples 1.11.26 and 1.11.31).

Case (ii) is simple: it simply means that $\mathcal{S}$ *contains* $\mathcal{S}_1$. It is only worth noting that every type constructor H maps a parameter of row kind $\mathit{Row}(\varnothing)$ to a result of row kind *Type*, that is, a complete row to an ordinary type. Thanks to this design choice, the type $\tilde{}\ (\mathsf{int}^{\mathit{Type}})$ is invalid: indeed, $\mathsf{int}^{\mathit{Type}}$ has row kind *Type*, while $\tilde{}$ expects a parameter of row kind $\mathit{Row}(\varnothing)$.

Cases (iii) and (iv) introduce new type constructors, which were not present in $\mathcal{S}_0$ or $\mathcal{S}_1$, and allow forming rows. They were informally described in §1.11.2. First, for every κ and L, there is a *constant row constructor* $\partial^{\kappa,\mathrm{L}}$. Its parameter must have row kind *Type*, while its result has row kind $\mathit{Row}(\mathrm{L})$: in other words, this type constructor maps an ordinary type to a row. It is worth noting that the row thus built may be complete or incomplete: for instance, $\partial^{\star,\varnothing}\,\mathsf{bool}$ is a complete row, and may be used *e.g.* to build the type $\tilde{}\ (\partial^{\star,\varnothing}\,\mathsf{bool})$, while $\partial^{\star,\{\ell\}}\,\mathsf{bool}$ is an incomplete row, and may be used *e.g.* to build the type $\tilde{}\ (\ell\!:\!\mathsf{int}\ ;\ \partial^{\star,\{\ell\}}\,\mathsf{bool})$. Second, for every κ, L, and $\ell \notin \mathrm{L}$, there is a *row extension constructor* $\ell^{\kappa,\mathrm{L}}$. We usually write $\ell^{\kappa,\mathrm{L}}\!:\mathtt{T}_1\ ;\ \mathtt{T}_2$ for $\ell^{\kappa,\mathrm{L}}\ \mathtt{T}_1\ \mathtt{T}_2$ and let this symbol be right associative, so as to recover the familiar list notation for rows. According to the definition of $\mathcal{S}$, if $\mathtt{T}_2$ has row kind $\mathit{Row}(\ell.\mathrm{L})$, then $\ell^{\kappa,\mathrm{L}} : \mathtt{T}_1\ ;\ \mathtt{T}_2$ has row kind $\mathit{Row}(\mathrm{L})$. Thanks to this design choice, the type $(\ell^{\star,\mathrm{L}} : \mathtt{T}_1\ ;\ \ell^{\star,\mathrm{L}} : \mathtt{T}_2\ ;\ \partial^{\star,\ell.\mathrm{L}}\,\mathsf{bool})$ is invalid: indeed, the outer ℓ expects a parameter of row kind $\mathit{Row}(\ell.\mathrm{L})$, while the inner ℓ produces a type of row kind $\mathit{Row}(\mathrm{L})$.

The superscripts carried by the type constructors G, ℓ, and ∂ in the signature $\mathcal{S}$ make all kind information explicit, obviating the need for assigning several kinds to a single type constructor. In practice, however, we often drop the superscripts and use *unannotated* types. No ambiguity arises because, given a type expression $\mathtt{T}$ of known kind, it is possible to reconstruct all superscripts in a unique manner. This is the topic of the next example and exercises.

1.11.4 EXAMPLE [ILL-KINDED TYPES]: Assume that $\mathcal{S}_0$ contains type constructors int and $\rightarrow$, whose signatures are respectively $\star$ and $\star \otimes \star \Rightarrow \star$, and that $\mathcal{S}_1$ contains a type constructor $\tilde{}$, whose signature is $\star \Rightarrow \star$.

The unannotated type $\mathtt{X} \rightarrow \tilde{}\,(\mathtt{X})$ is invalid. Indeed, because $\tilde{}$'s image row kind is *Type*, the arrow must be $\rightarrow^{\mathit{Type}}$. Thus, the leftmost occurrence of $\mathtt{X}$ must have row kind *Type*. On the other hand, because $\tilde{}$ expects a parameter of row kind $\mathit{Row}(\varnothing)$, its rightmost occurrence must have row kind $\mathit{Row}(\varnothing)$—a contradiction. The unannotated type $\mathtt{X} \rightarrow \tilde{}\,(\partial\mathtt{X})$ is, however, valid, provided $\mathtt{X}$ has kind $\star.\mathit{Type}$. In fact, it is the type of the primitive record creation operation (§1.11.2).

The unannotated type $(\ell : \mathtt{T}\ ;\ \ell : \mathtt{T}\ ;\ \mathtt{T}'')$ is also invalid: there is no way of reconstructing the missing superscripts so as to make it valid. Indeed, the

row $(\ell : \mathtt{T}' ; \mathtt{T}'')$ must have row kind $\mathit{Row}(\mathrm{L})$ for some L that does not contain ℓ. However, the context where it occurs requires it to also have row kind $\mathit{Row}(\mathrm{L})$ for some L that does contain ℓ. This makes it impossible to reconstruct consistent superscripts.

Any type of the form $\tilde{}(\tilde{}(\mathtt{T}))$ is invalid, because the outer $\tilde{}$ expects a parameter of row kind $\mathit{Row}(\varnothing)$, while the inner $\tilde{}$ constructs a type of row kind *Type*. This is an intentional limitation: unlike those of $\mathcal{S}_0$, the type constructors of $\mathcal{S}_1$ are not lifted to every row kind s. □

1.11.5 EXERCISE [RECOMMENDED, ★]: Consider the unannotated type

$$\mathtt{X} \to \tilde{}(\ell : \mathsf{int} ; (\mathtt{Y} \to \partial \mathtt{X})).$$

Can you guess the kind of the type variables X and Y, as well as the missing superscripts, so as to ensure that this type has kind $\star.\mathit{Type}$? □

1.11.6 EXERCISE [★★★, ↛]: Propose a *kind checking* algorithm that, given an unannotated type T, given the kind of T, and given the kind of all type variables that appear within T, ensures that T is well-kinded, and reconstructs the missing superscripts within T. Next, propose a *kind inference* algorithm that, given an unannotated type T, *discovers* the kind of T and the kind of all type variables that appear within T so as to ensure that T is well-kinded. □

We have given a very general definition of the syntax of types. In this view, types, ranged over by the meta-variable T, encompass both "ordinary" types and rows: the distinction between the two is established only via the kind system. In the literature, however, it is common to establish this distinction by letting *distinct* meta-variables, say T and R, range over ordinary types and rows, respectively, so as to give the syntax a more concrete aspect. The next two examples illustrate this style and suggest common choices for $\mathcal{S}_0$ and $\mathcal{S}_1$.

1.11.7 EXAMPLE: Assume that there is a single basic kind $\star$, that $\mathcal{S}_0$ consists of the arrow type constructor $\to$, whose signature is $\star \otimes \star \Rightarrow \star$, and that $\mathcal{S}_1$ consists of the record type constructor $\tilde{}$, whose signature is $\star \Rightarrow \star$. Then, the composite kinds are $\star.\mathit{Type}$ and $\star.\mathit{Row}(\mathrm{L})$, where L ranges over the finite subsets of $\mathcal{L}$. Let us employ T (resp. R) to range over types of the former (resp. latter) kind, and refer to them as ordinary types (resp. rows). Then, the syntax of types, as defined by the signature $\mathcal{S}$, may be presented under the following form:

$$\begin{array}{lcl} \mathtt{T} & ::= & \mathtt{X} \mid \mathtt{T} \to \mathtt{T} \mid \tilde{}\,\mathtt{R} \\ \mathtt{R} & ::= & \mathtt{X} \mid \mathtt{R} \to \mathtt{R} \mid (\ell : \mathtt{T} ; \mathtt{R}) \mid \partial \mathtt{T} \end{array}$$

Ordinary types T include ordinary type variables (that is, type variables of kind $\star.\mathit{Type}$), arrow types (where the type constructor $\to$ is really $\to^{\mathit{Type}}$), and

record types, which are formed by applying the record type constructor $''$ to a row. Rows R include row variables (that is, type variables of kind $\star.\mathit{Row}(\mathrm{L})$ for some L), arrow rows (where the row constructor $\rightarrow$ is really $\rightarrow^{\mathit{Row}(\mathrm{L})}$ for some L), row extension (whereby a row R is extended with an ordinary type T at a certain label ℓ), and constant rows (formed out of an ordinary type T). It would be possible to also introduce a syntactic distinction between ordinary type variables and row variables, if desired.

Such a presentation is rather pleasant, because the syntactic segregation between ordinary types and rows makes the syntax less ambiguous. It does not allow getting rid of the kind system, however: (row) kinds are still necessary to keep track of the domain of every row. □

1.11.8 EXAMPLE: Assume that there are two basic kinds $\star$ and $\circ$, that $\mathcal{S}_0$ consists of the type constructors $\rightarrow$, abs, and pre, whose respective signatures are $\star \otimes \star \Rightarrow \star$, $\circ$, and $\star \Rightarrow \circ$, and that $\mathcal{S}_1$ consists of the record type constructor $''$, whose signature is $\circ \Rightarrow \star$. Then, the composite kinds are $\star.\mathit{Type}$, $\star.\mathit{Row}(\mathrm{L})$, $\circ.\mathit{Type}$, and $\circ.\mathit{Row}(\mathrm{L})$, where L ranges over the finite subsets of $\mathcal{L}$. Let us employ T, R, FT, and FR, respectively, to range over types of these four kinds. Then, the syntax of types, as defined by the signature $\mathcal{S}$, may be presented under the following form:

$$
\begin{array}{rcl}
\mathtt{T} & ::= & \mathtt{X} \mid \mathtt{T} \rightarrow \mathtt{T} \mid {}'' \mathtt{FR} \\
\mathtt{R} & ::= & \mathtt{X} \mid \mathtt{R} \rightarrow \mathtt{R} \mid (\ell : \mathtt{T} \,;\, \mathtt{R}) \mid \partial \mathtt{T} \\
\mathtt{FT} & ::= & \mathtt{X} \mid \mathsf{abs} \mid \mathsf{pre}\ \mathtt{T} \\
\mathtt{FR} & ::= & \mathtt{X} \mid \mathsf{abs} \mid \mathsf{pre}\ \mathtt{R} \mid (\ell : \mathtt{FT} \,;\, \mathtt{FR}) \mid \partial \mathtt{FT}
\end{array}
$$

Ordinary types T are as in the previous example, except the record type constructor $''$ must now be applied to a row of field types FR. Rows R are unchanged. Field types FT include field type variables (that is, type variables of kind $\circ.\mathit{Type}$) and applications of the type constructors abs and pre (which are really $\mathsf{abs}^{\mathit{Type}}$ and $\mathsf{pre}^{\mathit{Type}}$). Field rows FR include field row variables (that is, type variables of kind $\circ.\mathit{Row}(\mathrm{L})$ for some L), applications of the row constructors abs and pre (which are really $\mathsf{abs}^{\mathit{Row}(\mathrm{L})}$ and $\mathsf{pre}^{\mathit{Row}(\mathrm{L})}$ for some L), row extension, and constant rows, where the row components are field types FT.

In many basic applications of rows, $\mathsf{abs}^{\mathit{Row}(\mathrm{L})}$ and $\mathsf{pre}^{\mathit{Row}(\mathrm{L})}$ are never required: that is, they do not appear in the the type schemes that populate the initial environment. (Applications where they *are* required appear in (Pottier, 2000).) In that case, they may be removed from the syntax. Then, the nonterminal R becomes unreachable from the nonterminal T, which is the grammar's natural entry point, so it may be removed as well. In that simpli-

fied setting, the syntax of types and rows becomes:

$$\begin{array}{rcl} \mathtt{T} & ::= & \mathtt{X} \mid \mathtt{T} \to \mathtt{T} \mid \text{''}\,\mathtt{FR} \\ \mathtt{FT} & ::= & \mathtt{X} \mid \mathsf{abs} \mid \mathsf{pre}\ \mathtt{T} \\ \mathtt{FR} & ::= & \mathtt{X} \mid (\ell : \mathtt{FT}\,;\,\mathtt{FR}) \mid \partial\mathtt{FT} \end{array}$$

This is the syntax found in some introductory accounts of rows (Rémy, 1993b; Pottier, 2000). □

1.11.4 Meaning

We now give meaning to the type grammar defined in the previous section by interpreting it within a model. We choose to define a regular tree model, but alternatives exist; see Remark 1.11.12 below. In this model, every type constructor whose image row kind is *Type* (that is, every type constructor of the form G^{Type} or H) is interpreted as itself, as in a free tree model. However, every application of a type constructor whose image row kind is $Row(L)$ for some L receives special treatment: it is interpreted as a family of types indexed by $\mathcal{L} \setminus L$, which we encode as an infinitely branching tree. To serve as the root label of this tree, we introduce, for every κ and for every L, a symbol L^κ, whose arity is $\mathcal{L} \setminus L$. More precisely,

1.11.9 DEFINITION: The model, which consists of a set $\mathcal{M}_{\kappa.s}$ for every κ and s, is the regular tree algebra that arises out the following signature:

Symbol	Signature	Conditions
G	$(K \Rightarrow \kappa).\mathit{Type}$	$(G : K \Rightarrow \kappa) \in \mathcal{S}_0$
H	$K.\mathit{Row}(\varnothing) \Rightarrow \kappa.\mathit{Type}$	$(H : K \Rightarrow \kappa) \in \mathcal{S}_1$
L^κ	$\kappa.(\mathit{Type}^{\mathcal{L}\setminus L} \Rightarrow \mathit{Row}(L))$	

□

The first two lines in this signature coincide with the definitions of G^{Type} and H in the signature $\mathcal{S}$. Indeed, as announced above, we intend to interpret these type constructors in a syntactic manner, so each of them must have a counterpart in the model. The third line introduces the symbols L^κ hinted at above.

According to this signature, if t is a ground type of kind $\kappa.\mathit{Type}$ (that is, an element of $\mathcal{M}_{\kappa.\mathit{Type}}$), then its head symbol $t(\epsilon)$ must be of the form G or H. If t is a ground type of kind $\kappa.\mathit{Row}(L)$, then its head symbol must be L^κ, and its immediate subtrees, which are indexed by $\mathcal{L} \setminus L$, are ground types of kind $\kappa.\mathit{Type}$; in other words, the ground row t is effectively a family of ordinary ground types indexed by $\mathcal{L} \setminus L$. Thus, our intuition that rows denote infinite families of types is made literally true.

We have defined the model; there remains to explain how types are mapped to elements of the model.

1.11.10 DEFINITION: The interpretation of the type constructors that populate $\mathcal{S}$ is defined as follows.

1. Let $(G : K \Rightarrow \kappa) \in \mathcal{S}_0$. Then, G^{Type} is interpreted as the function that maps $T \in \mathcal{M}_{K.Type}$ to the ground type $t \in \mathcal{M}_{\kappa.Type}$ defined by $t(\epsilon) = G$ and $t/d = T(d)$ for every $d \in dom(K)$. This is a syntactic interpretation.

2. Let $(H : K \Rightarrow \kappa) \in \mathcal{S}_1$. Then, H is interpreted as the function that maps $T \in \mathcal{M}_{K.Row(\varnothing)}$ to the ground type $t \in \mathcal{M}_{\kappa.Type}$ defined by $t(\epsilon) = H$ and $t/d = T(d)$ for every $d \in dom(K)$. (Because H is unary, there is exactly one such d.) This is also a syntactic interpretation.

3. Let $(G : K \Rightarrow \kappa) \in \mathcal{S}_0$. Then, $G^{Row(L)}$ is interpreted as the function that maps $T \in \mathcal{M}_{K.Row(L)}$ to the ground type $t \in \mathcal{M}_{\kappa.Row(L)}$ defined by $t(\epsilon) = L^\kappa$ and $t(\ell) = G$ and $t/(\ell \cdot d) = T(d)/\ell$ for every $\ell \in \mathcal{L} \setminus L$ and $d \in dom(K)$. Thus, when applied to a family of rows, the type constructor $G^{Row(L)}$ produces a row where every component has head symbol G. This definition may sound quite technical; its effect is summed up in a simpler fashion by the equations C-ROW-GD and C-ROW-GL in the next section.

4. $\partial^{\kappa,L}$ is interpreted as the function that maps $t_1 \in \mathcal{M}_{\kappa.Type}$ to the ground type $t \in \mathcal{M}_{\kappa.Row(L)}$ defined by $t(\epsilon) = L^\kappa$ and $t/\ell = t_1$ for every $\ell \in \mathcal{L} \setminus L$. Note that t/ℓ does not depend on ℓ: t is a constant ground row.

5. Let $\ell \notin L$. Then, $\ell^{\kappa,L}$ is interpreted as the function that maps $(t_1, t_2) \in \mathcal{M}_{\kappa.Type} \times \mathcal{M}_{\kappa.Row(\ell.L)}$ to the ground type $t \in \mathcal{M}_{\kappa.Row(L)}$ defined by $t(\epsilon) = L^\kappa$ and $t/\ell = t_1$ and $t/\ell' = t_2(\ell')$ for every $\ell' \in \mathcal{L} \setminus \ell.L$. This definition is precisely row extension: indeed, the ground row t maps ℓ to t_1 and coincides with the ground row t_2 at every other label ℓ'.

□

Defining a model and an interpretation allows our presentation of rows to fit within the formalism proposed earlier in this chapter (§1.3). It also provides a basis for the intuition that rows denote infinite families of types. From a formal point of view, the model and its interpretation allow proving several constraint equivalence laws concerning rows, which are given and discussed in §1.11.5. Of course, it is also possible to accept these equivalence laws as axioms and give a purely syntactic account of rows, without relying on a model; this is how rows were historically dealt with (Rémy, 1993a).

$$(\ell_1 : T_1 \,;\, \ell_2 : T_2 \,;\, T_3) = (\ell_2 : T_2 \,;\, \ell_1 : T_1 \,;\, T_3) \qquad \text{(C-ROW-LL)}$$
$$\partial T = (\ell : T \,;\, \partial T) \qquad \text{(C-ROW-DL)}$$
$$G\ \partial T_1\ \ldots\ \partial T_n = \partial(G\ T_1\ \ldots\ T_n) \qquad \text{(C-ROW-GD)}$$
$$G\ (\ell : T_1 \,;\, T_1')\ \ldots\ (\ell : T_n \,;\, T_n') = (\ell : G\ T_1\ \ldots\ T_n \,;\, G\ T_1'\ \ldots\ T_n') \qquad \text{(C-ROW-GL)}$$

Figure 1-17: Equational reasoning with rows

1.11.11 REMARK: We have not defined the interpretation of the subtyping predicate, because much of the material that follows is independent of it. One common approach is to adopt a nonstructural definition of subtyping (Example 1.3.9), where every L^κ is considered covariant in every direction, and where the variances and relative ordering of all other symbols (G and H) are chosen at will, subject to the restrictions associated with nonstructural subtyping and to the conditions necessary to ensure type soundness.

Recall that the arrow type constructor $\to$ must be contravariant in its domain and covariant in its codomain. The record type constructor $''$ is usually covariant. These properties are exploited in proofs of the subject reduction theorem. The type constructors $\to$ and $''$ are usually incompatible. This property is exploited in proofs of the progress theorem. In the case of Example 1.11.7, because no type constructors other than $\to$ and $''$ are present, these conditions imply that there is no sensible way of interpreting subtyping other than equality. In the case of Example 1.11.8, two sensible interpretations of subtyping exist: one is equality, while the other is the nonstructural subtyping order obtained by letting $\mathsf{pre} \leqslant \mathsf{abs}$. □

1.11.12 REMARK: The model proposed above is a regular tree model. Of course, it is possible to adopt a finite tree model instead. Furthermore, other interpretations of rows are possible: for instance, Fähndrich (1999) extends the set constraints formalism with rows. In his model, an ordinary type is interpreted as a set of values, while a row is interpreted as a set of functions from labels to values. While the definition of the model may vary, the key point is that the characteristic laws of rows, which we discuss in §1.11.5, hold in the model. □

1.11.5 Reasoning with rows

The interpretation presented in the previous section was designed so as to support the intuition that a row denotes an infinite family of types, indexed by labels, that the row constructor $\ell : \cdot \,;\, \cdot$ denotes row extension, and that the row constructor ∂ denotes the creation of a constant row. From a formal point

of view, the definition of the model and interpretation may be exploited to establish some reasoning principles concerning rows. These principles take the form of equations between types (Figure 1-17) and of constraint equivalence laws (Figure 1-18), which we now explain and prove.

1.11.13 REMARK: As announced earlier, we omit the superscripts of row constructors. We also omit the side conditions that concern the kind of the type variables ($\mathtt{X}$) and type meta-variables ($\mathtt{T}$) involved. Thus, each equation in Figure 1-17 really stands for the (infinite) family of equations obtained by reconstructing the missing kind information in a consistent way. For instance, the second equation may be read $\partial^{\ell.\mathtt{L}}\mathtt{T} = (\ell^{\kappa,\mathtt{L}} : \mathtt{T} \,;\, \partial^{\mathtt{L}}\mathtt{T})$, where $\ell \notin \mathtt{L}$ and $\mathtt{T}$ has kind $\kappa.\mathit{Type}$. □

1.11.14 EXERCISE [RECOMMENDED, ★, ↛]: Reconstruct all of the missing kind information in the equations of Figure 1-17. □

1.11.15 REMARK: There is a slight catch with the unannotated version of the second equation: its left-hand side admits strictly *more* kinds than its right-hand side, because the former has row kind $\mathit{Row}(\mathtt{L})$ for every $\mathtt{L}$, while the latter has row kind $\mathit{Row}(\mathtt{L})$ for every $\mathtt{L}$ such that $\ell \notin \mathtt{L}$ holds. As a result, while replacing the unannotated term $(\ell : \mathtt{T} \,;\, \partial\mathtt{T})$ with $\partial\mathtt{T}$ is always valid, the converse is not: replacing the unannotated term $\partial\mathtt{T}$ with $(\ell : \mathtt{T} \,;\, \partial\mathtt{T})$ is valid only if it does not result in an ill-kinded term. □

The first equation states that rows are equal up to commutation of labels. For the equation to be well-kinded, the labels ℓ_1 and ℓ_2 must be distinct. The equation holds under our interpretation because extension of a ground row at ℓ_1 and extension of a ground row at ℓ_2 commute. The second equation states that $\partial\mathtt{T}$ maps every label within its domain to $\mathtt{T}$, that is, $\partial^{\mathtt{L}}\mathtt{T}$ maps every label $\ell \notin \mathtt{L}$ to $\mathtt{T}$. This equation holds because $\partial\mathtt{T}$ is interpreted as a constant row. The last two equations deal with the relationship between the row constructors G and the ordinary type constructor G. Indeed, notice that their left-hand sides involve $\mathrm{G}^{\mathit{Row}(\mathtt{L})}$ for some $\mathtt{L}$, while their right-hand sides involve $\mathrm{G}^{\mathit{Type}}$. Both equations state that it is equivalent to apply $\mathrm{G}^{\mathit{Row}(\mathtt{L})}$ at the level of rows or to apply $\mathrm{G}^{\mathit{Type}}$ at the level of types. Our interpretation of $\mathrm{G}^{\mathit{Row}(\mathtt{L})}$ was designed so as to give rise to these equations: indeed, the application of $\mathrm{G}^{\mathit{Row}(\mathtt{L})}$ to n ground rows (where n is the arity of G) is interpreted as a pointwise application of $\mathrm{G}^{\mathit{Type}}$ to the rows' components (item 3 of Definition 1.11.10). Their use is illustrated in Examples 1.11.29 and 1.11.47.

1.11.16 LEMMA: Each of the equations in Figure 1-17 is equivalent to true. □

Proof: Each equation can be considered independently. It suffices to see that any ground assignment ϕ sends both sides of the equation to the same ele-

$$(\ell_1 : \mathtt{T}_1 \,;\, \mathtt{T}'_1) = (\ell_2 : \mathtt{T}_2 \,;\, \mathtt{T}'_2) \quad \equiv \quad \exists \mathtt{X}.(\mathtt{T}'_1 = (\ell_2 : \mathtt{T}_2 \,;\, \mathtt{X}) \wedge \mathtt{T}'_2 = (\ell_1 : \mathtt{T}_1 \,;\, \mathtt{X})) \qquad \text{(C-MUTE-LL)}$$
$$\text{if } \mathtt{X} \# \mathit{ftv}(\mathtt{T}_1, \mathtt{T}'_1, \mathtt{T}_2, \mathtt{T}'_2) \wedge \ell_1 \neq \ell_2$$

$$\partial \mathtt{T} = (\ell : \mathtt{T}' \,;\, \mathtt{T}'') \quad \equiv \quad \mathtt{T} = \mathtt{T}' \wedge \partial \mathtt{T} = \mathtt{T}'' \qquad \text{(C-MUTE-DL)}$$

$$\mathsf{G}\ \mathtt{T}_1\ \ldots\ \mathtt{T}_n = \partial \mathtt{T} \quad \equiv \quad \exists \mathtt{X}_1 \ldots \mathtt{X}_n.(\mathsf{G}\ \mathtt{X}_1\ \ldots\ \mathtt{X}_n = \mathtt{T} \wedge \textstyle\bigwedge_{i=1}^{n}(\mathtt{T}_i = \partial \mathtt{X}_i)) \qquad \text{(C-MUTE-GD)}$$
$$\text{if } \mathtt{X}_1 \ldots \mathtt{X}_n \# \mathit{ftv}(\mathtt{T}_1, \ldots, \mathtt{T}_n, \mathtt{T})$$

$$\mathsf{G}\ \mathtt{T}_1\ \ldots\ \mathtt{T}_n = (\ell : \mathtt{T} \,;\, \mathtt{T}') \quad \equiv \quad \exists \mathtt{X}_1 \ldots \mathtt{X}_n, \mathtt{X}'_1 \ldots \mathtt{X}'_n.(\mathsf{G}\ \mathtt{X}_1\ \ldots\ \mathtt{X}_n = \mathtt{T} \wedge \mathsf{G}\ \mathtt{X}'_1\ \ldots\ \mathtt{X}'_n = \mathtt{T}' \wedge \textstyle\bigwedge_{i=1}^{n}(\mathtt{T}_i = (\ell : \mathtt{X}_i \,;\, \mathtt{X}'_i)))$$
$$\text{if } \mathtt{X}_1 \ldots \mathtt{X}_n, \mathtt{X}'_1 \ldots \mathtt{X}'_n \# \mathit{ftv}(\mathtt{T}_1, \ldots, \mathtt{T}_n, \mathtt{T}, \mathtt{T}') \qquad \text{(C-MUTE-GL)}$$

Figure 1-18: Constraint equivalence laws involving rows

ment in the model, which follows directly from the meaning of row types. Notice that this fact only depends on the semantics of types and not on the semantics of the subtyping predicate. □

The four equations in Figure 1-17 show that two types with *distinct* head symbols may denote the *same* element of the model. In other words, in the presence of rows, the interpretation of types is no longer *free*: an equation of the form $\mathtt{T}_1 = \mathtt{T}_2$, where $\mathtt{T}_1$ and $\mathtt{T}_2$ have distinct head symbols, is not necessarily equivalent to false. In Figure 1-18, we give several constraint equivalence laws, known as *mutation* laws, that concern such "heterogeneous" equations, and, when viewed as rewriting rules, allow solving them. To each equation in Figure 1-17 corresponds a mutation law. The soundness of the mutation law, that is, the fact that its right-hand side entails its left-hand side, follows from the corresponding equation. The completeness of the mutation law, that is, the fact that its left-hand side entails its right-hand side, holds by design of the model.

1.11.17 EXERCISE [RECOMMENDED, ★, ↠]: Reconstruct all of the missing kind information in the laws of Figure 1-18. □

Let us now review the four mutation laws. For the sake of brevity, in the following informal explanation, we assume that a ground assignment ϕ that satisfies the left-hand equation is fixed, and write "the ground type $\mathtt{T}$" for "the ground type $\phi(\mathtt{T})$". C-MUTE-LL concerns an equation between two rows, which are both given by extension, but exhibit distinct head labels ℓ_1 and ℓ_2. When this equation is satisfied, both of its members must denote the same ground row. Thus, the ground row $\mathtt{T}'_1$ must map ℓ_2 to the ground type

$\mathtt{T}_2$, while, symmetrically, the ground row $\mathtt{T}_2'$ must map ℓ_1 to the ground type $\mathtt{T}_1$. This may be expressed by two equations of the form $\mathtt{T}_1' = (\ell_2 : \mathtt{T}_2 \ ; \ldots)$ and $\mathtt{T}_2' = (\ell_1 : \mathtt{T}_1 \ ; \ldots)$. Furthermore, because the ground rows $\mathtt{T}_1'$ and $\mathtt{T}_2'$ must agree on their common labels, the ellipses in these two equations must denote the same ground row. This is expressed by letting the two equations share a fresh, existentially quantified row variable $\mathtt{X}$. C-MUTE-DL concerns an equation between two rows, one of which is given as a constant row, the other of which is given by extension. Then, because the ground row $\partial\mathtt{T}$ maps every label to the ground type $\mathtt{T}$, the ground type $\mathtt{T}'$ must coincide with the ground type $\mathtt{T}$, while the ground row $\mathtt{T}''$ must map every label in its domain to the ground type $\mathtt{T}$. This is expressed by the equations $\mathtt{T} = \mathtt{T}'$ and $\partial\mathtt{T} = \mathtt{T}''$. C-MUTE-GD and C-MUTE-GL concern an equation between two rows, one of which is given as an application of a row constructor G, the other of which is given either as a constant row or by extension. Again, the laws exploit the fact that the ground row G $\mathtt{T}_1 \ \ldots \ \mathtt{T}_n$ is obtained by applying the type constructor G, pointwise, to the ground rows $\mathtt{T}_1, \ldots, \mathtt{T}_n$. If, as in C-MUTE-GD, it coincides with the constant ground row $\partial\mathtt{T}$, then every $\mathtt{T}_i$ must itself be a constant ground row, of the form $\partial\mathtt{X}_i$, and $\mathtt{T}$ must coincide with G $\mathtt{X}_1 \ \ldots \ \mathtt{X}_n$. C-MUTE-GL is obtained in a similar manner.

1.11.18 LEMMA: Each of the equivalence laws in Figure 1-18 holds. □

Proof: By examination of each of the laws.

∘ *Case* C-MUTE-LL: Let $\mathtt{X} \# \mathit{ftv}(\mathtt{T}_1, \mathtt{T}_1', \mathtt{T}_2, \mathtt{T}_2')$ **(1)** and $\ell_1 \neq \ell_2$. Let $\mathit{Row}(\mathrm{L})$ be the row kind of this equation. Let ϕ be a ground assignment that satisfies the constraint $(\ell_1 : \mathtt{T}_1 \ ; \ \mathtt{T}_1') = (\ell_2 : \mathtt{T}_2 \ ; \ \mathtt{T}_2')$. Then, ϕ maps both members of the equation to a ground type t of row kind $\mathit{Row}(\mathrm{L})$. According to the interpretation of row extension, we must have $t(\epsilon) = \mathrm{L}$, $t/\ell_1 = \phi(\mathtt{T}_1) = \phi(\mathtt{T}_2')/\ell_1$, $t/\ell_2 = \phi(\mathtt{T}_1')/\ell_2 = \phi(\mathtt{T}_2)$, and $t/\ell = \phi(\mathtt{T}_2')/\ell = \phi(\mathtt{T}_1')/\ell$ for every $\ell \in \mathcal{L}\backslash\ell_1.\ell_2.\mathrm{L}$. Now, let t' be the ground row of row kind $\mathit{Row}(\ell_1.\ell_2.\mathrm{L})$ defined by $t'(\epsilon) = \ell_1.\ell_2.\mathrm{L}$ and $t'/\ell = t/\ell$ for every $\ell \in \mathcal{L}\backslash\ell_1.\ell_2.\mathrm{L}$. By construction and by (1), $\phi[\mathtt{X} \mapsto t']$ satisfies both equations $\mathtt{T}_1' = (\ell_2 : \mathtt{T}_2 \ ; \mathtt{X})$ and $\mathtt{T}_2' = (\ell_1 : \mathtt{T}_1 \ ; \mathtt{X})$. Thus, by CM-EXISTS and (1), ϕ satisfies $\exists\mathtt{X}.(\mathtt{T}_1' = (\ell_2 : \mathtt{T}_2 \ ; \mathtt{X}) \wedge \mathtt{T}_2' = (\ell_1 : \mathtt{T}_1 \ ; \mathtt{X}))$. This proves that C-MUTE-LL is complete.

Conversely, we have:

$$
\begin{aligned}
& \exists\mathtt{X}.(\mathtt{T}_1' = (\ell_2 : \mathtt{T}_2 \ ; \mathtt{X}) \wedge \mathtt{T}_2' = (\ell_1 : \mathtt{T}_1 \ ; \mathtt{X})) \\
\equiv\ & \exists\mathtt{X}.((\ell_1 : \mathtt{T}_1 \ ; \mathtt{T}_1') = (\ell_1 : \mathtt{T}_1 \ ; \ell_2 : \mathtt{T}_2 \ ; \mathtt{X}) \wedge \\
& \qquad (\ell_2 : \mathtt{T}_2 \ ; \mathtt{T}_2') = (\ell_2 : \mathtt{T}_2 \ ; \ell_1 : \mathtt{T}_1 \ ; \mathtt{X})) && \textbf{(2)} \\
\Vdash\ & \exists\mathtt{X}.(\ell_1 : \mathtt{T}_1 \ ; \mathtt{T}_1') = (\ell_2 : \mathtt{T}_2 \ ; \mathtt{T}_2') && \textbf{(3)} \\
\equiv\ & (\ell_1 : \mathtt{T}_1 \ ; \mathtt{T}_1') = (\ell_2 : \mathtt{T}_2 \ ; \mathtt{T}_2') && \textbf{(4)}
\end{aligned}
$$

$$(\ell_1 : X_1 \,;\, X_1') = (\ell_2 : T_2 \,;\, T_2') = \epsilon \quad \rightarrow \quad \exists X.(X_1' = (\ell_2 : T_2 \,;\, X) \wedge T_2' = (\ell_1 : X_1 \,;\, X)) \wedge (\ell_1 : X_1 \,;\, X_1') = \epsilon \quad \text{(S-MUTE-LL)}$$
if $\ell_1 \neq \ell_2$

$$\partial X = (\ell : T \,;\, T') = \epsilon \quad \rightarrow \quad X = T \wedge \partial X = T' \wedge \partial X = \epsilon \quad \text{(S-MUTE-DL)}$$

$$G\ T_1\ \ldots\ T_n = \partial X = \epsilon \quad \rightarrow \quad \exists X_1 \ldots X_n.(G\ X_1\ \ldots\ X_n = X \wedge \textstyle\bigwedge_{i=1}^{n}(T_i = \partial X_i)) \wedge \partial X = \epsilon \quad \text{(S-MUTE-GD)}$$

$$G\ T_1\ \ldots\ T_n = (\ell : X \,;\, X') = \epsilon \quad \rightarrow \quad \exists X_1 \ldots X_n, X_1' \ldots X_n'.(G\ X_1\ \ldots\ X_n = X \wedge G\ X_1'\ \ldots\ X_n' = X' \wedge \textstyle\bigwedge_{i=1}^{n}(T_i = (\ell : X_i \,;\, X_i'))) \wedge (\ell : X \,;\, X') = \epsilon \quad \text{(S-MUTE-GL)}$$

$$F\,\vec{T} = F'\,\vec{T}' = \epsilon \quad \rightarrow \quad \mathsf{false} \quad \text{(S-CLASH')}$$
if $F \neq F'$ and none of the four rules above applies

Figure 1-19: Row unification (corrigendum to Figure 1-11)

where (2) is by congruence; (3) is by C-ROW-LL and transitivity of equality; and (4) follows from C-EX* and (1). This proves that C-MUTE-LL is sound.

◦ *Cases* C-MUTE-DL, C-MUTE-GD, and C-MUTE-GL are left to the reader.
□

1.11.6 Solving equality constraints in the presence of rows

We now extend the unification algorithm given in §1.8.1 with support for rows. The extended algorithm is intended to solve unification problems where the syntax and interpretation of types are as defined in §1.11.3 and §1.11.4, respectively. Its specification consists of the original rewriting rules of Figure 1-11, minus S-CLASH, which is removed and replaced with the rules given in Figure 1-19. Indeed, S-CLASH is no longer valid in the presence of rows: not all distinct type constructors are incompatible.

The extended algorithm features four *mutation* rules, which are in direct correspondence with the mutation laws of Figure 1-18, as well as a weakened version of S-CLASH, dubbed S-CLASH', which applies when neither S-DECOMPOSE nor the mutation rules are applicable. (Let us point out that, in S-DECOMPOSE, the meta-variable F ranges over all type constructors in the signature $\mathcal{S}$, so that S-DECOMPOSE is applicable to multi-equations of the form $\partial X = \partial T = \epsilon$ or $(\ell : X \,;\, X') = (\ell : T \,;\, T') = \epsilon$.) Three of the muta-

tion rules may allocate fresh type variables, which must be chosen fresh for the rule's left-hand side. The four mutation rules paraphrase the four mutation laws very closely. Two minor differences are (i) the mutation rules deal with multi-equations, as opposed to equations; and (ii) any subterm that appears more than once on the right-hand side of a rule is required to be a type variable, as opposed to an arbitrary type. Neither of these features is specific to rows: both may be found in the definition of the standard unification algorithm (Figure 1-11), where they help reason about sharing.

1.11.19 EXERCISE [★, ↠]: Check that the rewriting rules in Figure 1-19 preserve well-kindedness. Conclude that, provided its input constraint is well-kinded, the unification algorithm needs not keep track of kinds. □

1.11.20 REMARK: Like S-DECOMPOSE, the mutation rules accept a multi-equation with two non-variable members. One of these members is discarded, making the original multi-equation smaller; furthermore, new multi-equations may be created. One novelty with respect to S-DECOMPOSE is that the two non-variable terms at hand no longer necessarily have the same head symbol. Thus, *which* is discarded and *which* is kept may make a difference. How do we resolve this choice?

In the case of S-MUTE-LL, the two terms at hand have head symbols $(\ell_1 : \cdot\ ;\ \cdot)$ and $(\ell_2 : \cdot\ ;\ \cdot)$, respectively. Because a row label is a good as another, there does not seem, at first sight, to be a meaningful way of choosing which term to keep and which to discard. However, let us impose a fixed, arbitrary total order on labels, and let us modify S-MUTE-LL by replacing the side condition $\ell_1 \neq \ell_2$ with $\ell_1 < \ell_2$. In other words, among two syntactic representations of the same row, let us choose to keep the one that exhibits the smaller head label. It is believed that, under such a strategy, rows tend to "self-organize" so that labels appear in increasing order. Under this conjecture, two rows that are to be unified are more likely to be organized in the same order, so that S-DECOMPOSE, which is cheap, is more likely to be applicable than S-MUTE-LL, which is expensive, because it allocates a fresh row varible as well as two new row terms. Thus, it is believed that this strategy increases the efficiency of the unification algorithm. No rigorous study of this phenomenon exists, however.

In the case of S-MUTE-DL, we choose to keep a term of the form $\partial\cdot$, rather than a term of the form $(\ell : \cdot\ ;\ \cdot)$, since the former is more concise.

In the case of S-MUTE-GD and S-MUTE-GL, the choice is between favoring an application of G, on the one hand, or an application of one the row constructors ∂ or ℓ, on the other hand. The rules in Figure 1-19 choose the latter. However, depending on the arity of G, the former may be more concise, so a variation of these two rules is conceivable. □

The properties of the unification algorithm are preserved by this extension, as witnessed by the next three lemmas. Please note that the termination of reduction is ensured *only* when the initial unification problem is well-kinded. The ill-kinded unification problem $X = (\ell_1 : T ; Y) \wedge X = (\ell_2 : T ; Y)$, where ℓ_1 and ℓ_2 are distinct, illustrates this point.

1.11.21 LEMMA: The rewriting system $\rightarrow$ is strongly normalizing. □

Proof: We reuse the proof of Lemma 1.8.4. We refine the weight of type constructors as follows. Let $\mathcal{L}_f$ be a finite subsets of $\mathcal{L}$ that includes at least all labels appearing in the constraints. Note that labels are never created during reduction of constraints, so $\mathcal{L}_f$ can be chosen before any reduction occurs. We define the weight of a kind by $kw(Type) = 1$ and $kw(Row(L)) = 2 + |\mathcal{L}_f \setminus L|$. Finally, the weight of a type constructor F of row kind $s_1 \times \ldots s_n \Rightarrow s$ is a polynom in X, Y with non negative integer coefficients, defined by $cw(F) = (X^{kw(s)}Y^{k(F)} + 2)(2 + n)$ where $k(F)a$ is, depending on F, 0 for ∂ and $\check{}$, 1 for ℓ, and 2 for G. Polynoms are ordered in increasing power of X, then in increasing power of Y. Since this choice satisfies the condition found in the proof of Lemma 1.8.4, it suffices to check that the rules S-MUTE-LL, S-MUTE-LG, S-MUTE-DG, S-MUTE-DL, and S-CLASS-I are also weight-decreasing. We write $\delta(T)$ for $iw(T) - w(T)$, which is either 2 or -1, depending on whether $T \in \mathcal{V}$; thus, we always have $\delta(T) \geq -1$.

∘ *Case* S-MUTE-LL: This amounts to replacing the term $(\ell_2 : T_2 ; T_2')$ by terms X_1', $(\ell_2 : T_2 ; Y)$, T_2', and $(\ell_1 : X_1 ; Y)$. This decreases weight by (omitting internal weight of variables, which are null):

$$cw(\ell_2{}^L) - cw(\ell_2{}^{\ell_1 \cdot L}) - cw(\ell_1{}^{\ell_2 \cdot L}) + \delta(T_2') - 1.$$

The monom of highest degree, which is $cw(\ell_2{}^L)$, has a positive coefficient.

∘ *Case* S-MUTE-LG: This amounts to replacing the term $F^{Row(L)}\ T_i^{i\in I}$ by terms X, $F^{Type}\ Y_i^{i\in I}$, X', $F^{Row(\ell.L)}\ Y_i^{i\in I}$, $T_i^{i\in I}$, and $\ell^L : Y_i ; Y_i'$. This decreases the weight by:

$$cw(F^{Row(L)}) - cw(\ell^L) - cw(F^{Row(\ell.L)}) + \Sigma^{i\in I}\delta(T_i) - 2.$$

The monom of highest degree, which is the first one, has a positive coefficient.

∘ *Case* S-MUTE-DG: This amounts to replacing the term $G\ T_i^{i\in I}$ by terms X, $G\ Y_i^{i\in I}$, and $(T_i = \partial Y_i)^{i\in I}$. This decreases weight by:

$$cw(G^{Row(L)}) - \Sigma^{i\in I}(cw(\partial^L)) - cw(G^{Type}) + \Sigma^{i\in I}\delta(T_i) - 1$$

The monom of highest degree, which is the first one, has a positive coefficient.

∘ *Case* S-MUTE-DL: This amounts to replacing the term $(\ell^L : T\ ;\ T'$ by the terms X, T, $\partial^{\ell.L}X$, and T'. This decreases weight by:

$$cw(\ell^L) - cw(\partial^L) + \delta(T) + \delta(T') - 1$$

The monom of highest degree, which is the first one, has a positive coefficient. □

1.11.22 LEMMA: $U_1 \to U_2$ implies $U_1 \equiv U_2$. □

Proof: It suffices to check the property independently for each rule defining $\to$. The proof for rules of Figure 1-11 but S-CLASH remain valid for row terms. For S-DECOMPOSE, it follows by the invariance of all type constructors, which is preserved for row terms (Lemma **??**). For rule S-CLASS-I it follows by Lemma **??** and for mutation rules, it follows by Lemma 1.11.18. □

1.11.23 LEMMA: Every normal form is either false or of the form $\mathcal{X}[U]$, where $\mathcal{X}$ is an existential constraint context, U is a standard conjunction of multi-equations and, if the model is syntactic, U is acyclic. These conditions imply that U is satisfiable. □

The time complexity of standard first-order unification is quasi-linear. What is, then, the time complexity of row unification? Only a partial answer is known. In practice, the algorithm given in this chapter is extremely efficient, and appears to behave just as well as standard unification. In theory, the complexity of row unification remains unexplored, and forms an interesting open issue.

1.11.24 EXERCISE [★★★, $\nrightarrow$]: The unification algorithm presented above, although very efficient in practice, does *not* have linear or quasi-linear time complexity. Find a family of unification problems U_n such that the size of U_n is linear with respect to n and the number of steps required to reach its normal form is quadratic with respect to n. □

1.11.25 REMARK: *Mutation* is a common technique for solving equations in a large class of non-free algebras that are described by *syntactic theories* (Kirchner and Klay, 1990). The equations of Figure 1-17 happen to form a syntactic presentation of an equational theory. Thus, it is possible to derive a unification algorithm out of these equations in a systematic way (Rémy, 1993a). Here, we have presented the same algorithm in a direct manner, without relying on the apparatus of syntactic theories. □

1.11.7 Operations on records

We now illustrate the use of rows for typechecking operations on records. We begin with full records; our treatment follows (Rémy, 1992b).

1.11.26 EXAMPLE [FULL RECORDS]: As in §1.11.2, let us begin with full records, whose domain is exactly $\mathcal{L}$. The primitive operations are record creation $\{\cdot\}$, update $\{\cdot \text{ with } \ell = \cdot\}$, and access $\cdot.\{\ell\}$.

Let us first introduce some useful syntactic sugar. Let $<$ denote a fixed strict total order on row labels. For every set of labels L of cardinal n, let us introduce a $(n+1)$-ary constructor $\{\}_L$. We write $\{\ell_1 = t_1; \ldots; \ell_n = t_n; t\}$ for the application $\{\}_L\ t_{i_1}\ \ldots\ t_{i_n}\ t$, where $L = \{\ell_1, \ldots, \ell_n\} = \{\ell_{i_1}, \ldots, \ell_{i_n}\}$ and $\ell_{i_1} < \ldots < \ell_{i_n}$ holds. The use of the total order $<$ makes the meaning of record expressions independent of the order in which fields are defined; in particular, it allows fixing the order in which $t_1, \ldots, t_n$ are evaluated. We abbreviate the record value $\{\ell_1 = v_1; \ldots; \ell_n = v_n; v\}$ as $\{V; v\}$, where V is the finite function that maps ℓ_i to v_i for every $i \in \{1, \ldots, n\}$.

The operational semantics of the above three operations may now be defined in the following straightforward manner. First, record creation $\{\cdot\}$ is precisely the unary constructor $\{\}_\varnothing$. Second, for every $\ell \in \mathcal{L}$, let update $\{\cdot \text{ with } \ell = \cdot\}$ and access $\cdot.\{\ell\}$ be destructors of arity 1 and 2, respectively, equipped with the following reduction rules:

$$\begin{array}{rcll}
\{\{V; v\} \text{ with } \ell = v'\} & \xrightarrow{\delta} & \{V[\ell \mapsto v']; v\} & & \text{(R-UPDATE)} \\
\{V; v\}.\{\ell\} & \xrightarrow{\delta} & V(\ell) & (\ell \in \mathit{dom}(V)) & \text{(R-ACCESS-1)} \\
\{V; v\}.\{\ell\} & \xrightarrow{\delta} & v & (\ell \notin \mathit{dom}(V)) & \text{(R-ACCESS-2)}
\end{array}$$

In these rules, $V[\ell \mapsto v]$ stands for the function that maps ℓ to v and coincides with V at every other label, while $V(\ell)$ stands for the image of ℓ through V. Because these rules make use of the syntactic sugar defined above, they are, strictly speaking, rule *schemes*: each of them really stands for the infinite family of rules that would be obtained if the syntactic sugar was eliminated.

Let us now define the syntax of types as in Example 1.11.7. Let the initial environment Γ_0 contain the following bindings:

$$\begin{array}{rl}
\{\}_{\{\ell_1,\ldots,\ell_n\}}: & \forall X_1 \ldots X_n X. X_1 \to \ldots \to X_n \to X \to \text{''}\,(\ell_1 : X_1; \ldots; \ell_n : X_n; \partial X) \\
& \text{where } \ell_1 < \ldots < \ell_n \\
\{\cdot \text{ with } \ell = \cdot\}: & \forall X X' Y.\text{''}\,(\ell : X\ ;\ Y) \to X' \to \text{''}\,(\ell : X'\ ;\ Y) \\
\cdot.\{\ell\}: & \forall X Y.\text{''}\,(\ell : X\ ;\ Y) \to X
\end{array}$$

Please note that, in particular, the type scheme assigned to record creation $\{\cdot\}$ is $\forall X. X \to \text{''}\,(\partial X)$. As a result, these bindings are exactly as announced in §1.11.2.

To illustrate how these definitions work together, let us consider the program $\{\{0\}$ with $\ell_1 = \mathtt{true}\}.\{\ell_2\}$, which builds a record, extends it at ℓ_1, then accesses it at ℓ_2. Can we build an HM(X) type derivation for it, under the constraint true and the initial environment Γ_0? To begin, by looking up Γ_0 and using HMX-INST, we find that $\{\cdot\}$ has type $\mathsf{int} \to \tilde{}\ (\partial\mathsf{int})$. Thus, assuming that 0 has type int, the expression $\{0\}$ has type $\tilde{}\ (\partial\mathsf{int})$. Indeed, this expression denotes a record all of whose fields hold an integer value. Then, by looking up Γ_0 and using HMX-INST, we find that $\{\cdot$ with $\ell_1 = \cdot\}$ has type $\tilde{}\ (\ell_1 : \mathsf{int}\ ;\ \partial\mathsf{int}) \to \mathsf{bool} \to \tilde{}\ (\ell_1 : \mathsf{bool}\ ;\ \partial\mathsf{int})$. May we immediately use HMX-APP to typecheck the application of $\{\cdot$ with $\ell_1 = \cdot\}$ to $\{0\}$? Unfortunately, no, because there is an apparent mismatch between the expected type $\tilde{}\ (\ell_1 : \mathsf{int}\ ;\ \partial\mathsf{int})$ and the effective type $\tilde{}\ (\partial\mathsf{int})$. To work around this problem, let us recall that, by C-ROW-DL, the equation $\tilde{}\ (\partial\mathsf{int}) = \tilde{}\ (\ell_1 : \mathsf{int}\ ;\ \partial\mathsf{int})$ is equivalent to true. Thus, HMX-SUB allows proving that $\{0\}$ has type $\tilde{}\ (\ell_1 : \mathsf{int}\ ;\ \partial\mathsf{int})$. Assuming that $\mathtt{true}$ has type bool, we may now apply HMX-APP and deduce

$$\mathsf{true}, \Gamma_0 \vdash \{\{0\} \text{ with } \ell_1 = \mathtt{true}\} : \tilde{}\ (\ell_1 : \mathsf{bool}\ ;\ \partial\mathsf{int}).$$

We let the reader check that, in a similar manner, which involves C-ROW-DL, C-ROW-LL, and HMX-SUB, one may prove that $\{\{0\}$ with $\ell_1 = \mathtt{true}\}.\{\ell_2\}$ has type int, provided ℓ_1 and ℓ_2 are distinct. □

1.11.27 EXERCISE [★★, ↛]: Unfold the definition of the constraint $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\{\{0\}$ with $\ell_1 = \mathtt{true}\}.\{\ell_2\} : \mathtt{X}]\!]$, which states that $\mathtt{X}$ is a valid type for the above program. Assuming that subtyping is interpreted as equality, simulate a run of the constraint solver (§1.8), extended with support for rows (§1.11.6), so as to solve this constraint. Check that the solved form is equivalent to $\mathtt{X} = \mathsf{int}$. □

1.11.28 EXERCISE [★★★]: Check that the definitions of Example 1.11.26 meet the requirements of Definition 1.7.7. □

1.11.29 EXAMPLE [RECORD APPLICATION]: Let us now introduce a more unusual primitive operation on full records. This operation accepts two records, the first of which is expected to hold a function in every field, and produces a new record, whose contents are obtained by applying, pointwise, the functions in the first record to the values in the second record. In other words, this new primitive operation lifts the standard application combinator (which may be defined as $\lambda\mathtt{f}.\lambda\mathtt{z}.\mathtt{f}\ \mathtt{z}$), pointwise, to the level of records. For this reason, we refer to it as apply. Its operational semantics is defined by making

it a binary destructor and equipping it with the following reduction rules:

$$\begin{array}{lcll}
\mathsf{apply}\ \{\mathtt{V};\mathtt{v}\}\{\mathtt{V}';\mathtt{v}'\} & \xrightarrow{\delta} & \{\mathtt{V}\ \mathtt{V}';\mathtt{v}\ \mathtt{v}'\} & \text{(R-APPLY-1)}\\
\mathsf{apply}\ \{\mathtt{V};\mathtt{v}\}\{\mathtt{V}';\mathtt{v}'\} & \xrightarrow{\delta} & \mathsf{apply}\ \{\mathtt{V};\mathtt{v}\}\{\mathtt{V}'[\ell \mapsto \mathtt{v}'];\mathtt{v}'\} & \text{(R-APPLY-2)}\\
 & & \text{if } \ell \in \mathit{dom}(\mathtt{V}) \setminus \mathit{dom}(\mathtt{V}') & \\
\mathsf{apply}\ \{\mathtt{V};\mathtt{v}\}\{\mathtt{V}';\mathtt{v}'\} & \xrightarrow{\delta} & \mathsf{apply}\ \{\mathtt{V}[\ell' \mapsto \mathtt{v}];\mathtt{v}\}\{\mathtt{V}';\mathtt{v}'\} & \text{(R-APPLY-3)}\\
 & & \text{if } \ell' \in \mathit{dom}(\mathtt{V}') \setminus \mathit{dom}(\mathtt{V}) &
\end{array}$$

In the first rule, $\mathtt{V}\ \mathtt{V}'$ is defined only if $\mathtt{V}$ and $\mathtt{V}'$ have a common domain; it is then defined as the function that maps ℓ to the expression $\mathtt{V}(\ell)\ \mathtt{V}'(\ell)$. The second and third rules, which are symmetric, deal with the case where some field is explicitly defined in one input record, but not in the other; in that case, the field is made explicit by creating a copy of the record's default value.

The syntax of types remains as in Example 1.11.26. We extend the initial environment Γ_0 with the following binding:

$$\mathsf{apply}: \quad \forall \mathtt{XY}.''\,(\mathtt{X} \to \mathtt{Y}) \to ''\,\mathtt{X} \to ''\,\mathtt{Y}$$

To understand this type scheme, recall that the principal type scheme of the standard application combinator (which may be defined as $\lambda\mathtt{f}.\lambda\mathtt{z}.\mathtt{f}\ \mathtt{z}$) is $\forall \mathtt{XY}.(\mathtt{X} \to \mathtt{Y}) \to \mathtt{X} \to \mathtt{Y}$. The type scheme assigned to apply is very similar: the most visible difference is that both arguments, as well as the result, are now wrapped within the record type constructor $''$. A more subtle, yet essential change is that $\mathtt{X}$ and $\mathtt{Y}$ are now row variables: their kind is $\star.\mathit{Row}(\varnothing)$. As a result, the leftmost occurrence of the arrow constructor is really $\to^{\mathit{Row}(\varnothing)}$. Thus, we are exploiting the presence of type constructors of the form G^{s}, with $\mathsf{s} \neq \mathit{Type}$, in the signature $\mathcal{S}$.

To illustrate how these definitions work together, let us consider the program $\mathsf{apply}\ \{\ell = \mathtt{not};\mathtt{succ}\}\ \{\ell = \mathtt{true};0\}$, where the terms not and succ are assumed to have types $\mathsf{bool} \to \mathsf{bool}$ and $\mathsf{int} \to \mathsf{int}$, respectively. Can we build an HM(X) type derivation for it, under the constraint true and the initial environment Γ_0? To begin, it is straightforward to derive that the record $\{\ell = \mathtt{not};\mathtt{succ}\}$ has type $''\ (\ell : \mathsf{bool} \to \mathsf{bool}\ ;\ \partial(\mathsf{int} \to \mathsf{int}))$ **(1)**. In order to use apply, however, we must prove that this record has a type of the form $''\ (\mathtt{R}_1 \to \mathtt{R}_2)$, where $\mathtt{R}_1$ and $\mathtt{R}_2$ are rows. This is where C-ROW-GD and C-ROW-GL (Figure 1-17) come into play. Indeed, by C-ROW-GD, the type $\partial(\mathsf{int} \to \mathsf{int})$ may be written $\partial\mathsf{int} \to \partial\mathsf{int}$. So, (1) may be written $''\ (\ell : \mathsf{bool} \to \mathsf{bool}\ ;\ \partial\mathsf{int} \to \partial\mathsf{int})$ **(2)**, which by C-ROW-GL may be written $''\ ((\ell : \mathsf{bool}\ ;\ \partial\mathsf{int}) \to (\ell : \mathsf{bool}\ ;\ \partial\mathsf{int}))$ **(3)**. Thus, HMX-SUB allows deriving that the record $\{\ell = \mathtt{not};\mathtt{succ}\}$ has type (3). We let the reader continue and conclude that the program has type $''\ (\ell : \mathsf{bool}\ ;\ \partial\mathsf{int})$ under the constraint true and the initial environment Γ_0.

This example illustrates a very important use of rows, namely to *lift* an operation on ordinary values so as to turn it into a *pointwise* operation on records. Here, we have chosen to lift the standard application combinator, giving rise to apply on records. The point is that, thanks to the expressive power of rows, we were also able to lift the standard combinator's *type scheme* in the most straightforward manner, giving rise to a suitable type scheme for apply. □

1.11.30 EXERCISE [★★★]: Check that the definitions of Example 1.11.29 meet the requirements of Definition 1.7.7. □

The previous examples have illustrated the use of rows to typecheck operations on full records. Let us now move to records with finite domain. As explained in §1.11.1, they may be either encoded in terms of full records, or given a direct definition. The latter approach is illustrated below.

1.11.31 EXAMPLE [FINITE RECORDS]: For every set of labels L of cardinal n, let us introduce a n-ary constructor $\langle\rangle_L$. We define the notations $\langle \ell_1 = t_1; \ldots; \ell_n = t_n \rangle$ and $\langle V \rangle$, where V is a finite mapping of labels to values, in a manner similar to that of Example 1.11.26.

The three primitive operations on finite records, namely the empty record $\langle\rangle$, extension $\langle \cdot \text{ with } \ell = \cdot \rangle$, and access $\cdot.\langle\ell\rangle$, may be defined as follows. First, the empty record $\langle\rangle$ is precisely the nullary constructor $\langle\rangle_\varnothing$. Second, for every $\ell \in \mathcal{L}$, let extension $\langle \cdot \text{ with } \ell = \cdot \rangle$ and access $\cdot.\langle\ell\rangle$ be destructors of arity 1 and 2, respectively, equipped with the following reduction rules:

$$\begin{array}{lcll} \langle \langle V \rangle \text{ with } \ell = v \rangle & \xrightarrow{\delta} & \langle V[\ell \mapsto v] \rangle & \text{(R-EXTEND)} \\ \langle V \rangle.\langle \ell \rangle & \xrightarrow{\delta} & V(\ell) \qquad (\ell \in dom(V)) & \text{(R-ACCESS)} \end{array}$$

Let us now define the syntax of types as in Example 1.11.8. Let the initial environment Γ_0 contain the following bindings:

$$\begin{array}{rl} \langle\rangle_{\{\ell_1,\ldots,\ell_n\}} : & \forall X_1 \ldots X_n.X_1 \to \ldots \to X_n \to \tilde{}\,(\ell_1 : \text{pre } X_1; \ldots; \ell_n : \text{pre } X_n; \partial\text{abs}) \\ & \text{where } \ell_1 < \ldots < \ell_n \\ \langle \cdot \text{ with } \ell = \cdot \rangle : & \forall XX'Y.\tilde{}\,(\ell : X ; Y) \to X' \to \tilde{}\,(\ell : \text{pre } X' ; Y) \\ \cdot.\langle\ell\rangle : & \forall XY.\tilde{}\,(\ell : \text{pre } X ; Y) \to X \end{array}$$

Please note that, in particular, the type scheme assigned to the empty record $\langle\rangle$ is $\tilde{}\,(\partial\text{abs})$. □

1.11.32 EXERCISE [RECOMMENDED, ★, ↛]: Reconstruct all of the missing kind information in the type schemes given in Example 1.11.31. □

1.11.33 EXERCISE [RECOMMENDED, ★★, ↛]: Define an encoding of finite records in terms of full records, along the lines of §1.11.1. Check that the principal type schemes associated, via the encoding, with the three operations on finite records are precisely those given in Example 1.11.31. □

1.11.34 EXERCISE [RECOMMENDED, ★]: The extension operation, as defined above, may either change the value of an existing field or create a new field, depending on whether the field ℓ is or isn't present in the input record. This flavor is known as *free* extension. Can you define a *strict* flavor of extension that is not applicable when the field ℓ already exists? Can you define (free and strict flavors of) a *restriction* operation that removes a field from a record? □

1.11.35 EXERCISE [RECOMMENDED, ★]: Explain why, when $\mathsf{pre} \leqslant \mathsf{abs}$ holds, subsumption allows a record with *more* fields to be supplied in a context where a record with *fewer* fields is expected. This phenomenon is often known as *width subtyping*. Explain why such is not the case when subtyping is interpreted as equality. □

1.11.36 EXERCISE [★★★]: Check that the definitions of Example 1.11.31 meet the requirements of Definition 1.7.7. □

1.11.37 EXAMPLE [REFINEMENT OF RECORD TYPES]: In an equality-only model, records with more fields cannot be used in place of records with fewer fields. However, this may be partially recovered by a small refinement of the structure of types. The presence of fields can actually be split form their types, thus enabling some polymorphism over the presence of fields while type of fields themselves remains fixed. Let $\diamond$ be a new basic kind. Let type constructors abs and pre be both of kind $\diamond$ and let $\cdot$ be a new type constructor of kind $\diamond \otimes \star \Rightarrow \circ$. Let Γ_0 contain the following typing assumptions:

$$\begin{array}{rl} \langle\rangle : & \forall \mathtt{X}.\tilde{}(\partial(\mathsf{abs} \cdot \mathtt{X})) \\ \langle \cdot \text{ with } \ell = \cdot \rangle : & \forall \mathtt{Z}\mathtt{X}\mathtt{X}'\mathtt{Y}.\ \tilde{}(\ell : \mathtt{X}\ ;\ \mathtt{Y}) \to \mathtt{X}' \to \tilde{}(\ell : \mathtt{Z} \cdot \mathtt{X}'\ ;\ \mathtt{Y}) \\ \cdot.\langle \ell \rangle : & \forall \mathtt{X}\mathtt{Y}.\ \tilde{}(\ell : \mathsf{pre} \cdot \mathtt{X}\ ;\ \mathtt{Y}) \to \mathtt{X} \end{array}$$

The semantics of records remain unchanged. The new signature strictly generalizes the previous one (strictly more programs can be typed) while preserving type soundness. Here is a program that can now be typed and that could not be typed before:

$$(\mathtt{if}\ \mathtt{a}\ \mathtt{then}\ \langle\langle\rangle \text{ with } \ell' = \mathtt{true}\rangle \text{ with } \ell = 1\rangle\ \mathtt{else}\ \langle\langle\rangle \text{ with } \ell = 2\rangle).\langle\ell\rangle$$

Notice however, that when a present field is forgotten, the type of the field is not. Therefore two records defining the same field but with incompatible types can still not be mixed, which is possible in the subtyping model. □

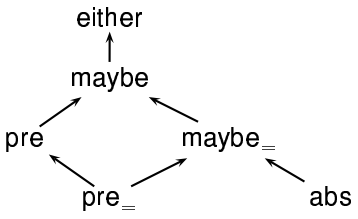
1.11.38 EXAMPLE [REFINED SUBTYPING]: The previous refinement for an equality-only model is not much interesting in the case of a subtyping model.

The subtyping assumption $\mathsf{pre} \leqslant \mathsf{abs}$ makes abs play the role of $\top$ for fields. That is, abs encodes the absence of information and not the information of absence. In other words, a value whose field ℓ has type abs may either be undefined or defined on field ℓ; in the latter case, the fact that field ℓ is actually defined has just been forgotten. Thus, types only provides a lower approximation of the actual domain of records. This is a lost of accuracy by comparison with the equality-only model, where a record domain is known from its type. As a result, some optimizations in the representation of records that are only possible when the exact domain of a record is statically known are lost.

Fortunately, there is a way to recover such accuracy. A conservative solution could of course to drop the inequality $\mathsf{pre} \leqslant \mathsf{abs}$. Notice that this would still be more expressive than using an equality model since, for instance $\tilde{}(\ell : \mathsf{pre}\ \ (\mathtt{T}_1 \to \mathtt{T}_2)\ ;\ \mathtt{T}) \leq \tilde{}(\ell : \mathsf{pre}\ \ \top\ ;\ \mathtt{T})$ would still hold, as long as $\to \leq \top$ does hold. This solution is known as depth-only subtyping for records, while the previous one provided both depth and width record subtyping. Conversely, one could also keep width subtyping and disallow depth subtyping, by preserving the relation $\mathsf{pre} \leqslant \mathsf{abs}$ while requiring pre to be invariant; in this case, presence of fields can be forgotten as a whole, but the types of fields cannot be weakened as long as fields remain visible.

Another more interesting solution consists in introducing another type constructor either of signature $\circ$ and assuming that $\mathsf{pre} \leqslant \mathsf{either}$ and $\mathsf{abs} \leqslant \mathsf{either}$ (but $\mathsf{pre} \not\leqslant \mathsf{abs}$). Here, either plays the role of $\top$ for fields and means either *present* (and forgotten) or *absent*. while abs really means *absent*. The accuracy of typechecking can be formally stated as the fact that a record value of type $\tilde{}(\ell : \mathsf{abs}\ ;\ \mathtt{T})$ cannot define field ℓ. □

1.11.39 EXAMPLE [MIXED SUBTYPING]: It is tempting to mix all variations of Example 1.11.38 together. As a first attempt, we may assume that the basic signature $\mathcal{S}_0$ contains covariant type constructors pre and maybe and invariant type constructors $\mathsf{pre}_=$ and $\mathsf{maybe}_=$, all of kind $\star \Rightarrow \circ$ and two type constructors abs and either of kind $\circ$, and that the subtype ordering $\leqslant$ is defined by the following diagram:



Intuitively, we wish that $\mathsf{pre}_=$ and $\mathsf{maybe}_=$ be *logically* invariant, pre and maybe be *logically* covariant, and the equivalences $\mathsf{pre}_= \mathtt{T} \leq \mathsf{maybe}_= \mathtt{T}' \equiv \mathtt{T} = \mathtt{T}'$ and

$$\mathsf{pre}_= \mathtt{T} \leq \mathsf{pre}\ \mathtt{T}' \equiv \mathsf{pre}\ \mathtt{T} \leq \mathsf{maybe}\ \mathtt{T}' \equiv \mathsf{maybe}_= \mathtt{T} \leq \mathsf{maybe}\ \mathtt{T}' \equiv \mathtt{T} \leq \mathtt{T}' \tag{1}$$

simultaneously hold. However, (1) requires, for instance, type constructors $\mathsf{pre}_=$ and pre to have the same direction, which is not currently possible since they do not have the same variance. Interestingly, this restriction may be relaxed by assigning variances of directions on a per type constructor basis and define structural subtyping accordingly (See Exercise 1.11.40). Then, replacing all occurrences of pre by $\mathsf{pre}_=$ in Γ_0 preserves type soundness and allows for both accurate record types and flexible subtyping in the same setting. □

1.11.40 EXERCISE [RELAXED VARIANCES, ★★★, ↛]: Let $\varnothing$ be allowed as a new variance, let extend the composition of variances defined in Example 1.3.9 with $\nu\varnothing = \varnothing$, and let $\leqslant^{\varnothing}$ stands for the full relation on type constructors. Let each type constructor F of signature $\mathsf{K} \Rightarrow \kappa$ now come with a mapping $\vartheta(\mathsf{F})$ from $dom(\mathsf{K})$ to variances. Let $\vartheta(\mathtt{t}, \mathtt{t}', \pi)$ be the variance of two ground types $\mathtt{t}$ and $\mathtt{t}'$ at a path π recursively defined by $\vartheta(\mathtt{t}, \mathtt{t}', \mathtt{d} \cdot \pi) = \big(\vartheta(\mathtt{t}(\epsilon))(\mathtt{d}) \cap \vartheta(\mathtt{t}'(\epsilon))(\mathtt{d})\big)\ \vartheta(\mathtt{t}/\mathtt{d}, \mathtt{t}'/\mathtt{d}, \pi)$ and $\vartheta(\mathtt{t}, \mathtt{t}', \epsilon) = +$. Then define the interpretation of subtyping as follows: if $\mathtt{t}, \mathtt{t}' \in \mathcal{M}_\kappa$, let $\mathtt{t} \leq \mathtt{t}'$ hold if and only if for all path $\pi \in dom(\mathtt{t}) \cap dom(\mathtt{t}')$, $\mathtt{t}(\pi) \leqslant^{\vartheta(\mathtt{t}, \mathtt{t}', \pi)} \mathtt{t}'(\pi)$ holds.

Check that the relation $\leq$ remains a partial ordering. Check that a type constructor whose direction $\mathtt{d}$ has been syntactically declared covariant (respectively contravariant, invariant) is still logically covariant (respectively contravariant, invariant) in $\mathtt{d}$. □

1.11.8 Record concatenation

Record concatenation takes two records and combines them into a new record whose fields are taken from whatever argument defines them. Of course, there is an ambiguity when the two records do not have disjoint domains and a choice should be made to disambiguate such cases. *Symmetric* concatenation let concatenation be undefined in this case (Harper and Pierce, 1991), while *asymmetric* concatenation let one-side (usually the right side) always take priority. For instance, concatenation with a right-priority semantics may be described by the following reduction rules, where @ of arity 2 stands for concatenation.

$$\mathtt{w} \mathbin{@} \langle \mathtt{w} \ \mathsf{with}\ \ell = \mathtt{v} \rangle \xrightarrow{\delta} \langle \mathtt{w} \mathbin{@} \mathtt{w}' \ \mathsf{with}\ \ell = \mathtt{v} \rangle \qquad \text{(ER-CONCAT-R)}$$

$$\mathtt{w} \mathbin{@} \langle\rangle \xrightarrow{\delta} \mathtt{w} \qquad \text{(ER-CONCAT-NIL-R)}$$

$$\langle\rangle \mathbin{@} \mathtt{w} \xrightarrow{\delta} \mathtt{w} \qquad \text{(ER-CONCAT-NIL-L)}$$

Despite a rather simple semantics, record concatenation remains hard to type (with either a strict or a priority semantics). To tackle this difficulty, let us consider an indirect semantics via encoding into full records. Moreover, we may consider records with a single field, or even simpler restrict our attention to field concatenation. That is, fields are of either form $\mathsf{pre}\ \mathtt{v}$ (defined with value v) or abs (undefined) and the concatenation primitive @ on fields is given by the two following reduction rules:

$$\mathtt{v} \mathbin{@} \mathsf{pre}\ \mathtt{w} \xrightarrow{\delta} \mathsf{pre}\ \mathtt{w} \qquad \text{and} \qquad \mathtt{v} \mathbin{@} \mathsf{abs} \xrightarrow{\delta} \mathtt{v}$$

Each rule suggest a different typing for @.

$$\forall \mathtt{XY}.\mathtt{X} \to \mathsf{pre}\ \mathtt{Y} \to \mathsf{pre}\ \mathtt{Y} \qquad \text{or} \qquad \forall \mathtt{X}.\mathtt{X} \to \mathsf{abs} \to \mathtt{X}$$

Indeed, both types are sound for concatenation. That is, both types should be instances of the typing assumption for @. Unfortunately, both types cannot be instances of a same sound type scheme for concatenation (the smallest type scheme that generalizes both of them being $\forall \mathtt{XX'X''}.\mathtt{X} \to \mathtt{X'} \to \mathtt{X''}$). Notice that strict concatenation is not any easier, and would lead to the following two cases: $\forall \mathtt{Y}.\mathsf{abs} \to \mathsf{pre}\ \mathtt{Y} \to \mathsf{pre}\ \mathtt{Y}$ and $\forall \mathtt{Y}.\mathsf{pre}\ \mathtt{Y} \to \mathsf{abs} \to \mathsf{pre}\ \mathtt{Y}$. A simple solution would be to replace type schemes with sets—or equivalently conjunctions—of type schemes. Then, the conjunction of the two types schemes above would be a good typing assumption for field concatenation (Wand, 1989). Such an extension should raise no theoretical problem, but serious practical issues: a naïve solver may in general require exponential time and produce solutions exponential in the size of the input. However, a restricted form of disjunction could also be considered and may lead to better performance, for instance by delaying resolution of constraints much as can be done for conditional constraints (Pottier, 2000).

Indeed, in the particular case of concatenation, disjunctions can be factored out with conditional type constraints as described in the example 1.3.10. Conditional constraints must now be interpreted in the row model as follows: If $t_0 \in \mathcal{M}_{\kappa.s}$ and $\mathtt{T}_1, \mathtt{T}_2 \in \mathcal{M}_{\kappa'.s'}$, then $G^s \leqslant \mathtt{T}_0 \Rightarrow \mathtt{T}_1 \leq \mathtt{T}_2$ holds, if and only if (i) $s = \mathit{Type}$ and $F = t_0(\epsilon)$ implies $t_1 = t_2$ or (ii) $s = \mathit{Row}(L)$ and for all $\ell \in \mathcal{L} \setminus L$, $F = t_0(\ell)$ implies $t_1/\ell = t_2/\ell$.

1.11.41 EXERCISE [RECOMMENDED, ★★★★, ↛]: Give an algorithm to solve conditional constraints and prove its correctness. □

With conditional constraints field concatenation can be given the unique type scheme:

$$\forall \mathtt{X}_1\mathtt{X}_2\mathtt{X}[\mathsf{pre} \leqslant \mathtt{X}_2 \Rightarrow \mathtt{X}_2 \leq \mathtt{X} \wedge \mathsf{abs} \leqslant \mathtt{X}_2 \Rightarrow \mathtt{X}_1 \leq \mathtt{X}].\mathtt{X}_1 \to \mathtt{X}_2 \to \mathtt{X}$$

Lifting field concatenation to record concatenation, we get following principal type scheme for record concatenation:

$$\forall \mathtt{X}_1\mathtt{X}_2\mathtt{X}[\mathsf{pre} \leqslant \mathtt{X}_2 \Rightarrow \mathtt{X}_2 \leq \mathtt{X} \wedge \mathsf{abs} \leqslant \mathtt{X}_2 \Rightarrow \mathtt{X}_1 \leq \mathtt{X}].''(\mathtt{X}_1) \to ''(\mathtt{X}_2) \to ''(\mathtt{X})$$

Notice that the conditional constraints above applies to rows of kind $\mathit{Row}(\varnothing)$: it will be distributed to fields by the solver (to be defined) as the row variable $\mathtt{X}_2$ gets instantiated.

1.11.42 EXERCISE [RECOMMENDED, ★★★★, ↛]: Prove type soundness for record concatenation with the conditional constraint typing given above. □

1.11.43 EXAMPLE: A solution to typing record concatenation without uses is based on a wrapper interpretation of records: a record field can be seen as a function that given a field value $\mathtt{v}$ as argument returns the field concatenation of $\mathtt{v}$ with itself. Using the map primitive of Exercise **??**, let the new interpretation of records be defined by:

$$\begin{aligned}
\langle\!\langle\rangle\!\rangle &\stackrel{\text{def}}{=} \{\lambda \mathtt{r}.\mathtt{r}\} \\
&\quad : \forall \mathtt{X}.''(\mathtt{X} \to \mathtt{X}) \\
\langle\!\langle \cdot \text{ with } \ell = \cdot \rangle\!\rangle &\stackrel{\text{def}}{=} \lambda \mathtt{z}.\lambda \mathtt{z}'.\{\mathtt{z} \text{ with } \ell = \lambda \mathtt{r}.\mathsf{pre}\ \mathtt{z}'\} \\
&\quad : \forall \mathtt{X}\mathtt{X}'\mathtt{Y}\mathtt{Y}'.''(\ell : \mathtt{Y}' ; \mathtt{Y}) \to \mathtt{X} \to ''(\ell : \mathsf{pre}\ (\mathtt{X}' \to \mathtt{X}) ; \mathtt{Y}) \\
\cdot.\langle\!\langle \ell \rangle\!\rangle &\stackrel{\text{def}}{=} \lambda \mathtt{z}.(\ell.\{(\}\mathtt{z})) \ \mathsf{abs} \\
&\quad : \forall \mathtt{Y}\mathtt{X}\mathtt{X}'.''(\ell : \mathsf{abs} \to \mathsf{pre}\ \mathtt{Y} ; \mathtt{X} \to \mathtt{X}') \to \mathtt{Y} \\
@ &\stackrel{\text{def}}{=} \mathsf{map}\ (\lambda \mathtt{z}.\lambda \mathtt{z}'.\lambda \mathtt{r}.\mathtt{z}\ (\mathtt{z}'\ \mathtt{r})) \\
&\quad : \forall \mathtt{X}\mathtt{X}'\mathtt{X}''.''(\mathtt{X} \to \mathtt{X}') \to ''(\mathtt{X}' \to \mathtt{X}'') \to ''(\mathtt{X} \to \mathtt{X}'')
\end{aligned}$$

Intuitively, the new interpretation models the old interpretation in the sense that any record value $\mathtt{v}$ obtained of the new interpretation can be turned into a record value for the old interpretation by reducing $\mathsf{map}\ \mathtt{v}\ \{\mathsf{abs}\}$. However, rather than proving such a property, it is simpler to take the types above as typing assumptions for extensible records and verify the requirements of Lemma 1.7.7. □

1.11.9 Polymorphic variants

So far, we have emphasized the use of rows for flexible typechecking of operations on records. The record type constructor $''$ expects one parameter, which is a row: informally speaking, one might say that it is a *product* constructor of infinite arity. It appears natural to also define *sums* of infinite arity. This may be done by introducing a new unary type constructor $^\circ$, whose parameter is a row.

As in the case of records, we use a nullary type constructor abs and a unary type constructor pre in order to associate information with every row label. Thus, for instance, the type $^\circ\,(\ell_1 : \mathsf{pre}\ \mathtt{T}_1\ ;\ \ell_2 : \mathsf{pre}\ \mathtt{T}_2\ ;\ \partial\mathsf{abs})$ is intended to contain values of the form $\ell_1\ \mathtt{v}_1$, where $\mathtt{v}_1$ has type $\mathtt{T}_1$, or of the form $\ell_2\ \mathtt{v}_2$, where $\mathtt{v}_2$ has type $\mathtt{T}_2$. The type constructors abs and pre are *not* the same type constructors as in the case of records. In particular, their subtyping relationship, if there is one, is reversed. Indeed, the type $^\circ\,(\ell_1 : \mathsf{pre}\ \mathtt{T}_1\ ;\ \ell_2 : \mathsf{abs}\ ;\ \partial\mathsf{abs})$ is intended to contain only values of the form $\ell_1\ \mathtt{v}_1$, where $\mathtt{v}_1$ has type $\mathtt{T}_1$, so it is safe to make it a subtype of the above type: in other words, it is safe to allow $\mathsf{abs} \leq \mathsf{pre}\ \mathtt{T}_2$. In spite of this, we keep the names abs and pre by tradition.

The advantages of this approach over algebraic data types are the same as in the case of records. The namespace of data constructors becomes global, so it becomes possible for two distinct sum types to share data constructors. Also, the expressiveness afforded by rows allows assigning types to new operations, such as *filtering* (see below), which allows functions that perform case analysis to be incrementally extended with new cases. One disadvantage is that it becomes more difficult to understand what it means for a function defined by pattern matching to be *exhaustive*; this issue is, however, out of the scope of this chapter.

1.11.44 EXAMPLE [POLYMORPHIC VARIANTS]: For every label $\ell \in \mathcal{L}$, let us introduce a unary constructor ℓ and a ternary destructor $[\,\ell : \cdot \mid \cdot\,]\,\cdot$. We refer to the former as a *data constructor*, and to the latter as a *filter*. Let us also introduce a unary destructor $[]$. We equip these destructors with the following reduction rules:

$$[\,\ell : \mathtt{v} \mid \mathtt{v}'\,]\ (\ell\ \mathtt{w}) \xrightarrow{\delta} \mathtt{v}\ \mathtt{w} \qquad \text{(R-FILTER-1)}$$

$$[\,\ell : \mathtt{v} \mid \mathtt{v}'\,]\ (\ell'\ \mathtt{w}) \xrightarrow{\delta} \mathtt{v}'\ (\ell'\ \mathtt{w}) \qquad \text{if } \ell \neq \ell' \qquad \text{(R-FILTER-2)}$$

Let us define the syntax of types as follows. Let there be two basic kinds $\star$ and $\bullet$. Let $\mathcal{S}_0$ consist of the type constructors $\to$, abs, and pre, whose respective signatures are $\star \otimes \star \Rightarrow \star$, $\bullet$, and $\star \Rightarrow \bullet$. Let $\mathcal{S}_1$ consist of the record type constructor $^\circ$, whose signature is $\bullet \Rightarrow \star$. Please note the similarity with the case of records (Example 1.11.8).

Subtyping is typically interpreted in one of two ways. One is equality. The other is the nonstructural subtyping order obtained by letting $\to$ be contravariant in its domain and covariant in its codomain, $^\circ$ be covariant, $\to$ and $^\circ$ be incompatible, and letting $\mathsf{abs} \leqslant \mathsf{pre}$. Please compare this definition with the case of records (Remark 1.11.11).

To complete the setup, let the initial environment Γ_0 contain the following

bindings:

$$\begin{array}{rl} \ell \cdot : & \forall \mathtt{XY}.\mathtt{X} \to {}^{\circ}\,(\ell : \mathsf{pre}\ \mathtt{X}\ ;\ \mathtt{Y}) \\ [\ell : \cdot \mid \cdot] \cdot : & \forall \mathtt{XX'YY'}.(\mathtt{X} \to \mathtt{Y}) \to ({}^{\circ}\,(\ell : \mathtt{X'}\ ;\ \mathtt{Y'}) \to \mathtt{Y}) \to {}^{\circ}\,(\ell : \mathsf{pre}\ \mathtt{X}\ ;\ \mathtt{Y'}) \to \mathtt{Y} \\ [] : & \forall \mathtt{X}.{}^{\circ}\,(\partial\mathsf{abs}) \to \mathtt{X} \end{array}$$

The first binding means, in particular, that if $\mathtt{v}$ has type $\mathtt{T}$, then a value of the form $\ell\ \mathtt{v}$ has type ${}^{\circ}\,(\ell : \mathsf{pre}\ \mathtt{T}\ ;\ \partial\mathsf{abs})$. This is a sum type with only one branch labelled ℓ, hence a very precise type for this value. However, it is possible to instantiate the row variable $\mathtt{Y}$ with rows other than $\partial\mathsf{abs}$. For instance, the value $\ell\ \mathtt{v}$ also has type ${}^{\circ}\,(\ell : \mathsf{pre}\ \mathtt{T}\ ;\ \ell' : \mathsf{pre}\ \mathtt{T'}\ ;\ \partial\mathsf{abs})$. This is a sum type with two branches, hence a somewhat less precise type, but still a valid one for this value. It is clear that, through this mechanism, the value $\ell\ \mathtt{v}$ admits an infinite number of types. The point is that, if $\mathtt{v}$ has type $\mathtt{T}$ and $\mathtt{v'}$ has type $\mathtt{T'}$, then both $\ell\ \mathtt{v}$ and $\ell'\ \mathtt{v'}$ have type ${}^{\circ}\,(\ell : \mathsf{pre}\ \mathtt{T}\ ;\ \ell' : \mathsf{pre}\ \mathtt{T'}\ ;\ \partial\mathsf{abs})$, so they may be stored together in a homogeneous data structure, such as a list.

Filters are used to perform case analysis on variants, that is, on values of a sum type. According to R-FILTER-1 and R-FILTER-2, a filter $[\ell : \mathtt{v} \mid \mathtt{v'}]$ is a function that expects an argument of the form $\ell'\ \mathtt{w}$ and reduces to $\mathtt{v}\ \mathtt{w}$ if ℓ' is ℓ and to $\mathtt{v'}\ (\ell'\ \mathtt{w})$ otherwise. Thus, a filter defines a two-way branch, where the label of the data constructor at hand determines which branch is taken. The expressive power of filters stems from the fact that they may be organized in a sequence, so as to define a multi-way branch. The inert filter $[]$, which does not have a reduction rule, serves as a terminator for such sequences. For instance, the composite filter $[\ell : \mathtt{v} \mid [\ell' : \mathtt{v'} \mid []]]$, which may be abbreviated as $[\ell : \mathtt{v} \mid \ell' : \mathtt{v'}]$, may be applied either to a value of the form $\ell\ \mathtt{w}$, yielding $\mathtt{v}\ \mathtt{w}$, or to a value of the form $\ell'\ \mathtt{w'}$, yielding $\mathtt{v'}\ \mathtt{w'}$. Applying it to a value $\mathtt{w}$ whose head symbol is not ℓ or ℓ' would lead to the term $[]\ \mathtt{w}$, which is stuck, since $[]$ does not have a reduction rule.

For the type system to be sound, we must ensure that every application of the form $[]\ \mathtt{w}$ is ill-typed. This is achieved by the third binding above: the domain type of $[]$ is ${}^{\circ}\,(\partial\mathsf{abs})$, a sum type with zero branches, which contains no values. The return type of $[]$ may be chosen at will, which is fine: since it can never be invoked, it can never return. The second binding above means that, if $\mathtt{v}$ accepts values of type $\mathtt{T}$ and $\mathtt{v'}$ accepts values of type ${}^{\circ}\,(\ell : \mathtt{T''}\ ;\ \mathtt{T'})$, then the filter $[\ell : \mathtt{v} \mid \mathtt{v'}]$ accepts values of type ${}^{\circ}\,(\ell : \mathsf{pre}\ \mathtt{T}\ ;\ \mathtt{T'})$. Please note that any choice of $\mathtt{T''}$ will do, including, in particular, abs. In other words, it is okay if $\mathtt{v'}$ does not accept values of the form $\ell\ \mathtt{w}$. Indeed, by definition of the semantics of filters, it will never be passed such a value. □

1.11.45 EXERCISE [★★★, ↛]: Check that the definitions of Example 1.11.44 meet the requirements of Definition 1.7.7. □

1.11.46 REMARK: It is interesting to study the similarity between the type schemes assigned to the primitive operations on polymorphic variants and those assigned to the primitive operations on records (Example 1.11.31). The type of [] involves the complete row $\partial\mathsf{abs}$, just like the empty record $\langle\rangle$. The type of $[\ell : \cdot \mid \cdot] \cdot$ is pretty much identical to the type of record extension $\langle \cdot \text{ with } \ell = \cdot \rangle$, provided the three continuation arrows $\to \mathtt{Y}$ are dropped. Last, the type of the data constructor ℓ is strongly reminiscent of the type of record access $\cdot.\langle\ell\rangle$. With some thought, this is hardly a surprise. Indeed, records and variants are *dual*: it is possible to encode the latter in terms of the former and vice-versa. For instance, in the encoding of variants in terms of records, a function defined by cases is encoded as a record of ordinary functions, in continuation-passing style. Thus, the encoding of [] is $\lambda\mathtt{f}.\mathtt{f}\ \langle\rangle$, the encoding of $[\ell : \mathtt{v} \mid \mathtt{v}']$ is $\lambda\mathtt{f}.\mathtt{f}\ \langle \mathtt{v}' \text{ with } \ell = \mathtt{v}\rangle$, and the encoding of $\ell\ \mathtt{v}$ is $\lambda\mathtt{r}.\mathtt{r}.\langle\ell\rangle\ \mathtt{v}$. The reader is encouraged to study the type schemes that arise out of this encoding and how they relate to the type schemes given in Example 1.11.44. □

1.11.47 EXAMPLE [FIRST-CLASS MESSAGES]: In a programming language equipped with both records and variants, it is possible to make the duality between these two forms of data explicit by extending the language with a primitive operation # that turns a record of ordinary functions into a single function, defined by cases. More precisely, # may be introduced as a binary destructor, whose reduction rule is

$$\#\ \mathtt{v}\ (\ell\ \mathtt{w}) \quad \xrightarrow{\delta} \quad \mathtt{v}.\langle\ell\rangle\ \mathtt{w} \qquad \text{(R-SEND)}$$

What type may we assign to such an operation? In order to simplify the answer, let us assume that we are dealing with full records (Example 1.11.26) and full variants; that is, we have a single basic kind $\star$, and do not employ abs and pre. Then, a suitable type scheme would be

$$\forall \mathtt{X}\mathtt{Y}.\text{''}\ (\mathtt{X} \to \partial\mathtt{Y}) \to {}^{\circ}\,\mathtt{X} \to \mathtt{Y}$$

In other words, this operation accepts a record of functions, all of which have the same return type Y, but may have arbitrary domain types, which are given by the row X. It produces a function that accepts a parameter of sum type $^{\circ}$ X and returns a result of type Y. The fact that the row X appears both in the $^{\circ}$ type and in the '' type reflects the operational semantics. Indeed, according to R-SEND, the label ℓ carried by the value ℓ w is used to extract, out of the record v, a function, which is then applied to w. Thus, the domain type of the function stored at ℓ within the record v should match the type of w. In other words, at every label, the domain of the contents of the record and the

contents of the sum should be type compatible. This is encoded by letting a single row variable $\mathtt{X}$ stand for both of these rows. Please note that the arrow in $\mathtt{X} \to \partial\mathtt{Y}$ is really $\to^{Row(\varnothing)}$: once again, we are exploiting the presence of type constructors of the form G^{s}, with $\mathsf{s} \neq \mathit{Type}$, in the signature $\mathcal{S}$.

If the record of functions $\mathtt{v}$ is viewed as an *object*, and if the variant $\ell\ \mathtt{w}$ is viewed as a *message* ℓ carrying a parameter $\mathtt{w}$, then R-SEND may be understood as *(first-class) message dispatch*, a common feature of object-oriented languages. (The *first-class* qualifier refers to the fact that the message name ℓ is not statically fixed, but is discovered at runtime.) The issue of type inference in the presence of such a feature has been studied in (Nishimura, 1998; Müller and Nishimura, 1998; Pottier, 2000). These papers address two issues that are not dealt with in the above example, namely (i) accommodating finite (as opposed to full) record and variants and (ii) allowing distinct methods to have distinct result types. This is achieved via the use of subtyping and of some form of conditional constraints. □

1.11.48 EXERCISE [★★★, ↛]: Check that the definitions of Example 1.11.47 meet the requirements of Definition 1.7.7. □

The name *polymorphic variants* stems from the highly polymorphic type schemes assigned to the operations on variants (Example 1.11.44). A row-based type system for polymorphic variants was first proposed by Rémy (1989). A somewhat similar, constraint-based type system for polymorphic variants was then studied by Garrigue (1998; 2000; 2002) and implemented by him as part of the programming language Objective Caml.

1.11.10 Other applications of rows

Typechecking records and variants is the best-known application of rows. Many variations of it are conceivable, some of which we have illustrated, such as the choice between *full* and *finite* records and variants. However, rows may also be put to other uses, of which we now list a few.

First, since objects may be viewed as records of functions, at least from a typechecking point of view, rows may be used to typecheck object-oriented languages in a structural style (Wand, 1994; Rémy, 1994). This is, in particular, the route followed in Objective Caml (Rémy and Vouillon, 1998). There, an object type consists of a row of method types, and gives the object's interface. Such a style is considered *structural*, as opposed to the style adopted by many popular object-oriented languages, such as C++, Java, and C#, where an object type consists of the *name* of its class. Thanks to rows, method invocation may be assigned a polymorphic type scheme, similar to that of record

access (Example 1.11.31), making it possible to invoke a specific method (say, ℓ) without knowing which class the receiver object belongs to.

Rows may also be used to encode set of properties within types or to encode type refinements, with applications in type-based program analysis. Some instances worth mentioning are soft typing (Cartwright and Fagan, 1991; Wright and Cartwright, 1994), exception analysis (Leroy and Pessaux, 2000; Pottier and Simonet, 2003), and static enforcement of an access control policy (Pottier, Skalka, and Smith, 2001). BANE (Fähndrich, 1999), a versatile program analysis toolkit, also implements a form of rows.

The key to rows is to decompose the set of row labels into a class of finite partitions that is closed by some operations. Here, those partitions are composed of singleton labels and cofinite sets of labels; the operations are merging (or conversely splitting) a singleton label and a cofinite set of labels. Other decompositions are possible, for instance, one could imagine to consider labels in a two-dimensional space. More generally, labels might also be given internal structure, for instance, one might consider automatons as labels. Notice also that record types are stratified, since rows, that is, expressions of kind $\mathit{Row}(\mathrm{L})$, may not themselves contain records —constructors of $\mathcal{S}_1$ are only given the image row kind *Type*. This restriction can be partially relaxed leading to rows of increasing degrees (Rémy, 1992b) ... and complexity! Yet more intriguing are type-indexed rows where labels are suppressed (Shields and Meijer, 2001).

1.11.11 Variations on rows

A type system may be said to have *rows*, in a broad sense, if mappings from labels to types may be (i) defined incrementally, via some syntax for extending an existing mapping with information about a new label and (ii) abstracted by a type variable. In this chapter, which follows Rémy's ideas (1993a; 1992a; 1992b), the former feature is provided by the row constructors $(\ell : \cdot \,;\, \cdot)$, while the latter is provided by the existence of row variables, that is, type variables of row kind $\mathit{Row}(\mathrm{L})$ for some L. There are, however, type systems that provide (i) and (ii) while departing significantly from the one presented here. These systems differ mainly in how they settle some important design choices:

1. does a row denote a *finite* or *infinite* mapping from labels to types?
2. is a row with duplicate labels considered well-formed? if not, by which mechanism is it ruled out?

In Rémy's approach, every row denotes an infinite (in fact, cofinite) mapping from labels to types. The type constructors abs and pre are used to encode

domain information within field types. A row with duplicate labels, such as $(\ell : \mathtt{T}_1 \ ; \ \ell : \mathtt{T}_2 \ ; \ \mathtt{T}_3)$, is ruled out by the kind system. Below, we mention a number of type systems that make different design choices.

The first use of rows for typechecking operations on records, including record extension, is due to Wand (1987; 1988). In Wand's approach, rows denote finite mappings. Furthermore, rows with duplicate labels are considered legal: row extension is interpreted as function extension, so that, if a label occurs twice, the later occurrence takes precedence. This leads to a difficulty in the constraint solving process: the constraint $(\ell : \mathtt{T}_1 \ ; \ \mathtt{R}_1) = (\ell : \mathtt{T}_2 \ ; \ \mathtt{R}_2)$ entails $\mathtt{T}_1 = \mathtt{T}_2$, but does *not* entail $\mathtt{R}_1 = \mathtt{R}_2$, because $\mathtt{R}_1$ and $\mathtt{R}_2$ may have different domains—indeed, their domains may differ at ℓ. Wand's proposed solution (1988) introduces a four-way disjunction, because each of $\mathtt{R}_1$ and $\mathtt{R}_2$ may or may not define ℓ. This gives type inference exponential time complexity.

Later work (Berthomieu, 1993; Berthomieu and le Moniès de Sagazan, 1995) interprets rows as *infinite* mappings, but sticks with Wand's interpretation of row extension as function extension, so that duplicate labels are allowed. The constraint solving algorithm rewrites the problematic constraint $(\ell : \mathtt{T}_1 \ ; \ \mathtt{R}_1) = (\ell : \mathtt{T}_2 \ ; \mathtt{R}_2)$ to $(\mathtt{T}_1 = \mathtt{T}_2) \wedge (\mathtt{R}_1 =_{\{\ell\}} \mathtt{R}_2)$, where the new predicate $=_{\mathtt{L}}$ is interpreted as row equality *outside* L. Of course, the entire constraint solver must then be extended to deal with constraints of the form $\mathtt{T}_1 =_{\mathtt{L}} \mathtt{T}_2$. The advantage of this approach over Wand's lies in the fact that no disjunctions are ever introduced, so that the time complexity of constraint solving apparently remains polynomial.

Several other works make opposite choices, by sticking with Wand's interpretation of rows as finite mappings, but forbidding duplicate labels. No kind discipline is imposed: some other mechanism is used to ensure that duplicate labels do not arise. In (Jategaonkar and Mitchell, 1988; Jategaonkar, 1989), somewhat *ad hoc* steps are taken to ensure that, if the row $(\ell : \mathtt{T} \ ; \ \mathtt{X})$ appears anywhere within a type derivation, then $\mathtt{X}$ is never instantiated with a row that defines ℓ. In (Gaster and Jones, 1996; Gaster, 1998; Jones and Peyton Jones, 1999), explicit constraints prevent duplicate labels from arising. This line of work uses *qualified types* (Jones, 1994), a constraint-based type system that bears strong similarity with HM($\mathtt{X}$). For every label ℓ, a unary predicate $\cdot$ lacks ℓ is introduced: roughly speaking, the constraint $\mathtt{R}$ lacks ℓ is considered to hold if the (finite) row $\mathtt{R}$ does not define the label ℓ. The constrained type scheme assigned to record access is

$$\cdot.\langle \ell \rangle : \forall \mathtt{X}\mathtt{Y}[\mathtt{Y} \text{ lacks } \ell].\tilde{} \ (\ell : \mathtt{X} \ ; \ \mathtt{Y}) \to \mathtt{X}.$$

The constraint $\mathtt{Y}$ lacks ℓ ensures that the row $(\ell : \mathtt{X} \ ; \ \mathtt{Y})$ is well-formed. Although interesting, this approach is not as expressive as that described in

this chapter. For instance, although it accommodates record update (where the field being modified is known to exist in the initial record) and *strict* record extension (where the field is known *not* to initially exist), it cannot express a suitable type scheme for *free* record extension, where it is *not known* whether the field initially exists. This approach has been implemented as the "Trex" extension to Hugs.

It is worth mentioning a line of type systems (Ohori and Buneman, 1988, 1989; Ohori, 1995) that do *not* have rows, because they lack feature (i) above, but are still able to assign a polymorphic type scheme to record access. One might explain their approach as follows. First, these systems are equipped with ordinary, structural record types, of the form $\{\ell_1 : \mathtt{T}_1; \ldots; \ell_n : \mathtt{T}_n\}$. Second, for every label ℓ, a binary predicate $\cdot$ has $\ell : \cdot$ is available. The idea is that the constraint $\mathtt{T}$ has $\ell : \mathtt{T}'$ holds if and only if $\mathtt{T}$ is a record type that contains the field $\ell : \mathtt{T}'$. Then, record access may be assigned the constrained type scheme

$$\cdot.\langle\ell\rangle : \forall \mathtt{XY}[\mathtt{X} \text{ has } \ell : \mathtt{Y}].\mathtt{X} \to \mathtt{Y}.$$

This technique also accommodates a restricted form of record update, where the field being written must initially exist and must keep its initial type; it does not, however, accommodate any form of record extension, because of the absence of row extension in the syntax of types. Although the papers cited above employ different terminology, we believe it is fair to view them as constraint-based type systems. In fact, Odersky, Sulzmann, and Wehr (1999) prove that Ohori's system (1995) may be viewed as an instance of HM(X). Sulzmann (2000) proposes several extensions of it, also presented as instances of HM(X), which accommodate record extension and concatenation using new, *ad hoc* constraint forms in addition to $\cdot$ has ℓ.

In the *label-selective* λ*-calculus* (Garrigue and Aït-Kaci, 1994; Furuse and Garrigue, 1995), the arrow type constructor carries a label, and arrows that carry distinct labels may *commute*, so as to allow labeled function arguments to be supplied in any order. Some of the ideas that underlie this type system are closely related to rows.

Pottier (2003) describes an instance of HM(X) where rows are *not* part of the syntax of types: equivalent expressive power is obtained via an extension of the constraint language. The idea is to work with constraints of the form $\mathtt{R}_1 \leq_L \mathtt{R}_2$, where L may be finite or cofinite, and to interpret such a constraint as row subtyping *inside* L. The point of this approach is to allow formulating constraint solving as a *closure* process. In this alternate formulation, no new type variables need be allocated during constraint solving; contrast this with S-MUTE-LL, S-MUTE-GD, and S-MUTE-GL in Figure 1-19.

Even though rows were originally invented with type inference in mind, they are useful in explicitly typed languages as well: indeed, other approaches to typechecking operations on records appear quite complex (Cardelli and Mitchell, 1991).

PART II

Reasoning About Programs

2 *Logical Relations and a Case Study in Equivalence Checking*

By Karl Crary

A fundamental technique for proving certain sorts of properties of programming language is the technique of *logical relations*. Logical relations arise when a property is to be proven of all well-formed terms in the language, but that property is not preserved by one or more of the language's elimination forms. For example, one such property is termination: if a function expression t_{fun} and its argument t_{arg} both terminate, it does not follow that the application of t_{fun} to t_{arg} necessarily terminates.

In cases such as this, it is impossible to form a direct proof by induction on typing derivations that all well-formed terms enjoy the property in question. However, such a property may nevertheless be true; for example, normalization holds for all well-typed terms in the simply typed lambda-calculus (TAPL Chapter 12). Logical relations surmount this difficulty by proving a stronger property based on a term's type. In the example above, for t_{fun} one would show (informally speaking) not only that t_{fun} itself terminates, but also that any application of t_{fun} to a terminating argument also terminates.

The classic application of logical relations is to prove various sorts of termination properties, especially *strong normalization* (Tait, 1967). A very simple example of a logical relation argument is given in TAPL Chapter 12 to prove a simple termination result. However, the technique has wide applicability beyond just termination properties. In this chapter, we will develop the technique of logical relations via a case study in decision procedures for equivalence of terms. Then in the next chapter, we will exploit this technique in developing a powerful theory of typed operational reasoning.

Syntax

$t ::=$		*terms:*
	x	*variable*
	$\lambda x{:}T.t$	*abstraction*
	$t\ t$	*application*
	k	*constant*
$T ::=$		*types:*
	b	*base type*
	$T{\to}T$	*type of functions*
$\Gamma ::=$		*contexts:*
	$\emptyset$	*empty context*
	$\Gamma, x{:}T$	*term variable binding*

Typing $\boxed{\Gamma \vdash t : T}$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}$$

$$\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.t_2 : T_1{\to}T_2} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : T_{11}{\to}T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-APP)}$$

$$\Gamma \vdash k : b \quad \text{(T-CONST)}$$

Equivalence $\boxed{\Gamma \vdash s \equiv t : T}$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash t \equiv t : T} \quad \text{(Q-REFL)}$$

$$\frac{\Gamma \vdash t \equiv s : T}{\Gamma \vdash s \equiv t : T} \quad \text{(Q-SYMM)}$$

$$\frac{\Gamma \vdash s \equiv t : T \qquad \Gamma \vdash t \equiv u : T}{\Gamma \vdash s \equiv u : T} \quad \text{(Q-TRANS)}$$

$$\frac{\Gamma, x{:}T_1 \vdash s_2 \equiv t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.s_2 \equiv \lambda x{:}T_1.t_2 : T_1{\to}T_2} \quad \text{(Q-ABS)}$$

$$\frac{\Gamma \vdash s_1 \equiv t_1 : T_1{\to}T_2 \qquad \Gamma \vdash s_2 \equiv t_2 : T_1}{\Gamma \vdash s_1\ s_2 \equiv t_1\ t_2 : T_2} \quad \text{(Q-APP)}$$

$$\frac{\Gamma, x{:}T_1 \vdash s_{12} \equiv t_{12} : T_2 \qquad \Gamma \vdash s_2 \equiv t_2 : T_1}{\Gamma \vdash (\lambda x{:}T_1.s_{12})\ s_2 \equiv [x \mapsto t_2]t_{12} : T_2} \quad \text{(Q-BETA)}$$

$$\frac{\Gamma, x{:}T_1 \vdash s\ x \equiv t\ x : T_2}{\Gamma \vdash s \equiv t : T_1{\to}T_2} \quad \text{(Q-EXT)}$$

Figure 2-1: Simply typed lambda-calculus with a base type ($\lambda_{\to b}$)

2.1 The Equivalence Problem

The problem with which we concern ourselves is to determine whether or not two terms in the simply typed lambda calculus are equivalent. The system we will consider is formulated in Figure 2-1. In order that the problem not be trivial, the system includes a single base type b that is inhabited by an unspecified set of constants. The constants are ranged over by the metavariable k.

The equivalence judgement, which will be our key subject of concern, is written $\Gamma \vdash s \equiv t : T$, meaning that (in context Γ) the terms s and t are equivalent, when considered as members of the type T. (The alert reader may observe that the type T at which terms are compared does not play an important role in the rules in Figure 2-1; however, it will be of critical importance in our future developments.) It is important to observe that this notion of

equivalence is defined directly by a set of inference rules, rather than by an appeal to an operational semantics. For this reason, this form of equivalence is often referred to as *definitional equivalence*.

The equivalence system consists of seven rules. The first three rules express that term equivalence is an equivalence relation, and the next two rules express that it is a congruence with respect to abstraction and application. Finally, there are two substantiative rules. The first expresses that a beta redex is equivalent to its contractum. The second is an *extensionality* principle; it says that two functions (*i.e.*, terms of type $\mathtt{T}_1 \to \mathtt{T}_2$) are equivalent if all applications to an argument are equivalent.

Motivation

Term equivalence is important for a variety of reasons, but one of the most important applications of the systems relates not to the equivalence of terms, but of *types*. Recall the language λ_ω from TAPL Chapter 29. In λ_ω the type system provided type expressions of higher kind, in order to provide a facility for type operators. As a result, the type system of λ_ω was essentially a copy of the simply typed lambda calculus "one level up," in which terms became types and types became kinds.

Conversely, we may view the simply typed lambda calculus as the type system of λ_ω viewed "one level down," in which types become terms and kinds become types. Thus, the type $\mathtt{b}$ in $\lambda_{\to\mathtt{b}}$ corresponds to the kind $*$ of types in λ_ω. By solving the problem of checking equivalence of terms, we also learn how to check equivalence of types, which is in turn essential to the problem of type checking.

Note that $\lambda_{\to\mathtt{b}}$ contains no *actual* types such as `Nat`; they correspond to the uninterpreted constants $\mathtt{k}$ of type $\mathtt{b}$. Built-in type operators such as $\to$ would correspond to constants of type $\mathtt{b}\to\mathtt{b}\to\mathtt{b}$. For simplicity, we include no such built-in operators, but would be easy to add.

2.2 Untyped Equivalence Checking

The most common strategy for checking for equivalence of terms or types is the *normalize-and-compare* algorithm. To determine whether two well-typed terms, say $\mathtt{s}$ and $\mathtt{t}$, are equal, the algorithm computes their normal forms $\mathtt{s}'$ and $\mathtt{t}'$ using a reduction relation derived from the equivalence rules and then compares to see if $\mathtt{s}'$ and $\mathtt{t}'$ are identical.[1] The original terms $\mathtt{s}$ and $\mathtt{t}$

1. Actually, nearly all implementations of equivalence checkers refine this algorithm to interleave comparison with the computation of normal forms. This is done for two reasons: First, if

Parallel reduction $\boxed{s \Rightarrow t}$

$$\frac{}{t \Rightarrow t} \text{(QR-REFL)}$$

$$\frac{s_2 \Rightarrow t_2}{\lambda x{:}T_1.\, s_2 \Rightarrow \lambda x{:}T_1.\, t_2} \text{(QR-ABS)}$$

$$\frac{s_1 \Rightarrow t_1 \qquad s_2 \Rightarrow t_2}{s_1\ s_2 \Rightarrow t_1\ t_2} \text{(QR-APP)}$$

$$\frac{s_1 \Rightarrow t_1 \qquad s_2 \Rightarrow t_2}{(\lambda x{:}T_1.\, s_1)\ s_2 \Rightarrow [x \mapsto t_2]t_1} \text{(QR-BETA)}$$

$$\frac{s \Rightarrow t \qquad x \text{ not free in } s}{\lambda x{:}T.\, s\ x \Rightarrow t} \text{(QR-ETA)}$$

Figure 2-2: Parallel reduction of terms

are equivalent if and only if the normal forms s' and t' are identical.

For this algorithm to work relies on three facts:

- One must be able to derive an untyped[2] reduction relation $s \Rightarrow t$ from the equivalence rules. This relation must be suitable in the sense that if $\Gamma \vdash s : T$ and $\Gamma \vdash t : T$ then $\Gamma \vdash s \equiv t : T$ iff $s \Leftrightarrow^* t$, where $\Leftrightarrow^*$ is the symmetric, transitive closure of $\Rightarrow$. For $\lambda_{\to b}$ a suitable relation is given in Figure 2-2.
- The reduction relation must be *confluent*, meaning that if $r \Rightarrow^* s$ and $r \Rightarrow^* t$ then there exists some term u such that $s \Rightarrow^* u$ and $t \Rightarrow^* u$. If the relation is confluent, its symmetric, transitive closure may be decided by comparing normal forms, as stated by the following lemma.

2.2.1 LEMMA: Suppose $\Rightarrow$ is confluent, and the normal forms of s and t are s' and t'. Then $s \Leftrightarrow^* t$ iff $s' = t'$. □

Proof: Exercise [★★]. □

- The reduction relation must be *normalizing*, meaning that every term must have a normal form, and those normal forms must be effectively computable. Given the existence and computability of normal forms, one may use the preceding two facts to show that two terms are equivalent exactly when their normal forms coincide.

---

the terms are inequivalent, it can be detected earlier. Second, once any corresponding components of the normal forms are determined to be equivalent, they are no longer necessary and can be discarded. Thus, one can avoid storing the entire normal forms, which can be large.

2. The relation is untyped in the sense that, unlike equivalence, it is not indexed by a type. Nevertheless, the relation is over typed terms.

New syntactic forms

t	::=	...	*terms:*
		⋆	*unit term*
T	::=	...	*types:*
		1	*unit type*

Typing $\boxed{\Gamma \vdash t : T}$

$$\Gamma \vdash \star : 1 \qquad \text{(T-UNIT)}$$

Equivalence $\boxed{\Gamma \vdash s \equiv t : T}$

$$\frac{\Gamma \vdash s : 1 \qquad \Gamma \vdash t : 1}{\Gamma \vdash s \equiv t : 1} \qquad \text{(Q-UNIT)}$$

Figure 2-3: Unit type ($\lambda_{\to b1}$)

In summary, to employ the normalize-and-compare algorithm, one must define a suitable reduction relation, and prove that it is suitable, confluent, and normalizing. In some cases, the algorithm is not applicable, either because the suitable relation fails to be confluent and normalizing, or because no suitable relation can be defined in the first place. It is this latter case that will arise in the next section. In fact, many proofs of normalization employ a logical relation, so the normalize-and-compare technique, even when applicable, often still involves logical relations.

2.2.2 EXERCISE [★★]: Because of the rule QR-ETA, the reduction relation $\Rightarrow$ is confluent only for well-typed terms. Give an example of an ill-typed term for which confluence fails. □

2.2.3 EXERCISE [★★★★ ↛]: In TAPL §30.3 there is sketched a proof of confluence for a reduction relation similar to $\Rightarrow$, except that it omits the QR-ETA rule. Extend that proof to show confluence for $\Rightarrow$ for well-typed terms. That is, prove that if r is well-typed, and if $r \Rightarrow^* s$ and $r \Rightarrow^* t$, then there exists u such that $s \Rightarrow^* u$ and $t \Rightarrow^* u$. □

2.3 Type-Driven Equivalence

One case in which the normalize-and-compare algorithm is inapplicable is when the definition of equivalence is type sensitive. The example we will work with arises when we add a second base type corresponding to `unit` (recall TAPL §11.2). The important thing about this type is that it contains exactly one element; to emphasize this, we will refer to the unit type as `1`. The sole element of `1` is written ⋆. These additions are summarized in Figure 2-3.

The interesting facet of the extension with unit is its equivalence rule:

$$\frac{\Gamma \vdash s : 1 \qquad \Gamma \vdash t : 1}{\Gamma \vdash s \equiv t : 1} \qquad \text{(Q-UNIT)}$$

This rule expresses the fact that, since the type `1` contains exactly one element, any two elements of `1` must actually be the same. It is important to note that this rule is strictly stronger than the alternative rule:

$$\Gamma \vdash \star \equiv \star : 1 \qquad \text{(Q-UNIT-WEAK)}$$

Although Q-UNIT-WEAK is sound (indeed, it is derivable from either Q-UNIT or Q-REFL), it cannot derive equivalences involving unit *variables*, such as:

$$\dfrac{\dfrac{}{x:1, y:1 \vdash x : 1}\ \text{T-UNIT} \qquad \dfrac{}{x:1, y:1 \vdash y : 1}\ \text{T-UNIT}}{x:1, y:1 \vdash x \equiv y : 1}\ \text{Q-UNIT}$$

This example also illustrates why the normalize-and-compare algorithm is inapplicable once the unit type is added. The terms `x` and `y` must be judged to be equivalent, but the reason for this has nothing to do with the *form* of `x` or `y`. Indeed, `x` and `y` are already in normal form according to the usual reduction relation. The terms must be judged equivalent because of their *types*, and the normalize-and-compare algorithm has no way to account for that.

There are a variety of ways to address this difficulty. Many repair the normalize-and-compare algorithm by, in one way or another, giving them knowledge of types. However, we will consider an entirely different algorithm, one that is explicitly driven by type information.

2.3.1 EXERCISE [RECOMMENDED, ★]: The need for types in deciding equivalence is not limited to open terms. Give two *closed* terms that are equivalent but have distinct normal forms. □

2.4 An Equivalence Algorithm

We wish to devise an algorithm for the equivalence problem. Stated precisely, the problem is this:

> Suppose $\Gamma \vdash s : T$ and $\Gamma \vdash t : T$. Determine whether or not $\Gamma \vdash s \equiv t : T$.

To solve this problem, we will employ a type-driven algorithm. This algorithm is based on two main observations:

1. If `T` is `1`, we can immediately return `true`. This is because we then have $\Gamma \vdash s : 1$ and $\Gamma \vdash t : 1$ from our assumptions, and $\Gamma \vdash s \equiv t : 1$ follows directly by Q-UNIT.

2. If $\mathtt{T}$ is $\mathtt{T}_1 \to \mathtt{T}_2$, we can reduce the problem to a related problem where $\mathtt{T}$ is just $\mathtt{T}_2$. To do so, we replace any query of the form:

   $$\Gamma \vdash \mathtt{s} \stackrel{?}{\equiv} \mathtt{t} : \mathtt{T}_1 \to \mathtt{T}_2$$

   with the equivalent query:

   $$\Gamma, \mathtt{x} : \mathtt{T}_1 \vdash \mathtt{s}\ \mathtt{x} \stackrel{?}{\equiv} \mathtt{t}\ \mathtt{x} : \mathtt{T}_2$$

   The latter judgement immediately implies the former using the Q-EXT rule, and the former implies the latter using the Q-APP rule and a weakening lemma:

2.4.1 LEMMA [WEAKENING]: If $\Gamma \vdash \mathtt{s} \equiv \mathtt{t} : \mathtt{T}$ then $\Gamma, \mathtt{x} : \mathtt{S} \vdash \mathtt{s} \equiv \mathtt{t} : \mathtt{T}$. □

Proof: Straightforward induction on equivalence and typing derivations. (Recall TAPL Lemma 9.3.7.) □

The significance of these observations is that they give us a means to reduce an equivalence problem at an arbitrary type to one at the base type $\mathtt{b}$. Therefore, it remains only to find an algorithm that decides equivalence at type $\mathtt{b}$.

Equivalence at Base Type

Since we have already dealt with equivalences at type $\mathtt{1}$, we can use a variant of the normalize-and-compare algorithm. However, one subtlety remains: Suppose, for example, that we wish to determine whether $\mathtt{f}\ \mathtt{s} \equiv \mathtt{f}\ \mathtt{t} : \mathtt{b}$, where $\mathtt{f}$ has the type $\mathtt{1} \to \mathtt{b}$, and $\mathtt{s}$ and $\mathtt{t}$ have type $\mathtt{1}$. This is an equivalence problem at type $\mathtt{b}$, but to decide it we nevertheless must still decide an equivalence problem at type $\mathtt{1}$, and we have seen that we cannot do so with a simple syntactic comparison.

Thus, once the normalization portion of the normalize-and-compare phase has written the problem in the form:

$$\Gamma \vdash \mathtt{x}\ \mathtt{s}_1 \ldots \mathtt{s}_n \stackrel{?}{\equiv} \mathtt{x}\ \mathtt{t}_1 \ldots \mathtt{t}_n : \mathtt{b}$$

the comparisons of $\mathtt{s}_i$ to $\mathtt{t}_i$ should be done using the entire type-driven algorithm, and not simple syntactic comparison. Note, however, that if normalization gives a problem in the form:

$$\Gamma \vdash \mathtt{x}\ \mathtt{s}_1 \ldots \mathtt{s}_m \stackrel{?}{\equiv} \mathtt{y}\ \mathtt{t}_1 \ldots \mathtt{t}_n : \mathtt{b}$$

where $\mathtt{x} \neq \mathtt{y}$, then the two terms cannot be identical and we may immediately return `false`.

Weak Head Normalization

One additional insight is necessary to understand the algorithm. Since the comparison portion works on a term of the form $\mathtt{x}\ \mathtt{t}_1\ \ldots\ \mathtt{t}_n$ by using the entire algorithm on the subterms $\mathtt{t}_i$, there is no point in normalizing the term any more than is required to put it in that form.

For example, if, as in the above example, we wish to determine whether $\mathtt{f}\ \mathtt{s} \equiv \mathtt{f}\ \mathtt{t} : \mathtt{b}$, where $\mathtt{f}$ has the type $\mathtt{1}\to\mathtt{b}$, and $\mathtt{s}$ and $\mathtt{t}$ have type $\mathtt{1}$, the answer will always be $\mathtt{true}$, so any effort spent normalizing $\mathtt{s}$ or $\mathtt{t}$ is wasted.

Consequently, our algorithm will employ a less aggressive form of normalization called *weak head normalization*. In weak head normalization, the leftmost, outermost redex is always selected for reduction, and the process is halted as soon as the term begins with something other than an application.

Such terms will always either be a constant $\mathtt{k}$ or be in the form $\mathtt{x}\ \mathtt{t}_1\ \ldots\ \mathtt{t}_n$. These terms are referred to as *paths*. (Although some other terms—such as abstractions—are also weak head normal, those other terms cannot have type $\mathtt{b}$ so they will not arise.)

The Algorithm

An algorithm based on the preceding observations is given in Figure 2-4. The algorithm is given in the form of rules defining four relations:

1. Algorithmic term equivalence: $\Gamma \vdash \mathtt{s} \Leftrightarrow \mathtt{t} : \mathtt{T}$. This portion of the algorithm is directed by the type $\mathtt{T}$. It works by driving the type $\mathtt{T}$ down to $\mathtt{b}$. All of Γ, $\mathtt{s}$, $\mathtt{t}$, and $\mathtt{T}$ are inputs to this relation.

2. Algorithmic path equivalence: $\Gamma \vdash \mathtt{p} \leftrightarrow \mathtt{q} : \mathtt{T}$. This portion of the algorithm is directed by the structure of the paths $\mathtt{p}$ and $\mathtt{q}$. It works by checking that the head variables of $\mathtt{p}$ and $\mathtt{q}$ are the same (or that $\mathtt{p}$ and $\mathtt{q}$ are the same constant), and then comparing corresponding subterms of the path for algorithmic term equivalence. The type $\mathtt{T}$ is an *output* of this relation, and Γ, $\mathtt{p}$, and $\mathtt{q}$ are inputs.

3. Weak head reduction: $\mathtt{s} \rightsquigarrow \mathtt{t}$. This portion of the algorithm reduces one redex at the head of the term $\mathtt{s}$. It implements one step of weak head normalization.

4. Weak head normalization: $\mathtt{s} \Downarrow \mathtt{t}$. This portion of the algorithm computes weak head normal forms by performing weak head reductions until no more can be performed. (We write $\mathtt{t} \not\rightsquigarrow$ to mean that there exists no term $\mathtt{t}'$ such that $\mathtt{t} \rightsquigarrow \mathtt{t}'$.)

Syntax

$p, q ::=$ *paths:*

x *variable*

$p\ t$ *application*

k *constant*

Weak head reduction $\boxed{s \rightsquigarrow t}$

$$(\lambda x{:}T_{11}.t_{12})\ t_2 \rightsquigarrow [x \mapsto t_2]t_{12} \quad \text{(QAR-BETA)}$$

$$\frac{t_1 \rightsquigarrow t_1'}{t_1\ t_2 \rightsquigarrow t_1'\ t_2} \quad \text{(QAR-APP)}$$

Weak head normalization $\boxed{s \Downarrow t}$

$$\frac{s \rightsquigarrow t \qquad t \Downarrow u}{s \Downarrow u} \quad \text{(QAN-REDUCE)}$$

$$\frac{t \not\rightsquigarrow}{t \Downarrow t} \quad \text{(QAN-NORMAL)}$$

Algorithmic term equivalence $\boxed{\Gamma \vdash s \Leftrightarrow t : T}$

$$\frac{s \Downarrow p \qquad t \Downarrow q \qquad \Gamma \vdash p \leftrightarrow q : b}{\Gamma \vdash s \Leftrightarrow t : b} \quad \text{(QAT-BASE)}$$

$$\frac{\Gamma, x{:}T_1 \vdash s\ x \Leftrightarrow t\ x : T_2}{\Gamma \vdash s \Leftrightarrow t : T_1 \to T_2} \quad \text{(QAT-ARROW)}$$

$$\Gamma \vdash s \Leftrightarrow t : 1 \quad \text{(QAT-ONE)}$$

Algorithmic path equivalence $\boxed{\Gamma \vdash p \leftrightarrow q : T}$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x \leftrightarrow x : T} \quad \text{(QAP-VAR)}$$

$$\frac{\Gamma \vdash p \leftrightarrow q : T_1 \to T_2 \qquad \Gamma \vdash s \Leftrightarrow t : T_1}{\Gamma \vdash p\ s \leftrightarrow t\ q : T_2} \quad \text{(QAP-APP)}$$

$$\Gamma \vdash k \leftrightarrow k : b \quad \text{(QAP-CONST)}$$

Figure 2-4: Equivalence algorithm for $\lambda_{\to b1}$

2.4.2 EXERCISE [★ ↛]: Convince yourself that the algorithm implements the ideas discussed above. □

2.4.3 EXERCISE [★★ ↛]: Hand-execute the algorithm on:

$$f{:}(1\to1)\to b \vdash f\ (\lambda x{:}1.\star) \stackrel{?}{\Leftrightarrow} f\ (\lambda x{:}1.x) : b\to b$$

□

2.4.4 EXERCISE [★★★★]: Prove that the algorithm is sound. That is, show that if $\Gamma \vdash s \Leftrightarrow t : T$ (where $\Gamma \vdash s : T$ and $\Gamma \vdash t : T$) then $\Gamma \vdash s \equiv t : T$. □

2.5 Completeness: A First Attempt

There are two parts to showing that the algorithm is correct. First, we wish to show that the algorithm is *sound;* that is, that the algorithm says yes only for equivalent terms. We considered soundness in Exercise 2.4.4. Second, we wish to show that the algorithm is *complete;* that is, that the algorithm says yes for *all* equivalent terms. Completeness is a good deal trickier than soundness, and is the subject of the remainder of this chapter.

We wish to show the following result:

2.5.1 PROPOSITION: If $\Gamma \vdash s \equiv t : T$ then $\Gamma \vdash s \Leftrightarrow t : T$. □

Our first attempt to prove this would naturally be to trying proving it directly by induction on derivations. This attempt encounters a variety of difficulties, most of which are surmountable. Because of the Q-REFL rule, we must immediately strengthen our statement to say something about typing:

2.5.2 PROPOSITION: If $\Gamma \vdash t : T$ then $\Gamma \vdash t \Leftrightarrow t : T$. □

With this addition, several rules go through without difficulty: T-CONST, T-UNIT, Q-REFL, Q-EXT, and Q-UNIT. Four other rules are more difficult, but can be dealt with:

- *Cases* Q-SYMM *and* Q-TRANS*:* For these cases, we must prove two lemmas stating that the algorithm is symmetric and transitive:

2.5.3 LEMMA [ALGORITHMIC SYMMETRY]: If $\Gamma \vdash s \Leftrightarrow t : T$ then $\Gamma \vdash t \Leftrightarrow s : T$. □

2.5.4 LEMMA [ALGORITHMIC TRANSITIVITY]: If $\Gamma \vdash s \Leftrightarrow t : T$ and $\Gamma \vdash t \Leftrightarrow u : T$ then $\Gamma \vdash s \Leftrightarrow u : T$. □

Proof: Both proofs are by induction on derivations, with a simultaneous induction showing the analogous property for algorithmic path equivalence. □

- *Case* Q-ABS *(and* T-ABS *is similar):* Here we wish to show that $\Gamma \vdash \lambda x{:}T_1.s_2 \Leftrightarrow \lambda x{:}T_1.t_2 : T_1 \to T_2$, for which we must prove that:

  $$\Gamma, x{:}T_1 \vdash (\lambda x{:}T_1.s_2)\, x = (\lambda x{:}T_1.t_2)\, x : T_2$$

  However, the induction hypothesis provides us a related but different fact:

  $$\Gamma, x{:}T_1 \vdash s_2 = t_2 : T_2$$

  Therefore, for us to conclude this case, we need another lemma stating that the algorithm is closed under weak head expansion:

2.5.5 LEMMA [ALGORITHMIC WEAK HEAD CLOSURE]: If $\Gamma \vdash s \Leftrightarrow t : T$ and $s' \rightsquigarrow^* s$ and $t' \rightsquigarrow^* t$ then $\Gamma \vdash s' \Leftrightarrow t' : T$. □

Proof: By induction on the derivation of algorithmic term equivalence. □

2.5.6 EXERCISE [★ ↛]: Verify Lemma 2.5.5. □

2.5.1 Trouble with Application

This leaves four rules: two dealing with application (T-APP and Q-APP), and two dealing with variables and substitution (T-VAR and Q-BETA). It is the application rules that bring the proof attempt to a standstill.[3]

The essential problem is that the induction hypothesis gives us no information about what happens when a term is applied to an argument. Consider the case Q-APP:

$$\frac{\Gamma \vdash s_1 \equiv t_1 : T_1 \to T_2 \qquad \Gamma \vdash s_2 \equiv t_2 : T_1}{\Gamma \vdash s_1\ s_2 \equiv t_1\ t_2 : T_2} \qquad \text{(Q-APP)}$$

The induction hypothesis gives us that $\Gamma \vdash s_1 \Leftrightarrow t_1 : T_1 \to T_2$ and $\Gamma \vdash s_2 \Leftrightarrow t_2 : T_1$. We wish to conclude that $\Gamma \vdash s_1\ s_2 \Leftrightarrow t_1\ t_2 : T_2$, but we cannot.

By inversion on $\Gamma \vdash s_1 \Leftrightarrow t_1 : T_1 \to T_2$ we obtain the fact that $\Gamma, x{:}T_1 \vdash s_1\ x \Leftrightarrow t_1\ x : T_2$, but that is as close as we get, since the behavior of the algorithm comparing $s_1\ x$ to $t_1\ x$ is entirely different from its behavior comparing $s_1\ s_2$ to $t_1\ t_2$.

2.6 Logical Relations

The problem discussed above arises because the algorithmic equivalence relation is not *logical*, in the following sense:[4]

2.6.1 DEFINITION: A relation $R(s, t, T)$ indexed by types T, is *logical* if whenever $R(s_1, t_1, T_1 \to T_2)$ and $R(s_2, t_2, T_1)$ hold, it follows that $R(s_1\ s_2, t_1\ t_2, T_2)$ also holds. □

That is, a relation is logical when the relatedness of two applications $s_1\ s_2$ and $t_1\ t_2$ is inherited from the pairwise relatedness of their constituent function and argument (s_1 with t_1, and s_2 with t_2). Such a relation is called "logical" because it respects the actions of the logical operators (in this case implications) that correspond to the language's type constructors.

The idea behind proof by logical relations is to circumvent problems resulting from the absence of logicality by defining another relation that *is* logical, and that also implies the desired property (in this case algorithmic equivalence). The new, logical relation is then used as the induction hypothesis in the proof.

3. The variable and substitution rules could be addressed using a device we develop in Section 2.9, but we will not bother to employ it here, since this proof attempt is doomed from the start.

4. Actually, it is more precise to say that algorithmic term equivalence is not *evidently* logical; that is, we cannot prove it at this stage in the proof.

Our overall proof strategy, then, consists of three stages:

1. Define a logical relation. When two terms are related by the logical relation, we will say that they are *logically equivalent*.
2. Show that logical equivalence implies algorithmic equivalence.
3. Show that definitional equivalence implies logical equivalence.

A Logical Relation

A direct way to define a logical relation is essentially by fiat. The definition proceeds inductively by cases on the index type, asserting algorithmic equivalence at base types, and asserting the exactly the necessary property at function types. This gives us the following first attempt at a definition of *logical equivalence:*

$\Gamma \vdash s$ **is** $t : T$ if and only if either:
$T{=}1$,
or $T{=}b$ and $\Gamma \vdash s \Leftrightarrow t : b$,
or $T{=}T_1 \to T_2$ and for all s', t',
if $\Gamma \vdash s'$ **is** $t' : T_1$
then $\Gamma \vdash s\ s'$ **is** $t\ t' : T_2$.

Note that the first clause of the definition could equivalently be "$T{=}1$ *and* $\Gamma \vdash s \Leftrightarrow t : 1$", since the algorithmic equivalence at 1 always holds.

Monotonicity

This relation is clearly logical, and at base types it just as clearly implies algorithmic equivalence. The question is, does it imply algorithmic equivalence at function types? The answer turns out to be *almost, but not quite*:

Suppose $\Gamma \vdash s$ **is** $t : T_1 \to T_2$. We wish to show that $\Gamma \vdash s \Leftrightarrow t : T_1 \to T_2$, for which it is sufficient to show:

$$\Gamma, x{:}T_1 \vdash s\ x \Leftrightarrow t\ x : T_2$$

Since T_2 is smaller than $T_1 \to T_2$, we can conclude by induction[5] that the desired algorithmic equivalence from the corresponding logical equivalence. Thus it remains to show:

$$\Gamma, x{:}T_1 \vdash s\ x \textbf{ is } t\ x : T_2$$

5. More precisely, we *could* conclude by induction, were this proof to work.

From the definition of logical equivalence, we can deduce the desired logical equivalence from:

1. $\Gamma, \mathtt{x}:\mathtt{T}_1 \vdash \mathtt{s}$ **is** $\mathtt{t} : \mathtt{T}_1 \to \mathtt{T}_2$, and

2. $\Gamma, \mathtt{x}:\mathtt{T}_1 \vdash \mathtt{x}$ **is** $\mathtt{x} : \mathtt{T}_1$.

The latter is not obvious, but we will be able to prove it (for our final definition). The former, on the other hand, is very similar to our assumption, $\Gamma \vdash \mathtt{s}$ **is** $\mathtt{t} : \mathtt{T}_1 \to \mathtt{T}_2$. All we need to know is that logical equivalence is preserved when bindings are added to the context. This property is called *monotonicity*.

Failure of Monotonicity

Unfortunately, monotonicity fails for our current definition of logical equivalence. It is not difficult to show monotonicity for the algorithm:

2.6.2 LEMMA [ALGORITHMIC MONOTONICITY]: Suppose $\Gamma' \supseteq \Gamma$. Then:

1. If $\Gamma \vdash \mathtt{s} \Leftrightarrow \mathtt{t} : \mathtt{T}$ then $\Gamma' \vdash \mathtt{s} \Leftrightarrow \mathtt{t} : \mathtt{T}$.

2. If $\Gamma \vdash \mathtt{p} \leftrightarrow \mathtt{q} : \mathtt{T}$ then $\Gamma' \vdash \mathtt{p} \leftrightarrow \mathtt{q} : \mathtt{T}$.

□

Proof: By induction on derivations. □

The lemma gives us monotonicity of logical equivalence for type `b`, and logical equivalence is trivially monotone for type `1`. The leaves $\mathtt{T}_1 \to \mathtt{T}_2$, where monotonicity fails.

To see why, let us suppose $\Gamma \vdash \mathtt{s}$ **is** $\mathtt{t} : \mathtt{T}_1 \to \mathtt{T}_2$ and attempt to show $\Gamma, \mathtt{x}:\mathtt{S} \vdash \mathtt{s}$ **is** $\mathtt{t} : \mathtt{T}_1 \to \mathtt{T}_2$. By the definition, it is sufficient to show that if:

$$\Gamma, \mathtt{x}:\mathtt{S} \vdash \mathtt{s}' \textbf{ is } \mathtt{t}' : \mathtt{T}_1$$

then:

$$\Gamma, \mathtt{x}:\mathtt{S} \vdash \mathtt{s}\ \mathtt{s}' \textbf{ is } \mathtt{t}\ \mathtt{t}' : \mathtt{T}_2$$

Since $\mathtt{T}_2$ is smaller than $\mathtt{T}_1 \to \mathtt{T}_2$, induction would give us monotonicity for $\mathtt{T}_2$, so it would be sufficient to show that:

$$\Gamma \vdash \mathtt{s}\ \mathtt{s}' \textbf{ is } \mathtt{t}\ \mathtt{t}' : \mathtt{T}_2$$

This follows from our supposition by the definition of logical equivalence, *provided* that $\Gamma \vdash \mathtt{s}'$ **is** $\mathtt{t}' : \mathtt{T}_1$. Unfortunately, our supposition provides only

$\Gamma, \mathtt{x}:\mathtt{S} \vdash \mathtt{s}'$ **is** $\mathtt{t}' : \mathtt{T}_1$. Thus, at type $\mathtt{T}_1$ we need not monotonicity (which induction could provide), but the converse, *antitonicity,* which is certainly false.

2.6.3 EXERCISE [★]: Let antitonicity be the property that if $\Gamma, \mathtt{x}:\mathtt{S} \vdash \mathtt{s}$ **is** $\mathtt{t} : \mathtt{T}$ then $\Gamma \vdash \mathtt{s}$ **is** $\mathtt{t} : \mathtt{T}$. Produce a counterexample for antitonicity. □

2.6.4 EXERCISE [★★★]: Suppose the language contains exactly one constant $\mathtt{k}$. Produce a counterexample for monotonicity, and prove that it is a counterexample. □

2.7 A Monotone Logical Relation

Earlier we addressed the problem of logicality by fiat, crafting a definition that provided exactly the necessary property. We can address the problem of monotonicity in essentially the same manner. The problem is that the definition's clause for function types could hold for a context Γ, but not evidently hold for a larger context Γ'.

To resolve this difficulty, we revise the definition so that the clause for function types must hold not only for the current context Γ, but also for any possible extended context $\Gamma' \supseteq \Gamma$. This gives us our final definition of logical equivalence:

2.7.1 DEFINITION [LOGICAL EQUIVALENCE]: Logical equivalence is defined as follows:

$\Gamma \vdash \mathtt{s}$ **is** $\mathtt{t} : \mathtt{T}$ if and only if either:
$\mathtt{T}=\mathtt{1}$,
or $\mathtt{T}=\mathtt{b}$ and $\Gamma \vdash \mathtt{s} \Leftrightarrow \mathtt{t} : \mathtt{b}$,
or $\mathtt{T}=\mathtt{T}_1 \to \mathtt{T}_2$ and for all $\mathtt{s}', \mathtt{t}'$, and for all $\Gamma' \supseteq \Gamma$,
if $\Gamma' \vdash \mathtt{s}'$ **is** $\mathtt{t}' : \mathtt{T}_1$
then $\Gamma' \vdash \mathtt{s}\ \mathtt{s}'$ **is** $\mathtt{t}\ \mathtt{t}' : \mathtt{T}_2$.

□

With this definition, we can easily prove the monotonicity of logical equivalence:

2.7.2 LEMMA [LOGICAL MONOTONICITY]: If $\Gamma \vdash \mathtt{s}$ **is** $\mathtt{t} : \mathtt{T}$ and $\Gamma' \supseteq \Gamma$ then $\Gamma' \vdash \mathtt{s}$ **is** $\mathtt{t} : \mathtt{T}$. □

Proof: By induction on the type $\mathtt{T}$, appealing to algorithmic monotonicity in the case where $\mathtt{T} = \mathtt{b}$. □

An Aside

A logical relation made monotone in this manner is often called a *Kripke* logical relation, by analogy to Kripke models for modal logic. Modal logic is a form of logic designed for reasoning about the difference between ordinary truths—truths that happen to hold given the existing state of affairs—and *necessary* (or "categorical") truths, which must hold in *any* reasonable state of affairs.

A Kripke model for modal logic is based on a set of "worlds," where each world provides a different set of truths. The set of worlds is additionally structured by a notion of which worlds are "reachable" from which other worlds. In such a model, an ordinary truth is one that holds in the current world, and a necessary truth is one that holds in all worlds reachable from the current world. (Any world not reachable from the current world is ignored, since there is no way to determine its existence.)

The connection to Kripke models arises in the logical relation's use of a context Γ, which we view as specifying a world. Each world provides a different set of variables, and we may reach another world by adding variables to the current context. Thus, when we require that our logical relation be monotone (that is, that it continues to hold in all reachable contexts), we are saying that we are interested in *necessary* equivalence, not accidental equivalence.[6]

2.8 The Main Lemma

With a monotone logical relation in hand, we are now ready to prove the completeness of the algorithm. Recalling our proof strategy from page 206, we have accomplished the first step, the definition of a logical relation. We now wish to show that logical equivalence implies algorithmic equivalence. It will then remain to show that definitional equivalence implies logical equivalence.

This fact is established by the following "main lemma." The main lemma actually establishes two facts simultaneously. First, it shows that logical equivalence implies algorithmic equivalence. To appreciate the second, recall that in order to show that logical implies algorithmic equivalence, we also need to establish that variables are logically equivalent to themselves. To do so, we prove the stronger result that algorithmically equivalent paths are logi-

6. Exercise 2.6.4 asks you to produce an example of an accidental equivalence, one that holds only in a certain world.

cally equivalent. The necessary result for variables follows, since a variable is always algorithmically path equivalent to itself (rule QAP-VAR).

In the following proof, observe how inseparably the two induction hypotheses of the lemma are intertwined. Although the details certainly differ from one application to another, it is very typical of logical relations proofs to use a lemma such as this, wherein one clause of the lemma establishes the logical relation and the other clause exploits it. This structure of the proof usually results from the definition of the logical relation in the arrow case, where the relation appears on both the left and the right of the implication.

2.8.1 LEMMA [MAIN LEMMA]:

1. If $\Gamma \vdash \mathtt{s}$ **is** $\mathtt{t} : \mathtt{T}$ then $\Gamma \vdash \mathtt{s} \Leftrightarrow \mathtt{t} : \mathtt{T}$.
2. If $\Gamma \vdash \mathtt{p} \leftrightarrow \mathtt{q} : \mathtt{T}$ then $\Gamma \vdash \mathtt{p}$ **is** $\mathtt{q} : \mathtt{T}$.

□

Proof: By induction on the structure of the type $\mathtt{T}$.

Case: $\mathtt{T} = \mathtt{b}$

1. Suppose $\Gamma \vdash \mathtt{s}$ **is** $\mathtt{t} : \mathtt{b}$. By definition, $\Gamma \vdash \mathtt{s} \Leftrightarrow \mathtt{t} : \mathtt{b}$.
2. Suppose $\Gamma \vdash \mathtt{p} \leftrightarrow \mathtt{q} : \mathtt{b}$. Since $\mathtt{p}$ and $\mathtt{q}$ are paths, it follows that $\mathtt{p} \not\rightarrow$ and $\mathtt{q} \not\rightarrow$, so $\mathtt{p} \Downarrow \mathtt{p}$ and $\mathtt{q} \Downarrow \mathtt{q}$ by QAN-NORMAL. Therefore $\Gamma \vdash \mathtt{p} \Leftrightarrow \mathtt{q} : \mathtt{b}$ by QAT-BASE, and $\Gamma \vdash \mathtt{p}$ **is** $\mathtt{q} : \mathtt{b}$ follows by the definition.

Case: $\mathtt{T} = \mathtt{1}$

1. For any $\mathtt{s}$ and $\mathtt{t}$, $\Gamma \vdash \mathtt{s} \Leftrightarrow \mathtt{t} : \mathtt{1}$ by QAT-ONE.
2. For any $\mathtt{p}$ and $\mathtt{q}$, $\Gamma \vdash \mathtt{p}$ **is** $\mathtt{q} : \mathtt{1}$ by the definition.

Case: $\mathtt{T} = \mathtt{T}_1 \rightarrow \mathtt{T}_2$

1. Suppose $\Gamma \vdash \mathtt{s}$ **is** $\mathtt{t} : \mathtt{T}_1 \rightarrow \mathtt{T}_2$. We wish to show that $\Gamma \vdash \mathtt{s} \Leftrightarrow \mathtt{t} : \mathtt{T}_1 \rightarrow \mathtt{T}_2$. It is sufficient to show that $\Gamma, \mathtt{x} : \mathtt{T}_1 \vdash \mathtt{s}\ \mathtt{x} \Leftrightarrow \mathtt{t}\ \mathtt{x} : \mathtt{T}_2$. By induction, it is sufficient to show that $\Gamma, \mathtt{x} : \mathtt{T}_1 \vdash \mathtt{s}\ \mathtt{x}$ **is** $\mathtt{t}\ \mathtt{x} : \mathtt{T}_2$.

   By induction (using the other clause), since $\Gamma, \mathtt{x} : \mathtt{T}_1 \vdash \mathtt{x} \leftrightarrow \mathtt{x} : \mathtt{T}_1$ (by QAP-VAR), we may deduce that $\Gamma, \mathtt{x} : \mathtt{T}_1 \vdash \mathtt{x}$ **is** $\mathtt{x} : \mathtt{T}_1$. Therefore, since $(\Gamma, \mathtt{x} : \mathtt{T}_1) \supseteq \Gamma$, we may conclude from the definition and our supposition that $\Gamma, \mathtt{x} : \mathtt{T}_1 \vdash \mathtt{s}\ \mathtt{x}$ **is** $\mathtt{t}\ \mathtt{x} : \mathtt{T}_2$, as desired.
2. Suppose $\Gamma \vdash \mathtt{p} \leftrightarrow \mathtt{q} : \mathtt{T}_1 \rightarrow \mathtt{T}_2$. We wish to show that $\Gamma \vdash \mathtt{p}$ **is** $\mathtt{q} : \mathtt{T}_1 \rightarrow \mathtt{T}_2$. Suppose further that $\Gamma' \supseteq \Gamma$ and $\Gamma' \vdash \mathtt{s}$ **is** $\mathtt{t} : \mathtt{T}_1$. Then we wish to

show that $\Gamma' \vdash \mathtt{p\ s}$ **is** $\mathtt{q\ t} : \mathtt{T}_2$. By induction, it is sufficient to show that $\Gamma' \vdash \mathtt{p\ s} \leftrightarrow \mathtt{q\ t} : \mathtt{T}_2$.

By induction (using the other clause), $\Gamma' \vdash \mathtt{s} \Leftrightarrow \mathtt{t} : \mathtt{T}_1$. By algorithmic monotonicity (Lemma 2.6.2), $\Gamma' \vdash \mathtt{p} \leftrightarrow \mathtt{q} : \mathtt{T}_1 \to \mathtt{T}_2$. Therefore, by rule QAP-APP, $\Gamma' \vdash \mathtt{p\ s} \leftrightarrow \mathtt{q\ t} : \mathtt{T}_2$, as desired.

□

2.9 The Fundamental Theorem

The final step in the completeness proof is to show that definition equivalence implies logical equivalence. We will refer to the theorem showing this fact as the "Fundamental Theorem."

Recall our attempted proof from Section 2.5. The principal problem we encountered was with application; we could not deduce from the algorithmic equivalence of two functions anything about the equivalence of their applications to arguments. We have solved that problem by using a logical relation, which explicitly provides the necessary conclusions about applications.

Since we are now using logical equivalence in place of algorithmic equivalence, we must revisit some of our other devices from Section 2.5. To address the Q-SYMM, Q-TRANS, Q-ABS, and T-ANS, we showed that the algorithm is symmetric, transitive, and closed under weak head reduction (Lemmas 2.5.3, 2.5.4, and 2.5.5). We will now require analogues of these lemmas applicable to logical equivalence:

2.9.1 LEMMA [LOGICAL SYMMETRY]: If $\Gamma \vdash \mathtt{s}$ **is** $\mathtt{t} : \mathtt{T}$ then $\Gamma \vdash \mathtt{t}$ **is** $\mathtt{s} : \mathtt{T}$. □

Proof: By induction on the structure of $\mathtt{T}$, appealing to Algorithmic Symmetry (Lemma 2.5.3) in the case where $\mathtt{T} = \mathtt{b}$. □

2.9.2 LEMMA [LOGICAL TRANSITIVITY]: If $\Gamma \vdash \mathtt{s}$ **is** $\mathtt{t} : \mathtt{T}$ and $\Gamma \vdash \mathtt{t}$ **is** $\mathtt{u} : \mathtt{T}$ then $\Gamma \vdash \mathtt{s}$ **is** $\mathtt{u} : \mathtt{T}$. □

Proof: Exercise [RECOMMENDED, ★★]. □

2.9.3 LEMMA [LOGICAL WEAK HEAD CLOSURE]: If $\Gamma \vdash \mathtt{s}$ **is** $\mathtt{t} : \mathtt{T}$ and $\mathtt{s}' \rightsquigarrow^* \mathtt{s}$ and $\mathtt{t}' \rightsquigarrow^* \mathtt{t}$ then $\Gamma \vdash \mathtt{s}'$ **is** $\mathtt{t}' : \mathtt{T}$. □

Proof: By induction on the structure of $\mathtt{T}$, appealing to Algorithmic Weak Head Closure (Lemma 2.5.5) in the case where $\mathtt{T} = \mathtt{b}$. □

Closure Under Substitution

There remain two more cases we did not consider in Section 2.5, T-VAR and Q-BETA. We could deal with T-VAR immediately, since a consequence of the

Main Lemma is that variables are logically equivalent to themselves. However, we will actually end up dealing with T-VAR somewhat differently in light of our last remaining complication.[7]

That final complication stems from the rule Q-BETA:

$$\frac{\Gamma, x:T_1 \vdash s_{12} \equiv t_{12} : T_2 \qquad \Gamma \vdash s_2 \equiv t_2 : T_1}{\Gamma \vdash (\lambda x:T_1.s_{12})\ s_2 \equiv [x \mapsto t_2]t_{12} : T_2} \quad \text{(Q-BETA)}$$

Suppose we employ the obvious induction hypothesis: if $\Gamma \vdash s \equiv t : T$ then $\Gamma \vdash s$ **is** $t : T$. Then for the Q-BETA case, we need to show that:

$$\Gamma \vdash (\lambda x:T_1.s_{12})\ s_2 \textbf{ is } [x \mapsto t_2]t_{12} : T_2$$

Using Logical Weak Head Closure, it is sufficient to show that:

$$\Gamma \vdash [x \mapsto s_2]s_{12} \textbf{ is } [x \mapsto t_2]t_{12} : T_2$$

Induction provides us with $\Gamma, x:T_1 \vdash s_{12} \equiv t_{12} : T_2$ and $\Gamma \vdash s_2 \equiv t_2 : T_1$.

Thus, we could complete the proof by showing that logical equivalence is closed under logically equivalent substitutions. Unfortunately, it is not clear how to prove that proposition at this stage in the completeness proof.

The Theorem

Fortunately, we can work around this difficulty by building this notion of equivalent substitutions into the Fundamental Theorem itself. First we require a few definitions:

2.9.4 DEFINITION [SUBSTITUTIONS]: A substitution is a function from some set of variables to terms. □

2.9.5 DEFINITION [SUBSTITUTIONS AND TERMS]: Suppose γ is a substitution and t is a term such that the free variables of t are contained in $dom(\gamma)$. Then we write $\gamma(t)$ to refer to the term resulting from the application of γ to all the free variables of t. □

2.9.6 DEFINITION [SUBSTITUTION EXTENSION]: Suppose $x \notin dom(\gamma)$. Then we define $\gamma[x \mapsto t]$ as the substitution with domain $dom(\gamma) \cup \{x\}$ such that:

$$(\gamma[x \mapsto t])(y) = \begin{cases} \gamma(y) & x \neq y \\ t & x = y \end{cases}$$

□

7. See the T-VAR case of Theorem 2.9.8.

2.9.7 DEFINITION [LOGICALLY EQUIVALENT SUBSTITUTIONS]: Logical equivalence of substitutions is defined as follows:

$\Gamma' \vdash \gamma$ **is** $\delta : \Gamma$ if $dom(\gamma) = dom(\delta) dom(\Gamma)$ and for every $x : T \in \Gamma$ we have $\Gamma' \vdash \gamma(x)$ **is** $\delta(x) : T$.

□

Now we can state the Fundamental Theorem by uniformly considering all equivalences under the application of equivalent substitutions:

2.9.8 THEOREM [FUNDAMENTAL THEOREM]:

1. If $\Gamma \vdash t : T$ and $\Gamma' \vdash \gamma$ **is** $\delta : \Gamma$ then $\Gamma' \vdash \gamma(t)$ **is** $\delta(t) : T$.

2. If $\Gamma \vdash s \equiv t : T$ and $\Gamma' \vdash \gamma$ **is** $\delta : \Gamma$ then $\Gamma' \vdash \gamma(s)$ **is** $\delta(t) : T$.

□

Proof: By induction on derivations. We show several cases; the rest are left as exercises.

Case T-VAR: $t = x$
with $x : T \in \Gamma$

By assumption, $\Gamma' \vdash \gamma(x)$ **is** $\delta(x) : T$.

Case T-ABS: $t = \lambda x : T_1 . t_2$
$T = T_1 \to T_2$

We wish to show that $\Gamma' \vdash \gamma(\lambda x : T_1 . t_2)$ **is** $\delta(\lambda x : T_1 . t_2) : T_1 \to T_2$. Suppose $\Gamma'' \supseteq \Gamma'$ and $\Gamma'' \vdash s'$ **is** $t' : T_1$. We wish to show that $\Gamma'' \vdash (\lambda x : T_1 . \gamma(t_2))\, s'$ **is** $(\lambda x : T_1 . \delta(t_2))\, t' : T_2$. By logical weak head closure, it is sufficient to show that $\Gamma'' \vdash [x \mapsto s']\gamma(t_2)$ **is** $[x \mapsto t']\delta(t_2) : T_2$.

By logical monotonicity, $\Gamma'' \vdash \gamma$ **is** $\delta : \Gamma$. Thus, $\Gamma'' \vdash \gamma[x \mapsto s']$ **is** $\delta[x \mapsto t'] : (\Gamma, x : T_1)$. Therefore, by induction, $\Gamma'' \vdash \gamma[x \mapsto s'](t_2)$ **is** $\delta[x \mapsto t'](t_2) : T_2$, which is equivalent to the desired conclusion.

Case T-APP: $t = t_1\, t_2$
$T = T_{12}$

By induction, $\Gamma' \vdash \gamma(t_1)$ **is** $\delta(t_1) : T_1 \to T_2$ and $\Gamma' \vdash \gamma(t_2)$ **is** $\delta(t_2) : T_1$. By the definition of the logical relation, since $\Gamma' \supseteq \Gamma$, we may conclude $\Gamma' \vdash \gamma(t_1)\gamma(t_2)$ **is** $\delta(t_1)\delta(t_2) : T_2$. That is, $\Gamma' \vdash \gamma(t_1\, t_2)$ **is** $\delta(t_1\, t_2) : T_2$.

Case Q-BETA: $s = (\lambda x : T_1 . s_{12})\, s_2$
$t = [x \mapsto t_2]t_{12} : T_2$
$T = T_2$

By induction, $\Gamma' \vdash \gamma(s_2)$ **is** $\delta(t_2) : T_1$. Thus $\Gamma' \vdash \gamma[x \mapsto \gamma(s_2)]$ **is** $\delta[x \mapsto \delta(t_2)] : (\Gamma, x:T_1)$. Therefore, by induction:

$$\Gamma' \vdash \gamma[x \mapsto \gamma(s_2)](s_{12}) \textbf{ is } \delta[x \mapsto \delta(t_2)](t_{12}) : T_2$$

By rearranging substitutions:

$$\Gamma' \vdash [x \mapsto \gamma(s_2)]\gamma(s_{12}) \textbf{ is } \delta([x \mapsto t_2]t_{12}) : T_2$$

Finally, by Logical Weak Head Closure:

$$\Gamma' \vdash (\lambda x:T_1.\gamma(s_{12}))\,\gamma(s_2) \textbf{ is } \delta([x \mapsto t_2]t_{12}) : T_2$$

That is:

$$\Gamma' \vdash \gamma((\lambda x:T_1.s_{12})\ s_2) \textbf{ is } \delta([x \mapsto t_2]t_{12}) : T_2$$

□

2.9.9 EXERCISE [RECOMMENDED, ★★★]: Complete the proof of Theorem 2.9.8. □

Now we can establish the algorithm's completeness, using the Fundamental Theorem with an identity substitution:

2.9.10 COROLLARY [COMPLETENESS]: If $\Gamma \vdash s \equiv t : T$ then $\Gamma \vdash s \Leftrightarrow t : T$. □

Proof: Suppose $\Gamma \vdash s \equiv t : T$. Let γ be the identity substitution on $dom(\Gamma)$. For all $x:T$ in Γ, observe that $\Gamma \vdash x$ **is** $x : T$ by the Main Lemma. Therefore $\Gamma \vdash \gamma$ **is** $\gamma : \Gamma$. By the Fundamental Theorem, $\Gamma \vdash \gamma(s)$ **is** $\gamma(t) : T$, which is to say $\Gamma \vdash s$ **is** $t : T$. Therefore $\Gamma \vdash s \Leftrightarrow t : T$ by the Main Lemma. □

2.9.11 EXERCISE [★]: An irony arises from the use logical relations to show completeness: it turns out that algorithmic equivalence actually is logical after all (at least for well-formed terms), we just cannot prove it until we have already proven the algorithm to be sound and complete. Show that if $\Gamma \vdash s_1 \Leftrightarrow t_1 : T_1 \to T_2$ and $\Gamma \vdash s_2 \Leftrightarrow t_2 : T_1$ (where $\Gamma \vdash s_1 : T_1 \to T_2$, $\Gamma \vdash t_1 : T_1 \to T_2$, $\Gamma \vdash s_2 : T_1$, and $\Gamma \vdash t_2 : T_1$), then $\Gamma \vdash s_1\ s_2 \Leftrightarrow t_1\ t_2 : T_2$. □

2.9.12 EXERCISE [★★★★]: We have shown that the equivalence algorithm is sound and complete. To show that the algorithm decides the equivalence problem, it remains to show that it terminates on all well-formed inputs. Show that if $\Gamma \vdash s : T$ and $\Gamma \vdash t : T$ then there exists no infinite proof search rooted in $\Gamma \vdash s \Leftrightarrow t : T$.

Hint: Termination is a corollary of completeness. You will not need to prove any additional non-trivial facts about the algorithm. □

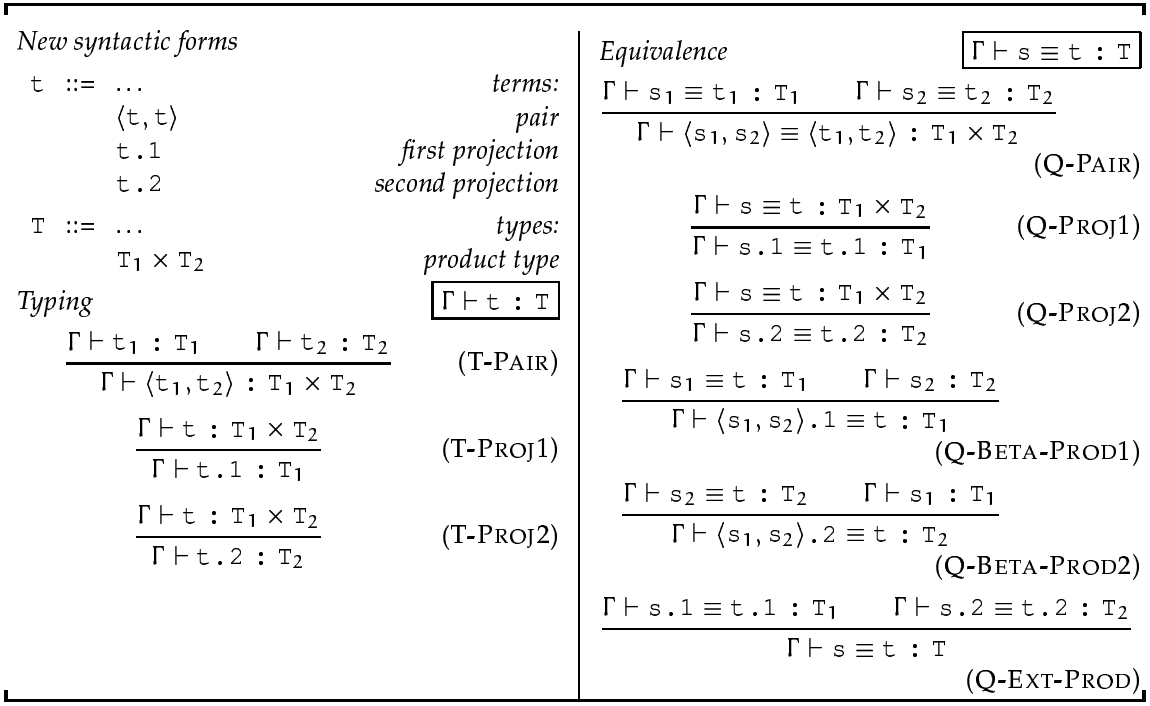


Figure 2-5: Product types ($\lambda_{\to\times b1}$)

2.9.13 EXERCISE [RECOMMENDED, ★★★★]: It is straightforward to extend our language with pairs. Figure 2-5 gives the extended syntax and type system, and Figure 2-6 extends the equivalence algorithm to account for pairs. Extend the completeness proof to cover the extended language and algorithm. □

2.10 Notes

The ideas behind logical relations were first developed by Tait (1967) and Howard (1973), and were developed further by Plotkin (1980). Logical relations were first proposed as a general proof technique by Statman (1985b).

The idea of using a Kripke logical relation to show the completeness of an equivalence algorithm is due to Coquand (1991a), who applies it to algorithms for various similar type systems. Unlike the algorithm we consider here, Coquand's algorithms are not type-directed. Coquand's technique was adopted by Stone and Harper (2000a), who use a more sophisticated form

Syntax

$p, q ::= \ldots$ *paths:*

$p.1$ *first projection*

$p.2$ *second projection*

Weak head reduction $\boxed{s \rightsquigarrow t}$

$\langle t_1, t_2\rangle.1 \rightsquigarrow t_1$ (QAR-BETA-PROD1)

$\langle t_1, t_2\rangle.2 \rightsquigarrow t_2$ (QAR-BETA-PROD2)

$$\frac{t \rightsquigarrow t'}{t.1 \rightsquigarrow t'.1}$$ (QAR-PROJ1)

$$\frac{t \rightsquigarrow t'}{t.2 \rightsquigarrow t'.2}$$ (QAR-PROJ2)

Algorithmic term equivalence $\boxed{\Gamma \vdash s \Leftrightarrow t : T}$

$$\frac{\Gamma \vdash s.1 \Leftrightarrow t.1 : T_1 \qquad \Gamma \vdash s.2 \Leftrightarrow t.2 : T_2}{\Gamma \vdash s \Leftrightarrow t : T_1 \times T_2}$$ (QAT-PROD)

Algorithmic path equivalence $\boxed{\Gamma \vdash p \leftrightarrow q : T}$

$$\frac{\Gamma \vdash p \leftrightarrow q : T_1 \times T_2}{\Gamma \vdash p.1 \leftrightarrow q.1 : T_1}$$ (QAP-PROJ1)

$$\frac{\Gamma \vdash p \leftrightarrow q : T_1 \times T_2}{\Gamma \vdash p.2 \leftrightarrow q.2 : T_2}$$ (QAP-PROJ2)

Figure 2-6: Equivalence algorithm for $\lambda_{\rightarrow \times \mathrm{b}1}$

of logical relation to show the completeness of an equivalence algorithm for a language with singleton kinds. (Singleton kinds arise from a form of type definitions, and are discussed in §10.3).

A broader survey of applications of logical relations appears in Mitchell (1996b).

Recommend some reading on modal logic and Kripke models.

3 *Typed Operational Reasoning*

By Andrew Pitts

The aim of this chapter is to explain, by example, some methods for reasoning about equivalence of programs based directly upon a type system and an operational semantics for the programming language in question. We will concentrate on methods for reasoning about equivalence of representations of abstract data types. This provides an excellent example, because it is easy to appreciate why such methods are useful and at the same time it is possible to give examples showing that non-trivial problems have to be solved to get a sound reasoning principle in the presence of non-termination and recursion. Rather than just treat abstract data types, we will cover full existential types, using a programming language combining a pure fragment of ML (including records and recursive functions) with second order lambda calculus.

3.1 Introduction

Type systems involving existentially quantified type variables provide a useful foundation for explaining and relating various features of programming languages to do with information hiding. For example, the paper (Mitchell and Plotkin, 1988a) popularised the idea that abstract data type declarations can be modelled by values of existential types; similarly, type-theoretic research into the foundations of object-oriented programming has made use of existential types, together with various combinations of function, record, and recursive types, to model objects and classes. We refer the reader to TAPL Chapter 24 for an introduction to the use of existential types for classifying forms of information hiding.

To establish the properties of such type-theoretic interpretations of information hiding requires a theory of semantic equivalence for expressions of

existential type. Methods involving type-indexed families of *relations* between between expressions have proved very useful in this respect. Study of relational properties of typed calculi goes back to the 'logical relations' for simply typed lambda calculus of (Plotkin, 1973; Statman, 1985a) and the notion of 'relational parametricity' for polymorphic types in (Reynolds, 1983a). More relevant is Mitchell's principle for establishing the denotational equivalence of programs involving higher order functions and different implementations of an abstract datatype, in terms of the existence of a 'simulation' relation between the implementations (Mitchell, 1991a). This principle was extended by Plotkin and Abadi to encompass all the (possibly impredicative) existential types of the Girard-Reynolds polymorphic lambda calculus in (Plotkin and Abadi, 1993). Theorem 7 in that paper shows that the principle gives a necessary and sufficient condition for equality at existential type in any model of the Plotkin-Abadi logic for parametric polymorphism.

One feature of the works mentioned above is that they develop proof principles for *denotational* models of programming languages. The relevance of such principles to the operational behaviour of programs relies upon 'goodness of fit' results (some published, some not) connecting operational and denotational semantics. Another feature of the above works is that they do not treat the use of general recursive definitions; and so the languages considered are not Turing powerful. It is folklore that a proof principle for denotational equality at existential type, phrased in terms of the existence of certain simulation relations, is still valid in the presence of recursively defined functions of higher type, provided one imposes some 'admissibility' conditions on the notion of relation. In fact using techniques for defining operationally-based logical relations developed in (Pitts, 2000), we will see in this chapter that suitable admissibility conditions for relations and an associated proof principle for operational equivalence at existential type can be phrased directly, and quite simply, in terms of the syntax and operational semantics of a programming language combining existential types with recursively defined, higher-order functions. The programming language we work with combines a pure fragment of ML (including records and recursive functions) with the polymorphic lambda calculus of Girard and Reynolds (Girard, 1972a; Reynolds, 1974a). Of course, the ability to define functions by unrestricted fixpoint recursion necessarily entails the presence of non-terminating computations. In contrast to the result of Plotkin and Abadi mentioned above (Plotkin and Abadi, 1993, Theorem 7), it turns out that in the presence of non-termination, the existence of a simulation relation is merely a sufficient, but not a necessary condition for operational equivalence at existential type (see Example 3.6.5 in §3.6).

3.2 Motivating Examples

In this section we motivate the use of logical relations for reasoning about existential types by giving some examples.

Before doing that, let us recall the syntax for expressions involving existentially quantified type variables from TAPL Chapter 24. If T is a type expression and X is a type variable, then we write {∃X, T} for the corresponding existentially quantified type. Free occurrences of X in T become bound in this type expression. We write [X ↦ S]T for the result of substituting a type S for all free occurrences of X in T, renaming bound type variables as necessary to avoid capture.[1] It t is a term of type [X ↦ S]T, then we can 'pack' the type S and the term t together to get a term

$$\{*S, t\} \text{ as } \{\exists X, T\} \tag{3.1}$$

of the indicated existential type. To eliminate such terms we use the form

$$\text{let } \{*X, x\} = t_1 \text{ in } t_2 \tag{3.2}$$

This is a binding construct: free occurrences of the type variable X and the value variable x in t_2 become bound in the term. The typing of such terms goes as follows:

> if t_1 has type {∃X, T} and t_2 has type T_2 when we assume the variable x has type T, then *provided* X *does not occur free in* T_2, we can conclude that the term in (3.2) has type T_2.

(Such rules are better presented symbolically, but we postpone doing that until giving a formal definition of the language we will be using, in the next section.) The italicised restriction on free occurrences of X in T_2 in the above rule is what distinguishes an existential type from a type-indexed dependent sum, where there is free access both to the type component as well as the term component of a 'packed' term: see (Mitchell and Plotkin, 1988a, p 474 *et seq*) for a discussion of this point.

Since we wish to consider existential types in the context of an ML-like language, we adopt an eager strategy for evaluating expressions like (3.1) and (3.2). Thus to evaluate the first, one evaluates t to canonical form, v say, and returns the canonical form {*S, v} as {∃X, T}; to evaluate the second, one evaluates t_1 to canonical form, {*S, v} as {∃X, T} say, and then evaluates [X ↦ S][x ↦ v]t_2.

1. Throughout this paper we will always identify expressions, be they types or terms, up to renaming of bound variables.

3.2.1 EXAMPLE: Consider the existentially quantified record type

```
type Cell = {∃X, {mk:X, inc:X → X, get:X → Int}}
```

where `Int` is a type of integers. Values of type `Cell` consist of some type together with values of the appropriate types implementing `mk`, `inc`, and `get`. For example

```
val cellplus = {*Int, {mk  = 0,
                       inc = λx:Int.x+1,
                       get = λx:Int.x   } as Cell
```

and

```
val cellminus = {*Int, {mk  = 0,
                        inc = λx:Int.x-1,
                        get = λx:Int.0-x } as Cell
```

are both values of type `Cell`. The terms

```
let {*X,x} = cellplus  in x.get(x.inc(x.mk))
let {*X,x} = cellminus in x.get(x.inc(x.mk))
```

(where we use the syntax `r.f` for selecting field `f` of record `r`) are both terms of type `Int` which evaluate to `1`. By contrast, of the terms

```
let {*X,x} = cellplus  in x.get(x.inc(1))
let {*X,x} = cellminus in x.get(x.inc(1))
```

the first evaluates to `2`, whereas the second evaluates to `0`; but in this case neither term is well-typed. Indeed, it is the case that *any* well-typed closed term involving occurrences of the term `cellplus` will exhibit precisely the same evaluation behaviour if we replace those occurrences by `cellminus`. In other words, `cellplus` and `cellminus` are equivalent in the following sense. □

3.2.2 DEFINITION [CONTEXTUAL EQUIVALENCE, INFORMALLY]: We write

$$t_1 =_{ctx} t_2 : T$$

to indicate that two terms t_1 and t_2 of the same type T are *contextually equivalent*. By definition, this means that for all well-typed terms $t[t_1]$ containing instances of t_1, if $t[t_2]$ is the term obtained by replacing those instances by t_2, then $t[t_1]$ and $t[t_2]$ give exactly the same observable results when evaluated. □

This notion of program equivalence assumes we have already fixed upon a definition of the 'observable results' of evaluating terms. It also presupposes that the meaning of a well-typed term should only depend upon the final result (if any) of evaluating it. This is reasonable for deterministic and non-interactive programming. Certainly, contextual equivalence is a widely used notion of program equivalence in the literature and it is the one we adopt here.

For the terms in Example 3.2.1, it is the case that

$$\texttt{cellplus} =_{\text{ctx}} \texttt{cellminus:Cell} \tag{3.3}$$

but the quantification over all possible contexts $\texttt{t}[-]$ in the definition of $=_{\text{ctx}}$ makes a direct proof of this and similar facts rather difficult. Thus one is led to ask whether there are proof principles for contextual equivalence that make proving such equivalences at existential types more tractable. Since values $\texttt{\{*S,v\} as \{}\exists\texttt{X,T\}}$ of a given existential type $\texttt{\{}\exists\texttt{X,T\}}$ are specified by pairs of data $\texttt{S}$ and $\texttt{v}$, as a first stab at such a proof principle one might try componentwise equality. Equality in the second component will of course mean contextual equivalence; but in the first component, where the expressions involved are types, what should equality mean? If we take it to mean syntactic identity up to α-conversion, $=_\alpha$, we obtain the following proof principle.

3.2.3 PRINCIPLE [EXTENSIONALITY FOR $\exists$-TYPES, VERSION I]: For each existential type $\texttt{E} \stackrel{\text{def}}{=} \texttt{\{}\exists\texttt{X,T\}}$, types $\texttt{T}_1$, $\texttt{T}_2$, and values $\texttt{v}_1$, $\texttt{v}_2$, if $\texttt{T}_1 =_\alpha \texttt{T}_2$ and $\texttt{v}_1 =_{\text{ctx}} \texttt{v}_2\texttt{:[X} \mapsto \texttt{T}_2\texttt{]T}$, then $\texttt{(\{*T}_1\texttt{,v}_1\texttt{\} as E)} =_{\text{ctx}} \texttt{(\{*T}_2\texttt{,v}_2\texttt{\} as E):\{}\exists\texttt{X,T\}}$. □

The hypotheses of Principle 3.2.3 are far too strong to make it very useful. For example, it cannot be used to prove (3.3), since in this case $\texttt{T}_1 =_\alpha \texttt{Int} =_\alpha \texttt{T}_2$, but

```
val v1 = {mk=0, inc=λx:Int.x+1, get=λx:Int.x}
```

and

```
val v2 = {mk=0, inc=λx:Int.x-1, get=λx:Int.0-x}
```

are clearly not contextually equivalent values of the record type

```
{mk:Int,inc:Int→Int,get:Int→Int}
```

(for example, we get different integers when evaluating $\texttt{t}[\texttt{v}_1]$ and $\texttt{t}[\texttt{v}_2]$ when $\texttt{t}[-]$ is $\texttt{(-.inc)0}$). However, they do become contextually equivalent if in the second term we use a version of integers in which the roles of positive

and negative are reversed. Such 'integers' are of course in bijection with the usual ones and this leads us to our second version of an extensionality principle for $\exists$-types—in which the use of α-equivalence as the notion of type equality is replaced by the more flexible one of *bijection*.

3.2.4 PRINCIPLE [EXTENSIONALITY FOR $\exists$-TYPES, VERSION II]: For each existential type $\mathtt{E} \stackrel{\text{def}}{=} \{\exists \mathtt{X}, \mathtt{T}\}$, types $\mathtt{T}_1$, $\mathtt{T}_2$, and values $\mathtt{v}_1$, $\mathtt{v}_2$, if there is a *bijection* $\mathtt{i} : \mathtt{T}_1 \cong \mathtt{T}_2$ such that $\mathtt{T[i]}(\mathtt{v}_1) =_{\text{ctx}} \mathtt{v}_2 : [\mathtt{X} \mapsto \mathtt{T}_2]\mathtt{T}$, then

$$(\{*\mathtt{T}_1, \mathtt{v}_1\} \text{ as } \mathtt{E}) =_{\text{ctx}} (\{*\mathtt{T}_2, \mathtt{v}_2\} \text{ as } \mathtt{E}) : \{\exists \mathtt{X}, \mathtt{T}\}.$$

Here a bijection $\mathtt{i} : \mathtt{T}_1 \cong \mathtt{T}_2$ means a closed term $\mathtt{i} : \mathtt{T}_1 \to \mathtt{T}_2$ for which there is a closed term $\mathtt{j} : \mathtt{T}_2 \to \mathtt{T}_1$ which is a two-sided inverse up to contextual equivalence: $\mathtt{j}(\mathtt{i}\ \mathtt{x}_1) =_{\text{ctx}} \mathtt{x}_1 : \mathtt{T}_1$ and $\mathtt{i}(\mathtt{j}\ \mathtt{x}_2) =_{\text{ctx}} \mathtt{x}_2 : \mathtt{T}_2$. Then given a type $\mathtt{T}$, possibly containing free occurrences of a type variable $\mathtt{X}$, one can define an induced bijection $\mathtt{T[i]} : [\mathtt{X} \mapsto \mathtt{T}_1]\mathtt{T} \cong [\mathtt{X} \mapsto \mathtt{T}_2]\mathtt{T}$ (with inverse $\mathtt{T[j]}$). For example, if $\mathtt{T}$ is the type

```
{mk:X, inc:X→X, get:X→Int}
```

then $\mathtt{T[i]}$ is

```
λx:{mk:T₁, inc:T₁→T₁, get:T₁→Int}.
  { mk  = i(x.mk),
    inc = λx₂:T₂.i(x.inc(j x₂)),
    get = λx₂:T₂.x.get(j x₂))}
```

and $\mathtt{T[j]}$ is

```
λx:{mk:T₂, inc:T₂→T₂, get:T₂→Int}.
  { mk = j(x.mk),
   inc = λx₁:T₁.j(x.inc(i x₁)),
   get = λx₁:T₁.x.get(i x₁))}
```

(In general, if $\mathtt{T}$ is a simple type then the definition of $\mathtt{T[i]}$ and $\mathtt{T[j]}$ can be done by induction on the structure of $\mathtt{T}$; for recursively defined types, the definition of the induced bijection is not so straightforward.) □

We can use this second version of the extensionality principle for $\exists$-types to prove the contextual equivalence in (3.3), using the bijection

$$\mathtt{i} \stackrel{\text{def}}{=} (\lambda \mathtt{x} : \mathtt{Int}.\, 0 - \mathtt{x}) : \mathtt{Int} \cong \mathtt{Int}.$$

This does indeed satisfy $\mathtt{T[i]}(\mathtt{v}_1) =_{\text{ctx}} \mathtt{v}_2 : \mathtt{Int}$ when $\mathtt{v}_1$, $\mathtt{v}_2$ and $\mathtt{T}$ are defined as above. (Of course these contextual equivalences, and indeed the fact

that this particular term `i` is a bijection, all require proof; but the methods developed in this chapter render this straightforward.) However, the use of bijections between types is still too restrictive for proving many common examples of contextual equivalence of abstract datatype implementations, such as the following.

3.2.5 EXAMPLE: Consider the existentially quantified record type

```
type Smph = {∃X, {bit:X, flip:X→X, read:X→Bool}}
```

(where `Bool` is a type of booleans) and the following terms of type `Smph`

```
val smph1 =
    {*Bool, {bit  = true
             flip = λx:Bool.not x,
             read = λx:Int.x      } as Smph;
val smph2 =
    {*Int,  {bit  = 1,
             flip = λx:Int.0-2*x,
             read = λx:Int.x >= 0} as Smph
```

There is no bijection $\texttt{Bool} \cong \texttt{Int}$, so one cannot use Principle 3.2.4 to prove

$$\texttt{smph}_1 =_{\text{ctx}} \texttt{smph}_2 : \texttt{Smph}$$

Nevertheless, this contextual equivalence does hold. An informal argument for this makes use of the following relation $r : \texttt{Bool} \leftrightarrow \texttt{Int}$ between values of type `Bool` and of type `Int`.

$$r \stackrel{\text{def}}{=} \{(\texttt{true}, m) \mid m = (-2)^n \text{ for some even } n \geq 0\} \cup \{(\texttt{false}, m) \mid m = (-2)^n \text{ for some odd } n \geq 0\}$$

Writing s_1 and s_2 for the second components of $\texttt{smph}_1$ and $\texttt{smph}_2$, note that

- $s_1\texttt{.bit}$ evaluates to `true`, $s_2\texttt{.bit}$ evaluates to `1` and $(\texttt{true}, 1) \in r$;
- if $(t_1, t_2) \in r$, then $(s_1\texttt{.flip})\,t_1$ and $(s_2\texttt{.flip})\,t_2$ evaluate to a pair of values which are again r-related;
- if $(t_1, t_2) \in r$, then $(s_1\texttt{.read})\,t_1$ and $(s_2\texttt{.read})\,t_2$ evaluate to the same boolean value.

Then the informal argument for the contextual equivalence goes as follows: "any context $t[-]$ which is well-typed whenever its hole $-$ is filled with a term of type `Smph` can only make use of a term placed in its hole by opening it as an abstract pair `{X, x}` and applying the methods `bit`, `flip`, and `read` in some combination; therefore the above observations about r are enough to show that $t[\texttt{smph}_1]$ and $t[\texttt{smph}_2]$ always have the same evaluation behaviour." □

The validity of this informal argument and in particular the assumptions it makes about the way a context can 'use' its hole are far from immediate and need formal justification. Leaving that for later, at least we can state the relational principle a bit more precisely.

3.2.6 PRINCIPLE [EXTENSIONALITY FOR ∃-TYPES, VERSION III]: For each existential type $\mathtt{E} \stackrel{\text{def}}{=} \{\exists \mathtt{X}, \mathtt{T}\}$, types $\mathtt{T}_1$, $\mathtt{T}_2$, and values $\mathtt{v}_1$, $\mathtt{v}_2$, if there is a *relation* $\mathtt{r} : \mathtt{T}_1 \leftrightarrow \mathtt{T}_2$ between terms of type $\mathtt{T}_1$ and of type $\mathtt{T}_2$, such that $(\mathtt{v}_1, \mathtt{v}_2) \in \mathtt{T}[\mathtt{r}]$, then $(\{*\mathtt{T}_1, \mathtt{v}_1\}\ \mathtt{as}\ \mathtt{E}) =_{\text{ctx}} (\{*\mathtt{T}_2, \mathtt{v}_2\}\ \mathtt{as}\ \mathtt{E}) : \{\exists \mathtt{X}, \mathtt{T}\}$. □

Evidently this principle presupposes the existence of an 'action' of types on term-relations, sending relations $\mathtt{r} : \mathtt{T}_1 \leftrightarrow \mathtt{T}_2$ to relations $\mathtt{T}[\mathtt{r}] : [\mathtt{X} \mapsto \mathtt{T}_1]\mathtt{T} \leftrightarrow [\mathtt{X} \mapsto \mathtt{T}_1]\mathtt{T}$ and with certain properties. It is the definition of this action which is at the heart of the matter. It has to be phrased with some care in order for the above extensionality principle to be valid for languages involving non-termination of evaluation (through the presence of fixpoint recursion for example). We will give a precise definition in §3.5 (Definition 3.5.6) for a language combining impredicative polymorphism with fixpoint recursion at the level of terms, that we introduce in the next section. How best to define such relational actions in the presence of recursion at the level of types is still a matter for research (see Exercise 3.7.1).

3.2.7 NOTE: Principle 3.2.4 generalises Principle 3.2.3, because if $\mathtt{T}_1 =_\alpha \mathtt{T}_2$, then the identity function $\mathtt{i} \stackrel{\text{def}}{=} \lambda \mathtt{x} : \mathtt{T}_1 . \mathtt{x}$ is a bijection $\mathtt{T}_1 \cong \mathtt{T}_2$ satisfying

$$(\mathtt{T}[\mathtt{i}]\ \mathtt{v}) =_{\text{ctx}} \mathtt{v} \quad \text{(for any } \mathtt{v})$$

so that $\mathtt{v}_1 =_{\text{ctx}} \mathtt{v}_2$ implies $(\mathtt{T}[\mathtt{i}]\ \mathtt{v}_1) =_{\text{ctx}} \mathtt{v}_2$. Principle 3.2.6 generalises Principle 3.2.4, because each bijection $\mathtt{i} : \mathtt{T}_1 \cong \mathtt{T}_2$ can be replaced by its *graph*

$$\mathtt{r}_\mathtt{i} \stackrel{\text{def}}{=} \{(\mathtt{u}_1, \mathtt{u}_2) \mid \mathtt{i}\ \mathtt{u}_1 =_{\text{ctx}} \mathtt{u}_2\}$$

which in fact has the property that

$$(\mathtt{v}_1, \mathtt{v}_2) \in \mathtt{T}[\mathtt{r}_\mathtt{i}] \quad \text{iff} \quad (\mathtt{T}[\mathtt{i}]\ \mathtt{v}_1) =_{\text{ctx}} \mathtt{v}_2 : [\mathtt{X} \mapsto \mathtt{T}_2]\mathtt{T}.$$

□

As mentioned in the Introduction, Principle 3.2.6 is an operational generalisation of similar principles for the denotational semantics of abstract datatypes over simply typed lambda calculus (Mitchell, 1991a) and relationally parametric models of the polymorphic lambda calculus (Plotkin and Abadi, 1993). It permits many examples of contextual equivalence at ∃-types to be proved rather easily. Nevertheless, we will see in §3.6 that in the presence of non-termination it is incomplete (Example 3.6.5).

3.3 The Language

In this section we define a small programming language that we will use in the rest of the chapter. It combines Girard's *System F* (Girard, 1972a) (in other words Reynolds' *polymorphic lambda calculus* (Reynolds, 1974a)) with an explicitly typed fragment of ML (Milner, Tofte, Harper, and MacQueen, 1997a) containing records and recursive functions. We will call the language F_{ML}. Its syntax and type system are specified in Figure 3-1 and its operational semantics in Figure 3-2.

3.3.1 Syntax

In Figure 3-1, `X` and `x` respectively range over disjoint countably infinite sets of *type variables* and *value variables*; `l` ranges over a countably infinite set of *field labels*; `c` ranges over the constants `true`, `false` and `n` (for $n \in \mathbb{Z}$); `Gnd` is either the type of booleans `Bool` or the type of integers `Int`; and `op` ranges over a fixed collection of arithmetic and boolean operations (such as `+`, `=`, `not`, etc).

To simplify the definition of the language's operational semantics we employ the now quite common device of using a 'reduced' syntax for terms in which all sequential evaluation has to be coded via `let`-expressions. For example, the general form of (left-to-right, call-by-value) function application is coded by

$$\texttt{t}_1\ \texttt{t}_2 \stackrel{\text{def}}{=} \texttt{let x}_1\texttt{=t}_1\ \texttt{in (let x}_2\texttt{=t}_2\ \texttt{in x}_1\ \texttt{x}_2\texttt{).} \tag{3.4}$$

As a further simplification, function abstraction and recursive function declaration have been rolled into the one form `fun f(x:T1) = t:T2`, which corresponds to the expressions

	`let fun f x = t in fn(x:T1)⇒(t:T2) end`	in Standard ML
or	`let rec f x = t in fun(x : T1) → (t : T2)`	in Ocaml.

Ordinary function abstraction can be coded as

$$\lambda\texttt{x:T}_1\texttt{.t} \stackrel{\text{def}}{=} \texttt{fun f(x:T}_1\texttt{) = t:T}_2 \tag{3.5}$$

where `f` does not occur freely in `t` (and $\texttt{T}_2$ is the type of `t`, given $\texttt{f:T}_1\texttt{→T}_2$ and $\texttt{x:T}_1$). In what follows we shall use the abbreviations (3.4) and (3.5) without further comment. We shall also use infix notation for application of constants that stand for arithmetic or boolean operators such as `+`, `=`, etc.

One slightly subtle aspect of F_{ML} compared with System F is that restricting the operation of polymorphic generalisation $\lambda\texttt{X}.(-)$ to apply only to values is a real restriction: one cannot define $\lambda\texttt{X.t}$ to be `let x=t in` $\lambda\texttt{X.x}$, since

Syntax

t ::=		*terms:*
	v	*value*
	if v then t else t	*conditional*
	op(v$_i$ $^{i\in 1..n}$)	*operation*
	v v	*application*
	v.l	*projection*
	v T	*type application*
	let {*X,x}=v in t	*unpacking*
	let x=t in t	*sequencing*
v ::=		*values:*
	x	*value variable*
	c	*constant*
	fun x(x:T)=t:T	*recursive function*
	{l$_i$=v$_i$ $^{i\in 1..n}$}	*record value*
	λX.v	*type abstraction*
	{*T,v} as {∃X,T}	*package value*
T ::=		*types:*
	X	*type variable*
	Gnd	*ground type*
	T→T	*function type*
	{l$_i$:T$_i$ $^{i\in 1..n}$}	*record type*
	∀X.T	*universally quantified type*
	{∃X,T}	*existentially quantified type*
Γ ::=		*typing contexts:*
	∅	*empty context*
	Γ, x:T	*non-empty context*

Typing terms $\boxed{\Gamma \vdash t : T}$

$$\frac{x:T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-Var)}$$

$$\Gamma \vdash c : \mathit{Typeof}(c) \quad \text{(T-Const)}$$

$$\frac{\Gamma, f:T_1\to T_2, x:T_1 \vdash t : T_2}{\Gamma \vdash \mathtt{fun}\, f(x:T_1)=t:T_2 : T_1\to T_2} \quad \text{(T-Fun)}$$

$$\frac{\text{for each } i \quad \Gamma \vdash v_i : T_i}{\Gamma \vdash \{l_i=v_i\ ^{i\in 1..n}\} : \{l_i:T_i\ ^{i\in 1..n}\}} \quad \text{(T-Rcd)}$$

$$\frac{\Gamma, X \vdash v : T \qquad X \notin \mathit{ftv}(\Gamma)}{\Gamma \vdash \lambda X.v : \forall X.T} \quad \text{(T-Tabs)}$$

$$\frac{\Gamma \vdash v_1 : [X \mapsto T_1]T}{\Gamma \vdash \{*T_1, v_1\}\ \mathtt{as}\ \{\exists X, T\} : \{\exists X, T\}} \quad \text{(T-Pack)}$$

$$\frac{\Gamma \vdash v : \mathtt{Bool} \qquad \Gamma \vdash t_1 : T \qquad \Gamma \vdash t_2 : T}{\Gamma \vdash \mathtt{if}\, v\, \mathtt{then}\, t_1\, \mathtt{else}\, t_2 : T} \quad \text{(T-If)}$$

$$\frac{\mathtt{op}:\mathtt{Gnd}_1,\ldots,\mathtt{Gnd}_n\to\mathtt{Gnd} \qquad \text{for each } i \quad \Gamma \vdash v_i : \mathtt{Gnd}_i}{\Gamma \vdash \mathtt{op}(v_i\ ^{i\in 1..n}) : \mathtt{Gnd}} \quad \text{(T-Op)}$$

$$\frac{\Gamma \vdash v_1 : T_1\to T_2 \qquad \Gamma \vdash v_2 : T_1}{\Gamma \vdash v_1\, v_2 : T_2} \quad \text{(T-App)}$$

$$\frac{\Gamma \vdash v : \{l_i:T_i\ ^{i\in 1..n}\}}{\Gamma \vdash v.l_j : T_j} \quad \text{(T-Proj)}$$

$$\frac{\Gamma \vdash v : \forall X.T}{\Gamma \vdash v\, T_1 : [X \mapsto T_1]T} \quad \text{(T-TApp)}$$

$$\frac{\Gamma \vdash v : \{\exists X, T\} \qquad \Gamma, X, x:T \vdash t : T_1 \qquad X \notin \mathit{ftv}(\Gamma, T_1)}{\Gamma \vdash \mathtt{let}\, \{*X, x\}=v\, \mathtt{in}\, t : T_1} \quad \text{(T-Unpack)}$$

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, x:T_1 \vdash t_2 : T_2}{\Gamma \vdash \mathtt{let}\, x=t_1\, \mathtt{in}\, t_2 : T_2} \quad \text{(T-Seq)}$$

Figure 3-1: F_{ML} syntax and typing

the latter will in general be an ill-typed term. In effect we are imposing an explicitly typed version of the 'value-restriction' on `let`-bound polymorphism that occurs in the 1996 revision of Standard ML (Milner, Tofte, Harper, and MacQueen, 1997a). It is convenient to do so, because then we do not have to consider evaluating under a type abstraction $\lambda X.(-)$ and hence can restrict attention to the evaluation of closed terms of *closed* type.

3.3.1 NOTE [BINDING AND FREE VARIABLES]: The constructions

$$\begin{array}{ll} & \texttt{let}\ \{*X, x\}=v\ \texttt{in}\ (-) \\ & \texttt{let}\ x=t\ \texttt{in}\ (-) \\ & \texttt{fun}\ f\,(x:T_1)=(-:T_2) \\ & \lambda X.(-) \\ & \forall X.(-) \\ & \{\exists X, (-)\} \\ \text{and} & S \circ (x.(-)) \end{array}$$

are binders and we will identify expressions up to renaming of bound value and type variables. We write $ftv(E)$ for the finite set of free type variables of the expression E (a type, a term, or a frame stack); and $fv(E)$ for the finite set of free value variables of an expression E (a term, or a frame stack, but not a type, since types do not contain occurrences of value variables). The result of capture-avoiding substitution of a type T for all free occurrences of a type variable X in E (a type, a term, or a frame stack) will be denoted $[X \mapsto T]E$. Similarly, $[x \mapsto v]E$ denotes the result of capture-avoiding substitution of a value v for all free occurrences of the value variable x in a term or frame stack E. Note that as their name suggests, value variables stand for unknown *values*—the substitution of a non-value term for a variable makes no sense syntactically, in that it may result in an ill-formed expression.

As usual, a type is closed if it has no free type variables; however, we will say that a term or frame stack is *closed* if it has no free value variables, whether or not it also has free type variables. □

3.3.2 Operational semantics

Although we do not do so, the operational semantics of F_{ML} could be specified in the style of the Definition of Standard ML (Milner, Tofte, Harper, and MacQueen, 1997a) as a syntax-directed, inductively defined relation between terms and values (parameterised by an environment assigning values to value variables—except it is technically convenient to 'substitute-in' the value environment parameter and only consider the evaluation relation be-

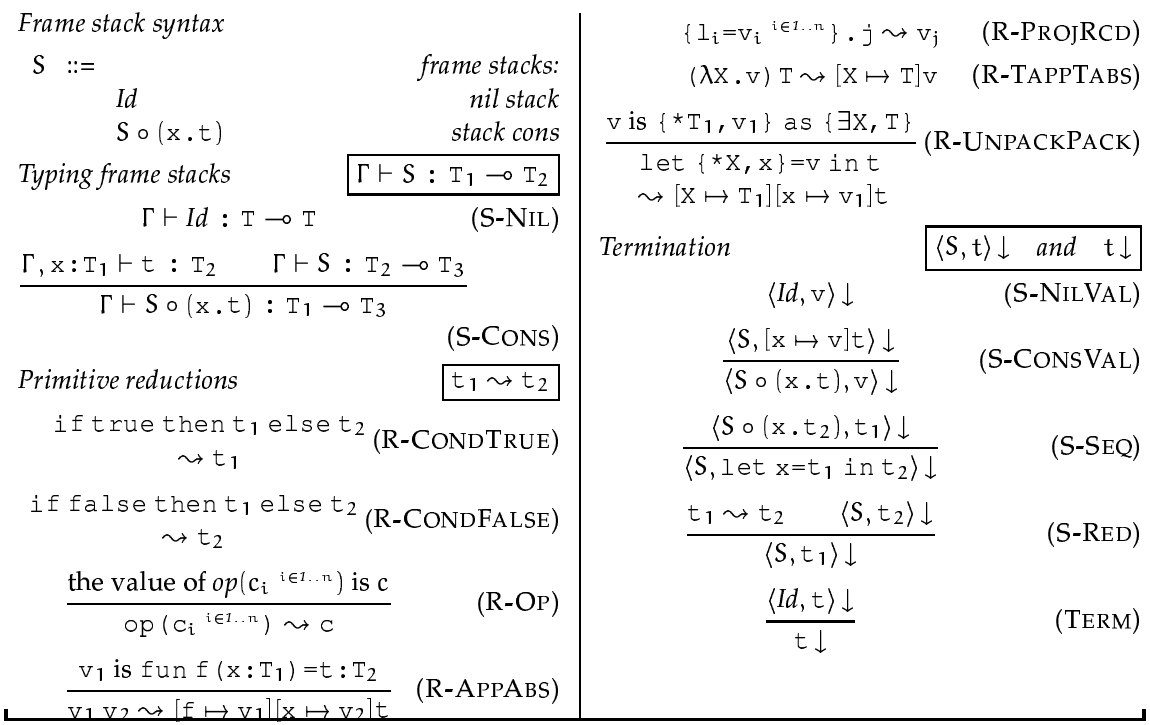


Figure 3-2: F_{ML} operational semantics

tween *closed* terms and values). Here we are interested primarily in the notion of contextual equivalence (Definition 3.2.2) that this evaluation relation determines by observing the results of evaluating terms in context. Because evaluation in F_{ML} is strict and the language has enough destructors of ground type, it turns out that the only result of evaluating a term that it is necessary to observe in order to determine contextual equivalence is *termination*, i.e. whether the term evaluates to some value, we care not which. Therefore for our purposes, it suffices to define the termination relation, $\mathtt{t}\downarrow$, for F_{ML}. This is done in Figure 3-2, using an auxiliary notion of *frame stack* that allows us to give a nice, syntax-directed definition of termination.

Frame stacks are finite lists of individual 'evaluation frames'. They provide a convenient syntax for the notion of *evaluation context* $\mathtt{E}[-]$ popularised by Felleisen *et al* (Felleisen and Hieb, 1992; Wright and Felleisen, 1994a). Every closed term can be decomposed uniquely as $\mathtt{E}[\mathtt{t}]$ where the evaluation context $\mathtt{E}[-]$ is a context with a unique hole $(-)$ occurring in the place where

the next step of evaluation (called a *primitive reduction* in Figure 3-2) will take place. With F_{ML}'s reduced syntax, such evaluation contexts turn out to be just nested sequences of the `let`-construct

$$\mathtt{E}[-] = \mathtt{let\ x_1{=}(\ldots(let\ x_n{=}(-)\ in\ t_n)\ldots)\ in\ t_1}$$

and the corresponding frame stack

$$S = \mathit{Id} \circ (\mathtt{x_1.t_1}) \circ \cdots \circ (\mathtt{x_n.t_n})$$

records this sequence as a list of *evaluation frames*, $\mathtt{x_i.t_i}$ (with free occurrences of $\mathtt{x_i}$ in $\mathtt{t_i}$ being bound in $\mathtt{x_i.t_i}$). Under this correspondence it can be shown that $\mathtt{E[t]}$ evaluates to some value in the standard evaluation-style (or 'big-step') structural operational semantics if and only if $\langle S, \mathtt{t}\rangle\downarrow$ holds, for the relation $\langle -, -\rangle\downarrow$ defined in Figure 3-2.

3.3.2 EXERCISE [RECOMMENDED, ★★]: Define a relation $\langle S_1, \mathtt{t_1}\rangle \longrightarrow \langle S_2, \mathtt{t_2}\rangle$ by cases according to the structure of the term $\mathtt{t_1}$ and the frame stack S_1, as follows:

- $\langle S \circ (\mathtt{x.t}), \mathtt{v}\rangle \longrightarrow \langle S, [\mathtt{x} \mapsto \mathtt{v}]\mathtt{t}\rangle$
- $\langle S, \mathtt{let\ x{=}t_1\ in\ t_2}\rangle \longrightarrow \langle S \circ (\mathtt{x.t_2}), \mathtt{t_1}\rangle$
- $\langle S, \mathtt{t_1}\rangle \longrightarrow \langle S, \mathtt{t_2}\rangle$, if $\mathtt{t_1} \rightsquigarrow \mathtt{t_2}$.

Show that

$$\langle S'@S, \mathtt{t}\rangle\downarrow \quad \text{iff} \quad (\exists \mathtt{v})\ \langle S, \mathtt{t}\rangle \longrightarrow^* \langle \mathit{Id}, \mathtt{v}\rangle\ \&\ \langle S', \mathtt{v}\rangle\downarrow \tag{3.6}$$

where as usual $\longrightarrow^*$ denotes the reflexive-transitive closure of the $\longrightarrow$ relation, and $S'@S$ is the frame stack obtained by appending the two lists of evaluation frames S' and S. Deduce that $\mathtt{t}\downarrow$ holds if and only if there is some value $\mathtt{v}$ with $\langle \mathit{Id}, \mathtt{t}\rangle \longrightarrow^* \langle \mathit{Id}, \mathtt{v}\rangle$. □

3.3.3 Typing

We will consider the termination relation only for frame stacks and terms that are *well-typed*. A term $\mathtt{t}$ is well-typed if a typing judgement

$$\Gamma \vdash \mathtt{t} : \mathtt{T} \tag{3.7}$$

can be derived for some type $\mathtt{T}$, where the typing environment

$$\Gamma = \mathtt{X_1}, \ldots, \mathtt{X_m}, \mathtt{x_1} : \mathtt{T_1}, \ldots, \mathtt{x_n} : \mathtt{T_n}$$

contains (at least) the free type and value variables occurring in $\mathtt{t}$ and $\mathtt{T}$. The axioms and rules for inductively generating the valid typing judgements of this form for F_{ML} are all quite standard and are given in Figure 3-1. (We have chosen to include sufficient explicit type information in terms to ensure that for any given Γ and $\mathtt{t}$, there is at most one $\mathtt{T}$ for which (3.7) holds.)

The judgement for typing frame stacks takes the form

$$\Gamma \vdash \mathtt{S} : \mathtt{T}_1 \multimap \mathtt{T}_2 \tag{3.8}$$

where, in terms of the evaluation context corresponding to $\mathtt{S}$, $\mathtt{T}_2$ is the overall type of the context, given that $\mathtt{T}_1$ is the type of the hole. The rules for generating this judgement are given in Figure 3-2. Unlike for terms, we have not included explicit type information in the syntax of frame stacks; for example, *Id* is not tagged with a type. However, it is not hard to see that, given Γ, $\mathtt{S}$, and $\mathtt{T}_1$, there is at most one $\mathtt{T}_2$ for which (3.8) holds. This property is enough for our purposes, since the argument type of a frame stack will always be supplied in any particular situation in which we use it.

3.3.3 EXERCISE [★ ↛]: Write $\Gamma \vdash \langle \mathtt{S}, \mathtt{t} \rangle : \mathtt{T}$ to mean that $\Gamma \vdash \mathtt{S} : \mathtt{T}' \multimap \mathtt{T}$ and $\Gamma \vdash \mathtt{t} : \mathtt{T}'$ hold for some type $\mathtt{T}'$. Show that if $\emptyset \vdash \langle \mathtt{S}_1, \mathtt{t}_1 \rangle : \mathtt{T}$ and $\langle \mathtt{S}_1, \mathtt{t}_1 \rangle \longrightarrow \langle \mathtt{S}_2, \mathtt{t}_2 \rangle$, then $\emptyset \vdash \langle \mathtt{S}_2, \mathtt{t}_2 \rangle : \mathtt{T}$. □

3.3.4 Unwinding recursive functions

In what follows we will need a compactness property of recursively defined functions with respect to the termination relation. This *unwinding property*, as it is called, is a syntactic analogue of the fact that the denotation of such functions is the least upper bound of finite approximations obtained by successively unfolding the definition starting with the totally undefined partial function. (For statements and proofs of analogous properties see for example: (Mason, Smith, and Talcott, 1996, Section 4.3), (Pitts and Stark, 1998, Theorem 3.2), (Birkedal and Harper, 1999, Section 3.1), and (Lassen, 1998, Section 4.5).) The proof of this unwinding property that we give here is made easier by our syntax-directed definition of termination using frame stacks.

3.3.4 THEOREM [UNWINDING]: Given any closed recursive function value

$$\mathtt{F} \stackrel{\text{def}}{=} \mathtt{fun\ f(x:T_1) = u : T_2}$$

define the following abbreviations:

$$\mathtt{F}_0 \stackrel{\text{def}}{=} \mathtt{fun\ f(x:T_1) = (f\ x) : T_2}$$

$$\mathtt{F}_{n+1} \stackrel{\text{def}}{=} \mathtt{fun\ f(x:T_1) = [f \mapsto F_n]u : T_2}$$

Thus $\mathtt{F}_0$ is a closed function value describing a totally undefined partial function of type $\mathtt{T}_1 \to \mathtt{T}_2$ and the $\mathtt{F}_n$ are obtained from this by repeatedly substituting for the the value variable $\mathtt{f}$ in the body $\mathtt{u}$ of the original function value $\mathtt{F}$. (Note that in the definition of $\mathtt{F}_{n+1}$, the outer binding instance of $\mathtt{f}$ is a dummy, since $\mathtt{f}$ does not occur free in $[\mathtt{f} \mapsto \mathtt{F}_n]\mathtt{u}$.) Then for all terms $\mathtt{t}$ containing at most $\mathtt{f}$ free we have

$$[\mathtt{f} \mapsto \mathtt{F}]\mathtt{t}\downarrow \quad \text{iff} \quad (\exists n)\, [\mathtt{f} \mapsto \mathtt{F}_n]\mathtt{t}\downarrow.$$

□

Proof: By definition of the relation $\mathtt{t}\downarrow$ in terms of the relation $\langle S, \mathtt{t}\rangle\downarrow$ (via rule (TERM) in Figure 3-2), it suffices to prove the more general property that for all terms $\mathtt{t}$ and frame stacks S (containing at most $\mathtt{f}$ free) we have

$$\langle [\mathtt{f} \mapsto \mathtt{F}]S, [\mathtt{f} \mapsto \mathtt{F}]\mathtt{t}\rangle\downarrow \quad \text{iff} \quad (\exists n)\, \langle [\mathtt{f} \mapsto \mathtt{F}_n]S, [\mathtt{f} \mapsto \mathtt{F}_n]\mathtt{t}\rangle\downarrow \tag{3.9}$$

The proof of (3.9) is via a series of straightforward, if somewhat tedious, inductions that we leave as an exercise. □

3.3.5 EXERCISE [★★★ ↛]: This exercise leads you through a proof of (3.9). First prove that

$$\langle [\mathtt{f} \mapsto \mathtt{F}_n]S, [\mathtt{f} \mapsto \mathtt{F}_n]\mathtt{t}\rangle\downarrow \quad \text{implies} \quad \langle [\mathtt{f} \mapsto \mathtt{F}]S, [\mathtt{f} \mapsto \mathtt{F}]\mathtt{t}\rangle\downarrow \tag{3.10}$$

holds for all n, S and $\mathtt{t}$ by induction on the derivation of $\langle [\mathtt{f} \mapsto \mathtt{F}_n]S, [\mathtt{f} \mapsto \mathtt{F}_n]\mathtt{t}\rangle\downarrow$ from the rules in Figure 3-2. Conversely show that

$$\langle [\mathtt{f} \mapsto \mathtt{F}]S, [\mathtt{f} \mapsto \mathtt{F}]\mathtt{t}\rangle\downarrow \quad \text{implies} \quad (\exists n)\, \langle [\mathtt{f} \mapsto \mathtt{F}_n]S, [\mathtt{f} \mapsto \mathtt{F}_n]\mathtt{t}\rangle\downarrow \tag{3.11}$$

holds for all S and $\mathtt{t}$, by induction on the derivation of $\langle [\mathtt{f} \mapsto \mathtt{F}]S, [\mathtt{f} \mapsto \mathtt{F}]\mathtt{t}\rangle\downarrow$ from the rules. To do this, you will first need to prove by induction on n that

$$\langle [\mathtt{f} \mapsto \mathtt{F}_n]S, [\mathtt{f} \mapsto \mathtt{F}_n]\mathtt{t}\rangle\downarrow \quad \text{implies} \quad \langle [\mathtt{f} \mapsto \mathtt{F}_{n+1}]S, [\mathtt{f} \mapsto \mathtt{F}_{n+1}]\mathtt{t}\rangle\downarrow \tag{3.12}$$

holds for all n, S and $\mathtt{t}$; the base case $n = 0$ involves yet another induction, this time over the derivation of $\langle [\mathtt{f} \mapsto \mathtt{F}_0]S, [\mathtt{f} \mapsto \mathtt{F}_0]\mathtt{t}\rangle\downarrow$ from the rules. □

3.4 Contextual Equivalence

Definition 3.2.2 gave an informal definition of the notion of *contextual equivalence* that applies to any (typed) programming language. In giving a precise definition of this notion for the F_{ML} language we will take the more

abstract, relational approach of (Gordon, 1998) and (Lassen, 1998), which de-emphasises the role of program contexts. By doing so, we avoid having to discuss the capture of free value variables by binders upon substitution of a term for the 'hole' in a context. In a nutshell, we will define contextual equivalence to be the largest type-respecting congruence relation between F_{ML} terms that is 'adequate' for observing termination.

3.4.1 DEFINITION: A *type-respecting binary relation* between F_{ML} terms is a set R of quadruples $(\Gamma, \mathtt{t}, \mathtt{t}', \mathtt{T})$ (consisting of a typing context, two terms and a type) satisfying $\Gamma \vdash \mathtt{t} : \mathtt{T}$ and $\Gamma \vdash \mathtt{t}' : \mathtt{T}$. Figure 3-3 defines the properties of *reflexivity*, *symmetry*, *transitivity*, *substitutivity* and *compatibility* for such relations; R has one of these properties if it is closed under the axioms and rules under the corresponding heading in the figure. In these figures, and elsewhere, we write $\Gamma \vdash \mathtt{t}\ \mathsf{R}\ \mathtt{t}' : \mathtt{T}$ instead of $(\Gamma, \mathtt{t}, \mathtt{t}', \mathtt{T}) \in \mathsf{R}$. We say that R is

- an *equivalence relation* if it has the reflexivity, symmetry and transitivity properties;
- a *congruence relation* if it is an equivalence relation with the *substitutivity* and *compatibility* properties;
- *adequate* (for the termination relation $\downarrow$ defined in Figure 3-2) if whenever $\emptyset \vdash \mathtt{t}\ \mathsf{R}\ \mathtt{t}' : \mathtt{T}$ holds, then $\mathtt{t}\downarrow$ holds if and only if $\mathtt{t}'\downarrow$ does.

□

3.4.2 DEFINITION: We will need to use the following constructions on type-respecting binary relations.

(i) The *identity* relation is $Id \stackrel{\text{def}}{=} \{(\Gamma, \mathtt{t}, \mathtt{t}, \mathtt{T}) \mid \Gamma \vdash \mathtt{t} : \mathtt{T}\}$.

(ii) The *reciprocal* of the relation R is $\mathsf{R}^{op} \stackrel{\text{def}}{=} \{(\Gamma, \mathtt{t}', \mathtt{t}, \mathtt{T}) \mid \Gamma \vdash \mathtt{t}\ \mathsf{R}\ \mathtt{t}' : \mathtt{T}\}$.

(iii) The *composition* of relations R_1 and R_2 is
$\mathsf{R}_1 \circ \mathsf{R}_2 \stackrel{\text{def}}{=} \{(\Gamma, \mathtt{t}, \mathtt{t}'', \mathtt{T}) \mid (\exists \mathtt{t}')\ \Gamma \vdash \mathtt{t}\ \mathsf{R}_1\ \mathtt{t}' : \mathtt{T} \text{ and } \Gamma \vdash \mathtt{t}'\ \mathsf{R}_2\ \mathtt{t}'' : \mathtt{T}\}$.

(iv) The *union* of a family $\{\mathsf{R}_i \mid i \in I\}$ of relations is
$\bigcup_{i\in I} \mathsf{R}_i \stackrel{\text{def}}{=} \{(\Gamma, \mathtt{t}, \mathtt{t}', \mathtt{T}) \mid (\exists i \in I)\ \Gamma \vdash \mathtt{t}\ \mathsf{R}_i\ \mathtt{t}' : \mathtt{T}\}$.

(v) The *transitive closure* of the relation R is $\mathsf{R}^+ \stackrel{\text{def}}{=} \bigcup_{i\in\mathbb{N}} \mathsf{R}_i$, where $\mathsf{R}_0 = \mathsf{R}$ and $\mathsf{R}_{i+1} = \mathsf{R} \circ \mathsf{R}_i$.

Reflexivity

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash t \; R \; t : T}$$

Symmetry

$$\frac{\Gamma \vdash t \; R \; t' : T}{\Gamma \vdash t' \; R \; t : T}$$

Transitivity

$$\frac{\Gamma \vdash t \; R \; t' : T \qquad \Gamma \vdash t' \; R \; t'' : T}{\Gamma \vdash t \; R \; t'' : T}$$

Substitutivity

$$\frac{\Gamma \vdash v \; R \; v' : T_1 \qquad \Gamma, x : T_1 \vdash t \; R \; t' : T_2}{\Gamma \vdash [x \mapsto v]t \; R \; [x \mapsto v']t' : T_2}$$

$$\frac{\Gamma, X \vdash t \; R \; t' : T}{\Gamma \vdash [X \mapsto T_1]t \; R \; [X \mapsto T_1]t' : [X \mapsto T_1]T}$$

Compatibility

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x \; R \; x : T}$$

$$\Gamma \vdash c \; R \; c : \mathit{Typeof}(c)$$

$$\frac{\Gamma, f : T_1 \to T_2, x : T_1 \vdash t \; R \; t' : T_2}{\Gamma \vdash \mathtt{fun}\, f(x:T_1) = t : T_2 \; R \; \mathtt{fun}\, f(x:T_1) = t' : T_2 : T_1 \to T_2}$$

$$\frac{\text{for each } i \quad \Gamma \vdash v_i \; R \; v_i' : T_i}{\Gamma \vdash \{l_i = v_i \; {}^{i \in 1..n}\} \; R \; \{l_i = v_i' \; {}^{i \in 1..n}\} : \{l_i : T_i \; {}^{i \in 1..n}\}}$$

$$\frac{\Gamma, X \vdash v \; R \; v' : T \qquad X \notin \mathit{ftv}(\Gamma)}{\Gamma \vdash \lambda X.v \; R \; \lambda X.v' : \forall X.T}$$

$$\frac{\Gamma \vdash v_1 \; R \; v_1' : [X \mapsto T_1]T}{\Gamma \vdash \{*T_1, v_1\} \,\mathtt{as}\, \{\exists X, T\} \; R \; \{*T_1, v_1'\} \,\mathtt{as}\, \{\exists X, T\} : \{\exists X, T\}}$$

$$\frac{\Gamma \vdash v \; R \; v' : \mathtt{Bool} \qquad \Gamma \vdash t_1 \; R \; t_1' : T \qquad \Gamma \vdash t_2 \; R \; t_2' : T}{\Gamma \vdash \mathtt{if}\, v \,\mathtt{then}\, t_1 \,\mathtt{else}\, t_2 \; R \; \mathtt{if}\, v' \,\mathtt{then}\, t_1' \,\mathtt{else}\, t_2' : T}$$

$$\frac{\mathtt{op} : \mathtt{Gnd}_1, \ldots, \mathtt{Gnd}_n \to \mathtt{Gnd} \qquad \text{for each } i \quad \Gamma \vdash v_i \; R \; v_i' : \mathtt{Gnd}_i}{\Gamma \vdash \mathtt{op}(v_i \; {}^{i \in 1..n}) \; R \; \mathtt{op}(v_i' \; {}^{i \in 1..n}) : \mathtt{Gnd}}$$

$$\frac{\Gamma \vdash v_1 \; R \; v_1' : T_1 \to T_2 \qquad \Gamma \vdash v_2 \; R \; v_2' : T_1}{\Gamma \vdash v_1 \, v_2 \; R \; v_1' \, v_2' : T_2}$$

$$\frac{\Gamma \vdash v \; R \; v' : \{l_i : T_i \; {}^{i \in 1..n}\}}{\Gamma \vdash v.l_j \; R \; v'.l_j : T_j}$$

$$\frac{\Gamma \vdash v \; R \; v' : \forall X.T}{\Gamma \vdash v \, T_1 \; R \; v' \, T_1 : [X \mapsto T_1]T}$$

$$\frac{\Gamma \vdash v \; R \; v' : \{\exists X, T\} \qquad \Gamma, X, x : T \vdash t \; R \; t' : T_1 \qquad X \notin \mathit{ftv}(\Gamma, T_1)}{\Gamma \vdash \mathtt{let}\, \{*X, x\} = v \,\mathtt{in}\, t \; R \; \mathtt{let}\, \{*X, x\} = v' \,\mathtt{in}\, t' : T_1}$$

$$\frac{\Gamma \vdash t_1 \; R \; t_1' : T_1 \qquad \Gamma, x : T_1 \vdash t_2 \; R \; t_2' : T_2}{\Gamma \vdash \mathtt{let}\, x = t_1 \,\mathtt{in}\, t_2 \; R \; \mathtt{let}\, x = t_1' \,\mathtt{in}\, t_2' : T_2}$$

Figure 3-3: Properties of a type-respecting relation R between F_{ML} terms

(vi) The *open extension* of the relation R is denoted R° and consists of all quadruples (Γ, t, t', T) such that $\emptyset \vdash \sigma(t) \; R \; \sigma(t') : \sigma(T)$ holds for all Γ-closing substitutions σ. If $\Gamma = X_1, \ldots, X_m, x_1 : T_1, \ldots, x_n : T_n$, then a Γ-*closing substitution* is given by a function $[X_i \mapsto T_i \mid i = 1..m]$ mapping the type variables X_i to closed types T_i, and by a function $[x_j \mapsto v_j \mid j = 1..n]$ mapping the value variables x_j to closed values v_j of appropriate type, namely satisfying $\emptyset \vdash v_j : [X_i \mapsto T_i \mid i = 1..m]T_j$.

(Note that R° only depends on the quadruples of the form $(\emptyset, \mathtt{t}, \mathtt{t}', \mathtt{T})$ in R.)

□

We wish to define contextual equivalence to be the largest adequate congruence relation, but it is not immediately clear why a largest such relation exists. Therefore we give a theorem rather than a definition.

3.4.3 THEOREM [F_{ML} CONTEXTUAL EQUIVALENCE, $=_{\mathrm{ctx}}$]: There is a largest type-respecting binary relation between F_{ML} terms which is a congruence and adequate. We call it *contextual equivalence* and write it $=_{\mathrm{ctx}}$. □

Proof: The proof makes use of the following series of easily verified facts.

(i) The collection of adequate relations is closed under taking unions of non-empty families.

(ii) The identity relation *Id* is an adequate congruence relation.

(iii) Every compatible relation is reflexive, i.e. contains *Id*.

(iv) The collections of compatible relations, substitutive relations and adequate relations is each closed under composition and reciprocation.

(v) If the union of a non-empty family of compatible relations is transitive, it is also compatible; similarly for a family of relations that is reflexive and substitutive.

Let $=_{\mathrm{ctx}}$ be the union of the family of relations which are adequate, compatible and substitutive. By (ii), this family is non-empty and hence by (i) $=_{\mathrm{ctx}}$ is adequate. So it suffices to show that it is a congruence relation. It is certainly reflexive by (ii); and (iv) implies that it is also symmetric and transitive. So it just remains to show that it is compatible and substitutive, and this follows from (v). □

It is not easy to use either the formulation in terms of contexts in Definition 3.2.2 or the more abstract characterisation of Theorem 3.4.3 to prove that a particular pair of terms are contextually equivalent. For example, it is not easy to see from these characterisations that terms with the same immediate reduction behaviour are contextually equivalent. That this is so follows from the coincidence of $=_{\mathrm{ctx}}$ with a notion of equivalence popularised by Mason and Talcott (Mason and Talcott, 1991).

3.4.4 DEFINITION [CIU-EQUIVALENCE, $=_{\text{ciu}}$]: Two closed F_{ML} terms of the same (closed) type are *ciu-equivalent* if they have the termination behaviour when paired with any frame stack (a 'use' of the terms); the relation is extended to open terms via closing substitutions (or 'closed intantiations'—thus we arrive at an explanation of the rather cryptic name for this equivalence).

More formally, we define $=_{\text{ciu}}$ to be the type-respecting relation $\mathtt{R}^\circ$ (using the operation from Definition 3.4.2(vi)), where $\mathtt{R}$ consists of quadruples $(\emptyset, \mathtt{t}, \mathtt{t}', \mathtt{T})$ satisfying $\emptyset \vdash \mathtt{t} : \mathtt{T}$, $\emptyset \vdash \mathtt{t}' : \mathtt{T}$ and for all frame stacks S, $\langle S, \mathtt{t}\rangle\downarrow$ if and only if $\langle S, \mathtt{t}'\rangle\downarrow$. □

3.4.5 THEOREM [CIU THEOREM FOR F_{ML}]: The relations of contextual equivalence and ciu-equivalence coincide. □

Proof: We first show that $=_{\text{ctx}}$ is contained in $=_{\text{ciu}}$. Suppose

$$\Gamma \vdash \mathtt{t} =_{\text{ctx}} \mathtt{t}' : \mathtt{T} \tag{3.13}$$

Since $=_{\text{ctx}}$ satisfies the substitutivity and reflexivity properties from Figure 3-3, it follows that

$$\emptyset \vdash \sigma(\mathtt{t}) =_{\text{ctx}} \sigma(\mathtt{t}') : \sigma(\mathtt{T}) \tag{3.14}$$

holds for any Γ-closing substitution σ. For any frame stack S, it follows easily from the rules in Figure 3-2 that

$$\langle S, \mathtt{t}\rangle\downarrow \quad \text{if and only if} \quad S[\mathtt{t}]\downarrow \tag{3.15}$$

where the term $S[\mathtt{t}]$ is defined by induction on the length of S:

$$\left.\begin{array}{rcl} \mathit{Id}[\mathtt{t}] & \stackrel{\text{def}}{=} & \mathtt{t} \\ S \circ (\mathtt{x}.\mathtt{t}')[\mathtt{t}] & \stackrel{\text{def}}{=} & S[\mathtt{let\ x{=}t\ in\ t}'] \end{array}\right\} \tag{3.16}$$

Since $=_{\text{ctx}}$ satisfies the compatibility (and reflexivity) properties from Figure 3-3, from (3.14) we deduce that $\emptyset \vdash S[\sigma(\mathtt{t})] =_{\text{ctx}} S[\sigma(\mathtt{t}')] : \sigma(\mathtt{T})$. Since $=_{\text{ctx}}$ is adequate, this means that $S[\sigma(\mathtt{t})]\downarrow$ if and only if $S[\sigma(\mathtt{t})]\downarrow$; hence by (3.15), $\langle S, \sigma(\mathtt{t})\rangle\downarrow$ if and only if $\langle S, \sigma(\mathtt{t}')\rangle\downarrow$. As this holds for all σ and S, we have $\Gamma \vdash \mathtt{t} =_{\text{ciu}} \mathtt{t}' : \mathtt{T}$, as required.

To complete the proof of the theorem we have to show conversely that $=_{\text{ciu}}$ is contained in $=_{\text{ctx}}$. We can deduce this as a corollary of a stronger characterisation of $=_{\text{ctx}}$ in terms of logical relations (Theorem 3.5.13) that we establish in §3.5; so we postpone the rest of this proof until then. □

3.4.6 COROLLARY [CONVERSIONS]: The following contextual equivalences are valid:

(i) $\Gamma \vdash \texttt{if true then } t_1 \texttt{ else } t_2 =_{\text{ctx}} t_1 : T$ and
$\Gamma \vdash \texttt{if false then } t_1 \texttt{ else } t_2 =_{\text{ctx}} t_2 : T$, where $\Gamma \vdash t_i : T$ for $i = 1, 2$.

(ii) $\Gamma \vdash \texttt{op}(c_i{}^{i \in 1..n}) =_{\text{ctx}} c : \texttt{Gnd}$, where c is the value of $op(c_i{}^{i \in 1..n})$ and $\mathit{Typeof}(c) = \texttt{Gnd}$.

(iii) $\Gamma \vdash v_1\ v_2 =_{\text{ctx}} [f \mapsto v_1][x \mapsto v_2]t : T_2$,
where $v_1 = \texttt{fun f(x:}T_1\texttt{)=t:}T_2$.

(iv) $\Gamma \vdash \{l_i{=}v_i{}^{i \in 1..n}\}.j =_{\text{ctx}} v_j : T_j$,
where $\Gamma \vdash \{l_i{=}v_i{}^{i \in 1..n}\} : \{l_i{:}T_i{}^{i \in 1..n}\}$.

(v) $\Gamma \vdash (\lambda X.v)\,T_1 =_{\text{ctx}} [X \mapsto T_1]v : [X \mapsto T_1]T$, where $\Gamma \vdash v : \forall X.T$.

(vi) $\Gamma \vdash \texttt{let \{*X,x\}=(\{*}T_1\texttt{,}v_1\texttt{\} as \{}\exists X, T\texttt{\}) in t} =_{\text{ctx}} [X \mapsto T_1][x \mapsto v_1]t : T_2$, where $\Gamma, X, x{:}T \vdash t : T_2$ with $X \notin \mathit{ftv}(\Gamma, T_2)$.

(vii) $\Gamma \vdash \texttt{let x=v in t} =_{\text{ctx}} [x \mapsto v]t : T_2$, where $\Gamma \vdash v : T_1$ and $\Gamma, x{:}T_1 \vdash t : T_2$.

(viii) $\Gamma \vdash \texttt{let } x_1{=}t_1 \texttt{ in (let } x_2{=}t_2 \texttt{ in t)} =_{\text{ctx}}$
$\texttt{let } x_2\texttt{=(let } x_1{=}t_1 \texttt{ in } t_2\texttt{) in t} : T$, where $\Gamma \vdash t_1 : T_1$,
$\Gamma, x_1{:}T_1 \vdash t_2 : T_2$ and $\Gamma, x_2{:}T_2 \vdash t : T$.

□

Proof: These are all ciu-equivalences, so we can just apply Theorem 3.4.5. That this is so follows easily from the definition of the termination relation (Figure 3-2) in all cases except the last one, where one can apply property (3.6) from Exercise 3.3.2 to reduce proving (viii) to the special case when t_1 is a value: see Exercise 3.4.7. □

3.4.7 EXERCISE [★ ↛]: Given $\emptyset \vdash t_1 : T_1$, $x_1{:}T_1 \vdash t_2 : T_2$ and $x_2{:}T_2 \vdash t : T$, use property (3.6) to show that for all frame stacks S

$$\langle S \circ (x_1.\texttt{let } x_2{=}t_2 \texttt{ in t}), t_1\rangle\downarrow \quad \text{iff} \quad \langle S \circ (x_2.t) \circ (x_1.t_2), t_1\rangle\downarrow.$$

Deduce part (viii) of Corollary 3.4.6. □

3.5 An Operationally-Based Logical Relation

Now that we have a precise definition of contextual equivalence for F_{ML} terms, before showing that the Extensionality Principle 3.2.6 holds for existential types in F_{ML}, we first have to give a precise definition of the action of types on term-relations, $r \mapsto T[r]$, mentioned in the principle. That is the

topic of this section. We will end up with a characterisation of $=_{\mathrm{ctx}}$ in terms of a 'logical relation' which yields several useful extensionality properties of contextual equivalence.

3.5.1 NOTATION: Let *Typ* denote the set of closed F_{ML} types. Given $\mathtt{T} \in \mathit{Typ}$, let

- $\mathit{Term}(\mathtt{T})$ denote the set of closed terms of type $\mathtt{T}$, i.e. those terms $\mathtt{t}$ for which $\emptyset \vdash \mathtt{t} : \mathtt{T}$ holds;
- $\mathit{Val}(\mathtt{T})$ denote the subset of $\mathit{Term}(\mathtt{T})$ whose elements are values;
- $\mathit{Stack}(\mathtt{T})$ denote the set of closed frame stacks whose argument type is $\mathtt{T}$, i.e. those frame stacks S for which $\emptyset \vdash S : \mathtt{T} \multimap \mathtt{T}'$ holds for some $\mathtt{T}' \in \mathit{Typ}$.

Given $\mathtt{T}, \mathtt{T}' \in \mathit{Typ}$, let

- $\mathit{TRel}(\mathtt{T}, \mathtt{T}')$ denote the set of all subsets of $\mathit{Term}(\mathtt{T}) \times \mathit{Term}(\mathtt{T}')$—we call such things *term-relations*;
- $\mathit{VRel}(\mathtt{T}, \mathtt{T}')$ denote the set of all subsets of $\mathit{Val}(\mathtt{T}) \times \mathit{Val}(\mathtt{T}')$—we call such things *value-relations*;
- $\mathit{SRel}(\mathtt{T}, \mathtt{T}')$ denote the the set of all subsets of $\mathit{Stack}(\mathtt{T}) \times \mathit{Stack}(\mathtt{T}')$—we call such things *stack-relations*.

□

Note that every value-relation is in particular a term-relation (since values are particular sorts of term): $\mathit{VRel}(\mathtt{T}, \mathtt{T}') \subseteq \mathit{TRel}(\mathtt{T}, \mathtt{T}')$. On the other hand we can obtain a value-relation from a term-relation just by restricting attention to values: given $\mathtt{r} \in \mathit{TRel}(\mathtt{T}, \mathtt{T}')$, define $\mathtt{r}^v \in \mathit{VRel}(\mathtt{T}, \mathtt{T}')$ by

$$\mathtt{r}^v \stackrel{\mathrm{def}}{=} \{(\mathtt{v}, \mathtt{v}') \in \mathit{Val}(\mathtt{T}) \times \mathit{Val}(\mathtt{T}') \mid (\mathtt{v}, \mathtt{v}') \in \mathtt{r}\}. \tag{3.17}$$

We will be particularly interested in term-relations $\mathtt{r}$ that are indistinguishable, as far as termination properties are concerned, from their value restrictions, $\mathtt{r}^v$. The following definition makes this precise, using the Galois connection between term-relations and stack-relations introduced in (Pitts, 2000, Definition 3.9). The definition may appear to be rather mysterious; its nature will emerge as we develop the action of types on term-relations and its properties.

3.5.2 DEFINITION [CLOSED AND VALUABLE TERM-RELATIONS]: Let $\mathtt{T}, \mathtt{T}' \in \mathit{Typ}$ be closed types. Given a term-relation $\mathtt{r} \in \mathit{TRel}(\mathtt{T}, \mathtt{T}')$, define a stack-relation $\mathtt{r}^s \in \mathit{SRel}(\mathtt{T}, \mathtt{T}')$ by

$(S, S') \in r^s$ if and only if for all $(t, t') \in r$, $\langle S, t\rangle\downarrow$ holds if and only if $\langle S', t'\rangle\downarrow$ does.

Conversely, given a stack-relation $s \in SRel(T, T')$, define a term-relation $s^t \in TRel(T, T')$ by

$(t, t') \in s^t$ if and only if for all $(S, S') \in s$, $\langle S, t\rangle\downarrow$ holds if and only if $\langle S', t'\rangle\downarrow$ does.

Call a term-relation $r \in TRel(T, T')$ *closed* if it satisfies $r = r^{st}$ and *valuable* if it satisfies $r = r^{vst}$. □

It is easy to see from the way they are defined that the operations $(-)^s$ and $(-)^t$ for turning term-relations into stack-relations and *vice versa*, form a Galois connection:

$$s \subseteq r^s \quad \text{if and only if} \quad r \subseteq s^t. \tag{3.18}$$

Hence the operator $(-)^{st}$ on term-relations is monotone ($r_1 \subseteq r_2$ implies $(r_1)^{st} \subseteq (r_2)^{st}$), inflationary ($r \subseteq r^{st}$) and idempotent ($(r^{st})^{st} = r^{st}$). In particular r is closed iff it is of the form $(r_1)^{st}$ for some term-relation r_1. In fact $(-)^{vst}$ is also an idempotent operator and therefore r is valuable iff it is of the form $(r_1)^{st}$ for some value-relation r_1.

3.5.3 EXERCISE [RECOMMENDED, ★★]: Given any value-relation $r \in VRel(T, T')$, show that r^{st} is valuable, i.e. satisfies $r^{st} = (r^{st})^{vst}$. □

3.5.4 NOTE: The closure operator $(-)^{st}$ is denoted $(-)^{\top\top}$ in (Pitts, 1998, 2000). □

Closed term-relations (and hence also valuable term-relations) have excellent 'admissibility' properties that we record in the following lemma.

3.5.5 LEMMA: If $r \in TRel(T, T')$ satisfies $r = r^{st}$ (and in particular if it is valuable), then it has the following properties.

Equality-respecting If $(t, t') \in r$, $\emptyset \vdash t =_{ctx} t_1 : T$ and $\emptyset \vdash t' =_{ctx} t'_1 : T$, then $(t_1, t'_1) \in r$.

Admissibility Given recursive function values $F \stackrel{\text{def}}{=} \texttt{fun f(x:}T_1\texttt{)=u:}T_2$ and $F' \stackrel{\text{def}}{=} \texttt{fun f(x:}T_1\texttt{)=u':}T_2$, let F_n and F'_n ($n = 0, 1, \ldots$) be their 'unwindings', as in Theorem 3.3.4. If $([x \mapsto F_n]t, [x \mapsto F'_n]t') \in r$ for all $n = 0, 1, \ldots$, then $([x \mapsto F]t, [x \mapsto F']t') \in r$.

□

Proof: We saw in the first half of the proof of Theorem 3.4.5 (the second half of which we have yet to complete) that contextual equivalence implies ciu-equivalence. Therefore it suffices to prove the **Equality-respecting** property of closed term-relations with respect to $=_{\text{ciu}}$, rather than $=_{\text{ctx}}$. So suppose $(\mathtt{t},\mathtt{t}') \in r$, $\emptyset \vdash \mathtt{t} =_{\text{ciu}} \mathtt{t}_1 : \mathtt{T}$ and $\emptyset \vdash \mathtt{t}' =_{\text{ciu}} \mathtt{t}'_1 : \mathtt{T}$. To see that $(\mathtt{t}_1,\mathtt{t}'_1) \in r$, since $r = (r^s)^t$, it suffices to show for all $(S,S') \in r^s$ that $\langle S,\mathtt{t}_1\rangle\downarrow$ iff $\langle S',\mathtt{t}'_1\rangle\downarrow$. But

$$\begin{array}{lll}
\langle S,\mathtt{t}_1\rangle\downarrow \text{ iff } \langle S,\mathtt{t}\rangle\downarrow & & (\text{since } \emptyset \vdash \mathtt{t} =_{\text{ciu}} \mathtt{t}_1 : \mathtt{T}) \\
\quad \text{iff } \langle S',\mathtt{t}'\rangle\downarrow & & (\text{since } (S,S') \in r^s \text{ and } (\mathtt{t},\mathtt{t}') \in r) \\
\quad \text{iff } \langle S',\mathtt{t}'_1\rangle\downarrow & & (\text{since } \emptyset \vdash \mathtt{t}' =_{\text{ciu}} \mathtt{t}'_1 : \mathtt{T}).
\end{array}$$

For the **Admissibility** property we apply the Unwinding Theorem. Suppose $([\mathtt{x}\mapsto \mathtt{F}_n]\mathtt{t},[\mathtt{x}\mapsto \mathtt{F}'_n]\mathtt{t}') \in r$ holds for all $n = 0,1,\ldots$. Then for any $(S,S') \in r^s$ we have

$$\begin{array}{lll}
\langle S,[\mathtt{x}\mapsto \mathtt{F}]\mathtt{t}\rangle\downarrow & & \\
\text{iff for all } n,\ \langle S,[\mathtt{x}\mapsto \mathtt{F}_n]\mathtt{t}\rangle\downarrow & & (\text{by Theorem 3.3.4}) \\
\text{iff for all } n,\ \langle S',[\mathtt{x}\mapsto \mathtt{F}'_n]\mathtt{t}'\rangle\downarrow & & (\text{since } (S,S') \in r^s \text{ and} \\
 & & \quad ([\mathtt{x}\mapsto \mathtt{F}_n]\mathtt{t},[\mathtt{x}\mapsto \mathtt{F}'_n]\mathtt{t}') \in r) \\
\text{iff } \langle S,[\mathtt{x}\mapsto \mathtt{F}']\mathtt{t}'\rangle\downarrow & & (\text{by Theorem 3.3.4 again})
\end{array}$$

and therefore $([\mathtt{x}\mapsto \mathtt{F}]\mathtt{t},[\mathtt{x}\mapsto \mathtt{F}']\mathtt{t}') \in (r^s)^t$; but $r^{st} = r$. □

3.5.6 DEFINITION [ACTION OF TYPES ON TERM-RELATIONS]: The action of types on term-relations takes the following form: if $\mathtt{T}(\overline{\mathtt{X}})$ is a type whose free type variables lie among the list $\overline{\mathtt{X}} = \mathtt{X}_1,\ldots,\mathtt{X}_n$, then given a corresponding list of term relations

$$r_1 \in \mathit{TRel}(\mathtt{T}_1,\mathtt{T}'_1),\ldots,r_n \in \mathit{TRel}(\mathtt{T}_n,\mathtt{T}'_n)$$

we define a term relation $\mathtt{T}[\overline{r}] \in \mathit{TRel}([\overline{\mathtt{X}}\mapsto\overline{\mathtt{T}}]\mathtt{T},[\overline{\mathtt{X}}\mapsto\overline{\mathtt{T}'}]\mathtt{T})$. The definition is by induction on the structure of $\mathtt{T}$ as follows.

$$\begin{array}{rcl}
\mathtt{X}_i[\overline{r}] & \stackrel{\text{def}}{=} & (r_i)^{v\,s\,t} \\
\mathtt{Gnd}[\overline{r}] & \stackrel{\text{def}}{=} & (\mathit{Id}_{\mathtt{Gnd}})^{s\,t} \\
(\mathtt{T}_1\to\mathtt{T}_2)[\overline{r}] & \stackrel{\text{def}}{=} & \mathtt{fun}(\mathtt{T}_1[\overline{r}],\mathtt{T}_2[\overline{r}])^{s\,t} \\
\{\mathtt{l}_i:\mathtt{T}_i\ ^{i\in 1..n}\}[\overline{r}] & \stackrel{\text{def}}{=} & \{\mathtt{l}_i{=}\mathtt{T}_i[\overline{r}]\ ^{i\in 1..n}\}^{s\,t} \\
(\forall\mathtt{X}.\mathtt{T})[\overline{r}] & \stackrel{\text{def}}{=} & (\lambda r.\,\mathtt{T}[r,\overline{r}])^{s\,t} \\
\{\exists\mathtt{X},\mathtt{T}\}[\overline{r}] & \stackrel{\text{def}}{=} & \{\exists r,\mathtt{T}[r,\overline{r}]\}^{s\,t}
\end{array}$$

In addition to the operations on term-, value- and stack-relations given in Definition 3.5.2, these definitions make use of the following operations for constructing value-relations from term-relations.

(i) $Id_{\mathtt{Gnd}} \stackrel{\text{def}}{=} \{(\mathtt{c},\mathtt{c}) \mid \mathit{Typeof}(\mathtt{c}) = \mathtt{Gnd}\} \in \mathit{VRel}(\mathtt{Gnd},\mathtt{Gnd})$.

(ii) $\mathtt{r}_1 \in \mathit{TRel}(\mathtt{T}_1,\mathtt{T}_1'), \mathtt{r}_2 \in \mathit{TRel}(\mathtt{T}_2,\mathtt{T}_2')$
$\mapsto \mathtt{fun}(\mathtt{r}_1,\mathtt{r}_2) \in \mathit{VRel}(\mathtt{T}_1{\to}\mathtt{T}_2, \mathtt{T}_1'{\to}\mathtt{T}_2')$:

$(\mathtt{v},\mathtt{v}') \in \mathtt{fun}(\mathtt{r}_1,\mathtt{r}_2)$ if and only if for all $(\mathtt{v}_1,\mathtt{v}_1') \in (\mathtt{r}_1)^v$, it is the case that $(\mathtt{v}\ \mathtt{v}_1, \mathtt{v}'\ \mathtt{v}_1') \in \mathtt{r}_2$.

(iii) $(\mathtt{r}_i \in \mathit{TRel}(\mathtt{T}_i,\mathtt{T}_i')^{\ i\in 1..n})$
$\mapsto \{\mathtt{l}_i{=}\mathtt{r}_i\ ^{i\in 1..n}\} \in \mathit{VRel}(\{\mathtt{l}_i{:}\mathtt{T}_i\ ^{i\in 1..n}\}, \{\mathtt{l}_i{:}\mathtt{T}_i'\ ^{i\in 1..n}\})$:

$(\mathtt{v},\mathtt{v}') \in \{\mathtt{l}_i{=}\mathtt{r}_i\ ^{i\in 1..n}\}$ if and only if for all $\mathtt{i} \in 1..\mathtt{n}$, it is the case that $(\mathtt{v}.\mathtt{l}_i, \mathtt{v}'.\mathtt{l}_i) \in \mathtt{r}_i$.

(iv) $(\lambda\mathtt{T}_1,\mathtt{T}_1' \in \mathit{Typ}, \mathtt{r} \in \mathit{TRel}(\mathtt{T}_1,\mathtt{T}_1')\ .\ \mathtt{R}(\mathtt{r}) \in \mathit{TRel}([\mathtt{X}\mapsto\mathtt{T}_1]\mathtt{T}, [\mathtt{X}\mapsto\mathtt{T}_1']\mathtt{T}'))$
$\mapsto \lambda\mathtt{r}.\mathtt{R}(\mathtt{r}) \in \mathit{VRel}(\forall\mathtt{X}.\mathtt{T}, \forall\mathtt{X}.\mathtt{T}')$:

$(\mathtt{v},\mathtt{v}') \in \lambda\mathtt{r}.\mathtt{R}(\mathtt{r})$ if and only if for all $\mathtt{T}_1,\mathtt{T}_1' \in \mathit{Typ}$ and all $\mathtt{r} \in \mathit{TRel}(\mathtt{T}_1,\mathtt{T}_1')$, it is the case that $(\mathtt{v}\ \mathtt{T}_1, \mathtt{v}'\ \mathtt{T}_1') \in \mathtt{R}(\mathtt{r})$.

(v) $(\lambda\mathtt{T}_1,\mathtt{T}_1' \in \mathit{Typ}, \mathtt{r} \in \mathit{TRel}(\mathtt{T}_1,\mathtt{T}_1')\ .\ \mathtt{R}(\mathtt{r}) \in \mathit{TRel}([\mathtt{X}\mapsto\mathtt{T}_1]\mathtt{T}, [\mathtt{X}\mapsto\mathtt{T}_1']\mathtt{T}'))$
$\mapsto \{\exists\mathtt{r},\mathtt{R}(\mathtt{r})\} \in \mathit{VRel}(\{\exists\mathtt{X},\mathtt{T}\}, \{\exists\mathtt{X},\mathtt{T}'\})$:

$(\mathtt{v},\mathtt{v}') \in \{\exists\mathtt{r},\mathtt{R}(\mathtt{r})\}$ if and only if there exist $\mathtt{T}_1,\mathtt{T}_1' \in \mathit{Typ}$, $\mathtt{r} \in \mathit{TRel}(\mathtt{T}_1,\mathtt{T}_1')$ and $(\mathtt{v}_1,\mathtt{v}_1') \in \mathtt{R}(\mathtt{r})$ with $\mathtt{v} = \{{*}\mathtt{T}_1,\mathtt{v}_1\}\ \mathtt{as}\ \{\exists\mathtt{X},\mathtt{T}\}$ and $\mathtt{v}' = \{{*}\mathtt{T}_1',\mathtt{v}_1'\}\ \mathtt{as}\ \{\exists\mathtt{X},\mathtt{T}'\}$.

□

The following lemma helps with calculations involving the action on term-relations of function types. (For related properties for record and ∀-types, see Exercise 3.5.8.)

3.5.7 LEMMA: The operation $\mathtt{fun}(-,-)$ from Definition 3.5.6(ii) satisfies

$$\mathtt{fun}(\mathtt{r}_1,(\mathtt{r}_2)^{st})^{stv} = \mathtt{fun}(\mathtt{r}_1,(\mathtt{r}_2)^{st}) = \mathtt{fun}((\mathtt{r}_1)^{vst},(\mathtt{r}_2)^{st}).$$

□

Proof: It is easy to see from the definition of $\mathtt{fun}(-,-)$ that it is order-reversing in its first argument and satisfies $\mathtt{fun}(\mathtt{r}_1,\mathtt{r}_2) = \mathtt{fun}((\mathtt{r}_1)^v,\mathtt{r}_2))$. So

since $(-)^{st}$ is inflationary, to prove the lemma we just have to prove the inclusions:

$$\mathtt{fun}(r_1,(r_2)^{st})^{st} \subseteq \mathtt{fun}(r_1,(r_2)^{st}) \tag{3.19}$$

$$\mathtt{fun}((r_1)^{vst},(r_2)^{st}) \subseteq \mathtt{fun}(r_1,(r_2)^{st}). \tag{3.20}$$

The first inclusion is a consequences of the following simple property of the termination relation (Figure 3-2) with respect to application:

$$(\langle S, v\ v_1\rangle\downarrow \Leftrightarrow \langle S', v'\ v_1'\rangle\downarrow) \text{ iff } (\langle S\circ(f\,.\,f\ v_1), v\rangle\downarrow \Leftrightarrow \langle S'\circ(f\,.\,f\ v_1'), v'\rangle\downarrow).$$

For from this we first get that if $(S,S') \in (r_2)^s$ and $(v_1,v_1') \in (r_1)^v$, then $(S\circ(f\,.\,f\ v_1), S'\circ(f\,.\,f\ v_1')) \in \mathtt{fun}(r_1,(r_2)^{st})^s$. Then using the same fact about termination again, given any $(v,v') \in \mathtt{fun}(r_1,(r_2)^{st})^{st}$ and any $(v_1,v_1') \in (r_1)^v$ we have $(v\ v_1, v'\ v_1') \in (r_2)^{st}$—which gives (3.19). The inclusion in (3.20) is proved in a similar way, but starting with the property of termination:

$$(\langle S, v\ v_1\rangle\downarrow \Leftrightarrow \langle S', v'\ v_1'\rangle\downarrow) \text{ iff } (\langle S\circ(x\,.\,v\ x), v_1\rangle\downarrow \Leftrightarrow \langle S'\circ(x\,.\,v'\ x), v_1'\rangle\downarrow).$$

□

3.5.8 EXERCISE [RECOMMENDED, ★]: Show that constructions (iii) and (iv) in Definition 3.5.6 satisfy

$$\{l_i{=}(r_i)^{st}\ ^{i\in 1..n}\}^{stv} = \{l_i{=}(r_i)^{st}\ ^{i\in 1..n}\} \tag{3.21}$$

$$(\lambda r\,.\,R(r)^{st})^{stv} = \lambda r\,.\,R(r)^{st}. \tag{3.22}$$

(Cf. the proof of Lemma (3.5.7).) □

We can use the action of types on term-relations to define a type-respecting binary relation between *open* terms (in the sense of Definition 3.4.1) by insisting that if we substitute related terms for the free value variables, the resulting terms are still related. This 'mapping related things to related things' property is the common characteristic of the wide variety of constructs called *logical relations* that have arisen since the seminal work in (Plotkin, 1973; Statman, 1985a) concerning simply typed λ-calculus.

3.5.9 DEFINITION [LOGICAL RELATION, Δ]: Given $\Gamma \vdash t : T$ and $\Gamma \vdash t' : T$, with $\Gamma = X_1,\ldots,X_m,\ x_1 : T_1,\ldots,x_n : T_n$ say, we write $\Gamma \vdash t\ \Delta\ t' : T$ to mean that for all Γ-closing substitutions σ, σ′ (cf. Definition 3.4.2(vi)) and all families of term-relations $\overline{r} = (r_i \in \mathit{TRel}(\sigma(X_i),\sigma'(X_i))\ ^{i\in 1..m})$, if $(\sigma(x_j),\sigma'(x_j')) \in T_j[\overline{r}]^v$ holds for each $j = 1,\ldots,n$, then $(\sigma(t),\sigma'(t')) \in T[\overline{r}]$. □

3.5.10 LEMMA [FUNDAMENTAL PROPERTY OF THE LOGICAL RELATION]: The logical relation Δ is has the substitutivity and compatibility properties defined in Figure 3-3. □

Proof: The first substitutivity property in Figure 3-3 (closure under substituting values for value variables) holds for Δ because of the way it is defined in terms of closing substitutions. The second substitutivity property (closure under substituting types for types variables) holds for Δ because of the following substitution property of the action of types on term-relations:

$$([\mathtt{X} \mapsto \mathtt{T}']\mathtt{T})[\overline{\mathtt{r}}] = \mathtt{T}[\mathtt{T}'[\overline{\mathtt{r}}], \overline{\mathtt{r}}] \qquad (3.23)$$
for any types $\mathtt{T}$ and $\mathtt{T}'$ with $ftv(\mathtt{T}) \subseteq \mathtt{X}, \overline{\mathtt{X}}$ and $ftv(\mathtt{T}') \subseteq \overline{\mathtt{X}}$.

This follows by induction on the structure of the type $\mathtt{T}$; for the base case ($\mathtt{T}$ a type variable) we use the fact that, because of the way the action is defined (Definition 3.5.6), each term-relation $\mathtt{T}'[\overline{\mathtt{r}}]$ is valuable (Definition 3.5.2), i.e. $(\mathtt{T}'[\overline{\mathtt{r}}])^{vst} = \mathtt{T}'[\overline{\mathtt{r}}]$.

The proof that Δ is closed under each of the compatibility properties given in Figure 3-3 follows a pattern similar to Lemmas 4.7–4.10 in (Pitts, 2000) (see also the proof of (Pitts and Stark, 1998, Proposition 4.8)), exploiting the syntax-directed nature of the termination relation (Figure 3-2). The only technically difficult step is for closure of Δ under formation of recursive function values (Exercise 3.5.11), where one needs the admissibility property of valuable term-relations established in Lemma 3.5.5. □

3.5.11 EXERCISE [RECOMMENDED, ★★★]: The following result, in conjunction with Lemma 3.5.7, shows that the logical relation Δ is closed under the compatibility property for recursive function values in Figure 3-3. Given

$$\mathtt{F} \stackrel{\text{def}}{=} \mathtt{fun\ f(x:T_1) = t : T_2} \in \mathit{Val}(\mathtt{T_1} \to \mathtt{T_2})$$
$$\mathtt{F}' \stackrel{\text{def}}{=} \mathtt{fun\ f(x:T_1') = t' : T_2'} \in \mathit{Val}(\mathtt{T_1'} \to \mathtt{T_2'})$$
$$\mathtt{r_1} \in \mathit{TRel}(\mathtt{T_1}, \mathtt{T_1'})$$
$$\mathtt{r_2} \in \mathit{TRel}(\mathtt{T_2}, \mathtt{T_2'})$$

satisfying $\mathtt{r_2} = (\mathtt{r_2})^{st}$ and

$$([\mathtt{f} \mapsto \mathtt{v}][\mathtt{x} \mapsto \mathtt{v_1}]\mathtt{t}, [\mathtt{f} \mapsto \mathtt{v}'][\mathtt{x} \mapsto \mathtt{v_1'}]\mathtt{t}') \in \mathtt{r_2}, \qquad (3.24)$$
for all $(\mathtt{v}, \mathtt{v}') \in \mathtt{fun}(\mathtt{r_1}, \mathtt{r_2})$ and $(\mathtt{v_1}, \mathtt{v_1'}) \in (\mathtt{r_1})^v$,

show that $(\mathtt{F}, \mathtt{F}') \in \mathtt{fun}(\mathtt{r_1}, \mathtt{r_2})$. □

3.5.12 LEMMA [ADEQUACY]: The logical relation Δ is adequate (Definition 3.4.1). □

Proof: Suppose $\emptyset \vdash \mathtt{t}\ \Delta\ \mathtt{t}' : \mathtt{T}$; we have to show that $\mathtt{t}\downarrow$ holds iff $\mathtt{t}'\downarrow$ does, or equivalently that

$$\langle \mathit{Id}, \mathtt{t} \rangle \downarrow \quad \text{iff} \quad \langle \mathit{Id}, \mathtt{t}' \rangle \downarrow. \qquad (3.25)$$

Unravelling Definition 3.5.9, the assumption that the closed terms $\mathtt{t}$ and $\mathtt{t}'$ of closed type $\mathtt{T}$ are Δ-related means that $(\mathtt{t},\mathtt{t}') \in \mathtt{T}[]$, the latter being the action of the type $\mathtt{T}$ on the empty list of term-relations. We noted in the proof of Lemma 3.5.10 that the action of types on term-relations always produces valuable term-relations; so $(\mathtt{t},\mathtt{t}') \in \mathtt{T}[]^{v\,s\,t}$. Hence to prove (3.25), it suffices to show that $(Id, Id) \in \mathtt{T}[]^{v\,s}$; but for any $(\mathtt{v},\mathtt{v}') \in \mathtt{T}[]^{v}$,

$$\langle Id, \mathtt{v}\rangle \downarrow \quad \text{iff} \quad \langle Id, \mathtt{v}'\rangle \downarrow$$

holds trivially by axiom (S-NILVAL) in Figure 3-2. □

3.5.13 THEOREM [$=_{\mathrm{ctx}}$ EQUALS Δ]: F_{ML} contextual equivalence (defined in Theorem 3.4.3) coincides with the logical relation of Definition 3.5.9: $\Gamma \vdash \mathtt{t} =_{\mathrm{ctx}} \mathtt{t}' : \mathtt{T}$ holds if and only if $\Gamma \vdash \mathtt{t}\ \Delta\ \mathtt{t}' : \mathtt{T}$ does. □

Proof: We have shown that Δ is compatible, substitutive and adequate. In Theorem 3.4.3 we constructed $=_{\mathrm{ctx}}$ as the union of all such type-respecting relations; therefore Δ is contained in $=_{\mathrm{ctx}}$.

For the converse inclusion, note that using the reflexivity and substitutivity properties of $=_{\mathrm{ctx}}$ (which it has by construction) and the fact that the action of types on term-relations produces valuable term-relations, we can apply Lemma 3.5.5 to conclude that

$$\text{if } \Gamma \vdash \mathtt{t} =_{\mathrm{ctx}} \mathtt{t}' : \mathtt{T} \text{ and } \Gamma \vdash \mathtt{t}'\ \Delta\ \mathtt{t}'' : \mathtt{T}, \text{ then } \Gamma \vdash \mathtt{t}\ \Delta\ \mathtt{t}'' : \mathtt{T}. \tag{3.26}$$

Since Δ is compatible it is in particular reflexive, so we can take $\mathtt{t}' = \mathtt{t}''$ in (3.26) to deduce that $\Gamma \vdash \mathtt{t} =_{\mathrm{ctx}} \mathtt{t}' : \mathtt{T}$ implies $\Gamma \vdash \mathtt{t}\ \Delta\ \mathtt{t}' : \mathtt{T}$. □

We can now complete the proof of Theorem 3.4.5, by showing that $=_{\mathrm{ciu}}$ is contained in $=_{\mathrm{ctx}}$. We noted in the proof of Lemma 3.5.5 that the equality-respecting property of valuable term-relations holds with respect to $=_{\mathrm{ciu}}$; therefore (3.26) also holds with $=_{\mathrm{ctx}}$ replaced by $=_{\mathrm{ciu}}$. Then as in the above proof, we deduce that $\Gamma \vdash \mathtt{t} =_{\mathrm{ciu}} \mathtt{t}' : \mathtt{T}$ implies $\Gamma \vdash \mathtt{t}\ \Delta\ \mathtt{t}' : \mathtt{T}$. But we now know that Δ coincides with $=_{\mathrm{ctx}}$, therefore $\Gamma \vdash \mathtt{t} =_{\mathrm{ciu}} \mathtt{t}' : \mathtt{T}$ implies $\Gamma \vdash \mathtt{t} =_{\mathrm{ctx}} \mathtt{t}' : \mathtt{T}$, as required for the second half of Theorem 3.4.5.

3.6 Operational Extensionality

In this section we develop some of the consequences of Theorem 3.5.13. Now that we know that contextual equivalence coincides with ciu-equivalence (Theorem 3.4.5), when giving general properties of $=_{\mathrm{ctx}}$ we restrict attention to closed terms of closed type where possible, since the corresponding property for open terms can be obtained via closing substitutions.

3.6.1 THEOREM [EXTENSIONALITY FOR VALUES]: We give extensionality principles for the various types of value; for package values, the principle is a formalisation of the final one discussed in the Introduction (Principle 3.2.6).

1. CONSTANTS: Given constants $\mathtt{c}$, $\mathtt{c}'$ of the same ground type, $\mathtt{Gnd}$ say, $\emptyset \vdash \mathtt{c} =_{\mathrm{ctx}} \mathtt{c}' : \mathtt{Gnd}$ holds if and only if $\mathtt{c} = \mathtt{c}'$.

2. FUNCTIONS: Given $\mathtt{f}:\mathtt{T}_1\to\mathtt{T}_2, \mathtt{x}:\mathtt{T}_1 \vdash \mathtt{t} : \mathtt{T}_2$ and $\mathtt{f}:\mathtt{T}_1\to\mathtt{T}_2, \mathtt{x}:\mathtt{T}_1 \vdash \mathtt{t}' : \mathtt{T}_2$, writing $\mathtt{v}$ and $\mathtt{v}'$ for the recursive function values $\mathtt{fun\ f(x:T_1)=t:T_2}$ and $\mathtt{fun\ f(x:T_1)=t':T_2}$ respectively, then $\emptyset \vdash \mathtt{v} =_{\mathrm{ctx}} \mathtt{v}' : \mathtt{T}_1\to\mathtt{T}_2$ if and only if for all $\emptyset \vdash \mathtt{v}_1 : \mathtt{T}_1$, it is the case that $\emptyset \vdash [\mathtt{f}\mapsto\mathtt{v}][\mathtt{x}\mapsto\mathtt{v}_1]\mathtt{t} =_{\mathrm{ctx}} [\mathtt{f}\mapsto\mathtt{v}'][\mathtt{x}\mapsto\mathtt{v}_1]\mathtt{t}' : \mathtt{T}_2$.

3. RECORDS: Given values $\emptyset \vdash \mathtt{v}_i : \mathtt{T}_i$ and $\emptyset \vdash \mathtt{v}'_i : \mathtt{T}_i$ for $i \in 1..n$, then $\emptyset \vdash \{\mathtt{l}_i=\mathtt{v}_i\ ^{i\in 1..n}\} =_{\mathrm{ctx}} \{\mathtt{l}_i=\mathtt{v}'_i\ ^{i\in 1..n}\} : \{\mathtt{l}_i:\mathtt{T}_i\ ^{i\in 1..n}\}$ if and only if for each $i \in 1..n$, $\emptyset \vdash \mathtt{v}_i =_{\mathrm{ctx}} \mathtt{v}'_i : \mathtt{T}_i$.

4. TYPE ABSTRACTIONS: Given $\mathtt{X} \vdash \mathtt{v} : \mathtt{T}$ and $\mathtt{X} \vdash \mathtt{v}' : \mathtt{T}$, then $\emptyset \vdash \lambda\mathtt{X}.\mathtt{v} =_{\mathrm{ctx}} \lambda\mathtt{X}.\mathtt{v}' : \forall\mathtt{X}.\mathtt{T}$ if and only if for all closed types $\mathtt{T}'$, $\emptyset \vdash [\mathtt{X}\mapsto\mathtt{T}']\mathtt{v} =_{\mathrm{ctx}} [\mathtt{X}\mapsto\mathtt{T}']\mathtt{v}' : [\mathtt{X}\mapsto\mathtt{T}']\mathtt{T}$.

5. PACKAGES: For any closed existential type $\{\exists\mathtt{X},\mathtt{T}\}$, closed types $\mathtt{T}_1$, $\mathtt{T}_2$, and values $\emptyset \vdash \mathtt{v}_i : [\mathtt{X}\mapsto\mathtt{T}_i]\mathtt{T}$ $(i = 1,2)$,

   $$\emptyset \vdash \{*\mathtt{T}_1,\mathtt{v}_1\}\ \mathtt{as}\ \{\exists\mathtt{X},\mathtt{T}\} =_{\mathrm{ctx}} \{*\mathtt{T}_2,\mathtt{v}_2\}\ \mathtt{as}\ \{\exists\mathtt{X},\mathtt{T}\} : \{\exists\mathtt{X},\mathtt{T}\}$$

   holds if there is some term-relation $\mathrm{r} \in \mathit{TRel}(\mathtt{T}_1,\mathtt{T}_2)$ with $(\mathtt{v}_1,\mathtt{v}_2) \in \mathtt{T}[\mathrm{r}]$.

□

Proof: We leave the extensionality properties for constants, records and type abstractions as exercises (Exercise 3.6.2 and 3.6.3). For the extensionality property of functions, suppose that for all $\emptyset \vdash \mathtt{v}_1 : \mathtt{T}_1$

$$\emptyset \vdash [\mathtt{f}\mapsto\mathtt{v}][\mathtt{x}\mapsto\mathtt{v}_1]\mathtt{t} =_{\mathrm{ctx}} [\mathtt{f}\mapsto\mathtt{v}'][\mathtt{x}\mapsto\mathtt{v}_1]\mathtt{t}' : \mathtt{T}_2 \tag{3.27}$$

where $\mathtt{v}$ and $\mathtt{v}'$ are as in 2. To show $\emptyset \vdash \mathtt{v} =_{\mathrm{ctx}} \mathtt{v}' : \mathtt{T}_1\to\mathtt{T}_2$, by Theorem 3.5.13 it suffices to show $\emptyset \vdash \mathtt{v}\ \Delta\ \mathtt{v}' : \mathtt{T}_1\to\mathtt{T}_2$, i.e. that $(\mathtt{v},\mathtt{v}') \in (\mathtt{T}_1\to\mathtt{T}_2)[] = \mathtt{fun}(\mathtt{T}_1[],\mathtt{T}_2[])^{st}$. In fact we show that $(\mathtt{v},\mathtt{v}') \in \mathtt{fun}(\mathtt{T}_1[],\mathtt{T}_2[])$. For this we have to prove that if $(\mathtt{v}_1,\mathtt{v}'_1) \in \mathtt{T}_1[]$, then $(\mathtt{v}\,\mathtt{v}_1,\mathtt{v}'\,\mathtt{v}'_1) \in \mathtt{T}_2[]$. By Theorem 3.5.13 again, this is the same as showing: if $\emptyset \vdash \mathtt{v}_1 =_{\mathrm{ctx}} \mathtt{v}'_1 : \mathtt{T}_1$, then $\emptyset \vdash \mathtt{v}\,\mathtt{v}_1 =_{\mathrm{ctx}} \mathtt{v}'\,\mathtt{v}'_1 : \mathtt{T}_2$. As noted in Corollary 3.4.6, we can turn the primitive reduction for function application into a contextual equivalence:

$$\emptyset \vdash \mathtt{v}\,\mathtt{v}_1 =_{\mathrm{ctx}} [\mathtt{f}\mapsto\mathtt{v}][\mathtt{x}\mapsto\mathtt{v}_1]\mathtt{t} : \mathtt{T}_2 \tag{3.28}$$

and similarly for $\mathtt{v}'\ \mathtt{v}_1'$. Therefore we just need to show: if $\emptyset \vdash \mathtt{v}_1 =_{\mathrm{ctx}} \mathtt{v}_1' : \mathtt{T}_1$, then $\emptyset \vdash [\mathtt{f} \mapsto \mathtt{v}][\mathtt{x} \mapsto \mathtt{v}_1]\mathtt{t} =_{\mathrm{ctx}} [\mathtt{f} \mapsto \mathtt{v}'][\mathtt{x} \mapsto \mathtt{v}_1']\mathtt{t} : \mathtt{T}_2$. But this follows from the assumption (3.27) using the reflexivity and substitutivity properties of $=_{\mathrm{ctx}}$. So we have established one half (the difficult half) of the property in 2. For the converse, if $\emptyset \vdash \mathtt{v} =_{\mathrm{ctx}} \mathtt{v}' : \mathtt{T}_1 \to \mathtt{T}_2$, then for any $\emptyset \vdash \mathtt{v}_1 : \mathtt{T}_1$, the compatibility properties of $=_{\mathrm{ctx}}$ give $\emptyset \vdash \mathtt{v}\ \mathtt{v}_1 =_{\mathrm{ctx}} \mathtt{v}'\ \mathtt{v}_1 : \mathtt{T}_2$; and then as before, we can compose with (3.28) to get (3.27).

Finally, let us consider the extensionality property for package values. (Note that unlike the other four, this only gives a sufficient condition for contextual equivalence; Example 3.6.5 below shows that the condition is not necessary.) If $(\mathtt{v}_1, \mathtt{v}_2) \in \mathtt{T}[\mathtt{r}]$, then from Definition 3.5.6 we have

$$\begin{aligned}(\{*\mathtt{T}_1, \mathtt{v}_1\}\ \mathtt{as}\ \{\exists \mathtt{X}, \mathtt{T}\}, \{*\mathtt{T}_2, \mathtt{v}_2\}\ \mathtt{as}\ \{\exists \mathtt{X}, \mathtt{T}\}) &\in \{\exists \mathtt{r}, \mathtt{T}[\mathtt{r}]\} \\ &\subseteq \{\exists \mathtt{r}, \mathtt{T}[\mathtt{r}]\}^{st} \\ &= \{\exists \mathtt{X}, \mathtt{T}\}[].\end{aligned}$$

Thus $\emptyset \vdash \{*\mathtt{T}_1, \mathtt{v}_1\}\ \mathtt{as}\ \{\exists \mathtt{X}, \mathtt{T}\} \ \Delta\ \{*\mathtt{T}_2, \mathtt{v}_2\}\ \mathtt{as}\ \{\exists \mathtt{X}, \mathtt{T}\} : \{\exists \mathtt{X}, \mathtt{T}\}$ and we can apply Theorem 3.5.13 to get the desired contextual equivalence. □

3.6.2 EXERCISE [★]: Given a constant $\mathtt{c}$, construct a frame stack $S_{\mathtt{c}}$ with the property that for all constants $\mathtt{c}'$ (of the same type), $\langle S_{\mathtt{c}}, \mathtt{c}' \rangle \downarrow$ holds if and only if $\mathtt{c}' = \mathtt{c}$. Deduce that $\emptyset \vdash \mathtt{c} =_{\mathrm{ctx}} \mathtt{c}' : \mathtt{Gnd}$ holds only if $\mathtt{c} = \mathtt{c}'$. □

3.6.3 EXERCISE [★★ ↛]: Use Theorem 3.5.13, Corollary 3.4.6 and the definition of the term-relations $\{\mathtt{l}_i{=}\mathtt{r}_i\ ^{i \in 1..n}\}$ and $\lambda \mathtt{r}.\mathtt{R}(\mathtt{r})$ in Definition 3.5.6 to deduce extensionality properties 3 and 4 of Theorem 3.6.1 □

To see how Theorem 3.6.1(5) can be used in practise, we will apply it to establish the contextual equivalence of Example 3.2.5 from the Introduction.

3.6.4 EXAMPLE: Let the type $\mathtt{Smph}$ and the values $\mathtt{smph}_1$, $\mathtt{smph}_2$ be as in Example 3.2.5. To prove $\emptyset \vdash \mathtt{smph}_1 =_{\mathrm{ctx}} \mathtt{smph}_2 : \mathtt{Smph}$ using Theorem 3.6.1(5), it suffices to show that $(\mathtt{v}_1, \mathtt{v}_2) \in \mathtt{T}[\mathtt{r}]$ where

$$\begin{aligned}\mathtt{T} &\stackrel{\mathrm{def}}{=} \{\mathtt{bit{:}X},\ \mathtt{flip{:}X{\to}X},\ \mathtt{read{:}X{\to}Bool}\} \\ \mathtt{v}_1 &\stackrel{\mathrm{def}}{=} \{\mathtt{bit{=}true},\ \mathtt{flip{=}\lambda x{:}Bool.not\ x},\ \mathtt{read{=}\lambda x{:}Int.x}\} \\ \mathtt{v}_2 &\stackrel{\mathrm{def}}{=} \{\mathtt{bit{=}1},\ \mathtt{flip{=}\lambda x{:}Int.0{-}2{*}x},\ \mathtt{read{=}\lambda x{:}Int.x >{=} 0}\}\end{aligned}$$

and $\mathtt{r} \in \mathit{VRel}(\mathtt{Bool}, \mathtt{Int})$ is as defined in that example. Since $\mathtt{r}$ is a value-relation, we can use Lemma 3.5.7 to slightly simplify $\mathtt{T}[\mathtt{r}]$:

$$\begin{aligned}\mathtt{T}[\mathtt{r}] &\stackrel{\mathrm{def}}{=} \{\mathtt{bit{=}}\mathtt{r}^{st}, \mathtt{flip{=}fun}(\mathtt{r}^{st}, \mathtt{r}^{st})^{st}, \mathtt{read{=}fun}(\mathtt{r}^{st}, \mathit{Id}^{st}_{\mathtt{Bool}})^{st}\}^{st} \\ &= \{\mathtt{bit{=}}\mathtt{r}^{st}, \mathtt{flip{=}fun}(\mathtt{r}, \mathtt{r}^{st})^{st}, \mathtt{read{=}fun}(\mathtt{r}, \mathit{Id}^{st}_{\mathtt{Bool}})^{st}\}^{st}.\end{aligned}$$

So since $(-)^{st}$ is inflationary, to prove $(\mathtt{v}_1, \mathtt{v}_2) \in \mathtt{T}[\mathtt{r}]$, it suffices to show

$$\begin{array}{rcl} (\mathtt{true}, 1) & \in & \mathtt{r} \\ (\lambda \mathtt{x{:}Bool.not\ x}, \lambda \mathtt{x{:}Int.0{-}2{*}x}) & \in & \mathtt{fun}(\mathtt{r}, \mathtt{r}^{st}) \\ (\lambda \mathtt{x{:}Int.x}, \lambda \mathtt{x{:}Int.x >= 0}) & \in & \mathtt{fun}(\mathtt{r}, Id^{st}_{\mathtt{Bool}}). \end{array}$$

These follow from the definition of r—the first trivially and the second two once we combine the definition of $\mathtt{fun}(-,-)$ with the fact (Lemma 3.5.5) that closed relations such as $\mathtt{r}^{st}$ and $Id^{st}_{\mathtt{Bool}}$ respect contextual equivalence and hence (Theorem 3.4.5) also respect ciu-equivalence. For example, if $(\mathtt{v}_1, \mathtt{v}'_1) \in \mathtt{r}$, then $(\lambda \mathtt{x{:}Bool.not\ x})\,\mathtt{v}_1$ and $(\lambda \mathtt{x{:}Int.0{-}2{*}x})\,\mathtt{v}'_1$ are ciu-equivalent to r-related values $\mathtt{v}_2$ and $\mathtt{v}'_2$. Therefore $(\mathtt{v}_2, \mathtt{v}'_2) \in \mathtt{r}^{st}$ and hence

$$((\lambda \mathtt{x{:}Bool.not\ x})\,\mathtt{v}_1, (\lambda \mathtt{x{:}Int.0{-}2{*}x})\,\mathtt{v}'_1) \in \mathtt{r}^{st}$$

and thus $(\lambda \mathtt{x{:}Bool.not\ x}, \lambda \mathtt{x{:}Int.0{-}2{*}x}) \in \mathtt{fun}(\mathtt{r}, \mathtt{r}^{st})$. □

Theorem 3.6.1(5) gives a sufficient condition for contextual equivalence of package values, but the condition is not necessary: it can be the case that $\{\mathtt{*\ T}_1, \mathtt{v}_1\}\ \mathtt{as}\ \{\exists \mathtt{X},\ \mathtt{T}\}$ is contextually equivalent to $\{\mathtt{*\ T}_2, \mathtt{v}_2\}\ \mathtt{as}\ \{\exists \mathtt{X},\ \mathtt{T}\}$ even though there is no $\mathtt{r} \in \mathit{TRel}(\mathtt{T}_1, \mathtt{T}_2)$ with $(\mathtt{v}_1, \mathtt{v}_2) \in \mathtt{T}[\mathtt{r}]$. The rest of this section is devoted to giving an example of this unpleasant phenomenon (based on a suggestion of Ian Stark arising out of our joint work on logical relations for functions and dynamically allocated names, (Pitts and Stark, 1993)).

3.6.5 EXAMPLE: Consider the following types and values.

```
type P(X)  = (X→Bool)→Bool
type Q     = {∃X,P}
type N     = ∀X.X
val  G     = fun g(f:N→Bool) = (g f):Bool
val  G'    = fun g(f:Bool→Bool) =
               (if f true  then
                  if f false then g f else true
                else g f):Bool
```

Thus N is a type with no values; G is a function that diverges when applied to any value of type N→Bool; and G′ is a function that diverges when applied to any value of type Bool→Bool except ones (such as the identity function) that map true to true and false to false, in which case it returns true. We claim that

(i) there is no $\mathtt{r} \in \mathit{TRel}(\mathtt{N}, \mathtt{Bool})$ for which $(\mathtt{G}, \mathtt{G}') \in \mathtt{P}[\mathtt{r}]$ holds,

(ii) but nevertheless $\emptyset \vdash \{\texttt{*N,G}\}\ \texttt{as Q} =_{\text{ctx}} \{\texttt{*Bool,G'}\}\ \texttt{as Q} : \texttt{Q}$.

□

Proof: For (i) note that the definition of N implies that $Val(\texttt{N}) = \emptyset$, i.e. there are no closed values of type N. So any $\texttt{r} \in TRel(\texttt{N},\texttt{Bool})$ satisfies $\texttt{r}^v = \emptyset$. Using Lemma 3.5.7, we have

$$\texttt{P}[\texttt{r}]^v = \texttt{fun}(\texttt{fun}(\texttt{r}^v, Id^{s\,t}_{\texttt{Bool}}), Id^{s\,t}_{\texttt{Bool}}).$$

Since $\texttt{r}^v = \emptyset$, we have $\texttt{fun}(\texttt{r}^v, Id^{s\,t}_{\texttt{Bool}}) = Val(\texttt{N}\to\texttt{Bool}) \times Val(\texttt{Bool}\to\texttt{Bool})$; and we know by Theorem 3.5.13 that $Id^{s\,t}_{\texttt{Bool}}$ is the relation $\emptyset \vdash (-) =_{\text{ctx}} (-) : \texttt{Bool}$. Therefore

$$\texttt{P}[\texttt{r}]^v = \{(\texttt{v},\texttt{v}') \mid \emptyset \vdash \texttt{v}\ \texttt{v}_1 =_{\text{ctx}} \texttt{v}'\ \texttt{v}_1' : \texttt{Bool} \text{ for all } \texttt{v}_1 \in Val(\texttt{N}\to\texttt{Bool}) \text{ and } \texttt{v}_1' \in Val(\texttt{Bool}\to\texttt{Bool})\}.$$

However, $\emptyset \vdash \texttt{G}\ \texttt{v}_1 =_{\text{ctx}} \texttt{G}'\ \texttt{v}_1' : \texttt{Bool}$ does not hold if we take $\texttt{v}_1$ and $\texttt{v}_1'$ to be the values

```
val v1  = fun f(x:N) = (f x):Bool
val v1' = fun f(x:Bool) = x:Bool
```

since evaluation of $\texttt{G}\ \texttt{v}_1$ does not terminate, whereas evaluation of $\texttt{G}'\ \texttt{v}_1'$ does. Therefore $(\texttt{G},\texttt{G}') \notin \texttt{P}[\texttt{r}]^v$, for any $\texttt{r} \in TRel(\texttt{N},\texttt{Bool})$.

Turning to the proof of (ii), now we know that it cannot be deduced from the extensionality principle for package values in Theorem 3.6.1, we have to prove this contextual equivalence by brute force. The termination relation defined in Fig. 3-2 provides a possible strategy, if rather a tedious one, for proving contextual equivalences—by what one might call *termination induction*. For example, to prove (ii) it suffices to prove for all terms t containing at most x free that

$$[\texttt{x} \mapsto \{\texttt{*N,G}\}\ \texttt{as Q}]\texttt{t}\downarrow \text{ iff } [\texttt{x} \mapsto \{\texttt{*Bool,G'}\}\ \texttt{as Q}]\texttt{t}\downarrow.$$

If one attempts to do this by induction on the definition of the termination relation in Fig. 3-2, it soon becomes clear that one must attempt to prove a stronger statement, namely that for all frame stacks S and terms t,

$$\langle [\texttt{x} \mapsto \{\texttt{*N,G}\}\ \texttt{as Q}]\texttt{S}, [\texttt{x} \mapsto \{\texttt{*N,G}\}\ \texttt{as Q}]\texttt{t}\rangle\downarrow \text{ iff}$$
$$\langle [\texttt{x} \mapsto \{\texttt{*Bool,G'}\}\ \texttt{as Q}]\texttt{S}, [\texttt{x} \mapsto \{\texttt{*Bool,G'}\}\ \texttt{as Q}]\texttt{t}\rangle\downarrow.$$

It is indeed possible to prove this by induction on the definition of $\langle -,-\rangle\downarrow$ (for all S and t simultaneously). The crucial induction step is that for the primitive reduction of a function application, where the following lemma is required.

For any value $\mathtt{v}$ satisfying $\mathtt{X}$, $\mathtt{g{:}P} \vdash \mathtt{v} : \mathtt{X}{\to}\mathtt{Bool}$, evaluation of $\mathtt{G}'\,(\mathtt{[X} \mapsto \mathtt{Bool][g} \mapsto \mathtt{G}']\mathtt{v})$ does not terminate.

This fact lies at the heart of the reason why the contextual equivalence in (ii) is valid: if an argument supplied to $\mathtt{G}'$ is sufficiently polymorphic (which is guaranteed by the existential abstraction), then when specialised to $\mathtt{Bool}$ it cannot have the functionality ($\mathtt{true} \mapsto \mathtt{true}$, $\mathtt{false} \mapsto \mathtt{false}$) needed to distinguish $\mathtt{G}'$ from the divergent behaviour of $\mathtt{G}$. To prove this we can use the logical relation from the previous section. Consider the following value-relation in $\mathit{VRel}(\mathtt{Bool}, \mathtt{Bool})$:

$$\mathtt{r} \stackrel{\text{def}}{=} \{(\mathtt{true}, \mathtt{true}), (\mathtt{false}, \mathtt{false}), (\mathtt{true}, \mathtt{false})\}.$$

As before, using Lemma 3.5.7 we have $\mathtt{P[r]}^v = \mathtt{fun}(\mathtt{fun}(\mathtt{r}, \mathit{Id}^{s\,t}_{\mathtt{Bool}}), \mathit{Id}^{s\,t}_{\mathtt{Bool}})$; and since $\mathit{Id}^{s\,t}_{\mathtt{Bool}}$ is contextual equivalence, the definition of $\mathtt{G}'$ implies that $(\mathtt{G}', \mathtt{G}') \in \mathtt{P[r]}^v$. So by Lemma 3.5.10 we have

$$(\mathtt{[X} \mapsto \mathtt{Bool][g} \mapsto \mathtt{G}']\mathtt{v}, \mathtt{[X} \mapsto \mathtt{Bool][g} \mapsto \mathtt{G}']\mathtt{v}) \in (\mathtt{X}{\to}\mathtt{Bool})\mathtt{[r]}^v = \mathtt{fun}(\mathtt{r}, \mathit{Id}^{s\,t}_{\mathtt{Bool}}).$$

Since $(\mathtt{true}, \mathtt{false}) \in \mathtt{r}$, we get

$$(\mathtt{[X} \mapsto \mathtt{Bool][g} \mapsto \mathtt{G}']\mathtt{v}\,\mathtt{true}, \mathtt{[X} \mapsto \mathtt{Bool][g} \mapsto \mathtt{G}']\mathtt{v}\,\mathtt{false}) \in \mathit{Id}^{s\,t}_{\mathtt{Bool}}.$$

Thus $(\mathtt{[X} \mapsto \mathtt{Bool][g} \mapsto \mathtt{G}']\mathtt{v})\,\mathtt{true}$ and $(\mathtt{[X} \mapsto \mathtt{Bool][g} \mapsto \mathtt{G}']\mathtt{v})\,\mathtt{false}$ are contextually equivalent closed terms of type $\mathtt{Bool}$. Therefore it cannot be the case that the first evaluates to $\mathtt{true}$ and the second to $\mathtt{false}$; but in that case, by definition of $\mathtt{G}'$, it must be that evaluation of $\mathtt{G}'\,(\mathtt{[X} \mapsto \mathtt{Bool][g} \mapsto \mathtt{G}']\mathtt{v})$ does not terminate. □

The last part of the above proof exploits 'relational parametricity' properties of polymorphic types in F_{ML}. In fact Theorem 3.5.13 tells us far more about the properties of type abstraction values than just the extensionality property of Theorem 3.6.1(5).

3.6.6 THEOREM [RELATIONAL PARAMETRICITY FOR ∀-TYPES]: Given $\mathtt{X} \vdash \mathtt{v} : \mathtt{T}$ and $\mathtt{X} \vdash \mathtt{v}' : \mathtt{T}$, then $\emptyset \vdash \lambda\mathtt{X}.\mathtt{v} =_{\mathrm{ctx}} \lambda\mathtt{X}.\mathtt{v}' : \forall\mathtt{X}.\mathtt{T}$ if and only if for all closed types $\mathtt{T}_1, \mathtt{T}'_1 \in \mathit{Typ}$ and all term-relations $\mathtt{r} \in \mathit{TRel}(\mathtt{T}_1, \mathtt{T}'_1)$ it is the case that $(\mathtt{[X} \mapsto \mathtt{T}_1]\mathtt{v}, \mathtt{[X} \mapsto \mathtt{T}'_1]\mathtt{v}') \in \mathtt{T[r]}$. □

Proof: By Theorem 3.5.13, we have that $\emptyset \vdash \lambda\mathtt{X}.\mathtt{v} =_{\mathrm{ctx}} \lambda\mathtt{X}.\mathtt{v}' : \forall\mathtt{X}.\mathtt{T}$ iff $\emptyset \vdash \lambda\mathtt{X}.\mathtt{v}\ \Delta\ \lambda\mathtt{X}.\mathtt{v}' : \forall\mathtt{X}.\mathtt{T}$, i.e. iff $(\lambda\mathtt{X}.\mathtt{v}, \lambda\mathtt{X}.\mathtt{v}') \in (\forall\mathtt{X}.\mathtt{T})[]^v = (\lambda\mathtt{r}.\mathtt{T[r]})^{s\,t\,v}$. By construction each term-relation $\mathtt{T[r]}$ is closed (i.e. $\mathtt{T[r]} = \mathtt{T[r]}^{s\,t}$). From this it follows that $(\lambda\mathtt{r}.\mathtt{T[r]})^{s\,t\,v} = \lambda\mathtt{r}.\mathtt{T[r]}$ (see Exercise 3.5.8). Hence $\emptyset \vdash \lambda\mathtt{X}.\mathtt{v} =_{\mathrm{ctx}} \lambda\mathtt{X}.\mathtt{v}' : \forall\mathtt{X}.\mathtt{T}$ iff $(\lambda\mathtt{X}.\mathtt{v}, \lambda\mathtt{X}.\mathtt{v}') \in \lambda\mathtt{r}.\mathtt{T[r]}$, as required. □

The force of Theorem 3.6.1(5) is to give a method for establishing that two type abstraction values are contextually equivalent. By contrast, the force of Theorem 3.6.6 is to give us useful properties of families of values parameterised by type variables. Given such a value, $\mathtt{X} \vdash \mathtt{v} : \mathtt{T}$, since $=_{\mathrm{ctx}}$ is reflexive, we have $\emptyset \vdash \Lambda\mathtt{X}.\mathtt{v} =_{\mathrm{ctx}} \Lambda\mathtt{X}.\mathtt{v} : \forall\mathtt{X}.\mathtt{T}$; hence the theorem has the following corollary.

3.6.7 COROLLARY: Given a value $\mathtt{X} \vdash \mathtt{v} : \mathtt{T}$, for all $\mathtt{T}_1, \mathtt{T}_1' \in \mathit{Typ}$ and all $\mathtt{r} \in \mathit{TRel}(\mathtt{T}_1, \mathtt{T}_1')$, it is the case that $([\mathtt{X} \mapsto \mathtt{T}_1]\mathtt{v}, [\mathtt{X} \mapsto \mathtt{T}_1']\mathtt{v}) \in \mathtt{T}[\mathtt{r}]$. □

Such 'relational parametricity' properties can often be exploited for proving contextual equivalences: examples can be found in (Pitts, 2000; Bierman, Pitts, and Russo, 2000; Johann, 2002). However, the strict nature of function application and type abstraction in F_{ML} means that it does not satisfy all the parametricity properties one might expect. For example, in (Pitts, 2000, § 7) it is shown that

$$\{\exists\mathtt{X},\mathtt{T}\} \cong \forall\mathtt{Y}.\,(\forall\mathtt{X}.\mathtt{T}\to\mathtt{Y})\to\mathtt{Y}$$

holds in the polymorphic version of PCF (Plotkin, 1977) studied in that paper (where $\cong$ is 'bijection up to contextual equivalence'—see Principle 3.2.4). However this bijection does not hold in general for F_{ML} (Exercise 3.6.8).

3.6.8 EXERCISE [★★★]: Consider the type $\mathtt{N} \stackrel{\mathrm{def}}{=} \forall\mathtt{X}.\mathtt{X}$ from Example 3.6.5 that has no closed values. Show that there cannot exist values

$$\mathtt{i} \in \mathit{Val}(\{\exists\mathtt{X},\mathtt{N}\}\to\forall\mathtt{Y}.\,(\forall\mathtt{X}.\mathtt{N}\to\mathtt{Y})\to\mathtt{Y})$$
$$\mathtt{j} \in \mathit{Val}((\forall\mathtt{Y}.\,(\forall\mathtt{X}.\mathtt{N}\to\mathtt{Y})\to\mathtt{Y})\to\{\exists\mathtt{X},\mathtt{N}\})$$

that are mutually inverse, in the sense that

$$\mathtt{p}:\{\exists\mathtt{X},\mathtt{N}\} \vdash \mathtt{j}(\mathtt{i}\,\mathtt{p}) =_{\mathrm{ctx}} \mathtt{p} : \{\exists\mathtt{X},\mathtt{N}\}$$
$$\mathtt{y}:\forall\mathtt{Y}.\,(\forall\mathtt{X}.\mathtt{N}\to\mathtt{Y})\to\mathtt{Y} \vdash \mathtt{i}(\mathtt{j}\,\mathtt{y}) =_{\mathrm{ctx}} \mathtt{y} : \forall\mathtt{Y}.\,(\forall\mathtt{X}.\mathtt{N}\to\mathtt{Y})\to\mathtt{Y}.$$

□

3.6.9 EXERCISE [★★★ ↛]: Verify the claim made in Note 3.2.7 that Principle 3.2.4 is a special case of Principle 3.2.6. To do so, you will first have to give a definition of the action of F_{ML} types on bijections mentioned in Principle 3.2.4. □

3.7 Notes

This chapter is a revised and expanded version of (Pitts, 1998) and also draws on some material from (Pitts, 2000).

In discussing typed operational reasoning we have focused on reasoning about *contextual equivalence* of program phrases. Being by construction a congruence, contextual equivalence permits us to use the usual forms of equational reasoning (replacing equals by equals) when deriving equivalences between phrases. However, its definition does not lend itself to establishing the basic laws that are needed to get such reasoning going. We studied two characterisations of contextual equivalence in order to get round this problem: *ciu-equivalence* (Definition 3.4.4) and a certain kind of operationally-based *logical relation* (Definition 3.5.9).

The informal notion of contextual equivalence (Definition 3.2.2) has been studied for a wide variety of programming languages. If the language's operational semantics involves non-determinism—usually because the language supports some form of concurrent or interactive computation—then contextual equivalence tends to identify too many programs and various coinductive notions of *bisimilarity* are used instead (see (Gordon, 1994), for example). But even if we remain within the realm of languages with deterministic operational semantics, one may ask to what extent the results of this chapter are stable with respect to adding further features such as recursive datatypes, references and object-oriented features à la Objective Caml.

Ciu-equivalence has the advantage of being quite robust in this respect—it can provide a characterisation of contextual equivalence in the presence of such features (Honsell, Mason, Smith, and Talcott, 1995; Talcott, 1998). However, its usefulness is mainly limited to establishing basic laws such as the conversions in Corollary 3.4.6; for example, as far as I know, it cannot be used to establish extensionality properties, such as those in Theorem 3.6.1. Ciu-equivalence is quite closely related to some notions of 'applicative bisimilarity' that have been applied to functional and object-based languages (Gordon, 1995, 1998), in that their congruence properties can both be established using a clever technique due to (Howe, 1996). The advantage of applicative bisimilarity is that it has extensionality built into its definition; so when it does coincide with contextual equivalence, this provides a method of establishing some extensionality properties for $=_{\mathrm{ctx}}$ (such as (1)–(4) in Theorem 3.6.1, but not, as far as I know, property (5) for package values).

The kind of operationally-based logical relation we developed in this chapter provides a very powerful analysis of contextual equivalence. We used it to prove not only conversions and simple extensionality principles for F_{ML}, but also quite subtle properties of $=_{\mathrm{ctx}}$ such as Theorems 3.6.1(5) and 3.6.6.

Similar logical relations can be used to prove some properties of ML-style references and of linear types: see (Pitts and Stark, 1998; Bierman, Pitts, and Russo, 2000; Pitts, 2002a). Unfortunately, the characteristic feature of logical relations—that functions are related iff they map related arguments to related results—makes it difficult to define them in the presence of 'recursive features'. I mean by the latter programming language features which in a denotational semantics lead one to have to solve non-trivial domain equations of mixed variance. Recursive datatypes involving function types are one such example; as are references to functions in ML. Suitable logical relations can be defined in the denotational semantics of languages with such features using techniques such as (Pitts, 1996), but they tell us properties of denotational equality, which is often a poor (if safe) approximation to contextual equivalence. For this reason people have tried to develop syntactical analogues of these denotational logical relations: see (Birkedal and Harper, 1999). The unwinding theorem (Theorem 3.3.4) provides the basis for such an approach. However, it seems like a fresh idea is needed to make further progress. Therefore I set a last exercise, whose solution is not included.

3.7.1 EXERCISE [★★★★..., $\twoheadrightarrow$]: Extend F_{ML} with *iso-recursive types*, $\mu \mathtt{X}.\mathtt{T}$, as in Figure 20-1 of TAPL Chapter 20. By finding an operationally-based logical relation as in §3.5 or otherwise, try to prove the kind of properties of contextual equivalence for this extended language that we developed for F_{ML} in this chapter. (For the special case of iso-recursive types $\mu \mathtt{X}.\mathtt{T}$ for which $\mathtt{T}$ contains no negative occurrences of $\mathtt{X}$, albeit for a non-strict functional language, see (Johann, 2002).) □

PART III

Precise Type Analyses

4 *Dependent Types*

By David Aspinall and Martin Hofmann

Dependent types are type-valued functions. This definition includes, for example, the type operators of F^ω such as `Pair`. When applied to two types `S` and `T`, this yields a type `Pair S T` whose elements are pairs `(s,t)` of elements `s` from `S` and `t` from `T` (see TAPL Chapter 29). However, the terminology "dependent types" usually indicates a particular kind of type-valued function: those functions sending *terms* to types. We begin this chapter by looking at a few representative examples of where type-term dependency occurs naturally.

Programming with vectors and format strings

The prototypical example of programming with dependent types is introduced by the *type family* of vectors (one-dimensional arrays):

```
Vector :: Nat → *
```

This kinding assertion states that `Vector` maps a natural number `k:Nat` to a type. The idea is that the type `Vector k` contains vectors of length `k` of elements of some fixed type, say `T`.

To use vectors, we need a way of introducing them. A useful intitialisation function takes a length `n`, a value `t` of `T`, and returns a vector with `n` elements all set to `t`. The typing of such an `init` function is written like this:

```
init : Πn:Nat. T → Vector n
```

The system studied in this chapter is the dependently typed lambda-calculus, λLF (Figures 4-1, 4-2), extended with Σ-types (Figure 4-5) and the Calculus of Constructions (Figure 4-7). The associated OCaml implementation is `deptypes`, to be found at `http://www.attapl.org/checkers/deptypes`.

And the application `init k t` has type `Vector k`.

The type of `init` introduces the *dependent product type* (or "Pi type"), written $\Pi x : S . T$. This type generalises the arrow type of the simply typed lambda-calculus. It is the type of functions which map elements $s : S$ to elements of $[x \mapsto s]T$. In contrast to the simply-typed case, the result type of a function with a Π-type can vary according to the argument supplied. According to Seldin, the Π-type goes back to Curry and is thus almost as old as the lambda calculus itself (Seldin, 2002).

A more interesting way of building up vectors is given by the constant empty vector `empty : Vector 0` together with a constructor for building longer vectors:

```
cons : T → Πn:Nat.Vector n → Vector (n+1).
```

The typing of `cons` expresses that `cons` takes three arguments: an element of type `T`, a natural number `n`, and a vector of length `n`. The result is a vector of length `n+1`. This means that, for example, when `v : Vector 5` and `x : T` then

```
cons x 5 v : Vector 6
```

The dependent product type $\Pi x : S . T$ is somewhat analogous to the universal type $\forall X . T$ of System F. The type of a term `t` with type $\forall X . T$ also varies with the argument supplied: but in the case of a type abstraction, the argument is a type rather than a term. If `A` is a type, then $t\ A : [X \mapsto A]T$. In System F (and F^ω), type variation occurs only with type arguments, whereas in dependent type theory it may occur with term-level arguments.

The reader familiar with programming with ordinary arrays will have realised that by using a type of arrays instead of vectors we could avoid dependent typing. The initialisation function for one-dimensional arrays could be given the simple type `Nat → T → Array`, where `Array` is the type of arrays with entries of type `T`. The point of the dependent typing is that it reveals more information about the behaviour of a term, which can be exploited to give more precise typings and exclude more of the badly behaved terms in a type system. For example, with the dependent type of vectors, we can type a function that returns the first element of a non-empty vector:

```
first : Πn:Nat.Vector(n+1) → T
```

The function `first` can never be applied to an empty vector — non-emptiness is expressed within the type system itself! This is a useful gain. With ordinary arrays instead of dependently-typed vectors, we would need some special way to deal with the case when `first` is applied to the empty array. We

could return an ad-hoc default element, or we might use a language-based exception mechanism to indicate the error. Either mechanism is more clumsy than simply prohibiting illegal applications of `first` from being written.

We suggested that $\Pi x : S . T$ generalises the function space $S \to T$ of simply typed lambda calculus. In fact, we can treat $S \to T$ simply as an abbreviation:

$$S \to T \quad = \quad \Pi x : S . T \qquad \textit{where } x \textit{ does not appear free in } T$$

For example, $\Pi x : \mathtt{Nat} . \mathtt{Nat}$ is exactly equivalent to $\mathtt{Nat} \to \mathtt{Nat}$. We will continue to write the arrow $\to$ whenever possible, to increase readability.

Another favourite example of a function with a useful dependent typing is `sprintf` of the C language.[1] Recall that `sprintf` accepts a format string and list of arguments whose types must correspond to the declarations made in the format string. It then converts the given arguments into a string and returns it. A simplified form of `sprintf` might have the typing:

```
sprintf : Πf:Format . Data(f) → String
```

where we suppose that `Format` is a type of valid print formats (for example, considered as character lists) and that `Data(f)` is the type of data corresponding to format `f`. The function `Data(f)` evaluates the type that the format string describes, which might include clauses like these:

```
Data([])        =  Unit
Data("%d"::cs)  =  Nat * Data(cs)
Data("%s"::cs)  =  String * Data(cs)
Data(c::cs)     =  Data(cs)
```

This example is rather different to the case of vectors. Vectors are uniform: we introduce operations that are parametric in the length n, and the family of types `Vector n` is indexed by n. In the case of format strings, we use case analysis over values to construct the type `Data(f)` which depends on `f` in an arbitrary way. Unsurprisingly, this non-uniform kind of dependent type is more challenging to deal with in practical type systems for programming.

III.1 EXERCISE [★]: Suggest some dependent typings for familiar data types and their operations. For example, consider matrices of size $n \times m$ and the typing of matrix multiplication, and a type of dates where the range of the day is restricted according to the month. □

1. We remark that a `sprintf`-like formatiing function can also be typed in ML without dependent types if formats are represented as appropriate higher-order functions rather than strings. For details see (Danvy, 1998).

The Curry-Howard correspondence

A rather different source for dependent typing is the Curry-Howard correspondence, also known by the slogan *propositions-as-types* (Howard, 1980). Under this correspondence simple types correspond to propositions in the implicational fragment of constructive logic. A formula has a proof if and only if the corresponding type is inhabited. For instance, the formula

```
((A→B) → A) → (A→B) → B
```

is valid in constructive logic and at the same time is inhabited, namely by `λf.λu.u(f u)`. The underlying philosophical idea behind this correspondence is that a constructive proof of an implication `A` $\Longrightarrow$ `B` ought to be understood as a procedure that transforms any given proof of `A` into a proof of `B`.

If propositions are types, then proofs are terms. We can introduce a type constructor `Prf(-)` which maps a formula (understood as a type) into the type of its proofs, and then a proof of `A` $\Longrightarrow$ `B` becomes a λ-term of type `Prf(A)→Prf(B)`. Often the type constructor `Prf(-)` is omitted, notationally identifying a proposition with the type of its proofs. In that case, a proof of `A` $\Longrightarrow$ `B` is simply any term of type `A→B`.

Generalising the correspondence to first-order predicate logic naturally leads to dependent types: a predicate `B` over type `A` is viewed as a type-valued function on `A`; a proof of the universal quantification `∀x:A.B(a)` is — constructively — a procedure that given an arbitrary element `x` of type `A` produces a proof of `B(x)`. Hence under the Curry-Howard correspondence we should identify universal quantification with dependent product: a proof of `∀x:A.B(x)` is a member of `Πx:A.B(x)`. Indeed, Per Martin-Löf, one of the protagonists of dependent typing (Martin-Löf, 1984), was motivated by this extension. In particular he introduced type-theoretic equivalents to existential quantification (Σ-types) and equality (identity types), used in the next example.

An important application of the Curry-Howard correspondence is that it allows one to freely mix propositions and (programming language) types. For example, an indexing function `ith(n)` to access elements of vectors of length `n` could be given the type

```
Πl:Nat.Lt(l,n)→Vector(n)→T
```

where `Lt(l,n)` is the proposition asserting that `l` is less than `n`. Perhaps more interestingly, we can package up types with axioms restricting their elements. For instance, the type of binary, associative operations on some

type T may be given as

$$\Sigma m{:}T{\to}T{\to}T.\,\Pi x{:}T.\,\Pi y{:}T.\,\Pi z{:}T.\,\mathrm{Id}(m(x,m(y,z)),\,m(m(x,y),z))$$

Here $\Sigma x{:}A.B(x)$ is the type of pairs (a,b) where $a{:}A$ and $b{:}B(a)$ and $\mathrm{Id}(t_1,t_2)$ is the type of proofs of the equality $t_1{=}t_2$. In Martin-L"of's type theory existential quatification is rendered with Σ-types the idea being that a constructive proof of an existential statement $\exists x{:}A.B(x)$ would consist of a member a of type A and a proof, thus a member, of $B(a)$—in other words, an element of $\Sigma a{:}A.B(a)$.

III.2 EXERCISE [★]: Write down a type which represents a constructive version of the axiom of choice, characterised by: *if for every element* a *of a type* A *there exists an element* b *of* B *such that* P(a,b) *then there exists a function* f *mapping an arbitrary* x:A *to an element of* B *such that* P(x, f x). □

III.3 EXERCISE [★]: Suppose that $f{:}A{\to}C$, $g{:}B{\to}C$ are two functions of equal target domain. Using set-theoretic notation we can form their *pullback* as $\{(a,b) \in A \times B \mid f(a) = g(b)\}$. Define an analogous type using Σ and Id. □

Logical frameworks

Dependent types have also found application in the representation of other type theories and formal systems. Suppose that we have an implementation of dependent types and want to get a rough-and-ready type checker for simply typed lambda calculus. We may then make the following declarations:

$$\begin{aligned}
&\mathrm{Ty} &&:: *\\
&\mathrm{Tm} &&:: \mathrm{Ty} \to *\\
&\mathrm{base} &&: \mathrm{Ty}\\
&\mathrm{arrow} &&: \mathrm{Ty} \to \mathrm{Ty} \to \mathrm{Ty}\\
&\mathrm{app} &&: \Pi A{:}\mathrm{Ty}.\,\Pi B{:}\mathrm{Ty}.\,\mathrm{Tm}(\mathrm{arrow}\,A\,B) \to \mathrm{Tm}\,A \to \mathrm{Tm}\,B\\
&\mathrm{lam} &&: \Pi A{:}\mathrm{Ty}.\,\Pi B{:}\mathrm{Ty}.\,(\mathrm{Tm}\,A \to \mathrm{Tm}\,B) \to \mathrm{Tm}(\mathrm{arrow}\,A\,B)
\end{aligned}$$

Here Ty represents the type of simple type expressions and for $A{:}\mathrm{Ty}$ the type $\mathrm{Tm}\,A$ represents the type of lambda terms of type A. We have a constant $\mathrm{base}{:}\mathrm{Ty}$ representing the base type and a function arrow representing the formation of arrow types. As for terms we have a function app that accepts to types A, B, a term of type $\mathrm{arrow}\,A\,B$, a term of type A and yields a term of type B: the application of the two.

Somewhat intriguingly, the function corresponding to lambda abstraction takes a "function" mapping terms of type A to terms of type B and returns a

term of type `arrow A B`. This way of using functions at one level to represent dependencies at another level is particularly useful for representing syntax with binders, and the technique is known as *higher-order abstract syntax*.

With these constants in place we can represent familiar terms such as the identity on `A:Ty` by:

```
idA = lam A A (λx:Tm A.x)
```

or the Church numeral 2 on type `A` by:

```
two = λA:Ty.lam A (arrow (arrow A A) A)
        (λx:Tm A.lam _ _ (λf:Tm(arrow A A).
              app _ _ f (app _ _ f x)));
```

Here we have replaced some obvious type arguments by underscores to aid readability.

Logical frameworks are systems which provide mechanisms for representing syntax and proof systems which make up a logic. The exact representation mechanisms depend upon the framework, but one approach exemplified in the Edinburgh Logical Framework (Harper, Honsell, and Plotkin, 1993a) is suggested by the slogan *judgements-as-types*, where types are used to capture the judgements of a logic.[2]

III.4 EXERCISE [★]: Write down some typing declarations which introduce a judgement expressing an evaluation relation for the representation of simply typed terms shown above. You should begin with a type family `Eval` which is parameterised on a simple type `A` and two terms of type `Tm A`, and declare four terms which represent the rules defining the compatible closure of one-step beta-reduction. □

4.1 Pure first-order dependent types

In this section we introduce one of the simplest systems of dependent types, in a presentation which we call λLF. As the name suggests, this type system is based on a simplified variant of the type system underlying the Edinburgh LF, mentioned above. The λLF type theory generalises simply typed lambda-calculus by replacing the arrow type `S→T` with the dependent product type `Πx:S.T` and by introducing type families. It is pure, in the sense that it has only Π-types; it is first-order, in the sense that it does not include higher-order

2. Judgements are the statements of a logic or a type system. For example, well-formedness, derivability, well-typedness. In LF these judgements are represented as types and derivations of a judgement are represented as members.

λLF

Syntax

t ::=		*terms:*
	x	*variable*
	λx:T.t	*abstraction*
	t t	*application*
T ::=		*types:*
	X	*type/family variable*
	Πx:T.T	*dependent product type*
	T t	*type family application*
K ::=		*kinds:*
	*	*kind of proper types*
	Πx:T.K	*kind of type families*
Γ ::=		*contexts:*
	∅	*empty context*
	Γ, x:T	*term variable binding*
	Γ, X:K	*type variable binding*

Well-formed kinds $\boxed{\Gamma \vdash K}$

$$\Gamma \vdash *$$ (WF-STAR)

$$\frac{\Gamma \vdash T :: * \qquad \Gamma, x:T \vdash K}{\Gamma \vdash \Pi x:T.K}$$ (WF-PI)

Kinding $\boxed{\Gamma \vdash T :: K}$

$$\frac{X :: K \in \Gamma \qquad \Gamma \vdash K}{\Gamma \vdash X :: K}$$ (K-VAR)

$$\frac{\Gamma \vdash T_1 :: * \qquad \Gamma, x:T_1 \vdash T_2 :: *}{\Gamma \vdash \Pi x:T_1.T_2 :: *}$$ (K-PI)

$$\frac{\Gamma \vdash S :: \Pi x:T.K \qquad \Gamma \vdash t : T}{\Gamma \vdash S\ t : [x \mapsto t]K}$$ (K-APP)

$$\frac{\Gamma \vdash T :: K \qquad \Gamma \vdash K \equiv K'}{\Gamma \vdash T :: K'}$$ (K-CONV)

Typing $\boxed{\Gamma \vdash t : T}$

$$\frac{x:T \in \Gamma \qquad \Gamma \vdash T :: *}{\Gamma \vdash x : T}$$ (T-VAR)

$$\frac{\Gamma \vdash S :: * \qquad \Gamma, x:S \vdash t : T}{\Gamma \vdash \lambda x:S.t : \Pi x:S.T}$$ (T-ABS)

$$\frac{\Gamma \vdash t_1 : \Pi x:S.T \qquad \Gamma \vdash t_2 : S}{\Gamma \vdash t_1\ t_2 : [x \mapsto t_2]T}$$ (T-APP)

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash T \equiv T' :: *}{\Gamma \vdash t : T'}$$ (T-CONV)

Figure 4-1: First-order dependent types (λLF)

type operators like those of F^ω. Under the Curry-Howard correspondence, this system corresponds to the $\forall, \rightarrow$-fragment of first-order predicate calculus.

Syntax

The main definition of λLF appears in Figure 4-1 and 4-2.

The terms are the same as those of the simply typed lambda calculus $\lambda_\rightarrow$.

The types include type variables X which can be declared in the context, but never appear bound. Type variables range over proper types as well as type familes such as `Vector :: Nat → *`. We may use type and term variables declared in a fixed initial context to simulate the built-in types and

λLF

Kind Equivalence $\boxed{\Gamma \vdash K \equiv K'}$

$$\frac{\Gamma \vdash T_1 \equiv T_2 :: * \qquad \Gamma, x:T_1 \vdash K_1 \equiv K_2}{\Gamma \vdash \Pi x:T_1.K_1 \equiv \Pi x:T_2.K_2} \text{(QK-PI)}$$

$$\frac{\Gamma \vdash K}{\Gamma \vdash K \equiv K} \text{(QK-REFL)}$$

$$\frac{\Gamma \vdash K_1 \equiv K_2}{\Gamma \vdash K_2 \equiv K_1} \text{(QK-SYM)}$$

$$\frac{\Gamma \vdash K_1 \equiv K_2 \qquad \Gamma \vdash K_2 \equiv K_3}{\Gamma \vdash K_1 \equiv K_3} \text{(QK-TRANS)}$$

Type Equivalence $\boxed{\Gamma \vdash S \equiv T :: K}$

$$\frac{\Gamma \vdash S_1 \equiv T_1 :: * \qquad \Gamma, x:T_1 \vdash S_2 \equiv T_2 :: *}{\Gamma \vdash \Pi x:S_1.S_2 \equiv \Pi x:T_1.T_2 :: *} \text{(QT-PI)}$$

$$\frac{\Gamma \vdash S_1 \equiv S_2 :: \Pi x:T.K \qquad \Gamma \vdash t_1 \equiv t_2 : T}{\Gamma \vdash S_1\ t_1 \equiv S_2\ t_2 : [x \mapsto t_1]K} \text{(QT-APP)}$$

$$\frac{\Gamma \vdash T : K}{\Gamma \vdash T \equiv T :: K} \text{(QT-REFL)}$$

$$\frac{\Gamma \vdash T \equiv S :: K}{\Gamma \vdash S \equiv T :: K} \text{(QT-SYM)}$$

$$\frac{\Gamma \vdash S \equiv U :: K \qquad \Gamma \vdash U \equiv T :: K}{\Gamma \vdash S \equiv T :: K} \text{(QT-TRANS)}$$

Term Equivalence $\boxed{\Gamma \vdash t_1 \equiv t_2 : T}$

$$\frac{\Gamma \vdash S_1 \equiv S_2 :: * \qquad \Gamma, x:S_1 \vdash t_1 \equiv t_2 : T}{\Gamma \vdash \lambda x:S_1.t_1 \equiv \lambda x:S_2.t_2 : \Pi x:S_1.T} \text{(Q-ABS)}$$

$$\frac{\Gamma \vdash t_1 \equiv s_1 :: \Pi x:S.T \qquad \Gamma \vdash t_2 \equiv s_2 : S}{\Gamma \vdash t_1\ t_2 \equiv s_1\ s_2 : [x \mapsto t_2]T} \text{(Q-APP)}$$

$$\frac{\Gamma, x:S \vdash t : T \qquad \Gamma \vdash s : S}{\Gamma \vdash (\lambda x:S.t)\ s \equiv [x \mapsto s]t : [x \mapsto s]T} \text{(Q-BETA)}$$

$$\frac{\Gamma \vdash t : \Pi x:S.T \qquad x \notin FV(t)}{\Gamma \vdash \lambda x:T.t\ x \equiv t : \Pi x:S.T} \text{(Q-ETA)}$$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash t \equiv t : T} \text{(Q-REFL)}$$

$$\frac{\Gamma \vdash t \equiv s : T}{\Gamma \vdash s \equiv t : T} \text{(Q-SYM)}$$

$$\frac{\Gamma \vdash s \equiv u : T \qquad \Gamma \vdash u \equiv t : T}{\Gamma \vdash s \equiv t : T} \text{(Q-TRANS)}$$

Figure 4-2: First-order dependent types (λLF) — Equivalence rules

operators of a programming language.[3] Apart from variables, types may be dependent products or type family applications. The latter allow us to instantiate families, for example, to give types such as `Vector k` for `k:Nat`.

Kinds allow us to distinguish between proper types and type families. Proper types have kind $*$ while type families have dependent product kinds of the form $\Pi x:T.K$.

3. To do this properly one ought to consider a *signature* as a special form of context, and consider the term and type variables declared in the signature to be the constants of the language. This isn't necessary when we move to richer type theories in which it is possible to define data types.

Contexts may bind term variables and type variables.

Type checking rules

The rules in Figure 4-1 define three judgement forms, for checking kind formation, kinding, and typing.

The characteristic typing rules of the system are the abstraction and application rules for terms, altered to use Π-types. The abstraction introduces a dependent product type, checking that the domain type $\mathtt{S}$ is well-formed:

$$\frac{\Gamma \vdash \mathtt{S} :: * \qquad \Gamma, \mathtt{x}:\mathtt{S} \vdash \mathtt{t} : \mathtt{T}}{\Gamma \vdash \lambda \mathtt{x}:\mathtt{S}.\mathtt{t} : \Pi \mathtt{x}:\mathtt{S}.\mathtt{T}} \qquad \text{(T-Abs)}$$

The term application rule eliminates a term with this type, substituting the operand in the Π-type:

$$\frac{\Gamma \vdash \mathtt{t}_1 : \Pi \mathtt{x}:\mathtt{S}.\mathtt{T} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{S}}{\Gamma \vdash \mathtt{t}_1\, \mathtt{t}_2 : [\mathtt{x} \mapsto \mathtt{t}_2]\mathtt{T}} \qquad \text{(T-App)}$$

The well-formedness check in T-Abs uses the kinding rules to ensure that $\mathtt{S}$ is a type. Notice that this check may again invoke the typing rules, in the rule K-App, which checks the instantiation of a type family. The kind formation judgement also invokes the well-formedness of types (in the first premise of Wf-Pi), so the three judgement forms are in fact mutually defined. One consequence of this is that proofs of properties in this system typically proceed by simultaneous proofs for the different judgement forms, using derivation height as an overall measure or alternatively simultaneous structural induction.

There are two conversion rules, K-Conv and T-Conv, which allow us to replace a kind or type with another one that is equivalent.

Kinds have the general form $\Pi \mathtt{x}_1:\mathtt{T}_1 \ldots \mathtt{x}_n:\mathtt{T}_n.*$ but in the typing rules we only ever need to check for proper types, with kind $*$. Nevertheless, we include the K-Conv to ensure that kinding is closed under conversion within the $\mathtt{T}_i$.

We consider the rules defining the equivalences next.

Equivalence rules

One of the main questions in any type system is when two types should be considered equivalent. Type equivalence comes into play in the application rules T-App and K-App. To show that some actual argument has an acceptable type for a function or type family, we may need to use the rule T-Conv to convert the type.

But what should our notion of type equivalence be? Without adding spe-

cial equality axioms, we can consider natural notions of equality which arise from the type structure. With dependent types, a natural notion is to equate types that differ only in their term components, when those term components themselves should be considered equal. So the question is reduced to considering notions of term equality.

A first example is a type-family application containing a β-redex, since we consider β-equivalent λ-terms to be equal:

```
T ((λx:S.x) z)  ≡ T z
```

A slightly different and more concrete example is two different applications of the `Vector` family:

```
Vector (3 + 4)  ≡ Vector 7
```

It seems reasonable that a type-checker should accept each of these pairs of types as being equivalent. But we quickly come across more complex equivalences involving more computation, or even, requiring proof in the general case. For example, supposing `x` is an unknown value of type `Nat` and `f:Nat → Nat` is a function whose behaviour is known. If it happens that `f x=7` for all `x` then we have:

```
Vector (f x) ≡ Vector 7
```

but this equality could be more difficult to add to an automatic type-checker.

The question of what form of type equivalence to include in a system of dependent types is therefore a central consideration in the design of the system. Many different choices have been studied, leading to systems with fundamentally different character: the most notable distinction being whether or not type-checking is decidable.

In the first case, we may choose to include only basic equalities which are manifestly obvious. This is the viewpoint favoured by Martin-Löf, who considers *definitional equality* to be the proper notion of equivalence. The first two equalities above are definitional: `3 + 4` is definitionally equal to `7` by the rules of computation for addition. Alternatively, one may prefer to include as many equalities as possible, to make the theory more powerful. This is the approach followed, for example, in the type theory implemented by the NuPrl system (Constable, 1986). This formulation of type theory includes type equivalences like the third example above, which may require arbitrary computation or proof to establish. In such a type system, type-checking is undecidable.

For λLF, we axiomatise definitional equality based on the type structure, which includes β and η equality on lambda terms. It is possible to define this

using a relation of equality defined via compatible closure of untyped reduction (this is the approach followed by Pure Type Systems, see Section 4.6). Instead, we give a declarative, typed definition of equivalence, using rules which follow the same pattern as the typing rules. The advantage of this approach is that it is more extensible than the "untyped" approach and avoids the need to establish properties of untyped reduction. See Chapter 2 in this volume for further explanation of the issues here.

The rules for equivalence are shown in Figure 4-2. Again there are three judgements, for equivalence of each of the syntactic categories of terms, types, and kinds. The only interesting rules are Q-BETA and Q-ETA which introduce β and η-equivalence on terms; the remaining rules are purely structural and express that equivalence is a congruence.

4.2 Properties

In this section we mention some basic properties of λLF. We don't go very far: in Section 4.3 we introduce an algorithmic presentation of λLF which allows us to establish further properties of the system indirectly but rather more easily.

Basic properties

The following properties use some additional notation. Inclusion is defined between contexts as $\Gamma \subseteq \Delta$ iff $\mathtt{x}:\mathtt{T} \in \Gamma$ implies $\mathtt{x}:\mathtt{T} \in \Delta$, in other words, $\Gamma \subseteq \Delta$ means that Δ is a permutation of an extension of Γ. We write $\Gamma \vdash \mathcal{J}$ for an arbitrary judgement, amongst the six defined in Figures 4-1 and 4-2. We write $\Gamma \vdash \mathtt{K}, \mathtt{K}'$ to stand for both $\Gamma \vdash \mathtt{K}$ and $\Gamma \vdash \mathtt{K}'$, and similarly for other judgements.

4.2.1 LEMMA [PERMUTATION AND WEAKENING]: Suppose $\Gamma \subseteq \Delta$. Then $\Gamma \vdash \mathcal{J}$ implies $\Delta \vdash \mathcal{J}$. □

4.2.2 LEMMA [SUBSTITUTION]: If $\Gamma, \mathtt{x}:\mathtt{S}, \Delta \vdash \mathcal{J}$ and $\Gamma \vdash \mathtt{s} : \mathtt{S}$, then $\Gamma, [\mathtt{x} \mapsto \mathtt{s}]\Delta \vdash [\mathtt{x} \mapsto \mathtt{s}]\mathcal{J}$. □

4.2.3 LEMMA [AGREEMENT]: Judgements in the system are in agreement, as follows:

1. If $\Gamma \vdash \mathtt{T} :: \mathtt{K}$ then $\Gamma \vdash \mathtt{K}$.
2. If $\Gamma \vdash \mathtt{t} : \mathtt{T}$ then $\Gamma \vdash \mathtt{T} :: *$.
3. If $\Gamma \vdash \mathtt{K} \equiv \mathtt{K}'$ then $\Gamma \vdash \mathtt{K}, \mathtt{K}'$.

4. If $\Gamma \vdash \mathtt{T} \equiv \mathtt{T}' :: \mathtt{K}$ then $\Gamma \vdash \mathtt{T}, \mathtt{T}' :: \mathtt{K}$.

5. If $\Gamma \vdash \mathtt{t} \equiv \mathtt{t}' : \mathtt{T}$ then $\Gamma \vdash \mathtt{t}, \mathtt{t}' : \mathtt{T}$. □

4.2.4 EXERCISE [★★↛]: Prove the lemmas above. □

Strong Normalization

As an auxiliary device for the soundness and completeness of algorithmic type checking we will now introduce general beta reduction which permits reductions within the scope of abstractions.

We define beta reduction on λLF terms by the four rules:

$$\frac{\mathtt{t}_1 \longrightarrow_\beta \mathtt{t}_1'}{\lambda \mathtt{x}:\mathtt{T}_1.\mathtt{t}_1 \longrightarrow_\beta \lambda \mathtt{x}:\mathtt{T}_1.\mathtt{t}_1'} \quad \text{(BETA-ABS)}$$

$$\frac{\mathtt{t}_1 \longrightarrow_\beta \mathtt{t}_1'}{\mathtt{t}_1\ \mathtt{t}_2 \longrightarrow_\beta \mathtt{t}_1'\ \mathtt{t}_2} \quad \text{(BETA-APP1)}$$

$$\frac{\mathtt{t}_2 \longrightarrow_\beta \mathtt{t}_2'}{\mathtt{t}_1\ \mathtt{t}_2 \longrightarrow_\beta \mathtt{t}_1\ \mathtt{t}_2'} \quad \text{(BETA-APP2)}$$

$$(\lambda \mathtt{x}:\mathtt{T}_1.\mathtt{t}_1)\ \mathtt{t}_2 \longrightarrow_\beta [\mathtt{x} \mapsto \mathtt{t}_2]\mathtt{t}_1 \quad \text{(BETA-APPABS)}$$

Notice that this reduction does not go inside the type labels of λ abstractions.

The following central result is required to ensure termination of type checking, proved in the next section.

4.2.5 THEOREM [STRONG NORMALIZATION]: The relation $\longrightarrow_\beta$ is strongly normalising on well-typed terms. More precisely, if $\Gamma \vdash \mathtt{t} : \mathtt{T}$ then there is no infinite sequence of terms $(\mathtt{t}_i)_{i \geq 1}$ such that $\mathtt{t} = \mathtt{t}_1$ and $\mathtt{t}_i \longrightarrow_\beta \mathtt{t}_{i+1}$ for $i \geq 1$. □

Proof: This can be proved by defining a reduction preserving translation from λLF to the simply-typed lambda-calculus as follows:

- for every type variable $\mathtt{X}$, no matter of what kind, we introduce a simple type variable $\mathtt{X}^\natural$,
- for each type expression $\mathtt{T}$, no matter of what kind we define a simple type expression $\mathtt{T}^\natural$ by $\Pi \mathtt{x}:\mathtt{S}.\mathtt{T}^\natural = \mathtt{S}^\natural \to \mathtt{T}^\natural$ and $(\mathtt{T}\ \mathtt{t})^\natural = \mathtt{T}^\natural$.
- the mapping $-^\natural$ is extended to terms and contexts by applying it to all type expressions occurring within.

Now we can show by induction on typing derivations in λLF that $\Gamma \vdash t : T$ implies $\Gamma^\natural \vdash t^\natural : T^\natural$, from which the result follows by appealing to the strong normalisation theorem for β-reduction of the simply typed lambda calculus. □

Since $\longrightarrow_\beta$ is finitely branching, this implies that for each term t there exists a number $\mu(t)$ with the property that if $(t_i)_{1 \le i \le k}$ is a reduction sequence starting from t, that is, $t{=}t_1$ and $t_i \longrightarrow_\beta t_{i+1}$ for $1 \le i < k$ then $k \le \mu(t)$. A term t' such that $t \longrightarrow_\beta^* t'$ and $t' \not\longrightarrow_\beta$ is called a (β) *normal form* of t. Since $\longrightarrow_\beta$ is confluent, normal forms are unique and we may write $t' = \mathrm{nf}(t)$.

4.3 Algorithmic typing and equality

To implement λLF, we need to find a formulation of the system which is closer to an algorithm. As usual, we follow the strategy of reformulating the rules to be *syntax directed*, so that they can be used to define an algorithm going from premises to conclusion (for further explanation, see the description of the implementation below in Section 4.7) . We also need to find an algorithm for deciding type equivalence in the cases where it needs to be tested.

The algorithmic presentation of λLF is shown in Figures 4-3 and 4-4. The judgements mirror the defining presentation, with the addition of a context checking judgement. (This is used only to check an initial context: the rules otherwise maintain context well-formation when extending contexts going from conclusions to premises).

The non-syntax directed rules K-CONV and T-CONV are removed. To replace T-CONV, we add equivalence testing in the algorithmic rules for applications, KA-APP and TA-APP.

The equivalence testing rules in Figure 4-4 assume that they are invoked on well-typed phrases. We show these rules with contexts Γ to facilitate extensions to type-dependent equalities or definitions in the context (used in the implementation), although in the rules for pure λLF, the context plays no role in equivalence testing.

The equivalence testing algorithm on terms that is suggested by these rules is similar to the one described in Chapter 2, but we do not make use of type information. (Similarly, the type equivalence rules do not record kinds). The algorithmic judgement $\Gamma \vdash\!\blacktriangleright s \equiv t$ for arbitrary terms is defined mutually with $\Gamma \vdash\!\blacktriangleright s \equiv_{\mathrm{wh}} t$ which is defined between *weak head normal forms*. Weak head reduction is a subset of the β reduction $\longrightarrow_\beta$, defined by the rules:

Algorithmic kind formation $\boxed{\Gamma \vdash K}$

$$\frac{}{\Gamma \vdash *} \quad \text{(WFA-Star)}$$

$$\frac{\Gamma \vdash T :: * \qquad \Gamma, x{:}T \vdash K}{\Gamma \vdash \Pi x{:}T.K} \quad \text{(WFA-Pi)}$$

Algorithmic kinding $\boxed{\Gamma \vdash T :: K}$

$$\frac{X :: K \in \Gamma}{\Gamma \vdash X :: K} \quad \text{(KA-Var)}$$

$$\frac{\Gamma \vdash T_1 :: * \qquad \Gamma, x{:}T_1 \vdash T_2 :: *}{\Gamma \vdash \Pi x{:}T_1.T_2 :: *} \quad \text{(KA-Pi)}$$

$$\frac{\Gamma \vdash S :: \Pi x{:}T_1.K \qquad \Gamma \vdash t : T_2 \qquad \Gamma \vdash T_1 \equiv T_2}{\Gamma \vdash S\ t : [x \mapsto t]K} \quad \text{(KA-App)}$$

Algorithmic typing $\boxed{\Gamma \vdash t : T}$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(TA-Var)}$$

$$\frac{\Gamma \vdash S :: * \qquad \Gamma, x{:}S \vdash t : T}{\Gamma \vdash \lambda x{:}S.t : \Pi x{:}S.T} \quad \text{(TA-Abs)}$$

$$\frac{\Gamma \vdash t_1 : \Pi x{:}S_1.T \qquad \Gamma \vdash t_2 : S_2 \qquad \Gamma \vdash S_1 \equiv S_2}{\Gamma \vdash t_1\ t_2 : [x \mapsto t_2]T} \quad \text{(TA-App)}$$

Figure 4-3: Algorithmic presentation of λLF

$$\frac{t_1 \longrightarrow_{wh} t_1'}{t_1\ t_2 \longrightarrow_{wh} t_1'\ t_2} \quad \text{(WH-App1)}$$

$$(\lambda x{:}T_1.t_1)\ t_2 \longrightarrow_{wh} [x \mapsto t_2]t_1 \quad \text{(WH-AppAbs)}$$

Weak head reduction only applies β-reduction in the head position. The implementation described in Section 4.7 adds expansion of definitions to this reduction; see Chapter 10 for a thorough treatment of how to do this.

4.3.1 Theorem [Weak head normal forms]: If $\Gamma \vdash t{:}T$ then there exists a unique term $t' = \text{whnf}(t)$ such that $t \longrightarrow_{wh}^* t' \not\longrightarrow_{wh}$. □

The theorem is a direct consequence of Theorem 4.2.5 and of the fact that $\longrightarrow_{wh}$ is deterministic (a partial function).

Correctness of the algorithm

We will now show that the type-checking algorithm defined by the algorithmic rules is sound, complete, and terminates on all inputs.

Since the algorithm checks the context only as it is extended, and (for efficiency) does not check the type of variables in the leaves, we can only expect to show soundness for contexts which are well-formed. The soundness lemma makes use of an auxiliary algorithmic judgement for context formation, defined by three rules:

Algorithmic kind equivalence $\boxed{\Gamma \vdash_\blacktriangleright K \equiv K'}$

$$\Gamma \vdash_\blacktriangleright * \equiv * \quad \text{(QKA-Star)}$$

$$\frac{\Gamma \vdash_\blacktriangleright T_1 \equiv T_2 \qquad \Gamma, x:T_1 \vdash_\blacktriangleright K_1 \equiv K_2}{\Gamma \vdash_\blacktriangleright \Pi x:T_1.K_1 \equiv \Pi x:T_2.K_2} \quad \text{(QKA-Pi)}$$

Algorithmic type equivalence $\boxed{\Gamma \vdash_\blacktriangleright S \equiv T}$

$$\Gamma \vdash X \equiv X \quad \text{(QTA-Var)}$$

$$\frac{\Gamma \vdash_\blacktriangleright S_1 \equiv T_1 \qquad \Gamma, x:T_1 \vdash_\blacktriangleright S_2 \equiv T_2}{\Gamma \vdash \Pi x:S_1.S_2 \equiv \Pi x:T_1.T_2} \quad \text{(QTA-Pi)}$$

$$\frac{\Gamma \vdash_\blacktriangleright S_1 \equiv S_2 \qquad \Gamma \vdash_\blacktriangleright t_1 \equiv t_2}{\Gamma \vdash_\blacktriangleright S_1\ t_1 \equiv S_2\ t_2} \quad \text{(QTA-App)}$$

Algorithmic term equivalence $\boxed{\Gamma \vdash_\blacktriangleright s \equiv t}$

$$\frac{\Gamma \vdash_\blacktriangleright \text{whnf}(s) \equiv_{\text{wh}} \text{whnf}(t)}{\Gamma \vdash_\blacktriangleright s \equiv t} \quad \text{(QA-WH)}$$

$$\Gamma \vdash_\blacktriangleright x \equiv_{\text{wh}} x \quad \text{(QA-Var)}$$

$$\frac{\Gamma, x:S \vdash_\blacktriangleright t_1 \equiv t_2}{\Gamma \vdash_\blacktriangleright \lambda x:S.t_1 \equiv_{\text{wh}} \lambda x:S.t_2} \quad \text{(QA-Abs)}$$

$$\frac{\Gamma \vdash_\blacktriangleright s_1 \equiv s_2 \qquad \Gamma \vdash_\blacktriangleright t_1 \equiv t_2}{\Gamma \vdash_\blacktriangleright s_1\ t_1 \equiv_{\text{wh}} s_2\ t_2} \quad \text{(QA-App)}$$

$$\frac{\Gamma, x:S \vdash_\blacktriangleright s\ x \equiv t \qquad s \text{ not a } \lambda}{\Gamma \vdash_\blacktriangleright s \equiv_{\text{wh}} \lambda x:S.t} \quad \text{(QA-Nabs1)}$$

$$\frac{\Gamma, x:S \vdash_\blacktriangleright s \equiv t\ x \qquad t \text{ not a } \lambda}{\Gamma \vdash_\blacktriangleright \lambda x:S.s \equiv_{\text{wh}} t} \quad \text{(QA-Nabs2)}$$

Figure 4-4: Algorithmic presentation of λLF— Equivalence rules

$$\vdash_\blacktriangleright \emptyset \quad \text{(WFA-Empty)}$$

$$\frac{\vdash_\blacktriangleright \Gamma \qquad \Gamma \vdash_\blacktriangleright T :: *}{\vdash_\blacktriangleright \Gamma, x:T} \quad \text{(WFA-Tm)}$$

$$\frac{\vdash_\blacktriangleright \Gamma \qquad \Gamma \vdash_\blacktriangleright K}{\vdash_\blacktriangleright \Gamma, X::K} \quad \text{(WFA-Ty)}$$

4.3.2 Lemma [Soundness of algorithmic λLF]: Each of the algorithmic judgements is sound, in the following sense:

1. If $\Gamma \vdash_\blacktriangleright K$ then $\Gamma \vdash K$
2. If $\Gamma \vdash_\blacktriangleright T :: K$ then $\Gamma \vdash T :: K$
3. If $\Gamma \vdash_\blacktriangleright t : T$ then $\Gamma \vdash t : T$
4. If $\Gamma \vdash_\blacktriangleright K, K'$ and $\Gamma \vdash_\blacktriangleright K \equiv K'$, then $\Gamma \vdash K \equiv K'$
5. If $\Gamma \vdash_\blacktriangleright T, T' :: K$ and $\Gamma \vdash_\blacktriangleright T \equiv T'$ then $\Gamma \vdash T \equiv T' :: K$.
6. If $\Gamma \vdash_\blacktriangleright t, t' : T$ and $\Gamma \vdash_\blacktriangleright t \equiv t'$ then $\Gamma \vdash t \equiv t' :: K$.

where in each case, we additionally assume $\vdash_\blacktriangleright \Gamma$. □

Proof: By induction on algorithmic derivations. □

4.3.3 LEMMA [COMPLETENESS OF ALGORITHMIC λLF]: Each of the algorithmic judgements is complete, in the following sense:

1. If $\Gamma \vdash \mathtt{K}$ then $\Gamma \mapsto \mathtt{K}$.
2. If $\Gamma \vdash \mathtt{T} : \mathtt{K}$ then for some $\mathtt{K}'$, we have $\Gamma \mapsto \mathtt{T} : \mathtt{K}'$ and $\Gamma \mapsto \mathtt{K} \equiv \mathtt{K}'$ and $\Gamma \mapsto \mathtt{K}'$.
3. If $\Gamma \vdash \mathtt{t} : \mathtt{T}$ then for some $\mathtt{T}'$, we have $\Gamma \mapsto \mathtt{t} : \mathtt{T}'$ and $\Gamma \mapsto \mathtt{T} \equiv \mathtt{T}'$ and $\Gamma \mapsto \mathtt{T}' :: *$.
4. If $\Gamma \vdash \mathtt{t}_1 \equiv \mathtt{t}_2 : \mathtt{T}$ then $\Gamma \mapsto \mathtt{t}_1 \equiv \mathtt{t}_2$.
5. If $\Gamma \vdash \mathtt{T}_1 \equiv \mathtt{T}_2 :: \mathtt{K}$ then $\Gamma \mapsto \mathtt{T}_1 \equiv \mathtt{T}_2$. □

Proof: By induction on derivations in the declarative system. □

Given soundness and completeness, we also want to know that our algorithm terminates on all inputs. This also demonstrates the decidability of the original judgements.

4.3.4 THEOREM [TERMINATION OF TYPE-CHECKING]: The algorithmic presentation yields a terminating algorithm for type checking. □

We highlight the crucial ideas of the proof of Theorem 4.3.4 here; the details are left to the diligent reader (Exercise 4.3.6 below). First, we note that the theorem must be generalised to terms-in-context, and must require well-formedness of the context (if it contained unkindable types, the algorithm might loop). We must also include statements for the other algorithmic judgements.

The equivalence judgement $\Gamma \mapsto \mathtt{t}_1 \equiv \mathtt{t}_2$ introduces a possible source of nontermination when invoked on non-well-typed terms (e.g., $\Omega = \Delta\Delta$ where $\Delta = \lambda \mathtt{x}:\mathtt{A}.\,\mathtt{x}\ \mathtt{x}$). Here, computation of weak head normal form runs into an infinite loop. We must be careful that equivalence testing is called only on well-typed terms.

The crucial termination property for term equality that we need is provided by the following lemma.

4.3.5 LEMMA: Suppose that $\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1$ and $\Gamma \vdash \mathtt{t}_2 : \mathtt{T}_2$. Then the syntax-directed backwards search for a derivation of $\Gamma \mapsto \mathtt{t}_1 \equiv \mathtt{t}_2$ always terminates. Equivalently, there is no infinite derivation ending in $\Gamma \mapsto \mathtt{t}_1 \equiv \mathtt{t}_2$. □

Proof: Recall that $\mu(s)$ denotes an upper bound on the length of any $\longrightarrow_\beta$ reduction sequence starting from s. We write $|s|$ for the size of the term s.

We associate an ω^2-valued weight to each judgement arising in a possible derivation of an equality judgement by

$$w(\Delta \vdash s_1 \equiv s_2) = w(\Delta \vdash s_1 \equiv_{wh} s_2) + 1$$
$$w(\Delta \vdash s_1 \equiv_{wh} s_2) = \omega \cdot (\mu(s_1) + \mu(s_2)) + |s_1| + |s_2| + 1$$

We claim that the weight of any premise of a rule is always less than the weight of the conclusion which excludes any infinite derivation tree. In other words we argue by induction on the length of reduction sequences and, subordinately, on the size of terms.

The claimed property is obviously satisfied for rules QA-WH, QA-ABS, QA-APP. To deal with rule QA-NABS1 (and similarly, QA-NABS2) we note that s must be of the form $y\ u_1 \ldots u_n$ whereby $\mu(s\ x) = \mu(s)$. The size, however, goes down, as the λ-symbol has disappeared. □

4.3.6 EXERCISE [★★★↛]: Complete the proof of Lemmas 4.3.2 and 4.3.3 and Theorem 4.3.4. □

Properties of λLF

We can use the algorithmic presentation of λLF to prove additional properties enjoyed by the main definition. We just mention one example: type preservation under β-reduction.

4.3.7 THEOREM [PRESERVATION]: If $\Gamma \vdash t : T$ and $t \longrightarrow_\beta t'$, then $\Gamma \vdash t' : T$ also. □

Proof: We show a slightly restricted form of the theorem, for well-formed contexts Γ. More precisely, well-formed contexts are those which can be built using the same rules as for $\vdash \Gamma$ (page 268), but in the declarative system; the corresponding assertion is written $\vdash \Gamma$. It is easy to extend the completeness lemma to show that $\vdash \Gamma$ implies $\vdash \Gamma$.

The crucial case is that of an outermost β-reduction (BETA-APPABS), when $t = (\lambda x{:}T_1.t_1)\ t_2$ for some T_1, t_1, t_2.

By Lemma 4.3.3, we know that $\Gamma \vdash (\lambda x{:}T_1.t_1)\ t_2 : T'$ for some T' with $\Gamma \vdash T \equiv T'$ and $\Gamma \vdash T' :: *$. The first judgement must have been derived with TA-APP preceded by TA-ABS, so we have the derivability of:

$$\Gamma \vdash T_1 :: *$$
$$\Gamma, x{:}T_1 \vdash t_1 : S_2$$
$$\Gamma \vdash t_2 : S_1$$
$$\Gamma \vdash T_1 \equiv S_1$$

Extends λLF *(4-1 and 4-2)*

New syntax

t ::=	...	*terms:*
	(t, t:Σx:T.T)	*typed pair*
	t.1	*first projection*
	t.2	*second projection*
T ::=	...	*types:*
	Σx:T.T	*dependent sum type*

Kinding $\boxed{\Gamma \vdash T :: K}$

$$\frac{\Gamma \vdash S :: * \qquad \Gamma, x{:}S \vdash T :: *}{\Gamma \vdash \Sigma x{:}S.T :: *} \text{ (K-SIGMA)}$$

Typing $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \vdash \Sigma x{:}S.T :: * \qquad \Gamma \vdash t_1 : S \qquad \Gamma \vdash t_2 : [x \mapsto t_1]T}{\Gamma \vdash (t_1, t_2 : \Sigma x{:}S.T) : \Sigma x{:}S.T} \text{ (T-PAIR)}$$

$$\frac{\Gamma \vdash t : \Sigma x{:}S.T}{\Gamma \vdash t.1 : S} \text{ (T-PROJ1)}$$

$$\frac{\Gamma \vdash t : \Sigma x{:}S.T}{\Gamma \vdash t.2 : [x \mapsto t.1]T} \text{ (T-PROJ2)}$$

Term Equivalence $\boxed{\Gamma \vdash t_1 \equiv t_2 : T}$

$$\frac{\Gamma \vdash \Sigma x{:}S.T :: * \qquad \Gamma \vdash t_1 : S \qquad \Gamma \vdash t_2 : [x \mapsto t_1]T}{\Gamma \vdash (t_1, t_2 : \Sigma x{:}S.T).1 = t_1 : \Sigma x{:}S.T} \text{ (Q-PROJ1)}$$

$$\frac{\Gamma \vdash \Sigma x{:}S.T :: * \qquad \Gamma \vdash t_1 : S \qquad \Gamma \vdash t_2 : [x \mapsto t_1]T}{\Gamma \vdash (t_1, t_2 : \Sigma x{:}S.T).2 = t_2 : \Sigma x{:}S.T} \text{ (Q-PROJ2)}$$

$$\frac{\Gamma \vdash t : \Sigma x{:}S.T}{\Gamma \vdash (t.t1, t.t2 : \Sigma x{:}S.T) \equiv t} \text{ (Q-SURJPAIR)}$$

Figure 4-5: Dependent sum types

in the algorithmic system, with $T' = [x \mapsto t_2]S_2$.

By the above and the assumptions about Γ, we have $\mapsto \Gamma, x{:}T_1$. Hence by Lemma 4.3.2, we can go back to get analogues of the statements above in the declarative system. For the last case, to establish the equivalence $\Gamma \vdash T_1 \equiv S_1 :: *$ we use Lemma 4.2.3 to get $\Gamma \vdash S_1 :: *$ and then $\Gamma \mapsto S_1 :: *$.

Now with substitution, Lemma 4.2.2, we get $\Gamma \vdash [x \mapsto t_2]t_1 : [x \mapsto t_2]S_2$ and then the result follows using T-CONV, with another hop between the systems and Lemma 4.2.3, to show the equivalence $\Gamma \vdash [x \mapsto t_2]S_2 \equiv T :: *$. □

4.3.8 EXERCISE [★★★↛]: Generalise the proof above to arbitrary contexts Γ. □

4.4 Dependent sum types

Figure 4-5 shows extensions to λLF to add dependent sum (or "Sigma") types, written $\Sigma x : T_1 . T_2$. Dependent sums were motivated briefly in the introduction. They generalise ordinary product types in a similar way to the way

that dependent products generalise ordinary function spaces. The degenerate non-dependent case, when x does not appear free in T_2, amounts to the ordinary product, written as $T_1 \times T_2$.

We extend the terms and types of λLF given in Figure 4-1 with pairs, projection operations, and the type constructor itself. Notice that the pair (t_1, t_2) is annotated explicitly with a type $\Sigma x{:}T_1.T_2$ in the syntax. This is necessary because the choice of dependent sum type is not determined by (either or both components of) the pair itself: we could assign the pair `(10,5)` either of the types $\Sigma x{:}\mathtt{Nat}.\mathtt{LessThan}(x)$ or $\Sigma x{:}\mathtt{Nat}.\mathtt{Divisors}(x)$, for example. In a practical programming language, however, these type annotations would be too heavy and might be constructed in some other way; perhaps from the surrounding context, or with the use of type names.

The most cluttered typing rule is the one which introduces a dependent pair, T-PAIR. It must check first that the Σ-type itself is allowed, and then that each component has the requested type. The projection rules are straightforward: compare the second projection with the application rule T-APP in Figure 4-1.

The equality relation on terms is extended to Σ-types by three rules. The first two define the elimination behaviour of projections on a pair (compare with the beta rule for Π-types). The third rule, Q-SURJPAIR, is known as *surjective pairing*. This rule is a form of eta rule for Σ-types: it states that every pair can be formed using the pair constructor.

Algorithmic typing with dependent sum types

To extend the algorithm to deal with Σ-types, we first extend the notions of β reduction and weak-head reduction. In both cases, the main clause is projection on a pair. Beta reduction also allows reduction inside the components of a pair.

$$(t_1, t_2{:}T).i \longrightarrow_\beta t_i \qquad \text{(BETA-PROJPAIR)}$$

$$\frac{t \longrightarrow_\beta t'}{t.i \longrightarrow_\beta t'.i} \qquad \text{(BETA-PROJ)}$$

$$\frac{t_1 \longrightarrow_\beta t_1'}{(t_1, t_2{:}T) \longrightarrow_\beta (t_1', t_2{:}T)} \qquad \text{(BETA-PAIR1)}$$

$$\frac{t_2 \longrightarrow_\beta t_2'}{(t_1, t_2{:}T) \longrightarrow_\beta (t_1, t_2{:}T)} \qquad \text{(BETA-PAIR2)}$$

Weak head reduction just has two new cases:

$$(t_1, t_2{:}T).i \longrightarrow_{wh} t_i \qquad \text{(WH-PROJPAIR)}$$

Extends λLF *algorithm (4-3 and 4-4)*

Algorithmic kinding $\boxed{\Gamma \vdash_{\blacktriangleright} T :: K}$

$$\frac{\Gamma \vdash_{\blacktriangleright} T_1 :: * \qquad \Gamma, x{:}T_1 \vdash_{\blacktriangleright} T_2 :: *}{\Gamma \vdash_{\blacktriangleright} \Sigma x{:}T_1.T_2 :: *}\ (\text{KA-SIGMA})$$

Algorithmic typing $\boxed{\Gamma \vdash_{\blacktriangleright} t : T}$

$$\frac{\begin{array}{l}\Gamma \vdash_{\blacktriangleright} \Sigma x{:}T_1.T_2 :: * \\ \Gamma \vdash_{\blacktriangleright} t_1 : T_1' \qquad \Gamma \vdash_{\blacktriangleright} T_1' \equiv T_1 \\ \Gamma \vdash_{\blacktriangleright} t_2 : T_2' \qquad \Gamma \vdash_{\blacktriangleright} T_2' \equiv [x \mapsto t_1]T_2\end{array}}{\Gamma \vdash_{\blacktriangleright} (t_1, t_2 : \Sigma x{:}T_1.T_2) : \Sigma x{:}T_1.T_2}\ (\text{TA-PAIR})$$

$$\frac{\Gamma \vdash_{\blacktriangleright} t : \Sigma x{:}T_1.T_2}{\Gamma \vdash_{\blacktriangleright} t.1 : T_1}\ (\text{TA-PROJ1})$$

$$\frac{\Gamma \vdash_{\blacktriangleright} t : \Sigma x{:}T_1.T_2}{\Gamma \vdash_{\blacktriangleright} t.2 : [x \mapsto t.1]T_2}\ (\text{TA-PROJ2})$$

Algorithmic type equivalence $\boxed{\Gamma \vdash_{\blacktriangleright} S \equiv T}$

$$\frac{\Gamma \vdash_{\blacktriangleright} S_1 \equiv T_1 \qquad \Gamma, x{:}T_1 \vdash_{\blacktriangleright} S_2 \equiv T_2}{\Gamma \vdash_{\blacktriangleright} \Sigma x{:}S_1.S_2 \equiv \Sigma x{:}T_1.T_2}\ (\text{QTA-SIGMA})$$

Algorithmic term equivalence $\boxed{\Gamma \vdash_{\blacktriangleright} t \equiv_{\text{wh}} t'}$

$$\frac{\Gamma \vdash_{\blacktriangleright} t_i \equiv t'_i}{\Gamma \vdash_{\blacktriangleright} (t_1, t_2 : T) \equiv_{\text{wh}} (t'_1, t'_2 : T')}\ (\text{QA-PAIR})$$

$$\frac{\Gamma \vdash_{\blacktriangleright} t_i \equiv t.i \qquad t \text{ not a pair}}{\Gamma \vdash_{\blacktriangleright} (t_1, t_2 : T) \equiv_{\text{wh}} t}\ (\text{QA-PAIR-NE})$$

$$\frac{\Gamma \vdash_{\blacktriangleright} t.i \equiv t_i \qquad t \text{ not a pair}}{\Gamma \vdash_{\blacktriangleright} t \equiv_{\text{wh}} (t_1, t_2 : T)}\ (\text{QA-NE-PAIR})$$

Figure 4-6: Algorithmic typing for Σ-types

$$\frac{t \longrightarrow_{\text{wh}} t'}{t.i \longrightarrow_{\text{wh}} t'.i}\ (\text{WH-PROJ})$$

Using the weak head reduction, the algorithmic typing and equality judgements are extended with the rules in Figure 4-6 to deal with Σ-types.

We leave the verification of this algorithmic presentation to the reader; no surprises are to be expected.

4.4.1 EXERCISE [★★★↛]: Extend Lemmas 4.3.2, 4.3.3 and 4.3.4 to Σ-types. □

4.5 The Calculus of Constructions

The Calculus of Constructions (CC) is one of the most famous system of dependent types. It was introduced by Coquand and Huet (Coquand and Huet, 1988) as a system in which to do all of constructive mathematics. While it has subsequently turned out that CC needs to be extended with certain features (in particular inductive types (Mohring, 1986)), its simplicity in relationship to its expressivity is unprecedented.

In our framework CC can be formulated as an extension of λLF which has a new basic type `Prop` and a new type family `Prf`. Elements of the type `Prop`

Extends λLF *(4-1 and 4-2)*

New syntax

t	::=	...	*terms:*
		all x:T.t	*universal quantification*
T	::=	...	*types:*
		Prop	*propositions*
		Prf	*family of proofs*

Kinding $\boxed{\Gamma \vdash \mathtt{T} :: \mathtt{K}}$

$$\frac{}{\Gamma \vdash \mathtt{Prop} :: *} \quad \text{(K-PROP)}$$

$$\frac{}{\Gamma \vdash \mathtt{Prf} :: \Pi \mathtt{x}:\mathtt{Prop}.\ *} \quad \text{(K-PRF)}$$

Typing $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\Gamma \vdash \mathtt{T} :: * \qquad \Gamma, \mathtt{x}:\mathtt{T} \vdash \mathtt{t} : \mathtt{Prop}}{\Gamma \vdash \mathtt{all\ x}:\mathtt{T}.\mathtt{t} : \mathtt{Prop}} \quad \text{(T-ALL)}$$

Type Equivalence $\boxed{\Gamma \vdash \mathtt{S} \equiv \mathtt{T} :: \mathtt{K}}$

$$\frac{\Gamma \vdash \mathtt{T} :: * \qquad \Gamma, \mathtt{x}:\mathtt{T} \vdash \mathtt{t} : \mathtt{Prop}}{\Gamma \vdash \mathtt{Prf\ (all\ x}:\mathtt{T}.\mathtt{t)} \equiv \Pi \mathtt{x}:\mathtt{T}.\mathtt{Prf\ t} :: *} \quad \text{(QT-ALL)}$$

Figure 4-7: The Calculus of Constructions

represent propositions, and also datatypes. Propositions and datatypes are identified in CC by taking the Curry-Howard isomorphism as an identity. The type family `Prf` assigns to each proposition or datatype `p : Prop` the type `Prf p` of its proofs, or, in the case of datatypes, its members. CC has a one new term former `all x:T.t`, and two rules which relate it to `Prf`. The additions to λLF are shown in Figure 4-7.

In most presentations and implementations of CC the type `Prf t` is notationally identified with the term `t`. This is convenient and enhances readability, however, we will not adopt it for the sake of compatibility. The original formulation of CC went as far as using the same notation, namely `(x:A)` for all three binders: Π, `all`, λ. That did not enhance readability at all and was thus given up after some time!

CC contains F^{ω} as a subsystem by an obvious translation which we introduce by example. Here is the type of an encoding of natural numbers in CC:

```
nat = all a:Prop.all z:Prf(a).all s:Prf(a)→Prf(a). a;
```

Recall that `A→B` abbreviates `Πx:A.B`.

Notice that `nat` is a member of type `Prop`. The natural numbers inhabit the type `Prf nat`. Accordingly, we have

```
zero = λa:Prop.λz:Prf(a).λs:Prf(a)→Prf(a).z : Prf(nat);

succ = λn:Prf nat.λa:Prop.λz:Prf(a).
          λs:Prf(a)→Prf(a).s(n a z s) : Prf nat → Prf nat;
```

```
add = λm:Nat.λn:Nat.m nat n succ : Prf nat → Prf nat → Prf nat;
```

Regarding higher-order polymorphism here is how we define existential types in CC:

```
exists = λf:Prop→Prop.all c:Prop.all m:Πx:Prop.Prf(f x)→Prf(c).c
```

4.5.1 EXERCISE [★↛]: Define the term corresponding to existential introduction of type: `Πf:Prop→Prop.Πa:Prop.Πi:Prf (f a).Prf (exists f).` □

4.5.2 EXERCISE [★★★]: Formalise the translation from F^ω into CC. □

The combination of type dependency and impredicativity á la Sytem F yields an astonishing expressive power. For example, we can define Leibniz equality as follows:

```
eq = λa:Prop.λx:Prf(a).λy:Prf(a).
          all p:Prf a→Prop.all h:Prf(p x).p y
   : Πa:Prop.Prf a → Prf a → Prop
```

We can now prove reflexivity of equality by exhibiting an inhabitant of the type

```
Πa:Prop.Πx:Prf(a).Prf(eq a x x)
```

Indeed,

```
eqRefl = λa:Prop.λx:Prf(a).λp:Prf(a)→Prop.
                    λh:Prf(p x).h
```

is such a term.

4.5.3 EXERCISE [★★↛]: State and prove symmetry and transitivity of equality. □

In a similar way, we can define other logical primitives such as boolean connectives and quantifiers and then prove mathematical theorems. Occasionally we have to assume additional axioms. For example, induction for the natural numbers can be stated, but not proved; it is thus convenient to work under the assumption:

```
natInd : Πp:Prf(nat)→Prop.Prf(p zero)
              → (Πx:Prf(nat).Prf(p x) → Prf(p(succ x)))
               → Πx:Prf(nat).Prf(p x)
```

With that assumption in place we can for example prove associativity of addition in the form of a term of type:

Extends λLF *algorithm (4-3 and 4-4)*

Algorithmic kinding $\boxed{\Gamma \vdash_{\blacktriangleright} T :: K}$

$$\Gamma \vdash_{\blacktriangleright} \texttt{Prop} :: * \qquad \text{(KA-PROP)}$$

$$\frac{\Gamma \vdash_{\blacktriangleright} t : \texttt{Prop}}{\Gamma \vdash_{\blacktriangleright} \texttt{Prf}(t) :: *} \qquad \text{(KA-PRF)}$$

Algorithmic typing $\boxed{\Gamma \vdash_{\blacktriangleright} t : T}$

$$\frac{\Gamma \vdash_{\blacktriangleright} T :: * \qquad \Gamma, x{:}T \vdash_{\blacktriangleright} t : \texttt{Prop}}{\Gamma \vdash_{\blacktriangleright} \texttt{all}\ x{:}T.t : \texttt{Prop}} \qquad \text{(QT-ALL-E)}$$

Algorithmic type equivalence $\boxed{\Gamma \vdash_{\blacktriangleright} S \equiv T}$

$$\frac{t \longrightarrow_{\text{wh}} \texttt{all}\ x{:}T_1.t_2 \qquad \Gamma \vdash_{\blacktriangleright} S_1 \equiv T_1 \qquad \Gamma, x{:}S_1 \vdash_{\blacktriangleright} S_2 \equiv \texttt{Prf}(t_2)}{\Gamma \vdash_{\blacktriangleright} \Pi x{:}S_1.S_2 \equiv \texttt{Prf}(t)} \qquad \text{(QKA-PI-PRF)}$$

$$\frac{\Gamma \vdash_{\blacktriangleright} \Pi x{:}S_1.S_2 \equiv \texttt{Prf}(t)}{\Gamma \vdash_{\blacktriangleright} \texttt{Prf}(t) \equiv \Pi x{:}S_1.S_2} \qquad \text{(QKA-PRF-PI)}$$

$$\frac{\Gamma \vdash_{\blacktriangleright} s \equiv t}{\Gamma \vdash_{\blacktriangleright} \texttt{Prf}(s) \equiv \texttt{Prf}(t)} \qquad \text{(QKA-PRF)}$$

Algorithmic term equivalence $\boxed{\Gamma \vdash_{\blacktriangleright} t \equiv_{\text{wh}} t'}$

$$\frac{\Gamma \vdash_{\blacktriangleright} S_1 \equiv T_1 \qquad \Gamma, x{:}S_1 \vdash_{\blacktriangleright} s \equiv t}{\Gamma \vdash_{\blacktriangleright} \texttt{all}\ x{:}S.s \equiv_{\text{wh}} \texttt{all}\ x{:}T.t} \qquad \text{(QA-ALL-E)}$$

Figure 4-8: Algorithmic typing for CC

```
Πx:Prf nat.Πy:Prf nat.Πz:Prf nat.
                Prf(eq nat (add x (add y z)) (add (add x y) z))
```

4.5.4 EXERCISE [★★★]: Find such a term. □

The task of finding proof terms inhabiting types is greatly simplified by an interactive goal-directed theorem prover such as LEGO (Luo and Pollack, 1992; Pollack, 1994) or Coq (Barras, Boutin, Cornes, Courant, Filliatre, Gimenez, Herbelin, Huet, Munoz, Murthy, Parent, Paulin-Mohring, Saibi, and Werner, 1997), or a structure driven text editor for programming, such as Agda or Alfa (Coquand, 1998; Hallgren and Ranta, 2000).

Algorithmic typing and equality for CC

We will now consider algorithmic type checking for the pure CC. The beta reduction relation is extended with a clause for `all`:

$$\frac{t \longrightarrow_\beta t'}{\texttt{all}\ x{:}T.t \longrightarrow_\beta \texttt{all}\ x{:}T.t'} \qquad \text{(BETA-ALL)}$$

4.5.5 THEOREM: The relation $\longrightarrow_\beta$ is strongly normalising on well-typed terms of CC. □

Proof: The mapping to F^ω defined in exercise 4.5.2 preserves $\longrightarrow_\beta$, therefore an alleged infinite reduction sequence in CC would entail an infinite reduction sequence in F^ω. □

With this result in place it is now possible to establish soundness, completeness, and termination of algorithmic typing. The additional rules for the algorithm (extending those for λLF) are presented in Figure 4-8.

The Calculus of Inductive Constructions

The fact that induction cannot be proved is a flaw of the impredicative encoding of datatypes. Not only is it aesthetically unappealing to have to make assumptions on an encoding; more seriously, the assumption of `natInd` destroys the analogue of the progress theorem (see TAPL §8.3). For example, the following term does not reduce to a canonical form:

```
natInd (λx:Prf nat.nat) zero (λx:Prf nat.λy:Prf nat.zero) zero
```

For these reasons, Paulin-Mohring and subsequent authors (Mohring, 1986; Werner, 1994; Altenkirch, 1993) have combined CC with inductive definitions as originally proposed (for a predicative system) by Martin-Löf (Martin-Löf, 1984). In the thus obtained *Calculus of Inductive Constructions* (CIC) as implemented in the Coq Theorem Prover (Barras, Boutin, Cornes, Courant, Filliatre, Gimenez, Herbelin, Huet, Munoz, Murthy, Parent, Paulin-Mohring, Saibi, and Werner, 1997) we can declare the type `nat:Prop` as an inductive type with constructors `zero:Prf nat` and `succ:Prf nat→Prf nat`. This generates a constant:

```
natInd : Πp:Prf(nat)→Prop.Prf(p zero) →
           (Πx:Prf(nat).Prf(p x) → Prf(p(succ x))) →
            Πx:Prf(nat).Prf(p x)
```

which obeys the following equality rules:

```
natInd p hz hs zero ≡ hz
natInd p hz hs (succ n) ≡ hs n (natInd p hz hs n)
```

This clearly provides induction, but it also allows us to define primitive recursive functions such as addition by

```
add = λx:Prf nat.λy:Prf nat.natInd (λx:nat.nat)
        y (λy:nat.λr:nat.succ r) x
```

Notice that we have instantiated `natInd` with the constant "predicate" `λx:nat.nat`.

The mechanism of inductive definitions is not restricted to simple inductive types such as `nat`. CIC, as well as Martin-Löf's predicative systems (as

implemented in ALF (Magnusson and Nordström, 1994)) admit the inductive definition of type families as well. For example, with `nat` already in place we may define an inductive family

```
vector : Prf nat → Prop
```

with constructors `nil : Prf(vector zero)` and

```
cons : Πx:Prf nat. Prf nat →
        Prf(vector x) → Prf(vector(succ x))
```

The (automatically generated) induction principle then has the typing

```
vecInd : Πp:Πx:nat.Prf(vector x) → Prop.
   Prf(p zero nil) →
   (Πx:Prf nat.Πy:Prf (vector x).
        Πa:Prf nat.Prf (p y)→Prf (cons x a y)) →
   Πx:Prf nat.Πy:Prf(vector x).Prf(p x y)
```

4.5.6 EXERCISE [★★↛]: What are the equality rules for this induction principle by analogy with the equations for `natInd`? □

Let us see how we can define the exception-free `first` function from the introduction for these vectors. We first define an auxiliary function `first'` that works for arbitrary vectors by

```
first' = vecInd (λx:Prf nat.λv:Prf(vector x).nat)
  zero
  (λx:Prf nat.λy:Prf (vector x).
       λa:Prf nat.λprev:Prf nat.a) :
   Πx:Prf nat.Πv:Prf(vector x).Prf nat
```

This function obeys the equations:

```
first' zero nil = zero
first' (succ x) (cons x a y) = a
```

We obtain the desired function `first` by instantiation

```
first = λx:Prf(nat).λy:Prf(vector (succ x)).
   first' (succ x) y)
```

The default value `zero` can be omitted in a system like ALF which allows the definition of dependently-typed functions by pattern matching. In that system one would merely declare the type of `first` and write down the single pattern

```
first x (cons x a y) = a
```

ALF can then work out that this defines a total function. The extension of pattern matching to dependent types was introduced in (Coquand, 1992) which also contains beautiful examples of proofs (as opposed to programs) defined by pattern matching. McBride (McBride, 2000) has studied translations of such pattern matching into traditional definitions using recursion/induction principles like `vecInd`.

4.5.7 EXERCISE [★★★↛]: Define using `vecInd` a function

```
concat : Πx:Prf nat.Πy:Prf nat.Prf (vector x) →
                                    Prf (vector y) →
                                    Prf (vector (add x y))
```

How does it typecheck? □

As a matter of fact, the CIC goes beyond the type system sketched here in that it allows quantification over *kinds*, so, for example, the "predicate" `p` in `natInd` may be an arbitrary type family. This means that using the constant type family $\mathtt{p} = \lambda \mathtt{x} : \mathtt{nat}.\mathtt{Prop}$ we can define a function $\mathtt{eqZero} : \mathtt{Prf\ nat} \to \mathtt{Prop}$ which equals `true` when applied to `zero` and `false` on all other arguments. This facility turns out to be useful to define the exception-free `first` function on vectors which was presented in the introduction.

Another additional feature of the CIC is the separation of propositions and datatypes into two disjoint universes `Prop` and `Set`. Both behave like our `Prop`, the difference lies in a program extraction feature that maps developments in the CIC to programs in an extension of F^ω with inductive types and general recursion. Types and terms residing in `Prop` are deleted by this translation; only types and terms in `Set` are retained. In this way, it is possible to extract correct programs from formal correctness proofs. For details we refer to (Paulin-Mohring, 1989).

Sigma types in CC

It is unproblematic and useful to combine CC with Σ-types as described in Section 4.4 and Figures 4-5 and **??**. This allows one to form types of algebraic structures, for instance

```
Semigrp = Σa:Prop.Σop:Prf(a)→Prf(a)→Prf(a).
            Πx:Prf(a).Πy:Prf(a).Πz:Prf(a).
             Prf(eq a (op x (op y z)) (op (op x y) z));
```

This system is contained in Luo's *Extended Calculus of Constructions* (ECC) (Luo, 1994) which additionally permits Π and Σ quantification over kinds. For consistency reasons which we will briefly describe next this requires an infinite

hierarchy of ever higher kinds $*_0, *_1, *_2, \ldots$. For instance, in ECC one has

$$\Sigma X{:}*_3.\ X : *_4$$

ECC has been implemented in the LEGO system (Luo and Pollack, 1992).

It is quite another matter to ask for a reflection of Σ-types into the universe `Prop` of datatypes and propositions, by analogy with the way `all` is treated. The temptation is to introduce a term former `ex y:T.t : Prop` when $\texttt{x:T} \vdash \texttt{t:Prop}$, together with an equality rule asserting that

$$\texttt{Prf(ex y:T.t)} \equiv \Sigma\texttt{y:T.Prf(t)}$$

Coquand (Coquand, 1986) has shown that the resulting system is unsound in the sense that all types are inhabited and strong normalisation fails. Intuitively, the reason is that in this system we are able to define

```
prop = ex x:Prop.nat
```

and now have a mapping $\texttt{i:Prop}\to\texttt{Prf(prop)}$ defined by

$$\texttt{i} = \lambda\texttt{x:Prop.(x,zero:prop)}$$

as well as a left inverse $\texttt{j:Prf(prop)}\to\texttt{Prop}$ given by

$$\texttt{j} = \lambda\texttt{x:Prf(prop).x.1}$$

Thus, we have reflected the universe `Prop` into one of its members, which allows one to encode (after some considerable effort) one of the set-theoretic paradoxes which shows that there cannot be a set of all sets.

This must be contrasted with the impredicative existential quantifier `exists` defined on page 276. The difference between `exists` and the hypothetical term former `ex` is that `exists` does not allow one to project out the existential witness in case it is of type `Prop`.

An existential quantifier which does not provide first and second projections, but only the impredicative elimination rule known from System F is called a *weak* sum, *weak* Σ-type, or existential. In contrast, the Σ-types with projections are called *strong*.

We conclude this section by remarking that it is unproblematic to have "small" strong Σ-types in the CC, that is, if $\texttt{t}_1\texttt{:Prop}$ and $\texttt{x:Prf(t}_1\texttt{)} \vdash \texttt{t}_2\texttt{:Prop}$ then $\sigma\texttt{x:Prf(t}_1\texttt{).t}_2\texttt{:Prop}$ with the equivalence

$$\texttt{Prf}(\sigma\,\texttt{x:Prf(t}_1\texttt{).t}_2\texttt{)} \equiv \Sigma\texttt{x:Prf(t}_1\texttt{).Prf(t}_2\texttt{)}$$

4.5.8 EXERCISE [★★★↛]: An "approximation" for $\sigma\,\texttt{x:Prf(t}_1\texttt{).t}_2$ is given by

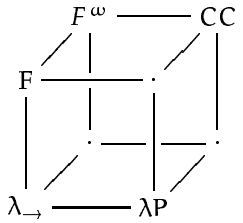
$$\texttt{exists} = \texttt{all c:Prop.all b:}\Pi\texttt{x:Prf t}_1\texttt{.Prf t}_2 \to \texttt{Prf c.c}$$

Define pairing and first projection for `exists`. Unfortunately, it is not possible to define a second projection. □

4.6 Relating abstractions: Pure Type Systems

The Calculus of Constructions is a very expressive system, but at first sight, somewhat difficult to understand because of the rich mix of different "levels" of typing (especially in its original formulation with `Prf` implicit). Given a lambda term $\lambda x : S.t$, we cannot tell without (possibly lengthy) further analysis of S and t whether this is a term-level function, a type abstraction, a type family, a type operator, or something else.

Partly as an attempt to explain the fine structure of CC, Barendregt introduced the *Lambda Cube* of typed calculi (briefly introduced in TAPL Chapter 30), illustrated below:



The cube relates previously known typed lambda calculi (recast within a uniform syntax) to CC, by visualising three "dimensions" of abstraction. In the bottom left corner, we have $\lambda_\rightarrow$ with ordinary term-term abstraction. Moving rightwards, we add the type-term abstraction characteristic of dependent types: λP is the Lambda Cube's version of our λLF. Moving upwards, we add the term-type abstraction of System F, capturing polymorphism. Finally, moving towards the back plane of the cube, we add the higher-order type-type abstraction characteristic of F^ω.

Pure Type Systems

The type systems of the Lambda Cube, and many others besides, can be described in the setting of *Pure Type Systems* (Terlouw, 1989; Berardi, 1988; Barendregt, 1991, 1992; Jutting, McKinna, and Pollack, 1994; McKinna and Pollack, 1993; Pollack, 1994). There is an simple and elegant central definition of Pure Type System (PTS) using just six typing rules, which captures a large family of systems constructed using Π-types. This uniform presentation allows one to establish basic properties for many systems at once, and also to consider mappings between type systems (so-called *PTS morphisms*).

A presentation of λLF as a Pure Type System is given in Figure 4-9.

The first thing to notice about PTSs is that there is a single syntactic category of terms, used to form types, terms, and abstractions and applications

λP

Syntax

t ::=		*terms:*
	s	*sort*
	x	*variable*
	λx:t.t	*abstraction*
	t t	*application*
	Πx:t.t	*dependent product type*
s ::=		*sorts:*
	∗	*sort of proper types*
	□	*sort of kinds*
Γ ::=		*contexts:*
	∅	*empty context*
	Γ, x:T	*variable binding*

Typing $\boxed{\Gamma \vdash t : T}$

$$\frac{}{\Gamma \vdash * : \Box} \quad \text{(T-STAR)}$$

$$\frac{x:T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash S : * \qquad \Gamma, x:S \vdash t : T}{\Gamma \vdash \lambda x:S.t : \Pi x:S.T} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : \Pi x:S.T \qquad \Gamma \vdash t_2 : S}{\Gamma \vdash t_1\, t_2 : [x \mapsto t_2]T} \quad \text{(T-APP)}$$

$$\frac{\Gamma \vdash S : s_i \qquad \Gamma, x:S \vdash T : s_j}{\Gamma \vdash \Pi x:S.T : s_j} \quad \text{(T-PI)}$$

$$\frac{\Gamma \vdash t : T \qquad T \equiv T' \qquad \Gamma \vdash T' : s}{\Gamma \vdash t : T'} \quad \text{(T-CONV)}$$

where $(s_i, s_j) \in \{(*, *), (*, \Box)\}$.

Figure 4-9: First-order dependent types, PTS-style (λP)

of diferent varieties. Although formally there is a single syntactic category, we use the same meta-variables as before, to aid intuition. (So the letters T and K and also range over the syntactic category of terms, but the system will determine that they are types and kinds, respectively).

To allow levels of types and kinds to be distinguished, the PTS framework uses tokens called *sorts* to classify different categories of term, within the formal system itself. The system λP requires two sorts: first, ∗, which is the kind of all proper types, as used before, and second, □, which is the sort that classifies well-formed kinds. Judgments of the form $\Gamma \vdash T : *$ replace $\Gamma \vdash T :: *$ from Figure 4-1, and judgements $\Gamma \vdash K : \Box$ replace $\Gamma \vdash K$.

The rule T-PI controls formation of Π-types, by restricting which sorts we are allowed to quantify over. In turn this restricts which λ-abstractions can be introduced by T-ABS. For λLF, there are two instances of λ-abstraction and two instances of Π-formation. In the PTS presentation, these are captured by the two pairs of sorts allowed in T-PI. When $s_i = s_j = *$, we have the first-order dependent product type, and when $s_j = \Box$ we have the kind of type families, corresponding respectively to K-PI and WF-PI in Figure 4-1.

The conversion rule is the main point of departure. The equivalence re-

lation $s \equiv t$ in Pure Type Systems is defined between untyped terms, as the compatible closure of β-reduction. This has a strong effect on the meta-theory.

4.6.1 EXERCISE [★★★★]: Using the obvious mapping from the syntax of λLF into the syntax of λP, give a proposition stating a connection between the two presentations. Try to prove your proposition. □

Systems of the Lambda-Cube and beyond

The other systems of the Lambda Cube can be expressed using the same rules as in Figure 4-9, with the single difference of changing the combinations of pairs of sorts (s_i, s_j) allowed in T-PI. This controls which kind of abstractions we can put into the context. The table below characterises the systems of the Lambda Cube:

System	PTS formation rules
$\lambda_{\to}$	$\{(*,*)$ $\}$
λP	$\{(*,*), (*,\Box)$ $\}$
λ2	$\{(*,*), (\Box,*)$ $\}$
F^{ω}	$\{(*,*), (\Box,*), (\Box,\Box)$ $\}$
CC	$\{(*,*), (*,\Box), (\Box,*), (\Box,\Box)$ $\}$

Further PTSs are given by adjusting the axiom T-STAR of Figure 4-9, which is another parameter in the formal definition of PTS. For example, if we take the axiom to be

$$\Gamma \vdash * : * \qquad \text{(T-TYPETYPE)}$$

(together with the T-PI restriction of $\{(*,*)\}$), we obtain a system where $*$ is the sort of all types including itself. In this system, all types are inhabited and there are non-normalizing terms (similarly to the result of Coquand (Coquand, 1986) mentioned on page 281). Although this renders the logical interpretation of the system meaningless, it is debatable whether systems like this may nonetheless be useful in some situations as type systems for programming languages.

For further details of Pure Type Systems, we refer the reader to the references given at the end of the chapter.

4.7 Implementation

In this closing section we describe an OCaml implementation of the dependent type theory described in preceding sections. The implementation allows

declarations and definitions of both terms and types. Type-checking occurs as soon as a declaration or definition is given. A term may be given with a type, which will be checked, or without, in which case one will be inferred. Similarly for kinds. Finally, we can ask to normalise well-typed terms.

The type checking algorithm proceeds by evaluating the rules in Figures 4-4 and 4-3 and the later tables extending these judgements. More precisely, we have (simultaneously defined) functions:

```
val whnf : term → term
val typeof : context → term → ty
val kindof : context → ty → kind
val checkkind :  context → kind → unit
val tyeqv : context → ty → ty → bool
val kindeqv : context → ty → ty → bool
val tmeqv : context → ty → ty → bool
```

These functions are implemented by encoding the algorithmic rules using pattern matching. For example, the definition of `tmeqv` begins like so:

```
tmeqv ctx tm1 tm2 =
  let tm1' = whred true ctx tm1 in
  let tm2' = whred true ctx tm2 in
  match (tm1',tm2') with
    (TmVar(fi,i,j), TmVar(fi',i',j')) → i=i'
  | (TmAbs(_,x,tyS1,tmS2),TmAbs(_,y,tyT1,tmT2))→
      let ctx' = addbinding ctx x (VarBind(tyS1)) in
      tmeqv ctx'  tmS2 tmT2
...
```

(the first argument of `whred` is a flag indicating whether to allow definitions in the context to be expanded).

We stress that the implementation is a direct rendition of the syntax and rules described earlier. It does not include any of the numerous desirable features that make programming with dependent types more convenient, such as argument synthesis (Harper and Pollack, 1991) or interactive, goal-directed construction of terms. Conversely, because the implementation is simple, it should be straightforward to experiment with extensions. The program is built on the F^ω implementation from TAPL and uses the same design and data structures (see TAPL Chapters 6,7,30).

We illustrate the use of the implementation by way of some examples. Notice that the ASCII input to the system to produce a type like Π`x:A.B` is `Pi x:A.B`.

Examples

With the commands

```
A : *;
Nat : *;
zero : Nat;
succ : Πn:Nat.Nat;
Vector : Πn:Nat.*;
```

we declare variables `A`, `Nat`, constants `zero` and `succ` intended to denote zero and successor on the natural numbers, and a type `Vector` depending on type `Nat`. Note that the implementation does not support $\rightarrow$; we must use Π-types throughout.

Next, we declare functions to form vectors by

```
nil : Vector zero
cons : Πn:Nat. Πx:A. Πv:Vector n. Vector (succ n)
```

allowing us to define a function for forming vectors of length three:

```
one = succ zero;two = succ one;
mkthree = λx:A.λy:A.λz:A.
        cons two z (cons one y (cons zero x nil));
```

The implementation will respond by inferring the type of `mkthree`:

```
mkthree : Πx:A. Πy:A. Πz:A. Vector (succ two)
```

We can now partially apply `mkthree` to two elements of type `A` by

```
a:A; b:A;
mkthree a b;
```

resulting in the response

```
λz:A.
 cons (succ (succ zero)) z
 (cons (succ zero) b (cons zero a nil)) :
      Πz:A.Vector(succ (succ (succ zero)))
```

This response exhibits two weaknesses of the implementation. First, definitions are always expanded in results; this will in practice almost always lead to unreadable outpt. Second, the first arguments to `cons` must be given explicitly and are printed out while they could be inferred from the types of the second arguments. Practical implementations of dependent types overcome both these problems. For instance, in LEGO (Luo and Pollack, 1992), `mkthree` would be defined (in our notation) as

```
mkthree = λx:A.λy:A.λz:A. cons z (cons y (cons x nil));
```

and the response to `mkthree a b` would be

```
λz:A.cons z (cons  b (cons a nil))) :  Πz:A.Vector three
```

For LEGO to know that the first argument to `cons` is implicit we must declare `cons` by

```
Πn|Nat. Πx:A. Πv:Vector n. Vector (succ n)
```

where the bar indicates implicitness for argument synthesis.

Returning to our experimental checker, let us illustrate Σ-types. We declare three types

```
A:*; B:Πx:A.*; C:Πx:A.Πy:B x.*;
```

and define

```
S = Σx:A.Σy:B x.C x y;
```

Supposing

```
a:A; b:B a; c: C a b;
```

then we can form

```
(a,(b,c:Σy:B a.C a y):S);
```

which is an element of `S`. The first type annotation is actually redundant and the implementation allows one to abbreviate the above by

```
(a,b,c:S)
```

If we declare

```
Q : Πx:S.*;
x:S; y:Q x;
```

Then the following typecast succeeds

```
y:Q (x.1,x.2.1,x.2.2:S);
```

thus illustrating the built-in surjective pairing.

Here, finally, is the definition of natural numbers in CC:

```
nat = all a:Prop.all z:Prf(a).all s:Πx:Prf(a).Prf(a). a;
```

Note that `Prf` always requires parentheses in the implementation.

We also remark that by default the implementation prints the weak-head normal form of input terms. The β-normal form of a term `t` is printed with the command `(Normal t);`.

4.8 Further reading

Dependent type theories have been widely investigated, much of the development building on the pioneering work of Per-Martin Löf. There is no space here for a comprehensive survey; instead we focus on a strands related to the develpment in this chapter, and the potentially rich future of programming with dependent types.

Dependent type theories

The Edinburgh Logical Framework and its type system are described in (Harper, Honsell, and Plotkin, 1993a). Our definition of λLF has the same type structure, but omits signatures, and includes declarative equality judgements rather than an untyped equivalence relation. A more complete recent development which also includes equality judgements is in (Harper, 2002).

Richer type theories than LF are considered in many places. The calculus of constructions was introduced in (Coquand and Huet, 1988) and further developed in related type theories (Mohring, 1986; Luo, 1994; Pollack, 1994). Algorithms for type checking with dependent types were first considered by Coquand (Coquand, 1991a), Cardelli (Cardelli, 1986, 1988b), and also within the closely related AUTOMATH system of de Brujijn (de Bruijn, 1980).

The best survey of Pure Type Systems remains Barendregt's handbook article (Barendregt, 1992), which includes a description of the λ-Cube. Although the definition of PTS is elegant and short, developing the meta-theory for PTSs has been surprisingly challenging. Several important results and improvements have been made since Barendregt's article. For example, Pollack (1994), studied formalization of the meta-theory of PTSs in type theory itself, Poll (1998) established the *expansion postponement* property for the class of normalizing PTSs, and Zwanenburg (1999), studied the addition of subtyping to PTSs.

Type theories which combine inductive types and quantification over kinds, such as CIC, do not permit an easy normalization proof by translation into a simply-typed normalising sytem as was the case for the pure CC. Therefore, strong normalisation must be proved from scratch for those systems. So far only partial proofs have been published for CIC as implemented in Coq. For UTT as implemented in LEGO, a strong normalization proof is given in (Goguen, 1994), which introduces the idea of a *typed-operational semantics* as a more controlled way of managing reduction in a typed setting.

There has been much study into semantic models for dependent type theories. See Hofmann (Hofmann, 1997b) for a particular approach and some comparison with the literature. Notable contributions include (Cartmell, 1986;

Erhard, 1988; Streicher, 1991; Jacobs, 1999).

Programming with dependent types

The task of building practical programming languages with dependent types is still a topic of research. Early languages include Pebble (Lampson and Burstall, 1988) and Cardelli's Quest (Cardelli and Longo, 1991). Programming in Martin-Löf's type theory is described in (Smith, Nordström, and Petersson, 1990). More recently, Augustsson developed the language Cayenne (Augustsson, 1998) which advertised programming with dependent types to a new audience. Cayenne has a syntax based on the functional programming language Haskell.

Recent work by several researchers, including (McBride, 2000; McBride and McKinna, 2002) (among others), seeks to make technological advances at the language level. The aim is to provide more comfortable high-level notations and new programming language abstractions for applying type theory. Underlying type theories such as CIC are amply expressive for this purpose; the current challenge lies in making these systems more convenient to use, by adding programming language constructions, notational conveniences and advanced inference techniques. Present implementations, oriented towards mathematical interactive proof development, need to be adapted to programming language setttings. We are confident that these exciting developments lead much to be expected for the future of programming with dependent types.

FIXME: more to add here (e.g., DML)

5 *Effect* Types and Region-based Memory Management

By Fritz Henglein, Henning Makholm, and Henning Niss

5.1 Type-based program analysis

Type-based program analysis is program analysis based on the concepts, theories and technologies developed for type systems employed in the definition of programming languages. It is a vast field of research with numerous applications and considerable practical impact. Applications include strictness analysis, data representation analysis, binding-time analysis, soft typing (also called dynamic typing inference), boxing analysis, pointer aliasing, value flow analysis (and all its applications), region-based memory management, communication topology analysis, Year 2000 type analysis, cryptographic protocol verification, locking, race detection, to name a few.

This chapter presents *type and effect systems*, or *effect type systems* and *region-based memory management*. Classical type systems express properties of values, not the computations leading to those values and thus do capture any side effects. Effect types capture properties of a computation, not just its result, in particular important effects. Region-based memory management, originally termed *region inference*.

Overview of chapter

Region-based memory management is a mature programming language technology involving a multitude of concepts and techniques. We approach it in a number of steps. First, in §5.2 we present a type-based formulation of value flow analysis by formulating it as a translation into a language with explicit labels that are interpreted operationally as tags for values. Value flow analysis is at the conceptual and technical core of many other analyses. This is also the case for region-based memory management: As we shall see, labels

can conceptually and technically be interpreted as memory regions. In §5.3 we introduce classical effect type systems. Effects are necessary to ensure soundness of lexically scoped region allocation and deallocation. In §5.4 we introduce the basics of region-based memory management. §5.5 presents, in essence, the Tofte-Talpin effect type system for a functional language with explicit region annotations, and §5.6 describes the automatic inference of region annotations. Finally, §5.7 presents extensions and §5.8 survey systems with statically checked region-based memory management.

5.2 Value flow analysis

Value flow analysis deals with the problem of figuring out where a value constructed at a particular program point may be used, usually in the same program. Here values can be any sort of data: atomic data such as integers, structured data such as records, or higher-order data such as function closures. Even though Reynolds (1969b) was first to look at the problem of computing flow for structured data and called it *data set computation*, we follow Schwartz (1975) in using the term *value flow* to emphasize its general applicabilty to primitive, structured and higher-order data. Classical data flow analysis corresponds to value flow analysis for primitive data (only); *closure analysis*, the term introduced by Sestoft, and *control flow analysis*, the term used by Shivers, focus on the flow of function values (function closures). Note that data and control flow are interdependent for higher-order languages; see Mossin (Mossin, 1997, Section 1.4) for a discussion of this.

A simple functional language: PCF_2

To provide a concrete setting for value flow analysis, consider the language PCF_2, *Finitary PCF* Jung and Stoughton (1993); Loader (2001), which is the simply typed lambda-calculus with general recursion (fix) and Boolean values. Its syntax and call-by-value small-step evaluation rules are given in Figure 5-1.

To define the observable result of evaluating an entire program — which we formally take to be any (closed) term — define:

5.2.1 DEFINITION: A term $\mathtt{t}$ is *final* if there is no term $\mathtt{t}'$ such that $\mathtt{t} \longrightarrow \mathtt{t}'$. □

5.2.2 DEFINITION: Define the relation $\rightarrow^{*!}$ between terms by: $\mathtt{t} \rightarrow^{*!} \mathtt{t}'$ iff $\mathtt{t} \longrightarrow^* \mathtt{t}'$ and $\mathtt{t}'$ is final. □

5.2.3 DEFINITION: Define the evaluation function $\mathrm{eval}(\cdot)$ from terms to the set $\{\mathtt{tt}, \mathtt{ff}, \bot, \mathtt{wrong}, \mathtt{fun}\}$ by

Terms

t ::=		*terms:*
	v	*value expression*
	x	*variable*
	t t	*application*
	if t then t else t	*conditional*
	fix x.t	*recursion*
v ::=		*value expressions:*
	λx.t	*abstraction*
	bv	*truth value*
bv ::=		*truth values:*
	tt	*true*
	ff	*false*

Evaluation rules $\boxed{t \xrightarrow{PCF_2} t'}$

$$(\lambda x.t_{12})\, v_2 \xrightarrow{PCF_2} [x \mapsto v_2]t_{12} \quad \text{(E-BETA)}$$

$$\texttt{fix}\ x.t \xrightarrow{PCF_2} [x \mapsto \texttt{fix}\ x.t]t \quad \text{(E-FIXBETA)}$$

$$\texttt{if tt then } t_2 \texttt{ else } t_3 \xrightarrow{PCF_2} t_2 \quad \text{(E-IFTRUE)}$$

$$\texttt{if ff then } t_2 \texttt{ else } t_3 \xrightarrow{PCF_2} t_3 \quad \text{(E-IFFALSE)}$$

$$\frac{t_1 \xrightarrow{PCF_2} t_1'}{t_1\ t_2 \xrightarrow{PCF_2} t_1'\ t_2} \quad \text{(E-APP1)}$$

$$\frac{t_2 \xrightarrow{PCF_2} t_2'}{v_1\ t_2 \xrightarrow{PCF_2} v_1\ t_2'} \quad \text{(E-APP2)}$$

$$\frac{t_1 \xrightarrow{PCF_2} t_1'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \xrightarrow{PCF_2} \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3} \quad \text{(E-IF)}$$

Types

T ::=		*types:*
	bool	*boolean type*
	T → T	*function type*

Typing rules $\boxed{\Gamma \vdash t : T}$

$$\frac{x \notin \Gamma'}{\Gamma, x : T, \Gamma' \vdash x : T} \quad \text{(T-VAR)}$$

$$\Gamma \vdash bv : \texttt{bool} \quad \text{(T-BOOL)}$$

$$\frac{\Gamma \vdash t_1 : \texttt{bool} \qquad \Gamma \vdash t_2 : T \qquad \Gamma \vdash t_3 : T}{\Gamma \vdash \texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : T} \quad \text{(T-IF)}$$

$$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x.t : T_1 \to T_2} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_0 : T_1 \to T_2 \qquad \Gamma \vdash t_1 : T_1}{\Gamma \vdash t_0\ t_1 : T_2} \quad \text{(T-APP)}$$

$$\frac{\Gamma, x : T \vdash t : T}{\Gamma \vdash \texttt{fix } x.t : T} \quad \text{(T-FIX)}$$

Derived form

$$\texttt{let } x = t_1 \texttt{ in } t_2 \stackrel{\text{def}}{=} (\lambda x.t_2)\, t_1$$

Figure 5-1: Base language PCF_2.

a) $\text{eval}(t_0) = \texttt{bv}$ iff $t_0 \to^{*!} \texttt{bv}$.

b) $\text{eval}(t_0) = \bot$ iff there is an infinite sequence $t_1, t_1, ..., t_i, ...$ such that $t_i \longrightarrow t_{i+1}$ for $0 \leq i$. "$\text{eval}(t) = \bot$" is pronounced "t diverges".

c) $\text{eval}(t_0) = \texttt{wrong}$ iff $t_0 \to^{*!} t$ where t is not a value.

d) $\text{eval}(t_0) = \texttt{fun}$ if $t_0 \to^{*!} \lambda x.t$. □

We say $\mathtt{t}$ *gets stuck* or *goes wrong* if $\mathrm{eval}(\mathtt{t}) = \mathtt{wrong}$.

Observe that the evaluation semantics is deterministic, so $\mathrm{eval}(\cdot)$ is well-defined (and total).

This definition does not allow us to distinguish between different functions as the final result of a program. To distinguish functions one needs to apply the function within the program itself. This convention will be convenient, but it does not lose any real expressivity – any program that computes something else can be wrapped in auxiliary code to do whatever observation we like and report the result coded as a Boolean.

Informal description

Intuitively, value flow analysis considers a value *produced* (generated, constructed) at some program point, traces its "flow" through the program, and figures out all the places where it may be *consumed* (observed, used, deconstructed).

Consider, e.g., the PCF_2-program $\mathtt{t}_0$:

```
let fst = λx.λy.x in
    (let x = λp.p tt ff in λy.λp.p (x fst) y)
    tt
```

Value flow analysis should tell us that `x` may be applied to `fst` (which is rather easy to see), `fst` may be applied to `tt` (which is not immediately obvious from the source code), and the λ-abstraction `λy.λp.p (x fst) y` may be applied to `tt`, but `λp.p (x fst) y` is not applied anywhere.

We insist on the analysis being *sound (conservative)*, but it need not be *complete*: whenever a value actually flows from a program point to another, then the analysis will predict this (soundness), but the analysis will also overshoot and predict spurious flows (incompleteness) in some cases. For Turing-complete programming languages any reasonable formulation of semantically exact value flow analysis as a decision problem is undecidable by Rice's Theorem. A consequence of this is that, as long as we are interested in precise problem formulations, there is not a single problem formulation for conservative value flow analysis, but a whole range of problem specifications, from the utterly conservative and blazingly efficiently implementable and useless "every producer may reach every consumer", to the exact and arbitrarily slow "this value may reach that consumer if and only if it actually reaches that consumer". It is important to remember that statements such as "value-flow analysis can be solved in cubic time" only make sense for a *particular* formulation of value flow anlysis.

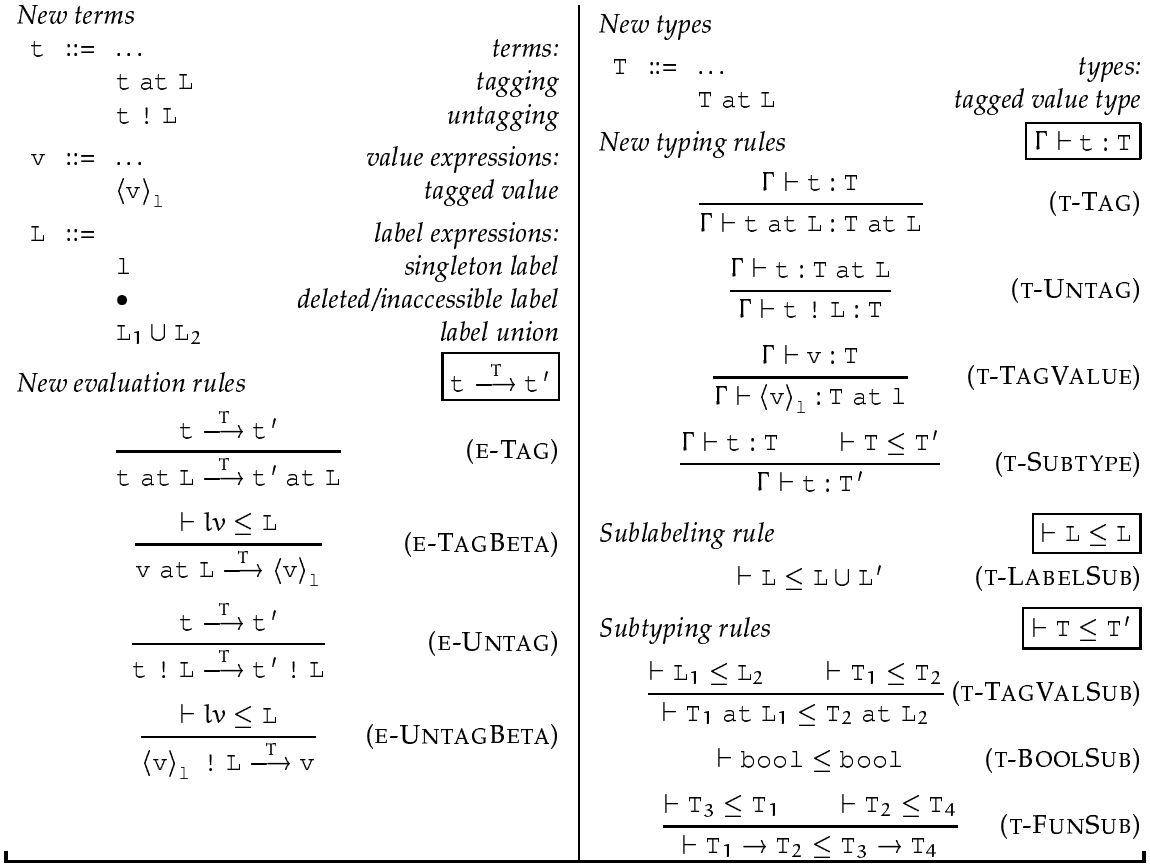


Figure 5-2: Tagged base language, TL.

Label tagging

In this subsection we introduce TL, which is basically PCF_2 with explicit label operations.

The language TL is defined by the definitions of Figure 5-1, extended with the definitions of Figure 5-2. Here the operator $\cup$ is assumed to be a set operator: it is associative, commutative and idempotent. As we have no representation of the empty set there is no neutral element for it in the language of label expresssions.

Operationally, evaluation of `t at L` consists of *tagging* the value of `t` with any of the labels in `L`. We write the result of tagging value `v` with `l` as $\langle v \rangle_l$.

The term `t ! L` is an *untagging* operation: `t` must evaluate to a tagged value $\langle v \rangle_l$, and if `L` contains `l` the underlying value `v` is returned. If `L` does not contain `l`, the evaluation gets stuck, which signals an error situation. As we shall see the type system guarantees that evaluation never can go wrong in this way.

The tagging and untagging operations serve as an operational model of value flow analysis. Consider a subexpression `t ! L` in a TL-program. The type system guarantees that only values with one of the labels `l` in `L` can reach this point. Those values can only be passed into the program from the environment or produced by the program itself. The former are identified by occurrences of `l` in "input position" of types (negative position in the judgments), the latter by finding all subexpressions `t at l` or `t at (l ∪ L)` in the TL-program.

A TL-term is to be thought of as an *instrumented* version of the underlying PCF_2-term. Syntactically, the underlying PCF_2-term of a TL-term is the term that arises by erasing anything that has to do with labels in the TL-term.

5.2.4 DEFINITION [ERASURE, COMPLETION]: Let `t` ∈ TL. Then $\|t\|$ is the PCF_2-term that arises by erasing all occurrences of `at L` and `! L` from `t` as well as repeatedly replacing any occurrence of $\langle v \rangle_l$ with `v`. We call $\|t\|$ the *erasure* of `t` and, conversely, any TL-term `t'` with $\|t'\| = \|t\|$ a *(label-tagged) completion* of $\|t\|$. □

A *fully tagged completion* is a completion where each value expression occurring in it is tagged (`bv at L` and `(λx.t) at L`) and untagging takes place in each destructive context (`if t ! L then t' else t''` and `(t ! L) t'`. Labels must not occur anywhere else. This ensures that each value gets tagged when it is created and every such tag is checked and removed immediately before the underlying untagged value is needed. This is the same as in the canonical implementation model for dynamically typed languages, where all values are created and stored with their (type) tags.

Intuitively, value flow analysis is the problem of finding a good — or best in a well-defined sense, if possible — fully tagged completion of a source program with label operations resulting in a TL program. The completion found in this process is the result of our value flow analysis. Correctness of completions and inference of good completions is covered below.

Conditional correctness and type soundness

Static correctness That erasure is a well-behaved notion with respect to static semantics is captured by the following property, which says that the TL-typing rules are conservative over the underlying PCF_2-typing rules:

5.2.5 PROPOSITION [CONSERVATIVITY OF TL-TYPING OVER PCF_2]:

1. If $\vdash \mathtt{T}_1 \leq \mathtt{T}_2$ then $\|\mathtt{T}_1\| = \|\mathtt{T}_2\|$.
2. If $\Gamma \vdash \mathtt{t} : \mathtt{T}$ then $\|\Gamma\| \vdash \|\mathtt{t}\| : \|\mathtt{T}\|$. □

The converse direction is trivial: since every PCF_2-term is also a TL-term, we are guaranteed that each PCF_2-term has at least one (well-typed) TL-completion.

Conditional dynamic correctness

5.2.6 DEFINITION [VALUE, STUCK STATE]: A *value* is a closed value expression. We say closed TL-term $\mathtt{t}$ is *stuck* if it is neither a value nor can be reduced to some $\mathtt{t}'$; that is, there is no $\mathtt{t}'$ such that $\mathtt{t} \xrightarrow{\mathrm{T}} \mathtt{t}'$. □

There are three possible mutually exclusive outcomes for the evaluation of a closed TL-term: either it terminates and results in a value, it gets stuck (terminates without reaching a value), or it does not terminate. The following theorem expresses that any completion of a closed PCF_2-term either "goes wrong" by reaching a stuck state or behaves exactly as the underlying PCF_2-term. This holds completely independently of any typing rules: neither PCF_2- nor TL-typing rules are needed.

5.2.7 THEOREM [CONDITIONAL CORRECTNESS]: Let $\mathtt{t}$ be a closed, not necessarily well-typed TL-term.

1. If $\mathtt{t} \xrightarrow{\mathrm{T}} \mathtt{v}$ then $\|\mathtt{t}\| \xrightarrow{\mathrm{PCF}_2} \|\mathtt{v}\|$.
2. If $\mathtt{t}$ does not terminate then $\|\mathtt{t}\|$ does not terminate.
3. If $\|\mathtt{t}\|$ gets stuck then $\mathtt{t}$ gets stuck. □

This guarantees only conditional correctness: It is possible for a TL-term $\mathtt{t}$ to get stuck where its erasure can continue computing. This is the case if evaluation of $\mathtt{t}$ reaches a state $\langle \mathtt{v} \rangle_{\mathtt{l}}\ !\ \mathtt{L}$ where $\mathtt{l} \notin \mathtt{L}$, which gets it stuck. The erasure, however, yields the value $\|\mathtt{v}\|$.

Soundness If we restrict ourselves to a subset of terms that do not get stuck then conditional correctness above indeed gives full correctness for those terms: $\mathtt{t} \xrightarrow{\mathrm{PCF}_2} \mathtt{v}$ if and only if $\|\mathtt{t}\| \xrightarrow{\mathrm{PCF}_2} \|\mathtt{v}\|$. This highlights the role that the type system and its soundness result play: define a subset of terms that do

not get stuck.[1] If this were all that we are interested in, the use of type systems is but one of many possible methods of defining such a subset. Any other method of doing so would be equally adequate.

5.2.8 THEOREM [SUBJECT REDUCTION]: If $\vdash \mathtt{t} : \mathtt{T}$ and $\mathtt{t} \xrightarrow{\mathrm{T}} \mathtt{t}'$ then $\vdash \mathtt{t}' : \mathtt{T}$. □

5.2.9 THEOREM [PROGRESS]: If $\vdash \mathtt{t} : \mathtt{T}$ then either $\mathtt{t} = \mathtt{v}$ for some value $\mathtt{v}$ or there exists $\mathtt{t}'$ such that $\mathtt{t} \xrightarrow{\mathrm{T}} \mathtt{t}'$. □

5.2.10 COROLLARY [SOUNDNESS (NO STUCK STATES)]: If $\vdash \mathtt{t} : \mathtt{T}$ then evaluation of $\mathtt{t}$ does not get stuck. □

No proofs of correctness — conditional correctness plus type soundness — are given here. They are completely analogous to the corresponding results for RAL in §5.5.

Inference

Note that a PCF_2-term $\mathtt{t}$ may have many different fully tagged completions, but they all can be written as substitution instances of a derivation template, in which

$\mathtt{t}[\vec{\mathtt{L}}]$; that is, the vector $\vec{\mathtt{L}}$ of label expressions characterizes the particular fully tagged completion. We shall now characterize the set of all vectors $\vec{\mathtt{L}}$ that give rise to a well-typed TL-term.

5.2.11 LEMMA: The type system in Figure 5-3 characterizes TL; that is, $\Gamma \vdash \mathtt{t} : \mathtt{T}$ according to Figure 5-3 if and only if $\Gamma \vdash \mathtt{t} : \mathtt{T}$ is a derivable TL-judgement. □

The typing rules of Figure 5-3 are derived from the TL-typing rules by:

1. eliminating the subtyping rule by systematically applying it to the conclusion of each syntax-oriented rule;
2. linearizing the inference rules by capturing all relations between types and labels by subtyping constraints.

This is the basis for constraint-based inference: Given an PCF_2-term $\mathtt{t}$, we can start with an *inference template*, a derivation tree where where all types and labels are replaced by type variables and label variables that each occur only once. We then derive a set of subtyping constraints $C(\mathtt{t})$. This can be done in linear time in the (untyped) size of $\mathtt{t}$. From this we can compute a principal solution, which can be turned into a principal typing. See Mossin (1997).

1. We need to choose a subset since, for Turing-complete languages, the set of terms that do not get stuck is not even recursively enumerable.

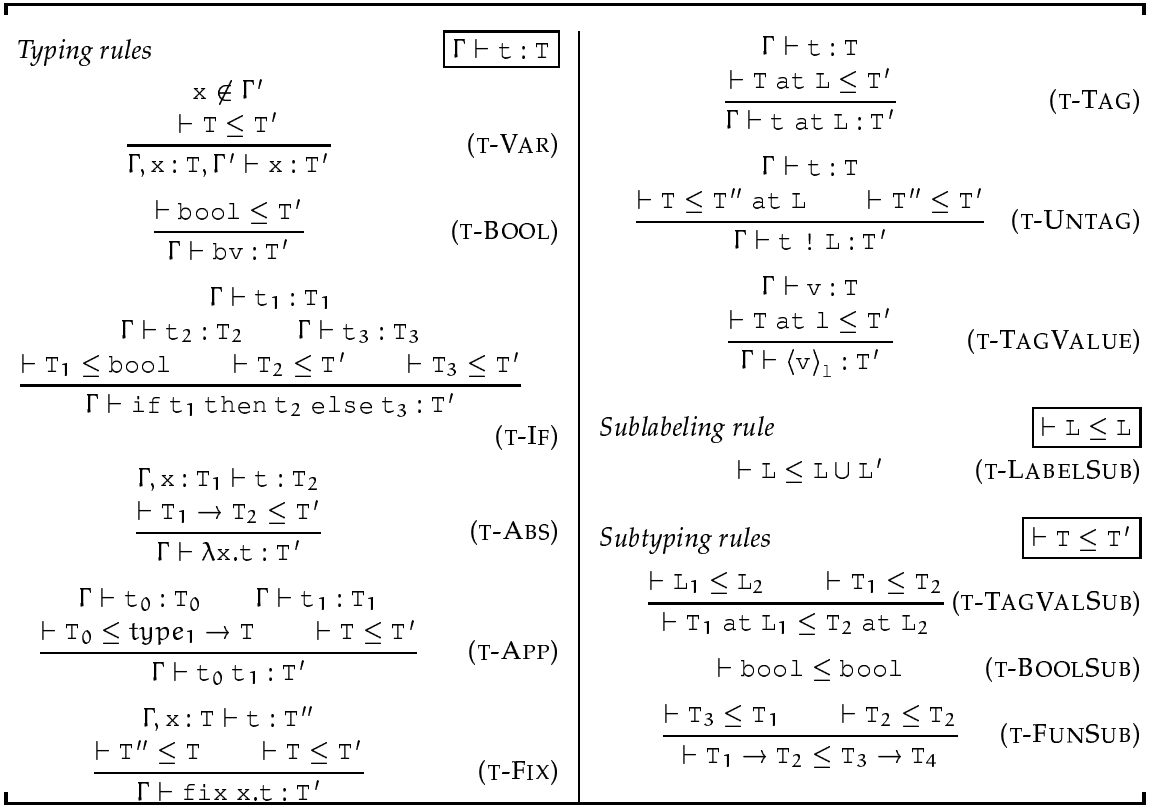


Figure 5-3: Syntax-oriented typing rules for TL.

Delimiting the scope of labels: Labels as memory regions

Consider a judgement $\Gamma \vdash t : T$. Intuitively, if label l occurs neither in Γ nor in T then evaluation of t does not refer to an l-tagged value nor does it return an l-tagged value to its context. In other words, label l does not need to exist before we start evaluation of t does it need to exist after. In particular, we can generate a new label before evaluating, which we write $\mathtt{new}\ l.t$.

Interpreting labels as memory regions We can exploit this basic idea as follows: Think of labels as variables bound to handles to memory regions at run-time. The tagging operation $t\ \mathtt{at}\ L$ is implemented as storing the value of t in any of the regions bound to a label in L. Its result is the pointer to where the value is stored. The untagging operation $t\ !\ L$ is implemented

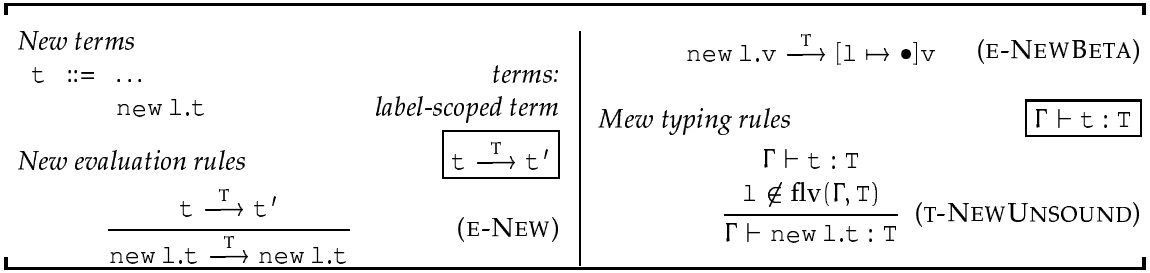


Figure 5-4: Scoped Tagged Language, STL.

as dereferencing the memory reference that t evaluates to (which is guaranteed to point into one of the regions in L); and $\langle v \rangle_l$ denotes any reference into region l whose dereferenced value is v. Finally, new l.t is implemented as follows: allocate a new memory region, bind its handle to l, evaluate t and finally, deallocate the memory region. This results in a stack-oriented memory management discipline for regions: the most recently allocated region is deallocated first.

Single, fixed sized chunks of memory will generally not do as regions because regions may grow unboundedly at run-time. So in practice, a region consists of a list of fixed sized pages taken from and returned to a free list of such pages. Other data structures for implementing unbounded regions are thinkable; e.g., "breathing" contiguous chunks of memory where the data stored in a chunk are copied to another chunk k times the size as the original one, with $k > 1$, once the original chunk has no more free space.

See §5.4 where we delve into region-based memory management in detail.

Unsoundness of naive typing for scoped regions Consider the type system given in Figure 5-4 for TL extended with new l.t. At first sight, it might seem curious that we substitute • for l in Rule (E-NEWBETA), given our intuition that l does not occur in the value v. The reason for this is, that our intuition is wrong: it is possible for l to occur in a value without it occurring in the type of the value. This is why we force the elimination of l by substitution • for all occurrences of l (the reason we have • in the first place). This reflects our intent that after evaluation of new l.t the region is deallocated and thus all values stored in it are inaccessible. The inaccessibility is captured in the evaluation rule for $\langle v \rangle_l$: l must be a label for v to be accessible — • does not qualify.

To see that this type system is unsound, consider the following fully tagged

STL-program $\mathtt{t_f}$ of type $\mathtt{bool\ at\ l_1} \to \mathtt{bool\ at\ l_1}$:

$$\mathtt{new\ l_0.let\ x = tt\ at\ l_0\ in\ \lambda y.if\ x\ !\ l_0\ then\ y\ else\ ff\ at\ l_1}$$

It reduces as follows:

$$\begin{array}{ll}
\mathtt{new\ l_0.let\ x = tt\ at\ l_0\ in\ \lambda y.if\ x\ !\ l_0\ then\ y\ else\ ff\ at\ l_1} & \longrightarrow \\
\mathtt{new\ l_0.let\ x = \langle tt\rangle_{l_0}\ in\ \lambda y.if\ x\ !\ l_0\ then\ y\ else\ ff\ at\ l_1} & \longrightarrow \\
\mathtt{new\ l_0.\lambda y.if\ \langle tt\rangle_{l_0}\ !\ l_0\ then\ y\ else\ ff\ at\ l_1} & \longrightarrow \\
\mathtt{\lambda y.if\ \langle tt\rangle_{\bullet}\ !\ \bullet\ then\ y\ else\ ff\ at\ l_1} &
\end{array}$$

Note that $\mathtt{l_0}$ occurs freely in $\mathtt{\lambda y.if\ \langle tt\rangle_{l_0}\ !\ l_0\ then\ y\ else\ ff\ at\ l_1}$ before performing the last reduction step even though the type of this value, $\mathtt{bool\ at\ l_1} \to \mathtt{bool\ at\ l_1}$, does not mention $\mathtt{l_0}$. It is easy to see now, how this can yield a stuck state. The program $\mathtt{t_f\ (tt\ at\ l_1)}$ is a well-typed STL-program of type $\mathtt{bool\ at\ l_1}$, yet evaluation gets stuck:

$$\begin{array}{ll}
\mathtt{t_f\ (tt\ at\ l_1)} & \stackrel{*}{\longrightarrow} \\
\mathtt{(\lambda y.if\ \langle tt\rangle_{\bullet}\ !\ \bullet\ then\ y\ else\ ff\ at\ l_1)\ (tt\ at\ l_1)} & \longrightarrow \\
\mathtt{if\ \langle tt\rangle_{\bullet}\ !\ \bullet\ then\ ff\ at\ l_1\ else\ ff\ at\ l_1} &
\end{array}$$

To continue evaluation would require reduction of $\mathtt{\langle tt\rangle_{\bullet}\ !\ \bullet}$, which, however, is not possible: the evaluation rule requires the tag to be a label (element of syntactic category $\mathtt{l}$), and $\bullet$, representing an invalidated or deleted tag, is not such a label. Indeed the exclusion of $\bullet$ in the evaluation rule for $\mathtt{\langle v\rangle_l\ !\ L}$ captures that deallocated values — values with tag $\bullet$ — are irrevocably lost. In the next session we introduce effects to to capture the need of access to $\mathtt{l}$ for soundness.

Notes on value flow analysis

Data flow analysis has been used in compilers already in the early 60s. See Aho et al. (1986) for its history. The historical starting point for value flow analysis for structured values (records, tuples), constraint-based flow analysis and soft typing is Reynolds' seminar work on data set definitions Reynolds (1969b). Schwartz (1975) developed value flow analysis independently for structured values in the context of SETL and was the first to suggest exploiting lifetime analysis based on value flow analysis for region-based memory management. Sestoft (1988, 1989) and Shivers (1988, 1991) developed value flow analysis for function values (closures). Shivers coined the term 0CFA, which in other literature is also used for monovariant value flow analysis (which is different from Shivers' 0CFA, however; see Mossin (1997) for a discussion of this). Shivers also invented a hierarchy kCFA of polyvariant value flow analyses.

Polymorphic value flow analysis was developed by Mossin (1997), extending on earlier work by Dussart et al. (1995b); Henglein and Mossin (1994) on combining subtyping, parametric polymorphism and polymorphic recursion for binding-time analysis. Polymorphic value flow analysis is modular and can be computed asymptotically in the same time as monomorphic value flow analysis. Transitive closure is the algorithmic bottleneck in both. Efficient algorithms are given in Fähndrich et al. (2000); Rehof and Fähndrich (2001) and Gustavsson and Svenningsson (2001a).

Palsberg and O'Keefe (1995) showed that safety analysis, a constraint-based analysis corresponding to monovariant value flow analysis, characterizes typability in Amadio and Cardelli (1991)'s type system with recursive subtyping. Constraint-based value flow analysis for object-oriented languages was pioneered by Palsberg and Schwartzbach (1990, 1994).

The presentation in this section covers only monovariant value flow analysis and greatly owes to Mossin (1997). See Nielson and Nielson for a presentation based on flow logic and abstract interpretation.

5.3 Effects

We have seen in the previous subsection that the soundness of a typing rule may depend not only on the properties of the results of evaluations, but on certain aspects of the evaluation itself, in other words on *how* a value is computed, not just *which* value is computed. To capture properties of evaluation we introduce *effects*.

Effect type judgements

The basic effect type judgement is

$$\Gamma \vdash \mathtt{t} :^{\varphi} \mathtt{T}$$

where φ is an *effect expression* (henceforth simply called *effect*) and $^{\varphi}\mathtt{T}$ is an *effect type* or *type and effect*. The judgement should be read informally as "Under the assumptions Γ, the evaluation of $\mathtt{t}$ may have the observable effect φ and eventually yields a value of type $\mathtt{T}$, if any." For program analysis purposes *observable* may also be understood as *interesting* (for the purposes of the analysis). When an evaluation has no observable effect, we say it has the *empty effect*, written $\emptyset$.

In a call-by-name language Γ is a sequence of *effect type assumptions* of the form $\mathtt{x} : {}^{\varphi}\mathtt{T}$ since $\mathtt{x}$ may be bound to unevaluated thunks, whereas in a call-by-value language we have *type assumptions* of the form $\mathtt{x} : \mathtt{T}$ since variables are bound to *values*, whose evaluation is guaranteed to always have the

empty effect. Analogously, in a call-by-name language we have general functional types of the form $\varphi_1 T_1 \to \varphi_2 T_2$; in a call-by-value language, however, we can restrict ourselves to functional types of the form $T_1 \to \varphi T_2$.[2]

Effect typed language ETL

We shall now present an effect typing system for language ETL. ETL has the same source terms and evaluation rules as STL. The only difference to STL is its effect type system. Syntax and effect typing rules for ETL are given in Figure 5-5.

The effect union operator is associative, commutative and idempotent with $\emptyset$ as its neutral element. Furthermore, the $\mathtt{get}()$-operator is used as a set homomorphism on label expression, that is modulo the equational theory induced by $\mathtt{get}(L_1 \cup L_2) = \mathtt{get}(L)_1 \cup \mathtt{get}(L)_2$. The upshot of this is that every effect expression has a normal form $\mathtt{get}(l)_1 \cup \ldots \cup \mathtt{get}(l)_n$ or $\mathtt{get}(l)_1 \cup \ldots \cup \mathtt{get}(l)_n \cup \mathtt{get}(\bullet)$ for distinct labels $l_1 \ldots l_n, n \geq 0$, that is unique up to the order of the individual get-effects. (It is a metaproperty of ETL, indeed the essence of its soundness property, that atomic effects of the form $\mathtt{get}(\bullet)$ can never arise in a typing derivation.) Given φ with normal form $\mathtt{get}(l)_1 \cup \ldots \cup \mathtt{get}(l)_n$ the notation $[\mathtt{get}(l)_i \mapsto \emptyset]\varphi$ denotes $\mathtt{get}(l)_1 \cup \ldots \cup \mathtt{get}(l)_{i-1} \mathtt{get}(l) i+1 \ldots \cup \mathtt{get}(l)_n$. In the absence of any other effects we could also have represented effects simply by label expressions since each effect is equivalent to $\mathtt{get}(L)$ for some L.

The guiding intuition for understanding the design of the effect type system is the following. The only interesting *atomic effect* we need to worry about is $\mathtt{get}(l)$, which indicates the effect of executing $\langle v \rangle_l \ ! \ L$; that is, it records that l is *dereferenced*. (Note that we do not record other effects such as allocation of value with label l. This is done in existing region-based memory management systems, but is not necessary for soundness.) Since in the end we are only interested in whether a particular label may be dereferenced during evaluation of a term t or not, our effect system does not record the *order* in which effects take place. We simply record the *set* of atomic effects that

2. The syntax φT has been chosen here for several reasons:

- It expresses that yielding a value of type T is the last "effect" of evaluation; that is it occurs after φ.
- Functional types in a call-by-value language end up being written $T_1 \to \varphi T_2$, which is consistent with the notation used in the literature where the *delayed effect* φ is written above the function type arrow.
- It is consistent with the syntax $M^{\varphi} T$ used in monadic interpretations of type and effect systems in the literature.

.

Terms

t ::=		*terms:*
	v	*value expression*
	x	*variable*
	t t	*application*
	if t then t else t	*conditional*
	t at L	*tagging*
	t ! L	*untagging*
	new l.t	*label-scoped term*
	fix x.t	*recursion*
v ::=		*value expressions:*
	λx.t	*abstraction*
	bv	*truth value*
	$\langle v \rangle_l$	*tagged value*
bv ::=		*truth values:*
	tt	*true*
	ff	*false*

Effect expressions

φ ::=		*effect expressions:*
	get(L)	*get effect*
	∅	*empty effect*
	φ ∪ φ	*effect union*

Types

T ::=		*types:*
	bool	*boolean type*
	$T \to^{\varphi} T$	*function type*
	T at L	*tagged value type*

Sublabeling rule $\boxed{\vdash L \leq L}$

$$\vdash L \leq L \cup L' \qquad \text{(TE-LabelSub)}$$

Subeffecting rule $\boxed{\vdash \varphi \leq \varphi}$

$$\vdash \varphi \leq \varphi \cup \varphi' \qquad \text{(TE-EffectSub)}$$

Subtyping rules $\boxed{\vdash T \leq T'}$

$$\frac{\vdash L_1 \leq L_2 \qquad \vdash T_1 \leq T_2}{\vdash T_1 \text{ at } L_1 \leq T_2 \text{ at } L_2} \text{ (TE-TagValSub)}$$

$$\vdash \text{bool} \leq \text{bool} \qquad \text{(TE-BoolSub)}$$

$$\frac{\vdash T_{21} \leq T_{11} \qquad \vdash T_{12} \leq T_{22} \qquad \vdash \varphi_1 \leq \varphi_2}{\vdash T_{11} \to^{\varphi_1} T_{12} \leq T_{21} \to^{\varphi_2} T_{22}} \text{ (TE-FunSub)}$$

Effect typing rules $\boxed{\Gamma \vdash t :^{\varphi} T}$

$$\frac{x \notin \Gamma'}{\Gamma, x : T, \Gamma' \vdash x :^{\emptyset} T} \qquad \text{(TE-Var)}$$

$$\Gamma \vdash bv :^{\emptyset} \text{bool} \qquad \text{(TE-Bool)}$$

$$\frac{\Gamma \vdash t_1 :^{\varphi_1} \text{bool} \qquad \Gamma \vdash t_2 :^{\varphi_2} T \qquad \Gamma \vdash t_3 :^{\varphi_3} T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 :^{\varphi_1 \cup \varphi_2 \cup \varphi_3} T} \text{ (TE-If)}$$

$$\frac{\Gamma, x : T_1 \vdash t :^{\varphi} T_2}{\Gamma \vdash \lambda x.t :^{\emptyset} T_1 \to^{\varphi} T_2} \qquad \text{(TE-Abs)}$$

$$\frac{\Gamma \vdash t_0 :^{\varphi_1} T_1 \to^{\varphi_3} T_2 \qquad \Gamma \vdash t_1 :^{\varphi_2} T_1}{\Gamma \vdash t_0\, t_1 :^{\varphi_1 \cup \varphi_2 \cup \varphi_3} T_2} \text{ (TE-App)}$$

$$\frac{\Gamma \vdash t :^{\varphi} T}{\Gamma \vdash t \text{ at } L :^{\varphi} T \text{ at } L} \qquad \text{(TE-At)}$$

$$\frac{\Gamma \vdash t :^{\varphi} T \text{ at } L}{\Gamma \vdash t \text{ ! } L :^{\varphi \cup \text{get}(L)} T} \qquad \text{(TE-From)}$$

$$\frac{\Gamma \vdash v :^{\varphi} T}{\Gamma \vdash \langle v \rangle_l :^{\varphi} T \text{ at } l} \qquad \text{(TE-Cell)}$$

$$\frac{\Gamma \vdash t :^{\varphi} T \qquad l \notin \text{flv}(\Gamma, T)}{\Gamma \vdash \text{new } l.t :^{[\text{get}(l) \mapsto \emptyset]\varphi} T} \qquad \text{(TE-New)}$$

$$\frac{\Gamma, x : T \vdash t :^{\varphi} T}{\Gamma \vdash \text{fix } x.t :^{\varphi} T} \qquad \text{(TE-Fix)}$$

$$\frac{\Gamma \vdash t :^{\varphi} T \qquad \vdash T \leq T' \qquad \vdash \varphi \leq \varphi'}{\Gamma \vdash t :^{\varphi'} T'} \qquad \text{(TE-Sub)}$$

Figure 5-5: Effect typed language ETL.

may take place during evaluation of $\mathtt{t}$. In this sense our effect type system is a basic *control-flow insensitive* effect type system. Effect type systems that capture evaluation order are discussed briefly later.

Soundness

Effects make the labels accessed during evaluation sufficiently "visible" to ensure that the typing rule for $\mathtt{new\ l.t}$ is sound. Consider the term

$$\mathtt{t_f = new\ l_0.let\ x = tt\ at\ l_0\ in\ \lambda y.if\ x\ !\ l_0\ then\ y\ else\ ff\ at\ l_1}$$

again. Whereas it is typable in STL, it is not typable in ETL. To see this, consider the let-expression $\mathtt{t_l}$

$$\mathtt{let\ x = tt\ at\ l_0\ in\ \lambda y.if\ x\ !\ l_0\ then\ y\ else\ ff\ at\ l_1.}$$

inside $\mathtt{t_f}$. Its ETL effect type is $\mathtt{bool\ at\ l_1} \to^{\mathtt{get(l_0)}} \mathtt{bool\ at\ l_1}$. Note that $\mathtt{l_0}$ occurs in the effect, but neither in the type's domain nor its range type. This reflects the fact that every application of $\mathtt{t_l}$ dereferences a value tagged with label $\mathtt{l_0}$. Since $\mathtt{l_0} \in \mathrm{flv}(\mathtt{bool\ at\ l_1} \to^{\mathtt{get(l_0)}} \mathtt{bool\ at\ l_1})$ rule (TE-NEW) is *not* applicable, and so there is no way of inferring a type for $\mathtt{t_f}$, which indeed would be unsound. Generally, we can prove the following soundness theorem.

5.3.1 THEOREM [SOUNDNESS OF ETL]: If $\vdash \mathtt{t} :^{\varphi} \mathtt{T}$ then evaluation of $\mathtt{t}$ does not get stuck. □

We shall not prove this result here. The techniques are the same as those in §5.5.

Notes on effect type systems

Type and effect systems were pioneered in Gifford and Lucassen (1986); Lucassen and Gifford (1988); Jouvelot and Gifford (1989) for integrating imperative operations, notably updatable references and control effects, into functional languages. Type and effect inference using unification technology, which is the basis for region inference, is developed in Jouvelot and Gifford (1991); Talpin and Jouvelot (1992, 1994). See below for more references in connection with region inference.

Nielson and Nielson (1994, 1996) pioneered type and effect systems with *behaviors* or *causal* effects, where effect types model order of evaluation. In such systems the language of effect expressions has operators for sequential composition and (internal) choice. They also provide for recursively defined effect expressions. The sequential composition operator captures the

sequential order of the execution of effects. The choice operator corresponds intuitively to our effect union operator. This changes the nature of effects substantially as they basically turn into process algebras and thus have a nontrivial theory of their own. Modeling order of execution is key to capturing synchronization properties of concurrent processes, where atomic effects include the sending and receiving of messages. See Amtoft et al. (1999) and Nielson et al. (1999) for references on soundness, inference and applications.

This is an active research area in terms of both foundations and applications. Applications include verification of cryptographic protocols by effect type checking Gordon and Jeffrey (2001b,a, 2002) and behavior type systems for asynchronous programming Igarashi and Kobayashi (2001); Rajamani and Rehof (2001); Chaki et al. (2002); Rajamani and Rehof (2002).

In terms of to computational λ-calculus of Moggi (1991), types are associated with values and effect types with *computations*; that is, intuitively, effect types correspond to monad types. This connection is investigated by Semmelroth and Sabry (1999); Wadler (1999).

5.4 Region-based memory management

Region-based memory management is a particular way to manage the dynamically (or *heap*) allocated memory of a program. Traditionally, the heap is managed either explicitly by the programmer using constructs such as C's `malloc` and `free` or automatically by a garbage collector leaving the programmer with only the responsibility of when to allocate memory. Region-based memory management uses explicit instructions for the allocation and deallocation of memory, but the safety of the explicit deallocations is guaranteed by a type system, and in some cases a compile-time analysis can insert the instructions automatically.

Basically a *region* is a sub-heap containing a number of heap-allocated values, and the heap is a collection of regions. A region starts out empty and grows when a value is allocated in it. A region can grow indenpently of the other regions constituting the heap; that is, one can allocate values in all the regions currently available. Regions can only shrink when the complete region is deallocated; that is, one does not deallocate individual values only complete regions.

In summary, we use three region primitives: (1) allocation of a new region, (2) allocation of a value in a region, and (3) deallocation of a complete region (thereby all values allocated in the region).

Terms

t	::=		*terms:*
		u	*value or almost value*
		x	*variable*
		if t then t else t	*conditional*
		fix x.u	*recursion*
		t t	*application*
		t [[p]]	*region app.*
		new ρ.t	*letregion*
u	::=		*almost values:*
		v	*value*
		(λx.t) at p	*abstraction*
		(λρ.u) at p	*region abs.*
v	::=		*values:*
		bv	*truth value*
		$\langle \lambda x.t \rangle_p$	*closure*
		$\langle \lambda \rho.u \rangle_p$	*region closure*
bv	::=		*truth values:*
		tt	*true*
		ff	*false*
p	::=		*places:*
		ρ	*region variable*
		•	*deallocated*

Evaluation $\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \longrightarrow \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3} \quad \text{(RE-IF)}$$

$$\texttt{if tt then } t_2 \texttt{ else } t_3 \longrightarrow t_2 \quad \text{(RE-IFTRUE)}$$

$$\texttt{if ff then } t_2 \texttt{ else } t_3 \longrightarrow t_3 \quad \text{(RE-IFFALSE)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(RE-APP1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(RE-APP2)}$$

$$\lambda x.t_{12} \texttt{ at } \rho \longrightarrow \langle \lambda x.t_{12} \rangle_\rho \quad \text{(RE-CLOS)}$$

$$\langle \lambda x.t_{12} \rangle_\rho\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \quad \text{(RE-BETA)}$$

$$\frac{u \longrightarrow u'}{\texttt{fix } x.u \longrightarrow \texttt{fix } x.u'} \quad \text{(RE-FIX)}$$

$$\texttt{fix } x.v \longrightarrow [x \mapsto \texttt{fix } x.v]v \quad \text{(RE-FIXBETA)}$$

$$\frac{t \longrightarrow t'}{t\ [[p]] \longrightarrow t'\ [[p]]} \quad \text{(RE-RAPP)}$$

$$\lambda \rho_1.u_{12} \texttt{ at } \rho \longrightarrow \langle \lambda \rho_1.u_{12} \rangle_\rho \quad \text{(RE-RCLOS)}$$

$$\langle \lambda \rho_1.u \rangle_\rho\ [[p]] \longrightarrow [\rho_1 \mapsto p]u \quad \text{(RE-RBETA)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{new } \rho.t_1 \longrightarrow \texttt{new } \rho.t_1'} \quad \text{(RE-LETREG)}$$

$$\texttt{new } \rho.v \longrightarrow [\rho \mapsto \bullet]v \quad \text{(RE-DEALLOC)}$$

Figure 5-6: Region-annotated language, RAL

5.4.1 A region-annotated language

Our region-annotated language is a lambda calculus with a fixed-point operator and (Boolean) constants, extended with explicit region annotations. Its syntax and evaluation semantics are defined in Figure 5-6. Note that we have replaced "labels" with "places"; this is to reflect their interpretation as region handles and for consistency with the literature.

5.4.1 DEFINITION: Define final region-annotated terms and $\rightarrow_R^{*!}$ by analogy with Definitions 5.2.1 and 5.2.2. Define the function "$\mathrm{eval}_R(\cdot)$" from terms to $\{\mathtt{tt}, \mathtt{ff}, \bot, \mathtt{wrong}\}$ by

a) $\mathrm{eval}_R(\mathtt{t}_0) = \mathtt{bv}$ iff $\mathtt{t}_0 \rightarrow_R^{*!} \mathtt{bv}$.

b) $\mathrm{eval}_R(\mathtt{t}_0) = \bot$ iff there is an infinite sequence $\mathtt{t}_1, \mathtt{t}_1, ..., \mathtt{t}_i, ...$ such that $\mathtt{t}_i \longrightarrow \mathtt{t}_{i+1}$ for $0 \leq i$.

c) $\mathrm{eval}_R(\mathtt{t}_0) = \mathtt{wrong}$ iff $\mathtt{t}_0 \rightarrow^{*!} \mathtt{t}$ where $\mathtt{t}$ is not a value. □

In comparison with Definition 5.2.3, we see that the possibility of `fun` is missing. The is a technical condition; it that our formal development does not need to worry about the case that a program computes a heap-allocated closure that has been deallocated before the program exits—on the contrary, we are free to require that a program deallocates *all* its heap memory before terminating.

The the `new` $\rho.\mathtt{t}$ construct introduces new region variables. The variable ρ can be used to annotate value-producing terms within `t`. The *allocation* of the new region in our system is implicit; it happens automatically when the execution focus moves inside the `new` binder. Implicit alpha-conversion makes sure that the `new` does not capture any foreign region variables before the allocation. On the other hand, *deallocation* is explicit in the evaluation semantics. The (RE-DEALLOC) rule records the fact that a value stored in the deallocated region is no longer available by replacing the region variable with the special marker $\bullet$. The "dangling pointers" to deallocated values can be manipulated freely as long as one does not attempt to read the values they point to. At that point execution will get stuck, because there is no reduction rule for an expression of the form "$\langle \lambda \mathtt{x}.\mathtt{t} \rangle_\bullet \ \mathtt{v}$". The (RE-BETA) that would ordinarily reduce it applies only when the place is a ρ, which explicitly does not include $\bullet$.

Observe that the substitution $[\rho \mapsto \bullet]$ in (RE-DEALLOC) can affect allocation expressions $(\cdots)$ `at` ρ as well as already allocated values $\langle \cdots \rangle_\rho$. In the former case we end up with a "$(\cdots)$ `at` $\bullet$" expression which asks to allocate something in a region that does not exist anymore. This is impossible, of course, but the *creation* of such an expression is not an error. The error happens if the expression is eventually executed; in which case execution will get stuck because (RE-CLOS) and (RE-RCLOS) demand a ρ rather than a `p` after the "`at`". Similarly a $\bullet$ can appear as the actual parameter in a region application, and the application can even be reduced without an error.

A novel aspect of the region-annotated language, compared to the tagged language described previously, is the presence of *region abstractions*. The in-

tention is that a region abstraction "λρ.u" is the natural counter-part to normal abstractions only at the level of regions; one can apply such an abstraction to an actual place parameter p in which case evaluation proceeds by substituting the place p for the formal parameter ρ in u, and then evaluates the result of this substitution. Region abstractions allows one to parameterize a function over the regions necessary for the evaluation of the function. Typically, this means paramerizing over the regions containing the input to the function, and the regions in which the output should be stored. We say that such a function is *region polymorphic* in the region parameters.

For example, consider the following program to compute Fibonacci numbers[3]

```
fix fib. λn.
    if n<2 then 1
    else fib(n-2)+fib(n-1)
```

One possible region annotation of this program is (ignore everything but the first line for now)

```
fix fib. (λρi. (λρo. (λn.
    if new ρ. (n < (2 at ρ) then 1 at ρo)
    else new ρ1.
              new ρ2.fib[[ρ2]][[ρ1]] (new ρ.n -at ρ2 (2 at ρ))
         +at ρo new ρ3.fib[[ρ3]][[ρ1]] (new ρ.n -at ρ3 (1 at ρ))
    ) at ρi) at ρi) at ρf
```

The point is that the fib function expects two region parameters at runtime: one, ρ_i, in which the input n is stored, and one, ρ_o, in which the function is supposed to store its result. Thus, any caller of fib is required to choose appropriate actual regions for these as witnessed in the two calls to fib in the body.

Observe that, since the only way to allocate and deallocate a region is via the new construct, it is not possible for the function to deallocate a region associated with a parameter, and similarly, the function cannot itself allocate such a region. The consequence is that the lifetime of regions passed as parameters to a function encompasses the lifetime of the complete function invocation. In order to avoid large, long-lived regions it is therefore important to allow the body of a recursive function to use actual parameters to recursive innovations different from the formal parameters.

3. In examples we shall allow ourselves to use features such as integers allocated in regions even though they are not part of the formal developments.

Continuing the Fibonacci example above, it is crucial that the two recursive calls to `fib` can each supply their own actual parameters (in this case ρ_2, ρ_1 and ρ_3, ρ_1). Thus, for each call we store the arguments in separate regions whose lifetimes are just the duration of the function call. The results need slightly longer lifetimes, since we need to add those up to give the result of the original call, but they can be stored in the same region. (The example is taken from Tofte and Talpin (1997).)

This aspect is referred to as *region polymorphic recursion* in the literature, as it allows us to choose different instantiations of the polymorphically bound region parameters for different invocations.

5.4.2 EXERCISE [★★]: What would happen to the region-behavior of the Fibonacci program if region polymorphic recursion was disallowed (ie., if the recursive calls were required to use the formal region parameters as actual region parameters)? □

5.4.2 SOLUTION: Both of the two recursive calls would have to specify ρ_i, ρ_o as actual parameters, and so all intermediate arguments and results would end up in the same two regions; namely the two argument regions supplied to the function at the outermost level.

The original Tofte-Talpin system restricts the places where region abstractions and recursive function definitions can occur. Region abstractions are only allowed in the definition of recursive functions, and a recursive function definition must appear in a `let` binding. Due to these restrictions, the original formulation of the Tofte-Talpin system has a single combined construction

$$\texttt{t} ::= \texttt{letrec f}[\rho_1, ..., \rho_k]\,\texttt{(x) at}\ \rho = \texttt{t}_1\ \texttt{in t}_2$$

which in our simplified region-annotated language corresponds to

$$\texttt{let f} = \texttt{fix f}.(\lambda\rho_1, ..., \rho_k, \rho'.(\lambda\texttt{x}.\texttt{t}_1)\ \texttt{at}\ \rho')\ \texttt{at}\ \rho\ \texttt{in t}_2$$

Using the `letrec` as an abbreviation we can rewrite the Fibonacci example to the following program:

```
letrec fib[ρi,ρo] (n) at ρf =
   if new ρ in (n < (2 at ρ) then 1 at ρo)
   else
     new ρ1 in
         new ρ2 in fib[ρ2,ρ1] (new ρ in n -at ρ2 (2 at ρ))
     +at ρonew ρ3 in fib[ρ3,ρ1] (new ρ in n -at ρ3 (1 at ρ))
```

5.4.3 EXERCISE [★]: What is the role of the ρ' parameter in the above expansion of the original TT `letrec` construction? Can you guess why it is not part of the original TT syntax? □

5.4.3 SOLUTION: When the region abstraction is applied (i.e., each time `f` is mentioned), a closure must be allocated to contain the region parameters and the free variables of the function body, because the ordinary PCF_2 parameter may not be supplied right away. The parameter ρ' selects the region in which this closure will be allocated. It is not part of the TT syntax for `letrec` because this closure allocation is implicit in the `letrec` construct; instead the original syntax for applying the region abstraction is

$$\texttt{t} ::= \texttt{f}[\rho_1, ..., \rho_k]\ \texttt{at}\ \rho'$$

which in our system would translate to $\texttt{f}\ [\![\rho_1, ..., \rho_k, \rho']\!]$.

5.4.4 EXERCISE [★]: What is the role of ρ in the `letrec` construction? Is it really operationally necessary? □

5.4.4 SOLUTION: ρ is the region where a closure for the region abstraction is allocated. This closure contains the values of the free variables of the lambda expression. The intention in the TT calculus was the this closure would be consulted when the region abstraction is applied, such that the variables could be moved to the final closure in ρ'. However, due to the syntactic requirement that the region abstraction is applied whenever `f` is mentioned in $\texttt{t}_1$ or $\texttt{t}_2$, the free variables will actually still be in scope at the application point. Since the body of the region abstraction is also statically known, nothing actually needs to be allocated in ρ, and indeed the ML Kit does not allocate this closure. But this was not realized when the TT calculus was first formulated.

5.4.5 EXERCISE [★★]: Our `letrec` unfolding uses abstraction over multiple regions at once, which is not actually part of the region-annotated language that we have proved things about. Show how n-ary region abstractions can be simulated using our unary ones. □

5.4.5 SOLUTION: An n-ary region abstraction can be converted into a stack of unary ones, but one must decide where the intermediate region closures that the semantics require are allocated. The following solution takes care not to cause any net heap allocation in the translation of an n-ary application:

$$(\lambda\rho_1, \ldots, \rho_n.\texttt{t})\ \texttt{at}\ \rho \Rightarrow (\lambda\rho'.(\lambda\rho_1.\cdots(\lambda\rho_n.\texttt{t})\ \texttt{at}\ \rho'\cdots)\ \texttt{at}\ \rho')\ \texttt{at}\ \rho$$
$$\texttt{f}\ [\![\rho_1, \ldots, \rho_n]\!] \Rightarrow \texttt{new}\ \rho'.\texttt{f}\ [\![\rho']\!]\ [\![\rho_1]\!]\cdots[\![\rho_n]\!]$$

The dynamic semantics presented by Tofte and Talpin (1997) goes to some length to stress that regions are allocated and deallocated according to a stack discipline. At runtime, a *region environment* maps region variables to concrete regions (denoted by `r`), and a *store* maps (concrete) regions to the values stored in them. Evaluation of a `new` ρ.`t` then proceeds as follows: (1) first bind a fresh region name `r` to ρ in the region environment and bind `r` to the empty region in the store, (2) then proceed with the evaluation of `t` in this extended runtime configuration, and (3) complete the evaluation of the entire term by removing the binding of ρ to `r` and the binding of `r` to the region.

That original formulation is closer to an operational understanding of how the region operations work, than the *store-less* semantics we use here. On the other hand, the store-less semantics is easier to reason about; a trick due to Helsen and Thiemann (2000) and Calcagno (2001). See Calcagno et al. (2002) for a proof that the two styles of semantics are indeed equivalent.

5.4.2 Reusing deallocated memory

Intuitively it should be safe to reuse deallocated memory (indicated by the special place $\bullet$) while executing a region-safe program. More formally, assume that $\mathtt{t}_\bullet$ is a term containing deallocated values and that `t` is constructed from $\mathtt{t}_\bullet$ by replacing some of these with new values. Then if $\mathtt{t}_\bullet$ evaluates to some value or loops indefinitely (ie., it does not go wrong), then so does `t`.

5.4.6 PROPOSITION: Let *Val* be the set of values and *Dead* be the subset of values of the form "$\langle ... \rangle_\bullet$". Let the relation $\preceq$ between terms be the compatible closure of *Dead* $\times$ *Val*. That is, $\mathtt{t}_\bullet \preceq \mathtt{t}$ if `t` arises from $\mathtt{t}_\bullet$ by replacing some (zero or more) deallocated values by arbitrary new values.

If $\mathtt{t}_\bullet \preceq \mathtt{t}$ and $\mathrm{eval}_R(\mathtt{t}_\bullet) = Y \neq \mathtt{wrong}$, then $\mathrm{eval}_R(\mathtt{t}) = Y$, too. □

Proof: Exercise (★★). □

5.4.6 SOLUTION: A simple induction over the derivation of the evaluation relation proves the lemma that if $\mathtt{t}_\bullet \longrightarrow \mathtt{t}'_\bullet$ and $\mathtt{t}_\bullet \preceq \mathtt{t}$, then there is a $\mathtt{t}'$ such that $\mathtt{t} \longrightarrow \mathtt{t}'$ and $\mathtt{t}'_\bullet \preceq \mathtt{t}'$.

Apply this lemma to each step of the reduction of the original $\mathtt{t}_\bullet$. In the case $Y = \mathtt{bv}$, note that $\mathtt{bv} \preceq \mathtt{t}$ implies $\mathtt{bv} = \mathtt{t}$.

5.4.3 Annotating programs with regions preserves meanings

Region annotating a program is the process of adding region annotations to it to make the memory management explicit (see §5.6 for how to do this automatically). Thus, the process takes a program written in PCF_2 and pro-

$$
\begin{array}{rcl}
\|\mathtt{bv}\| & = & \mathtt{bv} \\
\left\|\begin{array}{l}\mathtt{if}\ \mathtt{t}_0 \\ \quad \mathtt{then}\ \mathtt{t}_1 \\ \quad \mathtt{else}\ \mathtt{t}_2\end{array}\right\| & = & \begin{array}{l}\mathtt{if}\ \|\mathtt{t}_0\| \\ \quad \mathtt{then}\ \|\mathtt{t}_1\| \\ \quad \mathtt{else}\ \|\mathtt{t}_2\|\end{array} \\
\|\mathtt{x}\| & = & \mathtt{x} \\
\|(\lambda \mathtt{x}.\mathtt{t})\ \mathtt{at}\ \mathtt{p}\| & = & \lambda \mathtt{x}.\|\mathtt{t}\| \\
\left\|\langle \lambda \mathtt{x}.\mathtt{t}\rangle_{\mathtt{p}}\right\| & = & \lambda \mathtt{x}.\|\mathtt{t}\|
\end{array}
\qquad
\begin{array}{rcl}
\|\mathtt{t}_1\ \mathtt{t}_2\| & = & \|\mathtt{t}_1\|\ \|\mathtt{t}_2\| \\
\|\mathtt{fix}\ \mathtt{x}.\mathtt{t}\| & = & \mathtt{fix}\ \mathtt{x}.\|\mathtt{t}\| \\
\|\mathtt{new}\ \rho.\mathtt{t}\| & = & \|\mathtt{t}\| \\
\|(\lambda \rho.\mathtt{t})\ \mathtt{at}\ \mathtt{p}\| & = & \|\mathtt{t}\| \\
\left\|\langle \lambda \rho.\mathtt{t}\rangle_{\mathtt{p}}\right\| & = & \|\mathtt{t}\| \\
\|\mathtt{t}\ [\![\mathtt{p}]\!]\| & = & \|\mathtt{t}\|
\end{array}
$$

Figure 5-7: Definition of the erasure function

duces a program written in RAL. The intention is, of course, that the region-annotated program is supposed to have the same behavior as the original program. In other words, we shall prove that adding region annotations preserves the meaning of the program. We make this precise in the present section. We do so by starting with a region-annotated program and showing that it behaves the same as the program obtained by removing all region annotations (thereby obtaining a program in PCF_2, Figure 5-1),

Analogous to erasure for TL-terms, going from a term in the region-annotated language RAL to a term in the underlying base language PCF_2 is a matter of erasing all region annotations.

5.4.7 DEFINITION: The *erasure* $\|\mathtt{t}\|$ of a region-annotated term $\mathtt{t}$ is the ordinary PCF_2-term defined by removing the region annotations, as shown in Figure 5-7. □

The ideal meaning-preservation statement would be: For any region-annotated program $\mathtt{t}$, if $\mathrm{eval}_R(\mathtt{t}) = Y$ then $\mathrm{eval}(\|\mathtt{t}\|) = Y$ and vice versa. Unfortunately that is not true, since $\mathtt{t}$ can go wrong due to memory-management errors (such as trying to read a value after it has been deallocated) that have no counterpart in $\|\mathtt{t}\|$. What we *can* prove, however, is

5.4.8 THEOREM [CONDITIONAL CORRECTNESS]: Let $\mathtt{t}$ be a region-annotated program (formally: any term), and assume $\mathrm{eval}_R(\mathtt{t}) \neq \mathtt{wrong}$. Then $\mathrm{eval}_R(\mathtt{t}) = \mathrm{eval}(\|\mathtt{t}\|)$. □

In other words, a region-annotated program behaves the same as the original unannotated program, unless it goes wrong. Our semantics for region-annotated programs does not allow us to distinguish between going wrong because of memory-management errors and going wrong due to plain old

type errors, but it would be straightforward (though tedious) to extend the semantics with such a notion and then prove that if $\mathrm{eval}(\|\mathtt{t}\|) \neq \mathrm{eval}(\mathtt{t})$ then $\mathrm{eval}(\mathtt{t})$ is memory-wrong rather than type-wrong. Since we are primarily concerned with well-typed programs, we will not pursue that further.

The proof of the the theorem proceeds through a series of lemmas:

5.4.9 LEMMA: Assume that $\mathtt{t}$ is a value $\mathtt{v}$ *or* an almost-value $\mathtt{u}$. Then $\|\mathtt{t}\|$ is a value for PCF_2. □

Proof: By structural induction on $\mathtt{t}$. The induction hypothesis is used in the case of region abstractions and closures, which disappear during erasure. (This is why the body of a region abstraction is restricted to be an almost-value rather than an arbitrary term). □

5.4.10 LEMMA [SIMULATION]: Assume $\mathtt{t} \longrightarrow \mathtt{t}'$. Then either (a) $\|\mathtt{t}\| \xrightarrow{\mathrm{PCF}_2} \|\mathtt{t}'\|$ or (b) $\|\mathtt{t}\| = \|\mathtt{t}'\|$. □

Proof: By induction on the derivation of $\mathtt{t} \longrightarrow \mathtt{t}'$.

For the rules (RE-IF), (RE-APP1), and (RE-APP2), apply the induction hypothesis. If that yielded case (a), use the corresponding context rule from PCF_2. In the case of (RE-APP2), Lemma 5.4.9 ensures that erasure of the function expression is still a value, such that the corresponding PCF_2 rule is available.

For the rule (RE-FIX), first observe that since the body of the fix is an almost-value, the only rules that can establish the reduction $\mathtt{u} \longrightarrow \mathtt{u}'$ are (RE-CLOS) and (RE-RCLOS). Then, by inspection of each of these rules we find $\|\mathtt{u}\| = \|\mathtt{u}'\|$, and thus also $\|\mathtt{fix\ x.u}\| = \|\mathtt{fix\ x.u}'\|$.

For (RE-IFTRUE), (RE-IFFALSE), (RE-BETA), and (RE-FIXBETA), the $\|\mathtt{t}\| \xrightarrow{\mathrm{PCF}_2} \|\mathtt{t}'\|$ case applies through the corresponding PCF_2 reductions.

For (RE-RAPP), and (RE-NEW), use the induction hypothesis directly.

For (RE-CLOS) and (RE-RCLOS), $\|\mathtt{t}\| = \|\mathtt{t}'\|$ holds trivially. Similarly for also the case for (RE-RBETA) or (RE-DEALLOC), because the erasure hides the effect of region substitutions. □

5.4.11 LEMMA [SIMULATED PROGRESS]: Assume $\mathtt{t} \longrightarrow \mathtt{t}'$ yet not $\|\mathtt{t}\| \xrightarrow{\mathrm{PCF}_2} \|\mathtt{t}'\|$. Then $\mathtt{t}'$ is strictly smaller than $\mathtt{t}$, under a size measure where unevaluated abstractions are considered "larger" (e.g., twice as large) than closures. □

Proof: From the proof of Lemma 5.4.10 it is clear that the derivation of $\mathtt{t} \longrightarrow \mathtt{t}'$ must consist of a stack of context rules with one of the axioms (RE-CLOS), (RE-RCLOS), (RE-RBETA), or (RE-DEALLOC) at the top. Because the context rules do not themselves add material to the term, it is sufficient to check

the lemma for those four axioms. For (RE-CLOS) and (RE-RCLOS), the size measure is explicitly defined to make the lemma true. For (RE-RBETA) or (RE-DEALLOC), the region substitution does not change the size of its argument, whereas the reductions remove either the Λ or the ν binder. □

5.4.12 LEMMA: Assume $\mathrm{eval}_R(\mathtt{t}) = \mathtt{bv}$. Then $\mathrm{eval}(\|\mathtt{t}\|) = \mathtt{bv}$, too. □

Proof: We have that $\mathtt{t} \to^{*!} \mathtt{bv}$. By applying Lemma 5.4.10 to each of the reduction steps in turn, we get $\|\mathtt{t}\| \xrightarrow{\mathrm{PCF}_2}{}^{*} \|\mathtt{bv}\| = \mathtt{bv}$. Since $\mathtt{bv}$ has no successor, $\mathrm{eval}(\|\mathtt{t}\|) = \mathtt{bv}$. □

5.4.13 LEMMA: Assume $\mathrm{eval}_R(\mathtt{t}_0) = \bot$. Then $\mathrm{eval}(\|\mathtt{t}_0\|) = \bot$, too. □

Proof: The assumption gives us an infinite series of reductions

$$\mathtt{t}_0 \to \mathtt{t}_1 \to \cdots \to \mathtt{t}_i \ldots$$

By Lemma 5.4.10, we get for each $i \geq 0$ that either $\|\mathtt{t}_i\| = \|\mathtt{t}_{i+1}\|$ or $\|\mathtt{t}_i\| \xrightarrow{\mathrm{PCF}_2} \|\mathtt{t}_{i+1}\|$. Lemma 5.4.11 guarantees that there does not exist an N such that $\|\mathtt{t}_i\| \not\xrightarrow{\mathrm{PCF}_2} \|\mathtt{t}_{i+1}\|$ for all $i > N$. Therefore, by chosing certain i's, we get an infinite series of PCF_2 reductions

$$\|\mathtt{t}_0\| \xrightarrow{\mathrm{PCF}_2} \|\mathtt{t}_{i_1}\| \xrightarrow{\mathrm{PCF}_2} \cdots \xrightarrow{\mathrm{PCF}_2} \|\mathtt{t}_{i_j}\| \ldots$$

hence $\mathrm{eval}_R(\|\mathtt{t}\|) = \bot$. □

Proof: [of Theorem 5.4.8] Assume that $\mathrm{eval}_R(\mathtt{t}) \neq \mathtt{wrong}$. Then $\mathrm{eval}_R(\mathtt{t})$ is either $\mathtt{bv}$ or $\bot$, and one of the last two lemmas gives us $\mathrm{eval}(\|\mathtt{t}\|) = \mathrm{eval}_R(\mathtt{t})$. □

5.5 The Tofte-Talpin type system

One of the features that differentiates Tofte and Talpin's region language from other region-based systems (such as Hanson (1990), Ross (1967), and Schwartz (1975)) is the presence of a *type system*. The type system is sound (page 318) and thus well-typed programs do not go wrong at runtime. In the present setting, this means that if the term $\mathtt{t}$ is well-typed then $\mathrm{eval}_R(\mathtt{t}) \neq \mathtt{wrong}$. In particular, well-typed programs are memory safe. In contrast to this, in the systems mentioned above the programmer has to establish memory safety manually and this is essentially just as hard as estalishing memory safety of C-like `malloc/free` programs.

The region type system is defined in Figure 5-8. The judgment $\Gamma \vdash \mathtt{t} :^{\varphi} \mathtt{T}$ reads: in type environment Γ the term $\mathtt{t}$ has type $\mathtt{T}$ and effect φ. The effect

Type expressions

p	$\in$	Place	*places*
ϵ	$\in$	EffVar	*effect variables*
φ	$\in$	$\mathcal{P}_{\text{fin}}(\text{Place} \cup \text{EffVar})$	*effects*
τ	::=		*type expressions:*
		α	*type variable*
		$\texttt{bool}$	*Boolean type*
		$(\tau \to^{\varphi} \tau', p)$	*function type*
		$(\Pi\rho.^{\varphi}\tau, p)$	*region func.*
		$\forall\alpha.\tau$	*type polymorphism*
		$\forall\epsilon.\tau$	*effect polymorphism*

Typing rules $\boxed{\Gamma \vdash t :^{\varphi} \tau}$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x :^{\varphi} \tau} \quad \text{(TT-VAR)}$$

$$\Gamma \vdash \texttt{bv} :^{\varphi} \texttt{bool} \quad \text{(TT-BOOL)}$$

$$\frac{\Gamma \vdash t_1 :^{\varphi} \texttt{bool} \qquad \Gamma \vdash t_2 :^{\varphi} \tau \qquad \Gamma \vdash t_3 :^{\varphi} \tau}{\Gamma \vdash \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 :^{\varphi} \tau} \quad \text{(TT-IF)}$$

$$\frac{\Gamma, x : \tau_1 \vdash t :^{\varphi_2} \tau_2 \qquad p \in \varphi}{\Gamma \vdash (\lambda x.t)\ \texttt{at}\ p :^{\varphi} (\tau_1 \to^{\varphi_2} \tau_2, p)} \quad \text{(TT-ABS)}$$

$$\frac{\Gamma, x : \tau_1 \vdash t :^{\varphi_2} \tau_2}{\Gamma \vdash \langle \lambda x.t \rangle_p :^{\varphi} (\tau_1 \to^{\varphi_2} \tau_2, p)} \quad \text{(TT-CLOS)}$$

$$\frac{\Gamma \vdash t_0 :^{\varphi} (\tau_1 \to^{\varphi_2} \tau_2, p) \qquad \Gamma \vdash t_1 :^{\varphi} \tau_1 \qquad p \in \varphi \qquad \varphi_2 \subseteq \varphi}{\Gamma \vdash t_0\ t_1 :^{\varphi} \tau_2} \quad \text{(TT-APP)}$$

$$\frac{\Gamma, x : \tau \vdash u :^{\varphi} \tau}{\Gamma \vdash \texttt{fix}\ x.u :^{\varphi} \tau} \quad \text{(TT-FIX)}$$

$$\frac{\Gamma \vdash u :^{\varphi'} \tau \qquad \rho \notin \text{frv}(\Gamma) \qquad p \in \varphi}{\Gamma \vdash (\lambda\rho.u)\ \texttt{at}\ p :^{\varphi} (\Pi\rho.^{\varphi'}\tau, p)} \quad \text{(TT-RABS)}$$

$$\frac{\Gamma \vdash u :^{\varphi'} \tau \qquad \rho \notin \text{frv}(\Gamma)}{\Gamma \vdash \langle \lambda\rho.u \rangle_p :^{\varphi} (\Pi\rho.^{\varphi'}\tau, p)} \quad \text{(TT-RCLOS)}$$

$$\frac{\Gamma \vdash t :^{\varphi} (\Pi\rho.^{\varphi'}\tau, p) \qquad p \in \varphi \qquad [\rho \mapsto p']\varphi' \subseteq \varphi}{\Gamma \vdash t\ [\![p']\!] :^{\varphi} [\rho \mapsto p']\tau} \quad \text{(TT-RAPP)}$$

$$\frac{\Gamma \vdash t :^{\varphi,\rho} \tau \qquad \rho \notin \text{frv}(\Gamma, \tau)}{\Gamma \vdash \texttt{new}\ \rho.t :^{\varphi} \tau} \quad \text{(TT-LETREG)}$$

$$\frac{\Gamma \vdash t :^{\varphi} \tau \qquad \alpha \notin \text{ftv}(\Gamma)}{\Gamma \vdash t :^{\varphi} \forall\alpha.\tau} \quad \text{(TT-TGEN)}$$

$$\frac{\Gamma \vdash t :^{\varphi} \forall\alpha.\tau}{\Gamma \vdash t :^{\varphi} [\alpha \mapsto \tau']\tau} \quad \text{(TT-TINST)}$$

$$\frac{\Gamma \vdash t :^{\varphi} \tau \qquad \epsilon \notin \text{fev}(\Gamma, \varphi)}{\Gamma \vdash t :^{\varphi} \forall\epsilon.\tau} \quad \text{(TT-EGEN)}$$

$$\frac{\Gamma \vdash t :^{\varphi} \forall\epsilon.\tau}{\Gamma \vdash t :^{\varphi} [\epsilon \mapsto \varphi']\tau} \quad \text{(TT-EINST)}$$

Figure 5-8: Tofte-Talpin type system

captures the regions that has to be live (ie., allocated) for the term to evaluate without memory problems. In types, $\forall\alpha.\tau$ binds the type variable α in τ, $\Pi\rho.^{\varphi}\tau$ binds the region variable ρ in τ, and $\forall\epsilon.\tau$ binds the effect variable ϵ in τ. We denote the sets of free type variables, free region variables, and free effect variables of a type τ by $\text{ftv}(\tau)$, $\text{frv}(\tau)$, and $\text{fev}(\tau)$, respectively. These are extended to typing contexts in the standard manner. We write $[\alpha \mapsto \tau]$, $[\rho \mapsto p]$, and $[\epsilon \mapsto \varphi]$ for the capture-avoiding substitutions of type τ for the

type variable X, place p for the region variable ρ, and effect φ for the effect variable ϵ, respectively.

The typing rules in Figure 5-8 are natural extensions of the typing rules for the effect typed language in Figure 5-5 (page 304). The region-annotated language, is however, both type polymorphic and effect polymorphic. The type system includes standard rules for introducing and eliminating type polymorphism and the obvious variations for effect polymorphism. Compared to System *F* (TAPL Chapter 23) we do not have explicit syntax for these introductions and eliminations. As already mentioned, the language also contains region polymorphism. Region polymorphism is explicit in the syntax because it has operational significance.

Effect polymorphism is the natural complement to type polymorphism and higher-order functions. Consider a higher-order polymorphic function such as list `map`: it takes a function and a list as arguments and applies the function to each element in the list and has type $\forall\alpha,\beta.(\alpha \to \beta) \times \alpha\ \texttt{list} \to \beta\ \texttt{list}$ in the region-free base language. What is the effect of applying the equivalent of `map` in the region-annotated language to such arguments? It certainly has to include the effect of applying the function, and thus we would have to specify that in the type of the region-annotated `map` function: $\forall\alpha,\beta.(\alpha \to^{\varphi} \beta) \times (\alpha\ \texttt{list}, \rho) \to^{\{\rho,\rho'\}\cup\varphi} (\beta\ \texttt{list}, \rho')$ (see page 322 for the typing rules concerning lists). However, that would only allow us to apply `map` to functions with latent effect φ. We could of course inspect the complete program and make sure that all effects were large enough that this is not a problem, but then we would unnecessarily keep many regions alive. Instead we can employ effect polymorphism to propagate the effect of the functional argument to the effect of the complete evaluation of `map` as in $\forall\alpha,\beta.\forall\epsilon(\alpha \to^{\epsilon} \beta) \times (\alpha\ \texttt{list}, \rho) \to^{\{\rho,\rho'\}\cup\epsilon} (\beta\ \texttt{list}, \rho')$.

The type system presented in Figure 5-8 is based on the type system by Tofte and Talpin (1997).[4] Compared to that type system our system have moved effect enlargement up in the derivation tree so that it is the responsible of the axioms to introduce proper effects. This simplifies both the presentation of the rules and the soundness proof slightly, and it is possible to estalibsh a meta property of the type system that allows for effects to be enlarged. Moreover, the system presented here is more permissive than Tofte and Talpin's system due to the System *F*-like polymorphism in types, regions, and effects compared to the let-polymorphism of the original system. The restrictions present in the original system was there to make the job easier for the region inference algorithm.

4. The type system in that paper is not defined explicitly, instead it can be obtained from the "region inference rules" that specifies a typed translation from the region-free base language to the region-annotated language.

Compare again the `letrec` construction of the Tofte-Talpin system

$$\mathtt{t} ::= \mathtt{letrec\ f[}\rho_1,...,\rho_k\mathtt{]\ (x)\ at}\ \rho = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2$$

expressed in our system as

$$\mathtt{let\ f} = \mathtt{fix\ f.}(\lambda\rho_1,...,\rho_k,\rho'.(\lambda\mathtt{x.t}_1)\ \mathtt{at}\ \rho')\ \mathtt{at}\ \rho\ \mathtt{in}\ \mathtt{t}_2$$

The combined construction can be typed by a stack of our primitive rules:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma,\mathtt{f}\mapsto\mathtt{T}_{12},\mathtt{x}\mapsto\mathtt{T}\vdash\mathtt{t}_1:^{\varphi'}\mathtt{T}_1}{\Gamma,\mathtt{f}\mapsto\mathtt{T}_{12}\vdash\mathtt{t}_{14}:^{\rho'}\mathtt{T}_{14}}\,(\text{TT-ABS})}{\Gamma,\mathtt{f}\mapsto\mathtt{T}_{12}\vdash\mathtt{t}_{13}:^{\varphi}\mathtt{T}_{13}}\,(\text{TT-RABS})}{\vdots}\,(\text{TT-EGEN})}{\Gamma,\mathtt{f}\mapsto\mathtt{T}_{12}\vdash\mathtt{t}_{13}:^{\varphi}\mathtt{T}_{12}}\,(\text{TT-EGEN})}{\Gamma\vdash\mathtt{t}_{11}:^{\varphi}\mathtt{T}_{12}}\,(\text{TT-FIX})}{\vdots}\,(\text{TT-TGEN})}{\Gamma\vdash\mathtt{t}_{11}:^{\varphi}\mathtt{T}_{11}}\,(\text{TT-TGEN})\qquad\Gamma,\mathtt{f}\mapsto\mathtt{T}_{11}\vdash\mathtt{t}_2:^{\varphi}\mathtt{T}_2}{\Gamma\vdash\mathtt{letrec\ f[}\rho_1,...,\rho_k\mathtt{]\ (x)\ at}\ \rho=\mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2:^{\varphi}\mathtt{T}_2}\,(\text{TT-LET})$$

where

$$\begin{array}{ll}
\mathtt{t}_{14} = (\lambda\mathtt{x.t}_1)\ \mathtt{at}\ \rho' & \mathtt{T}_{14} = (\mathtt{T}\to^{\varphi'}\mathtt{T}_1,\rho')\\
\mathtt{t}_{13} = (\lambda\rho_1,...,\rho_k,\rho'.\mathtt{t}_{14})\ \mathtt{at}\ \rho & \mathtt{T}_{13} = (\Pi\rho_1,...,\rho_k,\rho'.^{\rho'}\mathtt{T}_{14},\rho)\\
 & \mathtt{T}_{12} = \forall\epsilon_1....\forall\epsilon_n.\mathtt{T}_{13}\\
\mathtt{t}_{11} = \mathtt{fix\ f.t}_{13} & \mathtt{T}_{11} = \forall\mathtt{X}_1....\forall\mathtt{X}_m.\mathtt{T}_{12}
\end{array}$$

As usual with let-polymorphism, each time `f` is mentioned in $\mathtt{t}_2$, its type scheme must be fully instantiated immediately. This principle is extended to the effect and region abstractions; these must *also* be fully instantiated (applied, in the case of region abstraction) each time `f` is mentioned in $\mathtt{t}_2$ *or* $\mathtt{t}_1$. Thus, the original Tofte-Talpin type system does not allow expressions in general to have type $(\Pi\rho.^{\varphi}\mathtt{T},\mathtt{p})$; in particular region abstractions cannot be passed as parameters to, or returned from, functions.

Syntactic type soundness

We will now prove that typable programs are memory safe. We do so by establishing *type soundness*, ie. that well-typed programs do no go wrong. The type soundness proof is a structured as a standard sequence of Substitution,

Subject Reduction, and Progress lemmas. This approach was pioneered by Helsen and Thiemann (2000) in the context of region-based languages, and apparently independently discovered by Calcagno (2001) for a big-step semantics.

We start by showing that one can massage derivations so that the typing context only mentions the free variables of the term, and so that the derivation does not end with one of the instantiation rules.

5.5.1 LEMMA: If $\Gamma \vdash \mathtt{t} :^{\varphi} \mathtt{T}$, $dom\Gamma' = \mathrm{fv}(\mathtt{t})$, and Γ and Γ' agree when both are defined, then $\Gamma' \vdash \mathtt{t} :^{\varphi} \mathtt{T}$. □

Proof: Straightforward induction on the typing derivation. □

5.5.2 LEMMA: Let S be a substitution of the form $[\rho \mapsto \mathtt{p}]$, $[\epsilon \mapsto \varphi]$, or $[\mathtt{X} \mapsto \mathtt{T}]$. If $\Gamma \vdash \mathtt{t} :^{\varphi} \mathtt{T}$ can be derived in n steps, then likewise can $S\Gamma \vdash S\mathtt{t} :^{S\varphi} S\mathtt{T}$. □

Proof: Left as an exercise (★). □

5.5.2 SOLUTION: The only interesting issue in the proof is that the substitutions substitutes from sets of variables to another syntactic class (from region variables to places, for example). Observe, that in the type system everywhere a type, region, or effect variable is mentioned, it occurs in a binding context.

5.5.3 LEMMA: Assume that $\Gamma \vdash \mathtt{v} :^{\varphi} \mathtt{T}$ has a derivation in n steps. Then it has a derivation in most n steps where the last rule used is neither (TT-TINST) nor (TT-EINST). □

Proof: By induction on n. All but the cases for (TT-TINST) and (TT-EINST) either are direct or are simple applications of the induction hypothesis. Thus, assume that the derivation ends with (TT-EINST) (the case for (TT-TINST) is similar). Apply the induction hypothesis to the derivation of its premiss; this gives a derivation that concludes in a type with the shape $\forall\epsilon.\mathtt{T}'$ yet does not end with (TT-EINST) or (TT-TINST). It cannot end with (TT-VAR) either, because a variable is not a value. The only other rule that can conclude $\forall\epsilon.\mathtt{T}'$ is (TT-EGEN), so the whole derivation now must end with

$$\dfrac{\dfrac{\Gamma \vdash \mathtt{v} :^{\varphi} \mathtt{T}' \qquad \epsilon \notin \mathrm{fev}(\Gamma, \varphi)}{\Gamma \vdash \mathtt{v} :^{\varphi} \forall\epsilon.\mathtt{T}'}\,(\text{TT-EGEN})}{\Gamma \vdash \mathtt{v} :^{\varphi} \mathtt{T}}\,(\text{TT-EINST})$$

where $\mathtt{T} = [\epsilon \mapsto \varphi']\mathtt{T}'$ for some φ'.

Lemma 5.5.2 now gives a derivation of $[\epsilon \mapsto \varphi']\Gamma \vdash \mathtt{v} :^{[\epsilon\mapsto\varphi']\varphi} [\epsilon \mapsto \varphi']\mathtt{T}'$ in $n - 2$ steps. But since $\epsilon \notin \mathrm{fev}(\Gamma, \varphi)$, this conclusion is the same as $\Gamma \vdash \mathtt{v} :^{\varphi} \mathtt{T}$, so we can use that instead of the original derivation. Now apply the induction hypothesis to this new derivation. □

5.5.4 LEMMA [CANONICAL FORMS]:

1. if $\mathtt{v}$ is a value of type $\mathtt{bool}$, then $\mathtt{v} = \mathtt{bv}$;
2. if $\mathtt{v}$ is a value of type $(\mathtt{T}_1 \to^{\varphi} \mathtt{T}_2, \mathtt{p})$ then $\mathtt{v} = \langle \lambda \mathtt{x}.\mathtt{t} \rangle_{\rho}$;
3. if $\mathtt{v}$ is a value of type $(\Pi \rho.^{\varphi} \mathtt{T}, \mathtt{p})$ then $\mathtt{v} = \langle \lambda \rho.\mathtt{u} \rangle_{\mathtt{p}}$. □

Proof: Left as an exercise (★↛). □

We next prove some lemmas about effects. Firstly, one can always enlarge the effect attributed to a term and obtain a derivable typing. Secondly, if a value can be typed, then it can also be typed with an empty effect; that is, evaluation of values does not cause any observable effects.

5.5.5 LEMMA: If $\Gamma \vdash \mathtt{t} :^{\varphi} \mathtt{T}$ and $\varphi \subseteq \varphi'$, then $\Gamma \vdash \mathtt{t} :^{\varphi'} \mathtt{T}$. □

Proof: Straightforward induction on the typing derivation. □

5.5.6 LEMMA: Let $\mathtt{v}$ be a value. If $\Gamma \vdash \mathtt{v} :^{\varphi} \mathtt{T}$ then $\Gamma \vdash \mathtt{v} :^{\emptyset} \mathtt{T}$. □

Proof: From Lemma 5.5.3 we get a derivation of $\Gamma \vdash \mathtt{v} :^{\varphi} \mathtt{T}$ that ends with neither (TT-TINST) nor (TT-EINST). Therefore, the derivation must end with one of the typing rules for values. By inspecting each of the rules (TT-BOOL), (TT-CLOS), and (TT-RCLOS) for values we see that we can construct a derivation of $\Gamma \vdash \mathtt{v} :^{\emptyset} \mathtt{T}$ (by choosing the effect to be empty in each case). □

Having established these basic lemmas we can now prove the lemmas leading to the type soundness result.

5.5.7 LEMMA [SUBSTITUTION]: If $\Gamma, \mathtt{x}_1 : \mathtt{T}_1 \vdash \mathtt{t} :^{\varphi} \mathtt{T}$ and $\Gamma \vdash \mathtt{t}_1 :^{\emptyset} \mathtt{T}_1$, then $\Gamma \vdash [\mathtt{x}_1 \mapsto \mathtt{t}_1]\mathtt{t} :^{\varphi} \mathtt{T}$. □

Proof: By induction on the typing derivation for $\mathtt{t}$. The cases are all standard, except for (TT-VAR) where $\mathtt{t} = \mathtt{x}_1$. By (TT-VAR) itself, $\mathtt{T} = \mathtt{T}_1$ and since $[\mathtt{x}_1 \mapsto \mathtt{t}_1]\mathtt{x}_1 = \mathtt{t}_1$ the second assumption combined with Lemma 5.5.5 gives us the desired derivation. □

5.5.8 PROPOSITION [SUBJECT REDUCTION]: If $\Gamma \vdash \mathtt{t} :^{\varphi} \mathtt{T}$ and $\mathtt{t} \longrightarrow \mathtt{t}'$ then $\Gamma \vdash \mathtt{t}' :^{\varphi} \mathtt{T}$. □

Proof: By induction on the derivation of $\to$.

For the rules (RE-IFTRUE) and (RE-IFFALSE) the term $\mathtt{t}$ is typable by assumption and therefore, by Lemma 5.5.3, there exist derivations of typings

of the `then` and `else` branches with the same type and effect as the complete term, as desired.

For the rules (RE-IF), (RE-APP1), (RE-APP2), (RE-FIX), (RE-NEW), and (RE-RAPP) again by the lemma there is a derivation of the typing of the redex. Next, apply the induction hypothesis to the redex and reconstruct the derivation of the original type and effect using the derivation of the reduct.

The cases for rules (RE-CLOS) and (RE-RCLOS) are immediate.

For the rule (RE-BETA) we first observe that, by Lemma 5.5.3, there is a derivation of the typing where the last rule is (TT-APP). Therefore by the lemma again we have (1) a derivation of $\Gamma \vdash \langle \lambda x_1.t_{12} \rangle_\rho :^{\varphi_1} T_2 \to^{\varphi} T$ that ends with (TT-CLOS), and therefore a subderivation of $\Gamma, x_1 : T_2 \vdash t_{12} :^{\varphi} T$, and (2) a derivation of $\Gamma \vdash v_2 :^{\varphi_2} T_2$ and next a derivation of $\Gamma \vdash v_2 :^{\emptyset} T_2$ by Lemma 5.5.6. We can thus apply Lemma 5.5.7 to get $\Gamma \vdash [x_1 \mapsto v_2]t_{12} :^{\varphi} T$.

The case for the rule (RE-RBETA) is similar.

For the rule (RE-FIXBETA) we again use the lemma to get a derivation where the last rule is (TT-FIX), and thus we have a derivation $\mathcal{D}$ of $\Gamma, x : T \vdash v :^{\varphi} T$. Apply Lemma 5.5.6 to get $\Gamma, x : T \vdash v :^{\emptyset} T$ and use this and (TT-FIX) to construct a derivation $\mathcal{D}_1$ of $\Gamma \vdash \mathtt{fix}\ x.v :^{\emptyset} T$. Now Lemma 5.5.7 applied to $\mathcal{D}$ and $\mathcal{D}_1$ yields a derivation of $\Gamma \vdash [x \mapsto \mathtt{fix}\ x.v]v :^{\varphi} T$.

For the rule (RE-DEALLOC), by the lemma there is a derivation of $\Gamma \vdash v_1 :^{\varphi_1} T_1$ where $\rho \notin \mathrm{frv}(\Gamma, T_1)$, and thus from Lemma 5.5.6 we have $\Gamma \vdash v_1 :^{\emptyset} T_1$. Applying Lemma 5.5.2 we then get a derivation of $[\rho \mapsto \bullet]\Gamma \vdash [\rho \mapsto \bullet]v_1 :^{[\rho \mapsto \bullet]\emptyset} [\rho \mapsto \bullet]T_1$, that is $\Gamma \vdash [\rho \mapsto \bullet]v_1 :^{\emptyset} T_1$ (since $\rho \notin \mathrm{frv}(\Gamma, T_1)$). Now use Lemma 5.5.5 to recover the original effect φ. □

5.5.9 PROPOSITION [PROGRESS]: If $\emptyset \vdash t :^{\varphi} T$ and $\bullet \notin \varphi$, then either t is a value or there is some t' such that $t \longrightarrow t'$. □

Proof: By induction over the structure of the term t.

The case $t = x$ is impossible (since t is typable in the empty context).

The cases for values are immediate.

If t is abstraction (or a region abstraction) then t is not a value, and there is a reduction using (RE-CLOS) (or (RE-RCLOS)).

If t is an application $t_0\ t_1$ then by Lemma 5.5.3 the last rule applied is (TT-APP) and therefore we have typings of $\emptyset \vdash t_0 :^{\varphi} T_1 \to^{\varphi_2} T$ and $\emptyset \vdash t_1 :^{\varphi} T_1$. Applying the induction hypothesis to the first of these we have that either t_0 is a value or there exists some term t_0' such that $t_0 \longrightarrow t_0'$. In the former case, by the Canonical Forms property t_0 must have the form $\langle \lambda x_0.t_0' \rangle_\rho$. Furthermore, we can next apply the induction hypothesis to the term t_1 and conclude that either t_1 is a value or there exists some term t_1' such that $t_1 \longrightarrow t_1'$. In the former case, since both t_0 and t_1 are values, we can apply the (RE-BETA) rule to obtain a reduction. In the latter case (where

t_0 is a value, and t_1 reduces) we can apply the (RE-APP2) rule to also obtain a reduction. Finally, in the event that t_0 is not a value (but it reduces), we can apply (RE-APP1) to obtain yet another reduction.

The cases where the term t is a region abstraction application or a conditional are similar, but simpler.

If t is $\texttt{new}\ \rho.t_1$ we can again apply the induction hypothesis to t_1 and get that either t_1 is a value, in which case we get a reduction using (RE-DEALLOC), or reduces to another term t_1', in which case we get a reduction using (RE-NEW). □

5.5.10 THEOREM: If $\emptyset \vdash t :^{\emptyset} T$, then either (1) there is some value v such that $t \to^* v$ and $\emptyset \vdash t :^{\emptyset} T$, or (2) for each t' such that $t \to^* t'$ there is some t'' such that $t' \to^+ t''$. □

Proof: Straightforward consequence of Propositions 5.5.8 and 5.5.9. □

5.5.11 COROLLARY [TYPE SOUNDNESS]: If $\emptyset \vdash t :^{\emptyset} \texttt{bool}$, then it is not the case that $\mathrm{eval}_R(t) = \texttt{wrong}$. □

Extensions

The region-annotated language we have presented so far only has Booleans and functions, but it is straightforward to add most other common types of data to it. As an example, in Figure 5-9 we give the necessary rules for extending the system with lists. The proofs of the metaproperties (in particular type soundness and conditional correctness) carry through to this extended system without any changes to the existing cases.

Rule (TT-CONS) implies that when adding a new element in front of a list, it must have the same type as the elements already there. That is hardly surprising, but note that it implies that the region part of the type must also be the same. Thus, the different element of a list are always allocated in the same region and therefore will be deallocated at the same time. If a single element of the list turns out to have a long lifetime, the region type system propagates that long lifetime to all the other elements!

5.5.12 EXERCISE [RECOMMENDED, ★★↛]: Using Figure 5-9 as a guideline, write rules to extend the system with one or more of: Let bindings (a lambda abstraction allocates a closure on the heap, so a let binding cannot simply be simulated as a β-redex). Pairs and records. Sums and variants. General recursive types (equi- or iso-). Verify that the Conditional Correctness and Type Soundness theorems still hold for your rules. □

New syntactic forms

$$
\begin{array}{lcll}
t & ::= & \ldots & \textit{terms:} \\
 & & (t :: t)\ \texttt{at}\ p & \textit{list constructor} \\
 & & \texttt{case}\ t_0\ \texttt{of}\ \begin{array}{l}\texttt{nil}\Rightarrow t_1\\ (x::x')\Rightarrow t_2\end{array} & \textit{case on lists} \\
v & ::= & \ldots & \textit{values:} \\
 & & \langle v :: v \rangle_p & \textit{cons cell} \\
 & & \texttt{nil} & \textit{empty list} \\
T & ::= & \ldots & \textit{types:} \\
 & & (T\ \texttt{list}, p) & \textit{type of lists}
\end{array}
$$

New erasure rules

$$
\begin{array}{rcl}
\|\texttt{nil}\| & = & \texttt{nil} \\
\|(t_1 :: t_2)\ \texttt{at}\ p\| & = & \|t_1\| :: \|t_2\| \\
\|\langle t_1 :: t_2 \rangle_p\| & = & \|t_1\| :: \|t_2\| \\
\left\|\texttt{case}\ t_0\ \texttt{of}\ \begin{array}{l}\texttt{nil}\Rightarrow t_1\\ (x::x')\Rightarrow t_2\end{array}\right\| & = & \\
\multicolumn{3}{r}{\texttt{case}\ \|t_0\|\ \texttt{of}\ \begin{array}{l}\texttt{nil}\Rightarrow \|t_1\|\\ (x::x')\Rightarrow \|t_2\|\end{array}}
\end{array}
$$

New evaluation rules $\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{(t_1 :: t_2)\ \texttt{at}\ p \longrightarrow (t_1' :: t_2)\ \texttt{at}\ p} \quad \text{(E-Cons1)}$$

$$\frac{t_2 \longrightarrow t_2'}{(v_1 :: t_2)\ \texttt{at}\ p \longrightarrow (v_1 :: t_2')\ \texttt{at}\ p} \quad \text{(E-Cons2)}$$

$$(v_1 :: v_2)\ \texttt{at}\ \rho \longrightarrow \langle v_1 :: v_2 \rangle_\rho \quad \text{(E-ConsAlloc)}$$

$$\frac{t_0 \longrightarrow t_0'}{\texttt{case}\ t_0\ \texttt{of}\ \texttt{nil} \Rightarrow t_1 \mid (x :: x') \Rightarrow t_2 \longrightarrow \texttt{case}\ t_0'\ \texttt{of}\ \texttt{nil} \Rightarrow t_1 \mid (x :: x') \Rightarrow t_2} \quad \text{(E-Case)}$$

$$\texttt{case nil of nil} \Rightarrow t_1 \mid (x :: x') \Rightarrow t_2 \longrightarrow t_1 \quad \text{(E-CaseNil)}$$

$$\texttt{case}\ \langle v :: v' \rangle_\rho\ \texttt{of nil} \Rightarrow t_1 \mid (x :: x') \Rightarrow t_2 \longrightarrow [x' \mapsto v'][x \mapsto v]t_2 \quad \text{(E-CaseCons)}$$

New typing rules $\boxed{\Gamma \vdash t :^\varphi T}$

$$\Gamma \vdash \texttt{nil} :^\varphi (T\ \texttt{list}, p) \quad \text{(TT-Nil)}$$

$$\frac{\Gamma \vdash t_1 :^\varphi T \qquad \Gamma \vdash t_2 :^\varphi (T\ \texttt{list}, p) \qquad p \in \varphi}{\Gamma \vdash (t_1 :: t_2)\ \texttt{at}\ p :^\varphi (T\ \texttt{list}, p)} \quad \text{(TT-Cons)}$$

$$\frac{\Gamma \vdash v_1 :^\varphi T \qquad \Gamma \vdash v_2 :^\varphi (T\ \texttt{list}, p)}{\Gamma \vdash \langle v_1 :: v_2 \rangle_p :^\varphi (T\ \texttt{list}, p)} \quad \text{(TT-ConsCell)}$$

$$\frac{\Gamma \vdash t_0 :^\varphi T' \qquad T' = (T\ \texttt{list}, p) \qquad p \in \varphi \qquad \Gamma \vdash t_1 :^\varphi T'' \qquad \Gamma, x : T, x' : T' \vdash t_2 :^\varphi T''}{\Gamma \vdash \texttt{case}\ t_0\ \texttt{of}\ \begin{array}{l}\texttt{nil}\Rightarrow t_1\\ (x::x')\Rightarrow t_2\end{array} :^\varphi T''} \quad \text{(TT-Case)}$$

Figure 5-9: Extending the system with a list type

5.5.13 Exercise [★★★★]: References can be added easily to the type system with rules like

$$\frac{\Gamma \vdash t :^\varphi T \qquad p \in \varphi}{\Gamma \vdash \texttt{ref}\ t\ \texttt{at}\ p :^\varphi (T\ \texttt{ref}, p)} \quad \text{(TT-Ref)}$$

$$\frac{\Gamma \vdash t :^\varphi (T\ \texttt{ref}, p) \qquad p \in \varphi}{\Gamma \vdash\ !t :^\varphi T} \quad \text{(TT-Deref)}$$

$$\frac{\Gamma \vdash t :^\varphi (T\ \texttt{ref}, p) \qquad \Gamma \vdash t' :^\varphi T \qquad p \in \varphi}{\Gamma \vdash t := t' :^\varphi \texttt{unit}} \quad \text{(TT-Assign)}$$

Of course, these rules need to be combined with the usual value restriction

on type *and effect* polymorphism, as discussed on pages 335–336 in TAPL. Which extensions to the semantics and soundness proofs are necessary for proving these rules sound? □

5.5.13 SOLUTION: The reference operations would need a formal semantics, so we would have to extend the evaluation and typing judgments with stores and store typings as in chapter 13 of TAPL. But that would break the lexical scoping of region variables, on which the correct operation of rule (RE-DEALLOC) depends critically. So the entire semantic treatment of region allocation and deallocation needs to be reworked. How to do this can be seen in Calcagno et al. (2002).

The typing rules presented in the above exercise correspond exactly to the way updatable references are handled in the ML Kit. Observe that since the (TT-ASSIGN) rule demands equality between the type of the value stored in the reference and the type of the new value, the rule forces the two values to have the same lifetime. For long-lived references, such as those found with container-classes in object-oriented programs, this behavior is inadequate.

5.6 Region inference

So far, we have said a lot about the TT region type system and how region-annotated programs are supposed to be executed, but next to nothing about where the region annotations come from.

One easy answer would be, "why, the programmer wrote them" – but alas, this answer would not be easy for the programmer. Realistic programs usually need quite a lot of region abstractions and applications in order to distribute their data over several regions while still being TT-typeable, and it is not always obvious exactly where they should be put. While it just might be possible to *write* a nontrivial well-typed program in the region-annotated language, it would be quite impossible to *maintain* it.

Therefore, the idea of using a region type system to check the safety of region annotations goes hand in hand with the idea that the region annotations themselves are the product of an automatic compile-time analysis. The human programmer writes a program $\mathtt{t}$ in PCF_2, whereupon the compiler will construct a region-annotated program $\mathtt{t}'$ such that $\|\mathtt{t}'\| = \mathtt{t}$ and $\mathtt{t}'$ is well-typed in the TT system. This process is known as *region inference*, because Tofte and Talpin (1994) viewed it as kin to a type reconstruction ("type inference") problem.

5.6.1 EXERCISE [RECOMMENDED, ★]: One can easily formulate a "trivial" region inference: Just choose a single fixed ρ, annotate each lambda abstraction (and

other allocating expressions) in the input program with " `at` ρ", and then wrap the entire program in a single `new` construction. This evidently always produces a region-annotated program that erases to the input program, but will it always be TT-typeable? □

5.6.1 SOLUTION: No, only if the input program satisfies our syntactic restriction on the use of the `fix` operator, and is typeable in (region-free) *F* with recursion. If the input program is ill-typed, it can never be region annotated; a derivation of $\emptyset \vdash \mathtt{t} :^{\varphi} \mathtt{T}$ can be converted into a derivation of $\emptyset \vdash \|\mathtt{t}\| : \|\mathtt{T}\|$ in System *F* with recursion simply by erasing all of the region-related syntax. Such an erasure transforms each TT type rule into either a conventional polymorphic typing rule or the identity rule that concludes any judgment from itself.

Of course, the trivial region inference is worthless from a memory-management point of view. It produces a region-annotated program that never deallocates anything until the entire computation is finished. What we want is the opposite: Region annotations that deallocate data as soon as allowed by the region type system. Unfortunately it is not known whether "best possible region annotations" in this sense always exist, but there are good approximate solutions available.

The articles by Tofte and Talpin (1994, 1997) do not themselves present an algorithm for region inference, but the nondeterministic "region inference system" they present has evidently been constructed with an inference algorithm in mind. The inference algorithm was published by Tofte and Birkedal (1998).

Recall (p. 310) that the original TT type system restricts the use of region abstractions to the `letrec` construction, and demands that all variables with a $\Pi\rho.^{\varphi}\mathtt{T}$ type are fully instantiated as soon as they are mentioned. This restriction simplifies the region inference problem somewhat, but region-polymorphic recursion is still a source of trouble.

If we were to further restrict the intended target language such that region-polymorphic recursion was forbidden, region inference would be relatively easy. This further restriction would amount to moving the region abstraction (as well as the invisible effect generalization) outside the `fix` operator in our translation of the `letrec` construction. Region inference can proceed like the familiar Algorithm $\mathcal{W}$, except that the types that are unified include region annotations, and that subset constraints between effects inside the types are collected during the reconstruction. Once the entire right-hand-side of a let binding has been processed, the set constraints that do not (directly or indirectly) constrain effects in the context can be reexpressed as effect variables

and generalized.[5] `new` expressions can be inserted eagerly whenever a region variable used within a subexpression is found not to occur either in its type or the context.

5.6.2 EXERCISE [★★★★→↠]: Implement region inference without region-polymorphic recursion. See Tofte and Birkedal (1998, Section 5) for hints on doing unification of region-annotated types with subset constraints on effects present. □

The region inference without region-polymorphic recursion is believed to be complete in the sense that it always produces region annotations with gives each heap-allocated object the shortest possible lifetime allowed by *any* set of region annotations that conform to the TT type systems with the added constraints that region abstraction and effect generalization are used only in analogy with what let-polymorphism allows.

However, completeness relative to these constraints is not really an impressive property – experience has shown that region annotations that to not use region-polymorphic recursion rarely yield tight enough lifetimes to be useful for actual memory management. The region inference algorithm needs to handle region-polymorphic recursion at least to some extent.

Polymorphic recursion in types is well known to lead to an undecidable type reconstruction problem (see references on page 338 of TAPL), intuitively because it allows values with unboundedly large monotypes to be constructed at run time, which means that natural iteration-based reconstruction algorithms may end up in an infinite loop, constructing ever larger approximations for a function's type scheme. Region-polymorphic recursion creates a similar problem for the region inference: The basic shape of the type can be known in advance (from an ordinary let-polymorphic type reconstruction without regions), but the effects embedded in the region-annotated types may grow without bounds during the region inference, as may the number of region and effect abstractions in a type scheme.

The original (Tofte and Birkedal, 1998) region inference algorithm avoided this problem by heuristically omitting certain opportunities for region and effect abstractions such that the iterative computation of a fixpoint for the recursive function's type scheme can be guaranteed to terminate. The cost

5. In the published formulation of the algorithm, syntactic effect variables are used to represent the operands in the constraints. This leads to the peculiar feature of the original TT system that it insists that the latent effect in each function type includes a distinguished effect variable called its *handle*. The handle is used to represent the entire effect during region inference. However, it has no special role in the soundness and correctness proofs, which has deeply mystified many readers of the Tofte–Talpin papers – including the authors of this chapter – because those papers did not provide details about the inference.

was that completeness fails: Example programs can be constructed for which the region inference algorithm lead to region annotations that are not the best possible.

Later Birkedal and Tofte (2001) rephrased the algorithm in terms of constraint solving. This reworked algorithm seems to be complete in the sense that for any region-annotated term `t` that can be TT-typed under the let-polymorphism restriction, the inference algorithm's output on $\|\texttt{t}\|$ will have as least as good space behavior as `t`. However, Birkedal and Tofte prove only a weaker "restricted completeness" result; full completeness was not even conjectured until well after the article's publication.

Another restricted case with an easy region inference problem is known. It is when the input program can be typed with *first order types*, that is, such that neither argument nor return type for any function includes a function type itself. Then there is no reason to use effect polymorphism, and one never needs to generalize over region variables that apppear only in latent effects. (For since there is only one arrow in each type, such a variable could just as well have been discharged by a `new` within the lambda abstraction). These two facts lead to a bounded representation of the latent effect: We simply need to know for each of the `p` positions in the argument and return types whether the actual `p` is in the latent effect. That solves the termination problem, and with a bit of ingenuity one does not even need fixpoint iteration for finding the best type scheme.

This principle has been used to derive a region inference for an adaptation of the TT system for a Prolog dialect which is naturally first-order; see (Makholm, 2000, Chapter 10).

5.6.3 EXERCISE [★↛]: Why does effect polymorphism not make sense in a first-order program? □

5.7 More powerful models for region-based memory management

Unfortunately, even with region-polymorphic recursion the TT model is not quite strong enough to achieve reasonable object lifetimes. At fault is the very idea of `new` – that the lifetime of a region must coincide with the time it takes to execute some subexpression of the original program. To see how this is a problem, let us look at how the TT system treats the classic "Game of Life" example. The task is to simulate a cellular automaton for n generations, starting from a specified state. This is a typical case of iterative programming, and the problems we will discover are common for iterative programs in general.

The standard way of programming an iteration in a functional language is to use tail recursion:

```
let rec nextgen(g) = ⟨read g; create and return new generation⟩
let rec life(n,g) = if n=0 then g
                    else life(n-1,nextgen(g))
```

We shall leave the details of `nextgen` unspecified here and in the following discussion. Furthermore we make the simplifying assumption that a single region holds all the pieces of a generation description, and for the sake of the argument we shall assume that the iteration count `n` needs to be heap-allocated, too.

The ordinary TT region inference algorithm annotates the Game of Life example as follows:

```
letrec nextgen[ρ](g) = ⟨read g from ρ; create new gen. at ρ⟩
letrec life[ρn,ρg](n,g) =
       if n=0 then g
       else new ρ'n
            in life[ρ'n,ρg]((n-1) at ρ'n, nextgen[ρg](g))
```

There are two major problems here. First, the recursive call of `life` is not a tail call anymore because it takes some work to deallocate a region at the end of `new`. Therefore all the ρ'_n regions will pile up on the call stack during the iteration and be deallocated only when the final result has been found.

Second, the `nextgen` function is forced to construct its result in the same region that contains its input. This means that the program has a serious space leak: all the intermediate-generation data will be deallocated only when the result of the iteration is deallocated.

Both of these problems are caused by the fact that `new` aligns the lifetime of its region with the hierarchical expression evaluation. Several solutions for this have been proposed, but because their formal properties have not been as thoroughly explored as the TT calculus, we will only present them briefly.

5.7.1 Region resetting in The ML Kit

The ML Kit's region implementation (Birkedal et al., 1996; Tofte et al., 1998) is based on the original TT system. Its solution to the tail recursion problem is based on a concept of *resetting* a region, meaning that its entire contents are deallocated while the region itself continues existing.

After a TT region inference, a special *storage-mode analysis* that runs after region inference amends the region annotations to control resetting: Each

" at ρ" annotation gets replaced by either "atbot ρ", meaning first reset the region and then allocate the new object as the new oldest object in the region, or "attop ρ", meaning allocate without resetting the region.

With this system one can rewrite the original Life program as follows to obtain better region behavior:

```
let rec copy(g) = ⟨read g; make fresh copy⟩
let rec life'((n,g) as p)
        = if n=0 then p
          else life'(n-1,copy(nextgen(g)))
let rec life(p) = snd (life' (p))
```

where copy (whose body is omitted here for brevity) takes apart a generation description and constructs a fresh, identical copy. Region inference and storage-mode analysis will then produce the region annotations

```
letrec nextgen[ρ,ρ'](g) = ⟨read g from ρ; new gen. at ρ'⟩
letrec copy[ρ',ρ](g) = ⟨read g from ρ'; fresh copy atbot ρ⟩
letrec life'[ρn,ρg]((n,g) as p)
       = if n=0 then p
         else life'[ρn,ρg]((n-1) atbot ρn,
                           new ρg'
                           in copy[ρg',atbot ρg]
                              (nextgen[ρg,ρg'](g)))
letrec life[ρn,ρg](p) = snd (life'[ρn,ρg](p))
```

Letting life' return the innermost n along with the result forces region inference to place all of the ns in the same region ρ_n. A memory leak in ρ_n is prevented by the atbot allocation, whose effect is that the region ρ_n is reset prior to placing n-1 in it.

The memory leak in ρ_g is prevented with the introduction of the copy function. Now the new generation can be constructed in a temporary region ρ_g' that gets deallocated before the recursive call; once the old generation is not needed anymore, the new generation is copied into ρ_g with the atbot mode which frees the old generation. (The atbot annotation in the passing of the region parameter serves to allow copy to actually reset the region; the need for this extra annotation has to do with aliasing between region variables.)

The storage-mode analysis works by changing the region annotation for an allocation to atbot if the value to be allocated is the only *live* value whose type includes the region name, as determined by a simple local liveness analysis. Neither a formal definition of the storage-mode analysis nor a proof that

it is safe has appeared in the literature, but it is described briefly by Birkedal et al. (1996), together with a number of other analyses that the ML Kit uses to implement the region model efficiently.

This solution does make it possible for iterative computations to run in constant space (assuming, in the Life example, that the size of a single `g` is bounded), but it is by no means obvious that precisely these changes to the original program would improve the space behavior. Furthermore, inserting such region optimizations in the program impede maintainability because they obscure the intended algorithm.

5.7.2 Aiken–Fähndrich–Levien's analysis for early deallocation

Aiken et al. (1995) extend the TT system in another direction, decoupling dynamic region allocation and deallocation from the introduction of region variables with the `new` construct.

In the AFL system, entry into a `new` block introduces a region variable, but does not allocate a region for it. During evaluation of the body of `new`, a region variable goes through precisely three states: unallocated, allocated, and finally deallocated. After a TT region inference (and possibly also storage-mode analysis as in the ML Kit), a constraint-based analysis – guided by a higher-order data-flow analysis for region variables – is used to insert explicit region allocation $[\![\texttt{alloc}\ \rho]\!]$ and deallocation commands $[\![\texttt{free}\ \rho]\!]$ into the program. Ideally the $[\![\texttt{alloc}\ \rho]\!]$ happens right before the first allocation in the region, and $[\![\texttt{free}\ \rho]\!]$ just after the last read from the region, but sometimes they need to be pushed farther away from the ideal placements, because the same region annotations on a function body must match all call sites.

With this system the Life example can be improved by rewriting the original program to

```
let rec copy(g) = ⟨read g; make fresh copy⟩
let rec life(n,g) = if n=0 then copy(g)
                    else life(n-1,nextgen(g))
```

where the only difference from the original program is that the base case returns a fresh copy of its input rather than the input itself. This program is analyzed as[6]

```
letrec nextgen[ρ,ρ'](g)
       = [[alloc ρ']] ⟨read g from ρ; new gen. at ρ'⟩ [[free ρ]]
```

6. The syntax here is not identical with the one used by Aiken et al. (1995); for example, they write "`free_after` ρ `t`" for what we write as "`t` $[\![\texttt{free}\ \rho]\!]$".

```
letrec copy[ρ, ρ′](g)
       = [[alloc ρ′]] ⟨read g from ρ; fresh copy at ρ′⟩ [[free ρ]]
letrec life[ρn, ρg, ρ′](n, g)
       = if n=0
         then [[free ρn]] copy[ρg, ρ′](g)
         else new ρ′n, ρ′g
              in life[ρ′n, ρ′g, ρ′]
                  ([[alloc ρ′n]] (n-1) at ρ′n [[free ρn]],
                   nextgen[ρg, ρ′g](g))
```

Because deallocation of each region is done explicitly and not by `new`, the body of `new` *is* a tail call context, and the regions containing the old `n` and `g` can be freed as soon as `n-1` and `nextgen(g)` have been computed. Without rewriting the original program this would not be the case, because a function must either *always* free one of its input regions or *never* do it.

5.7.3 Imperative regions: the Henglein–Makholm–Niss calculus

Recently Henglein et al. (2001) published a region system that completely severs the connection between region lifetimes and expression structures by eliminating the `new` construct. Instead, the region annotations form an imperative sublanguage for manipulating region handles asynchronously with respect to the expression structure.

The HMN system does not, as the two previously sketched solutions, build on top of the TT system and its region inference algorithm; instead it has its own region type system (proved sound by Niss (2002)) and inference algorithem (to appear in Makholm's Ph.D. thesis, hopefully before ATTAPL is finished). Starting anew in this fasion means that the system is conceptually simpler while still incorporating the essential features of ML Kit-like resetting and AFL-style early deallocation as special cases. On the other hand, the theory has not yet been extended to higher-order functions.

In the HMN system it is possible to handle the Game of Life with no rewriting at all. A function can pass regions as output (indicated by `o:` below) as well as receive them as input (indicated by `i:`); HMN region inference produces

```
letrec nextgen[i: ρ; o: ρ′](g)
       = [[new ρ′]] ⟨read g from ρ; new gen. at ρ′⟩ [[release ρ]]
letrec life[i: ρn, ρg; o: ρ′](n, g)
       = if n=0 then [[release ρn]] g [[ρ′ := ρg]]
                else life[i: ρ′n, ρ′g; o: ρ′]
                     ([[new ρ′n]] (n-1) at ρ′n [[release ρn]],
```

```
nextgen[i: ρg; o: ρ'g](g))
```

where each iteration of `life` decides for itself whether to release the region it gets as its second parameter or to return it back to the caller.

The $[\![\rho' := \rho_g]\!]$ operation serves the same purpose as the `copy` operation in the AFL solution, but is very cheap at runtime - it just renames the region that was previously called ρ_g to ρ', whereupon it is returned to the caller.

The renaming of regions means that the region-annotated types of values can change during the execution of the program. To manage that, the HMN region type system is based around a typing judgement with the shape

$$\Psi \vdash \{\Delta_1; \Gamma_1\}\ \mathtt{t} : \mathtt{T}\ \{\Delta_2; \Gamma_2\}$$

where the contexts Γ_1 and Γ_2 describe the types of the local variables before and after `t` is evaluated. The sets of region variables Ψ, Δ_1 and Δ_2 describe the available regions variables.

Other advanced features of the HMN system include reference-counted regions with a linear type discipline for region handles, "constant" region parameters that correspond to TT's region abstraction, and a subtyping discipline for regions that allows extensive manipulation of dangling pointers.

5.7.4 Other models

A number of more powerful region models and associated region type systems have been proposed without an accompanying inference algorithm.

Walker et al. (2000b) have developed a region model with a region type-system for a continuation-passing style language, intended to be used for translating TT-style region-based programs to certified machine code. To handle the CPS transformation of region abstractions, a very advanced type system with bounded quantification over regions and effects was necessary. The final system is much stronger than the TT system itself, but little is known about how to make an automatic region inference utilize this extra strength.

Walker and Watkins (2001b) have developed a region type system in which region references can be stored in data structures such as list. They are still not completely first-class, because they must have linear types (see Chapter 6), but the system is strong enough to reason about *heterogenous* lists (i.e., lists whose elements are allocated in different regions).

Another, more restricted, way of allowing heterogenous structures is found in the the Cyclone system. It employs a kind of *subtyping* on region lifetimes and a simpler notion of effects: A region variable ρ *outlives* another region variable ρ' if the lifetime of ρ encompasses the lifetime of ρ'. In that event, a value allocated in the region denoted by ρ can safely be used instead of the

same value allocated in the region denoted by ρ'. Cyclone supports coercion of values according to this principle.

5.8 Practical region-based memory management systems

5.8.1 The ML Kit

The ML Kit (`http://www.it-c.dk/research/mlkit`) essentially implements the theory described in §5.4 to §5.6 with two important extensions: firstly it includes region resetting and a storage-mode analysis as already described in §5.7, and secondly it includes a multiplicity inference allowing the compiler to allocate finite regions on the runtime stack Birkedal et al. (1996). The *multiplicity analysis* is a type-based analysis that determines, for all regions, whether they are finite or infinite. A *finite region* is a region into which the analysis can determine that there will only ever be written one value; all other regions are *infinite*. The importance of finite regions is that they can be stack-allocated since it is known in advance how large they are. Furthermore, since regions in the Tofte and Talpin region language follow a stack-discipline aligned with the expression structure of the program, such regions can even be allocated on the normal runtime stack giving a particularly simple and efficient implementation. The latest incarnation of the ML Kit even includes a garbage collector Hallenberg et al. (2002) suitable in situations where it is not practical to make a program more region friendly. See Tofte et al. (2001) for a comprehensive introduction to programming with regions in the ML Kit, and Tofte et al. (2003) for an excellent survey of the interplay between theory and practice in the context of the ML Kit.

5.8.2 Cyclone

Cyclone (`http://www.cs.cornell.edu/projects/cyclone/`) is a dialect of C that is designed to prevent safety violations. It uses regions both as a memory management discipline and as a way to guarantee safety (through a type soundness result). Cyclone includes three kinds of regions: a single *global* (or "heap") region; *stack* regions (corresponding to stack frames allocated from statement blocks); and *dynamic* regions (corresponding to the lexically scoped regions we have seen in the present chapter).

Instead of having effect variables as in the Tofte and Talpin system, Cyclone uses an operator on types (with no operational significance) called "regions of". The `regions_of` operator represents the region variables that occur free in a type; the crucial trick is that the regions of operator applied to a type variable is simply left abstract until the type variable is instantiated.

Intuitively, instead of propagating the effect of functional arguments via effect variables, they are propagated via the `regions_of` operator. Returning to the `map` example on page 317 we get the following type for `map`:

$$\forall\alpha,\beta.(\alpha \to \beta) \times (\alpha\ \texttt{list}, \rho) \to^{\{\rho,\rho'\}\cup\texttt{regions\_of}(\alpha\to\beta)} (\beta\ \texttt{list}, \rho').$$

For practical reasons a major aspect in the design of Cyclone was to make it easy for C programmers to write Cyclone applications and to port legacy C code to Cyclone. In particular, requiring programmers to write region-annotations as seen in the present chapter is out of the question. Cyclone addresses this by combining inference of region annotations with defaults that work in many cases.

Cyclone began as a compiler for producing *typed assembly language* (see Chapter 8) and as such can be seen as one way to realize *proof-carrying code* (see Chapter 7). The region aspects of Cyclone are described in (Grossman et al., 2002b), a systems overview is given in (Jim et al., 2002).

5.8.3 Other systems

The ML Kit and Cyclone are both mature systems. A number of research prototypes demonstrating various principles for region-based memory management has been described in the literature.

One trend has been to adapt Tofte and Talpin's system (and its spirit) to other languages. Velschow and Christensen (1998) describes *RegJava* which is a simple, region-annotated core subset of Java and an accompanying implementation. Makholm and Sagonas (2002) extend a Prolog compiler with region-based memory management and a region inference based on Henglein et al. (2001).

Another trend has been to experiment with the fundamental assumption that regions should be allocated and deallocated according to a stack-discipline. In this direction the present authors have constructed a prototype implementation of the system of Henglein et al. (2001) for a small functional language with function pointers (but not lexical closures containing free variables). Cyclone can also be seen as the practical realization of some of the ideas in the Calculus of Capabilities Walker et al. (2000b),

Finally, region-based memory management without the guarantees of memory safety offered by region type systems have a long history. The basic idea, of gaining extra efficiency by bulk allocations and deallocations, is certainly natural. Systems using region-like abstractions for their memory management dates as far back as 1967 Ross (1967); Schwartz (1975); Hanson (1990). In contrast to these special purpose region abstractions, the GNU C Library

provides an abstraction, called *obstacks*, to application programmers GNU (2001).

Also in this line of work is Gay and Aiken's *RC* compiler translating region annotated C programs to ordinary C programs with library support for regions Gay and Aiken (2001). At runtime, each region is equipped with a reference count keeping track of the number of (external) references to objects in the region. The operation for deleting a region can then flag instances that attempt to delete a region with non-zero reference count. A type system provides the compiler the opportunity to remove some of the reference count operations (but it does not guarantee memory safety as the type systems discussed in this chapter does).

6 *Substructural Type Systems*

By David Walker

Advanced type systems make it possible to restrict access to data structures and to limit the use of newly-defined operations. Oftentimes, this sort of access control is achieved through the definition of new abstract types under control of a particular module. For example, consider the following simplified file system interface.

```
type file

val open   : string → file option
val read   : file → string * file
val append : file * string → file
val write  : file * string → file
val close  : file → unit
```

By declaring that the type `file` is abstract, the implementer of the module can maintain strict control over the representation of files. A client has no way to accidentally (or maliciously) alter any of the file's representation invariants. Consequently, the implementer may assume that the invariants that he or she establishes upon opening a file hold before any `read`, `append`, `write` or `close`.

While abstract types are a powerful means of controlling the structure of data, they are not sufficient to limit the *ordering* and *number of uses* of functions in an interface. Try as we might, there is no way to prevent a file from being read after it has been closed. Likewise, we cannot stop a client from closing a file twice or forgetting to close file at all before terminating their program.

This chapter introduces *substructural* type systems that augment standard type abstraction mechanisms with the ability to control the number and order of uses of a data structure or operation. Substructural type systems are

particularly useful for constraining interfaces that provide access to system resources such as files, locks and memory. Each of these resources undergoes a series of changes of state throughout its lifetime. Files, as we have seen, may be open or closed; locks may be held or not; and memory may be allocated or deallocated. Substructural type systems provide sound static mechanisms for keeping track of just these sorts of state changes and prevent operations on objects in an invalid state.

The bulk of this chapter will focus on applications of substructural type systems to the control of memory resources. Memory is a pervasive resource that must be managed carefully in any programming system so it makes an excellent target of study. However, the general principles that we establish can be applied to other sorts of resources as well.

6.1 Structural Properties

All of the type systems we have seen so far in this book allow *unrestricted* use of variables in the type checking context. For instance, each variable may be used once, twice, three times, or not at all. It turns out that this is not just an idle observation: A precise analysis of the properties of such variables will suggest a whole new collection of type systems.

To begin our exploration, we will analyze the simply-typed lambda calculus, which is reviewed in Figure 6-1. In this discussion, we are going to be particularly careful when it comes to the form of the type-checking context Γ. We will consider such contexts to be simple lists of variable-type pairs. The "," operator appends a pair to the end of the list. We also write (Γ_1, Γ_2) for the list that results from appending Γ_2 onto the end of Γ_1. As usual, we allow a given variable to appear at most once in a context and to maintain this invariant, we implicitly alpha-convert bound variables before entering them into the context.

We are now in position to consider three basic *structural* properties satisfied by our simply-typed lambda calculus. The first property, *exchange*, indicates that the order in which we write down variables in the context is irrelevant. A corollary of exchange is that if we can type check a term with the context Γ, then we can type check that term with any permutation of the variables in Γ. The second property, *weakening*, indicates that adding extra, unneeded assumptions to the context, does not prevent a term from type checking. Finally, the third property, *contraction*, states that if we can type check a term using two identical assumptions ($\mathtt{x}_2 : \mathtt{T}_1$ and $\mathtt{x}_3 : \mathtt{T}_1$) then we can check the same term using a single assumption.

6.1.1 LEMMA [EXCHANGE]: If $\Gamma_1, \mathtt{x}_1 : \mathtt{T}_1, \mathtt{x}_2 : \mathtt{T}_2, \Gamma_2 \vdash \mathtt{t} : \mathtt{T}$ then

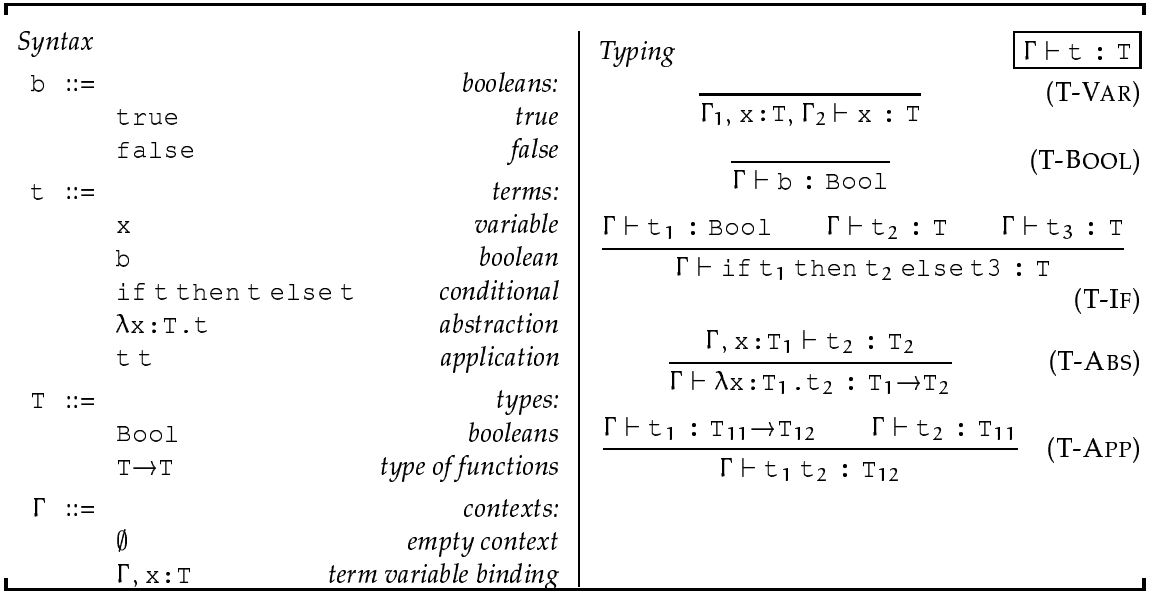


Figure 6-1: Simply-typed lambda calculus with booleans

$\Gamma_1, x_2:T_2, x_1:T_1, \Gamma_2 \vdash t : T$ □

6.1.2 LEMMA [WEAKENING]: If $\Gamma \vdash t : T$ then $\Gamma, x_1:T_1 \vdash t : T$ □

6.1.3 LEMMA [CONTRACTION]: If $\Gamma, x_2:T_1, x_3:T_1 \vdash t : T_3$ then $\Gamma, x_1:T_1 \vdash [x_2 \mapsto x_1][x_3 \mapsto x_1]t : T_3$ □

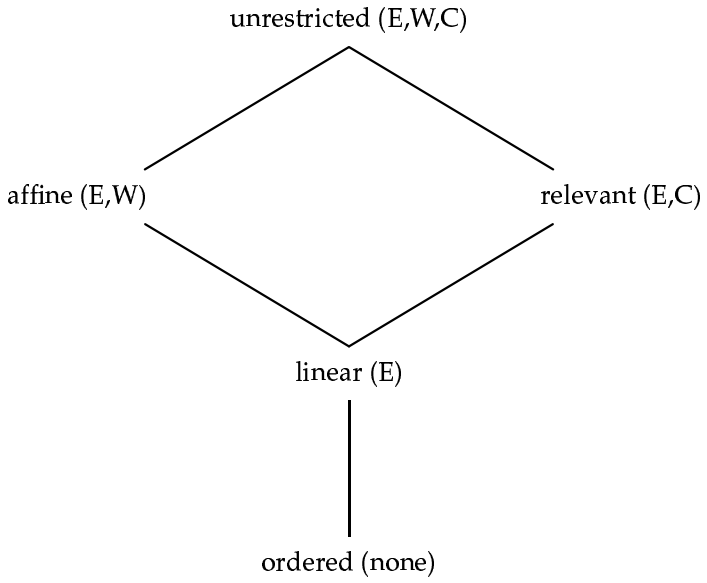
6.1.4 EXERCISE [★,RECOMMENDED]: Prove exchange, weakening and contraction hold for the simply-typed lambda calculus. □

A *substructural type system* is any type system that is designed so that one or more of the structural properties do not hold. Different substructural type systems arise when different properties are withheld.

- *Linear* type systems ensure that every variable is used exactly once by allowing exchange but not weakening or contraction.
- *Affine* type systems ensure that every variable is used at most once by allowing exchange and weakening, but not contraction.
- *Relevant* type systems ensure that every variable is used at least once by allowing exchange and contraction, but not weakening.

- *Ordered* type systems ensure that every variable is used exactly once and is used in the order in which it is introduced. Ordered type systems do not allow any of the structural properties.

The picture below can serve as a mnemonic for the relationship between these systems. The system at the bottom of the diagram (the ordered type system) admits no structural properties. As we proceed upwards in the diagram, we add structural properties: E stands for exchange; W stands for weakening; and C stands for contraction. It might be possible to define type systems containing other combinations of structural properties, such as contraction only or weakening only, but so far researchers have not found applications for such combinations. Consequently, we have excluded them from the diagram.



The diagram can be realized as a relation between the systems. We say system q_1 is more restrictive than system q_2 and write $q_1 \sqsubseteq q_2$ when system q_1 exhibits fewer structural rules than system q_2. Figure 6-2 specifies the relation, which we will find useful in the coming sections of this chapter.

$q ::=$		*system:*	
	`ord`	*ordered*	$\texttt{ord} \sqsubseteq \texttt{lin}$ (Q-OrdLin)
	`lin`	*linear*	$\texttt{lin} \sqsubseteq \texttt{rel}$ (Q-LinRel)
	`rel`	*relevant*	$\texttt{lin} \sqsubseteq \texttt{aff}$ (Q-LinAff)
	`aff`	*affine*	$\texttt{rel} \sqsubseteq \texttt{un}$ (Q-RelUn)
	`un`	*unrestricted*	$\texttt{aff} \sqsubseteq \texttt{un}$ (Q-AffUn)
			$q \sqsubseteq q$ (Q-Reflex)
			$\dfrac{q_1 \sqsubseteq q_2 \quad q_2 \sqsubseteq q_3}{q_1 \sqsubseteq q_3}$ (Q-Trans)

Figure 6-2: A Relation between Substructural Type Systems

6.2 A Linear Type System

In order to safely deallocate data, we need to know that the data we deallocate is never used in the future. Unfortunately, we cannot, in general, deduce whether data will be used after execution passes a certain program point: The problem is clearly undecidable. However, there are a number of sound, but useful approximate solutions. One such solution may be implemented using a *linear type system*. Linear type systems ensure that objects are used exactly once, so it is completely obvious that after the use of an object, it may be safely deallocated.

Syntax

Figure 6-3 presents the syntax of our linear language, which is an extension of the simply-typed lambda calculus. The main addition to be aware of, at this point, are the type qualifiers `q` that annotate the introduction forms for all data structures. The linear qualifier (`lin`) indicates that the data structure in question will be *used* (i.e., appear in the appropriate elimination form) exactly once in the program. Operationally, we deallocate these linear values immediately after they are used. The unrestricted qualifier (`un`) indicates that the data structure behaves as in the standard simply-typed lambda calculus. In other words, unrestricted data can be used as many times as desired and its memory resources will be automatically recycled by some extra-linguistic mechanism (a conventional garbage collector).

Apart from the qualifiers, the only slightly unusual syntactic form is the elimination form for pairs. The term $\texttt{split}\ t_1\ \texttt{as}\ x, y\ \texttt{in}\ t_2$ projects the

Syntax

q ::=		*qualifiers:*
	lin	*linear*
	un	*unrestricted*
b ::=		*booleans:*
	true	*true*
	false	*false*
t ::=		*terms:*
	x	*variable*
	q b	*boolean*
	if t then t else t	*conditional*
	q <t,t>	*pair*
	split t as x,y in t	*split*
	q λx:T.t	*abstraction*
	t t	*application*
P ::=		*pretypes:*
	Bool	*booleans*
	T*T	*pairs*
	T→T	*functions*
T ::=		*types:*
	q P	*qualified pretype*
Γ ::=		*contexts:*
	∅	*empty context*
	Γ, x:T	*term variable binding*

Figure 6-3: Linear lambda calculus: Syntax

first and second components from the pair t_1 and calls them x and y in t_2. This split operation allows us to extract two components while only counting a single use of a pair.[1]

To avoid dealing with an unnecessarily heavy syntax, we adopt a couple abbreviations in our examples in this section. First, we omit all unrestricted qualifiers and only annotate programs with the linear ones. Second, we freely use n-ary tuples (triples, quadruples, unit, etc.) in addition to pairs and also allow multi-argument functions. The latter may be defined as single-argument functions that take linear pairs (triples, etc) as arguments and immediately split them upon entry to the function body. Third, we often use ML-style type declarations, value declarations and let expressions where convenient; they all have the obvious meanings.

Typing

In order to ensure that linear objects are used exactly once, our type system maintains two important invariants.

1. Linear variables are used exactly once along every control-flow path.

2. Unrestricted data structures may not contain linear data structures. More

1. Extracting two components using the more conventional projections $\pi_1\ t_1$ and $\pi_2\ t_1$ requires two uses of the pair t_1. While it is possible to arrange our language so that operations of this sort are available, the set-up is somewhat trickier, so we simply avoid it here.

Context Split $\boxed{\Gamma = \Gamma_1 \circ \Gamma_2}$

$$\emptyset = \emptyset \circ \emptyset \quad \text{(M-EMPTY)}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, \texttt{x:un P} = (\Gamma_1, \texttt{x:un P}) \circ (\Gamma_2, \texttt{x:un P})} \quad \text{(M-UN)}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, \texttt{x:lin P} = (\Gamma_1, \texttt{x:lin P}) \circ \Gamma_2} \quad \text{(M-LIN1)}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, \texttt{x:lin P} = \Gamma_1 \circ (\Gamma_2, \texttt{x:lin P})} \quad \text{(M-LIN2)}$$

Figure 6-4: Linear Context Splitting Rules

generally, data structures with less restrictive type may not contain data structures with more restrictive type.

To understand why these invariants are useful, consider what could happen if either invariant is broken. When considering the first invariant, assume we have constructed a function `use` that makes some use of its argument before deallocating it. Now, if we allow a linear variable (say `x`) to appear twice, a programmer might write `<use x,use x>`, or, slightly more deviously, `(λz.λy.<use z,use y>) x x`. In either case, the program ends up "using" `x` twice, causing the program to crash.

Now considered the second invariant and suppose we allow a linear data structure (call it `x`) to appear inside an unrestricted pair (`un <x, 3>`). We can get exactly the same effect as above by using the unrestricted data structure multiple times:

```
let z = un <x,3> in
split z as x1,_ in
split z as x2,_ in
<use x1,use x2>
```

Fortunately, our type system ensures that none of these situations can occur.

We maintain the first invariant through careful context management. When type checking terms with two or more subterms, we pass all of the unrestricted variables in the context to each subterm. However, we split the linear variables between the different subterms in such a way as to ensure each variable is used exactly once. Figure 6-4 defines a relation, $\Gamma = \Gamma_1 \circ \Gamma_2$, which describes how to split a single context in a rule conclusion (Γ) into two contexts (Γ_1 and Γ_2) that will be used to type different subterms in a rule premise.

To help check the second invariant, we define the predicate `q(T)` (and its extension to contexts `q(Γ)`) to express the types `T` that can appear in a `q`-qualified data structure. These containment rules state that linear data structures can hold objects with linear or unrestricted type, but unrestricted data structures can only hold objects with unrestricted type.

Typing $\boxed{\Gamma \vdash t : T}$

$$\frac{\mathtt{un}(\Gamma_1, \Gamma_2)}{\Gamma_1, x{:}T, \Gamma_2 \vdash x : T} \quad \text{(T-VAR)}$$

$$\frac{\mathtt{un}(\Gamma)}{\Gamma \vdash q\ b : q\ \mathtt{Bool}} \quad \text{(T-BOOL)}$$

$$\frac{\Gamma_1 \vdash t_1 : q\ \mathtt{Bool} \qquad \Gamma_2 \vdash t_2 : T \qquad \Gamma_2 \vdash t_3 : T}{\Gamma_1 \circ \Gamma_2 \vdash \mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3 : T} \quad \text{(T-IF)}$$

$$\frac{\Gamma_1 \vdash t_1 : T_1 \qquad \Gamma_2 \vdash t_2 : T_2 \qquad q(T_1) \qquad q(T_2)}{\Gamma_1 \circ \Gamma_2 \vdash q\ \langle t_1, t_2 \rangle : q\ (T_1 * T_2)} \quad \text{(T-PAIR)}$$

$$\frac{\Gamma_1 \vdash t_1 : q\ (T_1 * T_2) \qquad \Gamma_2, x{:}T_1, y{:}T_2 \vdash t_2 : T}{\Gamma_1 \circ \Gamma_2 \vdash \mathtt{split}\ t_1\ \mathtt{as}\ x, y\ \mathtt{in}\ t_2 : T} \quad \text{(T-SPLIT)}$$

$$\frac{q(\Gamma) \qquad \Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash q\ \lambda x{:}T_1.t_2 : q\ T_1 \to T_2} \quad \text{(T-ABS)}$$

$$\frac{\Gamma_1 \vdash t_1 : q\ T_{11} \to T_{12} \qquad \Gamma_2 \vdash t_2 : T_{11}}{\Gamma_1 \circ \Gamma_2 \vdash t_1\ t_2 : T_{12}} \quad \text{(T-APP)}$$

Figure 6-5: Linear lambda calculus: Typing

- $q(T)$ if and only if $T = q'\ P$ and $q \sqsubseteq q'$
- $q(\Gamma)$ if and only if $(x{:}T) \in \Gamma$ implies $q(T)$

Recall, we have already defined $q \sqsubseteq q'$ such that it is reflexive, transitive and $\mathtt{lin} \sqsubseteq \mathtt{un}$.

Now that we have defined the rules for containment and context splitting, we are ready for the typing rules proper, which appear in Figure 6-5. Keep in mind that these rules are constructed anticipating a call-by-value operational semantics.

It is often the case when designing a type system that the rules for the base cases, variables and constants, are hardly worth mentioning. However, in substructural type systems these base cases have a special role in defining the nature of the type system, and subtle changes can make all the difference. In our linear system, the base cases must ensure that no linear variable is discarded without being used. To enforce this invariant in rule (T-VAR), we explicitly check that Γ_1 and Γ_2 contain no linear variables using the condition $\mathtt{un}(\Gamma_1, \Gamma_2)$. We make a similar check in rule (T-BOOL). Notice also that rule (T-VAR) is written carefully to allow the variable x to appear anywhere in the context, rather than just at the beginning or at the end.

6.2.1 EXERCISE [★]: What is the effect of rewriting the variable rule as follows?

$$\frac{\mathtt{un}(\Gamma)}{\Gamma, x{:}T \vdash x : T} \quad \text{(T-BROKENVAR)}$$

□

The inductive cases of the typing relation take care to use context splitting to partition linear variables between various subterms. For instance, rule (T-IF) splits the incoming context into two parts, one which is used to check subterm $\mathtt{t}_1$ and the other which is used to check both $\mathtt{t}_2$ and $\mathtt{t}_3$. As a result, a particular linear variable will occur once in $\mathtt{t}_2$ and once in $\mathtt{t}_3$. However, the linear object bound to the variable in question will be used (and hence deallocated) exactly once at run time since only one of $\mathtt{t}_2$ or $\mathtt{t}_3$ will be executed.

The rules for creation of pairs and functions make use of the containment rules. In each case, the data structure's qualifier `q` is used in the premise of the typing rule to limit the sorts of objects it may contain. For example, in the rule (T-ABS), if the qualifier `q` is `un` then the variables in Γ, which will inhabit the function closure, must satisfy $\mathtt{un}\ (\Gamma)$. In other words, they must all have unrestricted type. If we omitted this constraint, we could write the following badly behaved functions.[2]

```
type T = un (un bool → lin bool)

val discard =
  lin λx:lin bool.
    (λf:T.lin true) (un λy:un bool.x)

val duplicate =
  lin λx:lin bool.
    (λf:T.lin <f (un true),f (un true)>)) (un λy:un bool.x)
```

The first function discards a linear argument `x` without using it and the second duplicates a linear argument and returns two copies of it in a pair. Hence, in the first case, we fail to deallocate `x` and in the second case, a subsequent function may project both elements of the pair and use `x` twice, which would result in a memory error as `x` would be deallocated immediately after the first use. Fortunately, the containment constraint disallows the linear variable `x` from appearing in the unrestricted function `(λy:bool. x)`.

Now that we have defined our type system, we should verify our intended structural properties: exchange for all variables, and weakening and contraction for unrestricted variables.

6.2.2 LEMMA [EXCHANGE]: If $\Gamma_1, \mathtt{x}_1:\mathtt{T}_1, \mathtt{x}_2:\mathtt{T}_2, \Gamma_2 \vdash \mathtt{t} : \mathtt{T}$ then $\Gamma_1, \mathtt{x}_2:\mathtt{T}_2, \mathtt{x}_1:\mathtt{T}_1, \Gamma_2 \vdash \mathtt{t} : \mathtt{T}$. □

2. For clarity, we have not omitted the unrestricted qualifiers in this example.

6.2.3 LEMMA [UNRESTRICTED WEAKENING]: If $\Gamma \vdash \mathtt{t} : \mathtt{T}$ then $\Gamma, \mathtt{x}_1 : \mathtt{un}\ \mathtt{P}_1 \vdash \mathtt{t} : \mathtt{T}$. □

6.2.4 LEMMA [UNRESTRICTED CONTRACTION]: If $\Gamma, \mathtt{x}_2 : \mathtt{un}\ \mathtt{P}_1, \mathtt{x}_3 : \mathtt{un}\ \mathtt{P}_1 \vdash \mathtt{t} : \mathtt{T}_3$ then $\Gamma, \mathtt{x}_1 : \mathtt{un}\ \mathtt{P}_1 \vdash [\mathtt{x}_2 \mapsto \mathtt{x}_1][\mathtt{x}_3 \mapsto \mathtt{x}_1]\mathtt{t} : \mathtt{T}_3$. □

Proof: The proofs of all three lemmas follow by induction on the structure of the appropriate typing derivation. □

Finally, any type system that does not satisfy a substitution property is dubious indeed. We should also verify this property before proceeding further.

6.2.5 LEMMA [LINEAR SUBSTITUTION]: Let $\Gamma_3 = \Gamma_1 \circ \Gamma_2$. If $\Gamma_1, \mathtt{x} : \mathtt{T} \vdash \mathtt{t}_1 : \mathtt{T}_1$ and $\Gamma_2 \vdash \mathtt{t} : \mathtt{T}$ then $\Gamma_3 \vdash [\mathtt{x} \mapsto \mathtt{t}]\mathtt{t}_1 : \mathtt{T}_1$. □

Proof: The proof follows by induction on the structure of the derivation $\Gamma_1, \mathtt{x} : \mathtt{T} \vdash \mathtt{t}_1 : \mathtt{T}_1$. □

Algorithmic Linear Type Checking

The inference rules provided in the previous subsection give a clear, concise specification of the linearly-typed programs. However, these rules are also highly non-deterministic and cannot be implemented directly. The primary difficulty is that to implement the non-deterministic splitting operation, $\Gamma = \Gamma_1 \circ \Gamma_2$, we must guess how to split an input context Γ into two parts. Fortunately, it is relatively straightforward to restructure the type checking rules to avoid having to make these guesses. This restructuring leads directly to a practical type checking algorithm.

The central idea is that rather than splitting the context into parts before checking a complex expression composed of several subexpressions, we can pass the entire context as an input to the first subexpression and have it return the unused portion as an output. This output may then be used to check the next subexpression, which may also return some unused portions of the context as an output, and so on. Figure 6-6 makes these ideas concrete. It defines a new algorithmic type checking judgment with the form $\Gamma_{in} \vdash \mathtt{t} : \mathtt{T}; \Gamma_{out}$, where Γ_{in} is the input context, some portion of which will be consumed during type checking of $\mathtt{t}$, and Γ_{out} is the output context, which will be synthesized alongside the type $\mathtt{T}$.

There are several key changes in our reformulated system. First, the base cases for variables and constants allow any context to pass through the judgment rather than restricting the number of linear variables that appear. In order to ensure that linear variables are used, we move these checks to the rules

Algorithmic Typing $\boxed{\Gamma_{in} \vdash t : T; \Gamma_{out}}$

$$\Gamma_1, x{:}\mathtt{un}\ P, \Gamma_2 \vdash x : \mathtt{un}\ P; \Gamma_1, x{:}\mathtt{un}\ P, \Gamma_2 \quad \text{(A-UVAR)}$$

$$\Gamma_1, x{:}\mathtt{lin}\ P, \Gamma_2 \vdash x : \mathtt{lin}\ P; \Gamma_1, \Gamma_2 \quad \text{(A-LVAR)}$$

$$\Gamma \vdash q\ b : q\ \mathtt{Bool}; \Gamma \quad \text{(A-BOOL)}$$

$$\frac{\Gamma_1 \vdash t_1 : q\ \mathtt{Bool}; \Gamma_2 \qquad \Gamma_2 \vdash t_2 : T; \Gamma_3 \qquad \Gamma_2 \vdash t_3 : T; \Gamma_3}{\Gamma_1 \vdash \mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3 : T; \Gamma_3} \quad \text{(A-IF)}$$

$$\frac{\Gamma_1 \vdash t_1 : T_1; \Gamma_2 \qquad \Gamma_2 \vdash t_2 : T_2; \Gamma_3 \qquad q(T_1) \qquad q(T_2)}{\Gamma_1 \vdash q\ \langle t_1, t_2\rangle : q\ (T_1 * T_2); \Gamma_3} \quad \text{(A-PAIR)}$$

$$\frac{\Gamma_1 \vdash t_1 : q\ (T_1 * T_2); \Gamma_2 \qquad \Gamma_2, x{:}T_1, y{:}T_2 \vdash t_2 : T; \Gamma_3}{\Gamma_1 \vdash \mathtt{split}\ t_1\ \mathtt{as}\ x, y\ \mathtt{in}\ t_2 : T; \Gamma_3 \div (x{:}T_1, y{:}T_2)} \quad \text{(A-SPLIT)}$$

$$\frac{q{=}\mathtt{un} \Rightarrow \Gamma_1 = \Gamma_2 \div (x{:}T_1) \qquad \Gamma_1, x{:}T_1 \vdash t_2 : T_2; \Gamma_2}{\Gamma_1 \vdash q\ \lambda x{:}T_1.t_2 : q\ T_1 \to T_2; \Gamma_2 \div (x{:}T_1)} \quad \text{(A-ABS)}$$

$$\frac{\Gamma_1 \vdash t_1 : q\ T_{11} \to T_{12}; \Gamma_2 \qquad \Gamma_2 \vdash t_2 : T_{11}; \Gamma_3}{\Gamma_1 \vdash t_1\ t_2 : T_{12}; \Gamma_3} \quad \text{(A-APP)}$$

Figure 6-6: Linear lambda calculus: Algorithmic Type Checking

where variables are introduced. For instance, consider the rule (A-SPLIT). The second premise has the form

$$\Gamma_2, x{:}T_1, y{:}T_2 \vdash t_2 : T; \Gamma_3$$

If T_1 and T_2 are linear, then they should be used in t_2, and should not appear in Γ_3. Conversely, T_1 and T_2 are unrestricted, then they will always appear in Γ_3, but we should delete them from the final outgoing context of the rule so that the ordinary scoping rules for the variables are enforced. To handle both the check that linear variables do not appear and the removal of unrestricted variables, we use a special "context difference" operator ($\div$). Using this operator, the final outgoing context of the rule (A-SPLIT) is defined to be $\Gamma_3 \div (x{:}T_1, y{:}T_2)$. Formally, context difference is defined as follows.

$$\Gamma \div \emptyset = \Gamma$$

$$\frac{\Gamma_1 \div \Gamma_2 = \Gamma_3 \qquad (x{:}\mathtt{lin}\ P) \notin \Gamma_3}{\Gamma_1 \div (\Gamma_2, x{:}\mathtt{lin}\ P) = \Gamma_3}$$

$$\frac{\Gamma_1 \div \Gamma_2 = \Gamma_3 \qquad \Gamma_3 = \Gamma_4, x{:}\mathtt{un}\ P, \Gamma_5}{\Gamma_1 \div (\Gamma_2, x{:}\mathtt{un}\ P) = \Gamma_4, \Gamma_5}$$

Notice that this operator is undefined when we attempt to take the difference of two contexts, Γ_1 and Γ_2, that contain bindings for the same linear variable ($x{:}\mathtt{lin}\ P$). If the undefined quotient $\Gamma_1 \div \Gamma_2$ were to appear any-

where in a typing rule, the rule itself would not be considered defined and could not be part of a valid typing derivation.

The rule for abstraction (A-ABS) also introduces a variable and hence it also uses context difference to manipulate the output context for the rule. Abstractions must also satisfy the appropriate containment conditions. In other words, rule (A-ABS) must check that unrestricted functions do not contain linear variables. We perform this last check by verifying that when the function qualifier is unrestricted, the input and output contexts from checking the function body are the same. This equivalence check is sufficient because if a linear variable was used in the body of an unrestricted function (and hence captured in the function closure), that linear variable would not show up in the outgoing context.

It is completely straightforward to check that every rule in our algorithmic system is syntax directed and that all our auxiliary functions including context membership tests and context difference are easily computable. Hence, we need only show that our algorithmic system is equivalent to the simpler and more elegant declarative system specified in the previous section. The proof of equivalence can be a broken down into the two standard components: *soundness* and *completeness* of the algorithmic system with respect to the declarative system. However, before we can get to the main results, we will need to show that our algorithmic system satisfies some basic structural properties of its own. In the following lemmas, we use the notation $\mathcal{L}(\Gamma)$ and $\mathcal{U}(\Gamma)$ to refer to the list of linear and unrestricted assumptions in Γ respectively.

6.2.6 LEMMA [ALGORITHMIC MONOTONICITY]: If $\Gamma \vdash \mathtt{t} : \mathtt{T}; \Gamma'$ then $\mathcal{U}(\Gamma') = \mathcal{U}(\Gamma)$ and $\mathcal{L}(\Gamma') \subseteq \mathcal{L}(\Gamma)$. □

6.2.7 LEMMA [ALGORITHMIC EXCHANGE]: If $\Gamma_1, \mathtt{x}_1 : \mathtt{T}_1, \mathtt{x}_2 : \mathtt{T}_2, \Gamma_2 \vdash \mathtt{t} : \mathtt{T}; \Gamma_3$ then $\Gamma_1, \mathtt{x}_2 : \mathtt{T}_2, \mathtt{x}_1 : \mathtt{T}_1, \Gamma_2 \vdash \mathtt{t} : \mathtt{T}; \Gamma_3'$ and Γ_3 is the same as Γ_3' up to transposition of the bindings for $\mathtt{x}_1$ and $\mathtt{x}_2$. □

6.2.8 LEMMA [ALGORITHMIC WEAKENING]: If $\Gamma \vdash \mathtt{t} : \mathtt{T}; \Gamma'$ then $\Gamma, \mathtt{x} : \mathtt{T}' \vdash \mathtt{t} : \mathtt{T}; \Gamma', \mathtt{x} : \mathtt{T}'$. □

6.2.9 LEMMA [ALGORITHMIC LINEAR STRENGTHENING]: If $\Gamma, \mathtt{x} : \mathtt{lin}\ \mathtt{P} \vdash \mathtt{t} : \mathtt{T}; \Gamma', \mathtt{x} : \mathtt{lin}\ \mathtt{P}$ then $\Gamma \vdash \mathtt{t} : \mathtt{T}; \Gamma'$. □

Each of these lemmas may be proven directly by induction on the initial typing derivation. The algorithmic system also satisfies a contraction lemma, but since it will not be necessary in the proofs of soundness and completeness, we have not stated it here.

w	::=	*prevalues:*	S	::=	*stores:*
	b	*boolean*		$\emptyset$	*empty context*
	<x,y>	*pair*		S, x $\mapsto$ v	*store binding*
	λx:T.t	*abstraction*	E	::=	*evaluation contexts:*
v	::=	*values:*		[ ]	*context hole*
	q w	*qualified prevalue*		if E then t else t	*if context*
				q <E,t>	*fst context*
				q <x,E>	*snd context*
				split E as x,y in t	*split context*
				E t	*fun context*
				x E	*arg context*

Figure 6-7: Linear lambda calculus: Run-time Data

6.2.10 THEOREM [ALGORITHMIC SOUNDNESS]: If $\Gamma_1 \vdash t : T; \Gamma_2$ and $\mathcal{L}(\Gamma_2) = \emptyset$ then $\Gamma_1 \vdash t : T$. □

Proof: As usual, the proof is by induction on the typing derivation. The structural lemmas we have just proven are required to push through the result, but it is mostly straightforward. □

6.2.11 THEOREM [ALGORITHMIC COMPLETENESS]: If $\Gamma_1 \vdash t : T$ then $\Gamma_1 \vdash t : T; \Gamma_2$ and $\mathcal{L}(\Gamma_2) = \emptyset$. □

Proof: The proof is by induction on the typing derivation. □

Operational Semantics

To make the memory management properties of our language clear, we will evaluate terms in an abstract machine with an explicit store. As indicated in Figure 6-7, stores are a sequence of variable-value pairs. We will implicitly assume that any variable appears at most once on the left-hand side of a pair so the sequence may be treated as a finite partial map.

A value is a pair of a qualifier together with some data (a *prevalue* w). For the sake of symmetry, we will also assume that all values are stored, even base types such as booleans. As a result, both components of any pair will be pointers (variables).

We define the operation of our abstract machine using a context-based, small-step semantics. Figure 6-8 defines the computational contexts E, which are terms with a single hole. Contexts define the order of evaluation of terms—

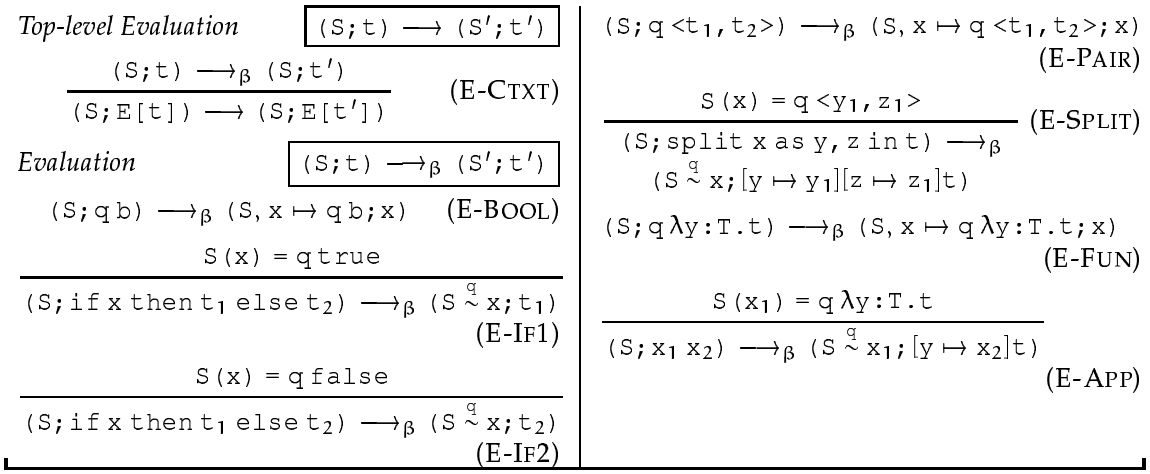


Figure 6-8: Linear lambda calculus: Operational semantics

they specify the places in a term where a computation can occur. In our case, evaluation is left-to-right since, for example, there is a context with the form `E t` indicating that we can reduce the term in the function position before reducing the term in the argument position. However, there is no context with the form `t E`. Instead, there is only the more limited context `x E`, indicating that we must reduce the term in the function position to a pointer `x` before proceeding to evaluate the term in the argument position. We use the notation `E[t]` to denote the term composed of the context `E` with its hole plugged by the computation `t`.

The operational semantics, defined in Figure **??**, is factored into two relations. The first relation, $(S;t) \longrightarrow (S';t')$, picks out a subcomputation to evaluate. The second relation, $(S;t) \longrightarrow_\beta (S';t')$, does all the real work. In order to avoid creation of two sets of operational rules, one for linear data, which is deallocated when used, and one for unrestricted data, which is never deallocated, we define an auxiliary function, $S \overset{q}{\sim} x$, to manage the differences.

$$\begin{array}{lcl} (S_1, x \mapsto v, S_2) \overset{lin}{\sim} x & = & S_1, S_2 \\ S \overset{un}{\sim} x & = & S \end{array}$$

Aside from these details, the operational semantics is standard.

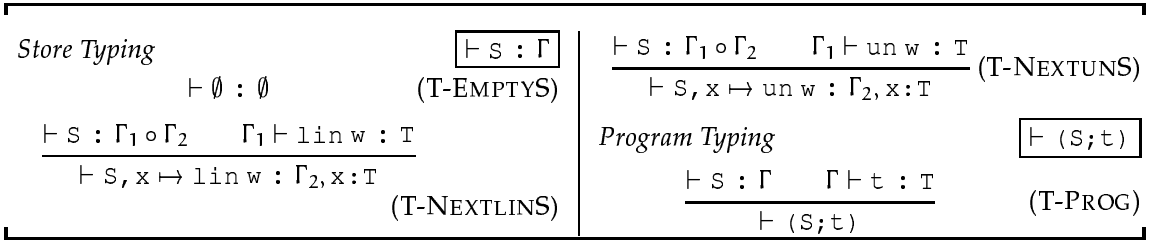


Figure 6-9: Linear lambda calculus: Program Typing

Preservation and Progress

In order to prove the standard safety properties for our language, we need to be able to show that programs are well-formed after each step in evaluation. Hence, we will define typing rules for our abstract machine. Since these typing rules are only necessary for the proof of soundness, and have no place in an implementation, we will extend the declarative typing rules rather than the algorithmic typing rules.

Figure 6-9 presents the machine typing rules in terms of two judgments, one for stores and the other for programs. The store typing rules generate a context that describes the available bindings in the store. The program typing rule uses the generated bindings to check the expression that will be executed.

With this new machinery in hand, we are able to prove the standard progress and preservation theorems.

6.2.12 THEOREM [PRESERVATION]: If $\vdash$ `(S;t)` and `(S;t)` $\longrightarrow$ `(S′;t′)` then $\vdash$ `(S′;t′)`. □

6.2.13 THEOREM [PROGRESS]: If $\vdash$ `(S;t)` then `(S;t)` $\longrightarrow$ `(S′;t′)` or `t` is a value. □

6.2.14 EXERCISE [★★,↠]: Prove progress and preservation using TAPL Chapters 9,13 as an approximate guide. □

6.3 Extensions and Variations

Most features found in modern programming languages can be defined to interoperate successfully with linear type systems, although some are trickier than others. In this section, we will consider a variety of practical extensions to our simple linear lambda calculus.

Sums and Recursive Types

Complex data structures, such as the recursive data types found in ML-like languages, pose little problem for linear languages. To demonstrate the central ideas involved, we extend the syntax for the linear lambda calculus with the standard introduction and elimination forms for sums and recursive types. The details are presented in Figure 6-10.

Values with sum type are introduced by injections `q inl`$_{\texttt{P}}$ `t` or `q inr`$_{\texttt{P}}$ `t`, where `P` is $\texttt{T}_1+\texttt{T}_2$, the resulting pretype of the term. In the first instance, the underlying term `t` must have type $\texttt{T}_1$, and in the second instance, the underlying term `t` must have type $\texttt{T}_2$. The qualifier `q` indicates the linearity of the argument in exactly the same way as for pairs. The case expression will execute its first branch if its primary argument is a left injection and it will execute its second branch if its primary argument is a right injection. We will assume that + binds more tightly that $\rightarrow$ but less tightly than `*`.

Recursive types are introduced with a `roll`$_{\texttt{P}}$ `t` expression, where `P` is the recursive pretype the expression will assume. Unlike all the other introduction forms, roll expressions are not annotated with a qualifier. Instead, they take on the qualifier of the underlying expression `t`. The reason for this distinction is that we will treat this introduction form as a typing coercion that has no real operational effect. Unlike functions, pairs or sums, recursive data types have no data of their own and therefore do not need a separate qualifier to control their allocation behavior. To simplify the notational overhead of sums and recursive types, we will normally omit the typing annotations on their introduction forms in our examples.

In order to write computations that process recursive types, we have added recursive function declarations to our language as well. Since the free variables in a recursive function closure will be used on each recursive invocation of the function, we cannot allow the closure to contain linear variables. Hence, all recursive functions are unrestricted data structures.

A simple, but useful data structure is the linear list of `T`s:

```
type T llist = rec a.lin (unit + lin (T * lin a))
```

Here, the entire spine (aside from the terminating value of unit type) is linear while the underlying `T` objects may be linear or unrestricted. To create a fully unrestricted list, we simply omit the linear qualifiers on the sum and pairs that make up the spine of the list:

```
type T list = rec a.unit + T * a
```

6.3.1 EXERCISE [★]: What is "wrong" with the following list type declaration. Does this example reveal an error in our language design?

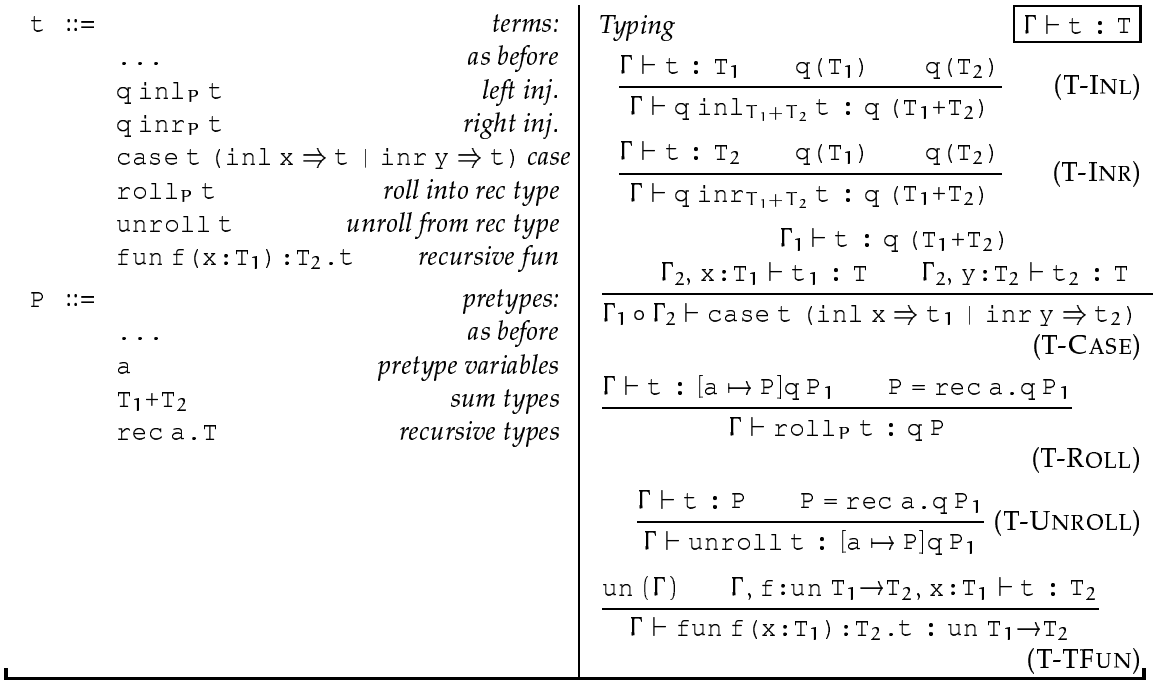


Figure 6-10: Linear lambda calculus: Sums and Recursive Types

```
type T badlist = rec a.lin (unit + T * a)
```

□

After defining the linear lists, the memory conscious programmer can write many familiar list-processing functions in a minimal amount of space. For example, here is how we map an unrestricted function across a linear list.

```
fun nil(_:unit) : T2 llist =
   roll (lin inl ())

fun cons(hd:T2, tl:T2 llist) : T2 llist =
   roll (lin inr (lin <hd,tl>))

fun map(f:T1→T2, xs:T1 llist) : T2 llist =
  case unroll xs (
```

```
    inl _  ⇒ nil()
  | inr xs ⇒
      split xs as hd,tl in
      cons(f hd,map lin <f,tl>))
```

In this implementation of `map`, we can observe that on each iteration of the loop, it is possible to reuse the space deallocated by `split` or `case` operations for the allocation operations that follow in the body of the function (inside the calls to `nil` and `cons`).

Hence, at first glance, it appears that `map` will execute with only a constant space overhead. Unfortunately, however, there are some hidden costs as `map` executes. A typical implementation will store local variables and temporaries on the stack before making a recursive call. In this case, the result of `f hd` will be stored on the stack while map iterates down the list. Consequently, rather than having a constant space overhead, our map implementation will have an $O(n)$ overhead, where n is the length of the list. This is not too bad, but we can do better.

In order to do better, we need to avoid implicit stack allocation of data each time we iterate through the body of a recursive function. Fortunately, most functional programming languages guarantee that if the last operation in a function is itself a function call then the language implementation will deallocate the current stack frame before calling the new function. We name such function calls *tail calls* and we say that any language implementation that guarantees that the current stack frame will be deallocated before a tail call is *tail-call optimizing*.

Assuming that our language is tail-call optimizing, we can now rewrite map so that it executes with only a constant space overhead. The main trick involved is that we will explicitly keep track of both the part of the input list we have yet to process and the ouput list that we have already processed. The output list will wind up in reverse order, so we will reverse it at the end. Both of the loops in the code, `mapRev` and `reverse` are *tail-recursive* functions. That is, they end in a tail call and have a space-efficient implementation.

```
fun map(f:T1→T2, input:T1 llist) : T2 llist =
  reverse(mapRev(f,input,nil()),nil())

and mapRev(f:T1→T2,
           input:T1 llist,
           output:T2 llist) : T2 llist =
  case unroll input (
    inl _  ⇒ output
  | inr xs ⇒
      split xs as hd,tl in
```

```
      mapRev (f,tl,cons(f hd,output)))

and reverse(input:T₂ llist, output:T₂ llist)
  case unroll input (
    inl _  ⇒ output
  | inr xs ⇒
      split xs as hd,tl in
      reverse(tl,cons(hd,output)))
```

This *link reversal* algorithm is a well-known way of traversing a list in constant space. It is just one of a class of algorithms developed well before the invention of linear types. A similar algorithm was invented by Deutsch, Schorr and Waite for traversing trees and graphs in constant space. Such constant space traversals are essential parts of mark-sweep garbage collectors—at garbage collection time there is no extra space for a stack so any traversal of the heap must be done in constant space.

6.3.2 EXERCISE [★★★]: Define a recursive type that describes linear binary trees that hold data of type `T` in their internal nodes (nothing at the leaves). Write a constant-space function `treeMap` that produces an identically-shaped tree on output as it was given on input, modulo the action of the function `f` that is applied to each element of the tree. Feel free to use reasonable extensions to our linear lambda calculus including mutually recursive functions, n-ary tuples and n-ary sums. □

Polymorphism

Parametric polymorphism is a crucial feature of almost any functional language, and our linear lambda calculus is no exception. The main function of polymorphism in our setting is to support two different sorts of code reuse.

1. Reuse of code to perform the same algorithm, but on data with different shapes.

2. Reuse of code to perform the same algorithm, but on data governed by different memory management strategies.

To support the first kind of polymorphism, we will allow quantification over pretypes. To support the second kind of polymorphism, we will allow quantification over qualifiers. A good example of both sorts of polymorphism arises in the definition of a polymorphic `map` function. In the code below, we use `a` and `b` to range over pretype variables as we did in the previous section, and `p` to range over qualifier variables.

q ::=		*qualifiers:*		q Λp.t	*qualifier abstraction*
	...	*as before*		t [q]	*qualifier application*
	p	*polymorphic qualifier*	P ::=		*pretypes:*
t ::=		*terms:*		...	*as before*
	...	*as before*		∀a.T	*pretype polymorphism*
	q Λa.t	*pretype abstraction*		∀p.T	*qualifier polymorphism*
	t [P]	*pretype application*			

Figure 6-11: Linear lambda calculus: Polymorphism Syntax

```
type (p1,p2,a) list =
  rec a.p1 (unit + p1 (p2 a * (p1,p2,a) list))

map :
  ∀a,b.
    ∀pa,pb.
      lin ((pa a → pb b)*(lin,pa,a) list)→(lin,pb,b) list
```

The type definition in the first line defines lists in terms of three parameters. The first parameter, p_1, gives the usage pattern (linear or unrestricted) for the spine of the list, while the second parameter gives the usage pattern for the elements of the list. The third parameter is a pretype parameter, which gives the (pre)type of the elements of list. The `map` function is polymorphic in the argument (`a`) and result (`b`) element types of the list. It is also polymorphic (via parameters p_a and p_b) in the way those elements are used. Overall, the function maps lists with linear spines to lists with linear spines.

Developing a system for polymorphic, linear type inference is a challenging research topic, beyond the scope of this book, so we will assume that, unlike in ML, polymorphic functions are introduced explicitly using the syntax Λa.t or Λp.t. Here, `a` or `p` are the type parameters to a function with body `t`. The body does not need to be a value, like in ML, since we will run the polymorphic function every time a pretype or qualifier is passed to the function as an argument. The syntax `t' [P]` or `t' [q]` applies the function `t'` to its pretype or qualifier argument. Figure 6-11 summarizes the syntactic extensions to the language.

Before we get to writing the `map` function, we will take a look at the polymorphic constructor functions for linear lists. These functions will take a pretype parameter and two qualifier parameters, just like the type definition for lists.

```
val nil : ∀a,p2.(lin,p2,a) list =
```

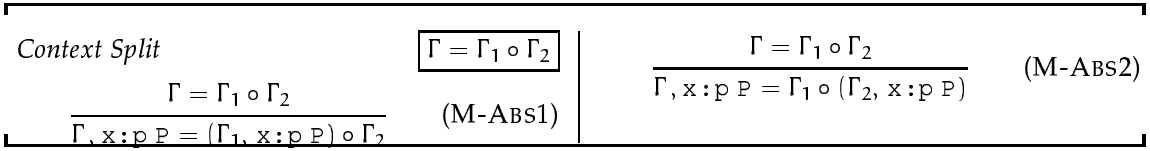


Figure 6-12: Linear Context Manipulation Rules

```
  Λa,p2.roll (lin inl ())

val list :
  ∀a,p2.lin (p2 a * (lin,p2,a) list)→(lin,p2,a) list =
  Λa,p2.
    λcell : lin (p2 a * (lin,p2,a) list).
      roll (lin inr (lin cell))
```

Now our most polymorphic `map` function may be written as follows.

```
val map =
  Λa,b.
  Λpa,pb.
  fun aux(f:(pa a → pb b),
          xs:(lin,pa,a) list)) : (lin,pb,b) list =
    case unroll xs (
      inl _  ⇒ nil [b,pb] ()
    | inr xs ⇒
        split xs as hd,tl in
         cons [b,pb] (pb <f hd,map (lin <f,tl>)>))
```

In order to ensure that our type system remains sound in the presence of pretype polymorphism, we will add the obvious typing rules, but change very little else. However, adding qualifier polymorphism, as we have done, is a little more involved. Before arriving at the typing rules themselves, we need to adapt some of our basic definitions to account for abstract qualifiers that may either be linear or unrestricted.

First, we need to ensure that we propagate contexts containing abstract qualifiers safely through the other typing rules in the system. Most importantly, we will add additional cases to the context manipulation rules defined in the previous section. We need to ensure that linear hypotheses are not duplicated and therefore we cannot risk duplicating unknown qualifiers, which might turn out to be linear. Figure 6-12 specifies the details.

Second, we need to conservatively extend the relation on type qualifiers $q_1 \sqsubseteq q_2$ so that it is sound in the presence of qualifier polymorphism. Since

$\Delta ::=$ *type contexts:*

$\emptyset$ *empty*

$\Delta, \mathtt{a}$ *pretype var.*

$\Delta, \mathtt{p}$ *qualifier var.*

Typing $\boxed{\Delta; \Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\mathtt{q}(\Gamma) \qquad \Delta, \mathtt{a}; \Gamma \vdash \mathtt{t} : \mathtt{T}}{\Delta; \Gamma \vdash \mathtt{q}\,\Lambda\mathtt{a}.\mathtt{t} : \mathtt{q}\,\forall\mathtt{a}.\mathtt{T}} \text{(T-PABS)}$$

$$\frac{\Delta; \Gamma \vdash \mathtt{t} : \mathtt{q}\,\forall\mathtt{a}.\mathtt{T} \qquad FV(\mathtt{P}) \subseteq \Delta}{\Delta; \Gamma \vdash \mathtt{t}\ [\mathtt{P}] : [\mathtt{a} \mapsto \mathtt{P}]\mathtt{T}} \text{(T-PAPP)}$$

$$\frac{\mathtt{q}(\Gamma) \qquad \Delta, \mathtt{p}; \Gamma \vdash \mathtt{t} : \mathtt{T}}{\Delta; \Gamma \vdash \mathtt{q}\,\Lambda\mathtt{p}.\mathtt{t} : \mathtt{q}\,\forall\mathtt{p}.\mathtt{T}} \text{(T-QABS)}$$

$$\frac{\Delta; \Gamma \vdash \mathtt{t} : \mathtt{q}_1\,\forall\mathtt{p}.\mathtt{T} \qquad FV(\mathtt{q}) \subseteq \Delta}{\Delta; \Gamma \vdash \mathtt{t}\ [\mathtt{q}] : [\mathtt{p} \mapsto \mathtt{q}]\mathtt{T}} \text{(T-QAPP)}$$

Figure 6-13: Linear lambda calculus: Polymorphic Typing

the linear qualifier is the least qualifier in the current system, the following rule should hold.

$$\mathtt{lin} \sqsubseteq \mathtt{p} \qquad \text{(Q-LINP)}$$

Likewise, since `un` is the greatest qualifier in the system, we can be sure the following rule is sound.

$$\mathtt{p} \sqsubseteq \mathtt{un} \qquad \text{(Q-PUN)}$$

Aside from these rules, we will only be able to infer that an abstract qualifier `p` is related to itself via the general reflexivity rule. Consequently, linear data structures can contain abstract ones; abstract data structures can contain unrestricted data structures; and data structure with qualifier `p` can contain other data with qualifier `p`.

In order to define the typing rules for the polymorphic linear lambda calculus proper, we will need to change the judgment form to keep track of the type variables that are allowed to appear free in a term. The new judgment uses the type context Δ for this purpose. The typing rules for the introduction and elimination forms for each sort of polymorphism are fairly straightforward now and are presented in Figure 6-13.

The typing rules for the other constructs we have seen will be almost unchanged. One relatively minor alteration is that the incoming type context Δ will be propagated through the rules to account for the free type variables. Unlike term variables, type variables can always be used in an unrestricted fashion; it is difficult to understand what it would mean to restrict the use of a type variable to one place in another type or term. Consequently, all parts of Δ are propagated from the conclusion of any rule to all premises. We will also need the occasional side condition to check that whenever a programmer writes down a type, its free variables are contained in the current type context Δ. For instance the rules for function abstraction and application will now be written as follows.

$$\begin{array}{ll} \texttt{E} ::= & \textit{evaluation contexts:} \\ \quad \texttt{E [P]} & \textit{pretype app context} \\ \quad \texttt{E [q]} & \textit{qualifier app context} \end{array}$$

$$\texttt{(S;q}\Lambda\texttt{a.t)} \longrightarrow_\beta \texttt{(S, x} \mapsto \texttt{q}\Lambda\texttt{a.t;x)} \quad \text{(E-PFun)}$$

$$\frac{\texttt{S(x) = q}\Lambda\texttt{a.t}}{\texttt{(S;x [P])} \longrightarrow_\beta \texttt{(S} \overset{\texttt{q}}{\sim} \texttt{x;[a} \mapsto \texttt{P]t)}} \quad \text{(E-PApp)}$$

$$\texttt{(S;q}\Lambda\texttt{p.t)} \longrightarrow_\beta \texttt{(S, x} \mapsto \texttt{q}\Lambda\texttt{p.t;x)} \quad \text{(E-QFun)}$$

$$\frac{\texttt{S(x) = q}\Lambda\texttt{p.t}}{\texttt{(S;x [q}_1\texttt{])} \longrightarrow_\beta \texttt{(S} \overset{\texttt{q}}{\sim} \texttt{x;[p} \mapsto \texttt{q}_1\texttt{]t)}} \quad \text{(E-QApp)}$$

Figure 6-14: Linear lambda calculus: Polymorphic Operational Semantics

$$\frac{\texttt{q}(\Gamma) \qquad FV(\texttt{T}_1) \subseteq \Delta \qquad \Delta;\Gamma, \texttt{x:T}_1 \vdash \texttt{t}_2 \texttt{ : T}_2}{\Delta;\Gamma \vdash \texttt{q}\,\lambda\texttt{x:T}_1\texttt{.t}_2 \texttt{ : q T}_1\texttt{}\rightarrow\texttt{T}_2} \quad \text{(T-Abs)}$$

$$\frac{\Delta;\Gamma_1 \vdash \texttt{t}_1 \texttt{ : q T}_1\texttt{}\rightarrow\texttt{T}_2 \qquad \Delta;\Gamma_2 \vdash \texttt{t}_2 \texttt{ : T}_1}{\Delta;\Gamma_1 \circ \Gamma_2 \vdash \texttt{t}_1\,\texttt{t}_2 \texttt{ : T}_2} \quad \text{(T-App)}$$

The most important check we need to do to test our system for faults is to prove the type substitution lemma. In particular, the proof will demonstrate that we have made safe assumptions about how abstract type qualifiers may be used.

6.3.3 Lemma [Type Substitution]: 1. If Δ, $\texttt{p};\Gamma \vdash \texttt{t} \texttt{ : T}$ and $FV(\texttt{q}) \in \Delta$ then $\Delta;[\texttt{p} \mapsto \texttt{q}]\Gamma \vdash [\texttt{p} \mapsto \texttt{q}]\texttt{t} \texttt{ : } [\texttt{p} \mapsto \texttt{q}]\texttt{T}$

2. If Δ, $\texttt{a};\Gamma \vdash \texttt{t} \texttt{ : T}$ and $FV(\texttt{P}) \in \Delta$ then $\Delta;[\texttt{a} \mapsto \texttt{P}]\Gamma \vdash [\texttt{a} \mapsto \texttt{P}]\texttt{t} \texttt{ : } [\texttt{a} \mapsto \texttt{P}]\texttt{T}$ □

6.3.4 Exercise [★]: Sketch the proof of the type substitution lemma. What structural rule(s) do you need to carry out the proof? □

Operationally, we will choose to implement polymorphic instantiation using substitution. As a result, our operational semantics changes very little. We only need to specify the new computational contexts and to add the evaluation rules for polymorphic functions and application as in Figure 6-14.

Arrays

Arrays pose a special problem for linearly typed languages. If we try to provide an operation fetches an element from an array in the usual way, perhaps using an array index expression `a[i]`, we would need to reflect the fact that

the $\texttt{i}^{\texttt{th}}$ element (and only the $\texttt{i}^{\texttt{th}}$ element) of the array had been "used." However, there is no simple way to reflect this change in the type of an array as the usual form of array types (`array(T)`) provides no mechanism to distinguish between the properties of different elements of the array.

We dodged this problem when we constructed our tuple operations by defining a pattern matching construct that simultaneously extracted all of the elements of a tuple. Unfortunately, we cannot follow the same path for arrays because in modern languages like Java and ML, the length of an array (and therefore the size of the pattern) is unknown at compile time.

Yet another non-solution to the problem is to add a special built-in iterator to process all the elements in an array at once. However, this last straw man proposal prevents programmers from using arrays as efficient, constant-time, random-access data structures; they might as well use lists instead.

One way out of this jam is to design the central array access operations so that, unlike the ordinary "get" and "set" operations, they *preserve* the number of pointers to the array and the number of pointers to each of its elements. We avoid our problem because there is no change to the array data structure that needs to be reflected in the type system. Using this idea, we will be able to allow programmers to define linear arrays that can hold a collection of arbitrarily many linear objects. Moreover, programmers will be able to access any of these linear objects, one at a time, using a convenient, constant-time, random-access mechanism.

So, what are the magic pointer-preserving array access operations? Actually, we need only one: a swap operation with the form `swap(a[i],t)`. The swap replaces the $\texttt{i}^{\texttt{th}}$ element of the array `a` (call it `t'`) with `t` and returns a (linear) pair containing the new array and `t'`. Notice the number of pointers to `t` and `t'` does not change during the operation. If there was one pointer to `t` (as an argument to `swap`) before the call, then there is one pointer to `t` afterward (from within the array `a`) and vice versa for `t'`. If, in addition, all of the elements of `a` had one pointer to them before the swap, then they will all have one pointer to them after the swap as well. Consequently, we will find it easy to type the swap operation, even when it works over linear arrays of linear objects.

In addition to swap, we provide functions to allocate an array given its list of elements (`array`), to determine array length (`length`) and to deallocate arrays (`free`). The last operation is somewhat unusual in that it takes two arguments `a` and `f`, where `a` is an array of type `lin array(T)` and `f` is a function with type `T→unit` that is run on each element of `T`. The function may be thought of as a finalizer for the elements; it may be used to deallocate any linear components of the array elements, thereby preserving the single pointer property.

Our definition of arrays is compatible with the polymorphic system from the previous subsection, but for simplicity, we present the formal syntax and semantics in the context of the simply-typed lambda calculus (see Figure 6-15).

6.3.5 EXERCISE [★,RECOMMENDED]: The typing rule for array allocation (T-ARRAY) contains the standard containment check to ensure that unrestricted arrays cannot contain linear objects. What kinds of errors can occur if this check is omitted? □

The `swap` and `free` functions are relatively low-level operations. Fortunately, it is easy to build more convenient, higher-level abstractions out of them. For instance, the following code defines some simple functions for manipulating linear matricies of unrestricted integers.

```
type iArray = lin array(int)
type matrix = lin array(iArray)

fun dummy(x:unit):iArray = lin array()

fun freeElem(x:int):unit = ()
fun freeArray(a:iArray):unit = free(a,freeElem)
fun freeMatrix(m:matrix):unit = free(m,freeArray)

fun get(a:matrix,i:int,j:int):lin (matrix * int) =
  split swap(a[i],dummy()) as a,b in
  split swap(b[j],0) as b,k in
  split swap(b[j],k) as b,_ in
  split swap(a[i],b) as a,junk in
  freeArray(junk);
  lin <a,k>

fun set(a:matrix,i:int,j:int,e:int):matrix =
  split swap(a[i],dummy()) as a,b in
  split swap(b[j],e) as b,_ in
  split swap(a[i],b) as a,junk in
  freeArray(junk);
  a
```

6.3.6 EXERCISE [★★,↛]: Use the functions provided above to write matrix-matrix multiply. Your multiply function should return an integer and deallocate both arrays in the process. Use any standard integer operations necessary. □

P ::=		*pretypes:*
	...	*as before*
	array(T)	*array pretypes*
t ::=		*terms:*
	...	*as before*
	q array(t,...,t)	*array creation*
	swap(t[t],t)	*swap*
	length(t)	*length*
	free(t,t)	*deallocate*
w ::=		*prevalues:*
	...	*as before*
	array[n,x,...,x]	*array*
E ::=		*evaluation contexts:*
	...	*as before*
	q array(v,...,v,E,t,...,t)	*array context*
	swap(E(t),t)	*swap context*
	swap(v(E),t)	*swap context*
	swap(v(v),E)	*swap context*
	length(E)	*length context*
	free(E,t)	*free context*
	free(v,E)	*free context*

Typing $\boxed{\Gamma \vdash t : T}$

$$\frac{q(T) \qquad \Gamma \vdash t_i : T \quad (\text{for } 1 \le i \le n)}{\Gamma \vdash q\ \mathtt{array}(t_1,\ldots,t_n) : q\ \mathtt{array}(T)} \text{ (T-ARRAY)}$$

$$\frac{\Gamma \vdash t_1 : q_1\ \mathtt{array}(T_1) \quad \Gamma \vdash t_2 : q_2\ \mathtt{int} \quad \Gamma \vdash t_3 : T_1}{\Gamma \vdash \mathtt{swap}(t_1[t_2],t_3) : \mathtt{lin}\,(q_1\ \mathtt{array}(T_1) * T_1)} \text{ (T-SWAP)}$$

$$\frac{\Gamma \vdash t : q\ \mathtt{array}(T)}{\Gamma \vdash \mathtt{length}(t) : \mathtt{lin}\,(q\ \mathtt{array}(T) * \mathtt{int})} \text{ (T-LENGTH)}$$

$$\frac{\Gamma \vdash t_1 : q\ \mathtt{array}(T) \qquad \Gamma \vdash t_2 : T \to \mathtt{unit}}{\Gamma \vdash \mathtt{free}(t_1,t_2) : \mathtt{unit}} \text{ (T-FREE)}$$

Evaluation $\boxed{(S;t) \longrightarrow_\beta (S';t')}$

$$(S; q\ \mathtt{array}(x_0,\ldots,x_{n-1})) \longrightarrow_\beta ((S, x \mapsto q\ \mathtt{array}[n,x_0,\ldots,x_{n-1}]; x) \text{ (E-ARRAY)}$$

$$\frac{S(x_i) = q_i\ j \qquad S = S_1, x_a \mapsto q\ \mathtt{array}[n,\ldots,x_j,\ldots], S_2 \qquad S' = S_1, x_a \mapsto q\ \mathtt{array}[n,\ldots,x_e,\ldots], S_2}{(S;\ \mathtt{swap}(x_a[x_i],x_e)) \longrightarrow_\beta (S' \overset{q_i}{\sim} x_i; \mathtt{lin}\,\langle x_a, x_j\rangle)} \text{ (E-SWAP)}$$

$$\frac{S(x) = q\ \mathtt{array}[n,x_0,\ldots,x_n]}{(S;\mathtt{length}(x)) \longrightarrow_\beta (S;\mathtt{lin}\,\langle x, \mathtt{un}\ n\rangle)} \text{ (E-LENGTH)}$$

$$\frac{S(x_a) = q\ \mathtt{array}[n,x_0,\ldots,x_n]}{(S;\mathtt{free}(x_a,x_f)) \longrightarrow_\beta (S \overset{q}{\sim} x_a; \mathtt{App}(x_f,x_0,\ldots,x_{n-1}))} \text{ (E-FREE)}$$

where

$$\mathtt{App}(x_f,\cdot) = ()$$
$$\mathtt{App}(x_f,x_1,\ldots) = x_f\ x_1; \mathtt{App}(x_f,\ldots)$$

Figure 6-15: Linear lambda calculus: Arrays

In the examples above, we needed some sort of dummy value to swap into an array to replace the value we wanted to extract. For integers and arrays it was easy to come up with a dummy value. However, when dealing with polymorphic or abstract types, it may not be possible to conjure up a dummy value of the right type. Consequently, rather than manipulating arrays with type `q array(a)` for some abstract type `a`, we may need to manipulate arrays of options with type `q array(a + unit)`. In this case, when we need to read out a value, we always have another value (`inr ()`) to swap in in its place. Normally such operations are called *destructive reads*; they are quite a common way to preserve the single pointer property when managing complex structured data.

Reference Counting

Array swaps and destructive reads are dynamic techniques that can help overcome a lack of compile-time knowledge about the number of uses of a particular object. *Reference counting* is another dynamic technique that serves a similar purpose. Rather than restricting the number of pointers to an object to be exactly one, we can allow any number of pointers to the object and keep track of that number dynamically. Only when the last reference is used will the object be deallocated.

There are various ways to integrate reference counts into the current system. Here, we choose the simplest, which is to add a new qualifier `rc` for reference-counted data structures, and operations that allow the programmer to explicitly increment (`inc`) and decrement (`dec`) the counts. More specifically, the increment operation takes a pointer argument, increments the reference count for the object pointed to, and returns two copies of the pointer as a (linear) pair. The decrement operation takes two arguments, a pointer and a function, and works as follows. In the case the object pointed to (call it `x`) has a reference count of 1 before the decrement, the function executes with `x` as an argument. The function treats `x` linearly and deallocates it before it completes. It may also recursively decrement the reference counts of any subcomponents. In the other case, when `x` has a reference count greater than 1, the reference count is simply decremented and the function is not called; `unit` is returned in its place.

The main typing invariant in this system is that whenever a reference-counted variable appears in the static type-checking context, there is one dynamic reference count associated with it. Linear typing will ensure the number of references to an object are properly preserved.

The new `rc` qualifier should be treated in the same manner as the linear qualifier when it comes to context splitting. In other words, a reference-

Syntax

q ::=		*qualifiers:*
	...	*as before*
	rc	*ref. count*
t ::=		*terms:*
	...	*as before*
	inc(t)	*increment count*
	dec(t,t)	*decrement count*

Qualifier Relations

$$\texttt{rc} \sqsubseteq \texttt{un} \qquad \text{(Q-RCUN)}$$

$$\texttt{lin} \sqsubseteq \texttt{rc} \qquad \text{(Q-LINRC)}$$

Typing $\boxed{\Gamma \vdash \texttt{t} : \texttt{T}}$

$$\frac{\Gamma \vdash \texttt{t} : \texttt{rc P}}{\Gamma \vdash \texttt{inc(t)} : \texttt{lin (rc P * rc P)}} \qquad \text{(T-INC)}$$

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{rc P} \qquad \Gamma \vdash \texttt{t}_2 : \texttt{lin P} \to \texttt{unit}}{\Gamma \vdash \texttt{dec(t}_1\texttt{,t}_2\texttt{)} : \texttt{unit}} \qquad \text{(T-DEC)}$$

Figure 6-16: Linear lambda calculus: Reference counting syntax and typing

counted variable should be placed in exactly one of the left-hand context or the right-hand context (not both). In terms of containment, the rc qualifier sits between unrestricted and linear qualifiers: A reference-counted data structure may not be contained in unrestricted data structures and may not contain linear data structures. Figure 6-16 presents syntax, appropriate qualifier relation and typing rules for our reference counting additions.

In order to define the execution behavior of reference-counted data structures, and later prove that execution is sound, we will define a new sort of stored value with the form rc(n,X) w. The integer n is the reference count: it keeps track of the number of pointers the value. Given a mapping x↦rc(n,X) w, the meta-variable X represents a list of n pseudonyms for x. More specifically, rather than having x show up n times in other parts of the store, each of the variables in the list X will show up exactly once. This arrangement facilitates the proof of type safety, which we will discuss in a moment.

The operational semantics for the new commands and for reference-counted pairs and functions are summarized in Figure 6-17. Several new bits of notation show up here to handle the relatively complex computation that must go on to increment and decrement reference counts. First, in a slight abuse of notation, we allow q to range over static qualifiers un, lin and rc as well as dynamic qualifiers un, lin and rc(n,X). Context will disambiguate the two different sorts of uses. Second, we extend the notation $\texttt{S} \overset{\texttt{q}}{\sim} \texttt{x}$ so that q may be rc(n,X) as well as lin and un. If X is the singleton set {x} then $\texttt{S} \overset{\texttt{rc(n,X)}}{\sim} \texttt{x}$ removes the binding y↦rc(n,X) w from S when x∈X. Otherwise,

$S \overset{rc(n,X)}{\sim} x$ repaces the binding $y \mapsto rc(n, X)\ w$ with $y \mapsto rc(n, Y - \{x\})\ w$ Finally, to increment a set of reference counts and add new pseudonyms for the appropriate pointers, we define the function $incr(S; X)$.

$$
\begin{array}{lcl}
incr(S; \cdot) & = & (S; \cdot) \\
incr(S; x, X) & = & (S''; x', X') \\
 & & \text{if} \\
 & & incr(S; X) = (S'; X') \\
 & & S' = S'_1, y \mapsto rc(n, Y)\ w, S'_2 \\
 & & x \in Y \\
 & & x' \notin FV(S') \\
 & & S'' = S'_1, y \mapsto rc(n+1; x', Y)\ w, S'_2 \\
incr(S; x, X) & = & (S'; X') \\
 & & \text{if} \\
 & & incr(S; X) = (S'; X') \\
 & & S'(x) = \text{un}\ w \\
incr(S; X) & & \text{undefined otherwise}
\end{array}
$$

To understand how the reference counting operational semantics works, we will focus on the rules for pairs. Allocation and use of linear and unrestricted pairs stays unchanged from before as in rules (E-PAIR′) and (E-SPLIT′). Rule (E-PAIRRC) specifies that allocation of reference-counted pairs is similar to allocation of other data, except for the fact that the dynamic reference count must be initialized to 1. Use of reference-counted pairs is identical to use of other kinds of pairs when the reference count is 1: We remove the pair from the store via the function $S \overset{rc(n,X)}{\sim} x$ as shown in rule and substitute the two components of the pair in the body of the term as shown in (E-SPLIT′). When the reference count is greater than 1, rule (E-SPLITRC) shows there are additional complications. More precisely, if one of the components of the pair, say y_1, is reference-counted then y_1's reference count must be increased by 1 due to the fact that y_1's container is not deallocated and an additional copy of y_1 is substituted through the body of t. We use the $incr$ function to handle the possible increase. In most respects, the operational rules for reference-counted functions follow the same principles as reference-counted pairs. Increment and decrement operations are also relatively straightforward.

The main trickiness in carrying out the proof of safety for the language is finding store typing rules that enforce the appropriate properties and are preserved at each step in the computation. In anticipation of the difficulties, we have set up the operational semantics so that reference-counted values carry with them a set X of pseudonyms for the actual pointer x in the store binding $x \mapsto rc(n, X)\ w$. These pseudonyms each have one occurrence some-

where else in the store. This arrangement allows us to maintain our invariant that a context Γ has at most one occurence of any variable and yet place `n` assumptions that do not obey weakening or contraction laws in the context. In summary, the rule for typing reference-counted values can be the following.

$$\frac{\vdash \mathtt{S} : \Gamma_1 \circ \Gamma_2 \qquad \Gamma_1 \vdash \mathtt{rc\ w} : \mathtt{T}}{\vdash \mathtt{S}, \mathtt{x} \mapsto \mathtt{rc(n, x_1, \ldots, x_n)\ w} : \Gamma_2, \mathtt{x_1{:}T}, \ldots, \mathtt{x_n{:}T}} \quad \text{(T-NEXTRCS)}$$

Note that we require that none of the variables $\mathtt{x}_1, \ldots, \mathtt{x}_n$ be repeated either in the domain of the store or in `Y` when `Y` appears in other reference-counted data structures (`rc(n,Y) w`).

6.3.7 EXERCISE [★★,↛]: State and prove progress and preservation for the simply-typed linear lambda calculus (functions and pairs) with reference counting. □

The list of pseudonyms is really just an artifact of our proof of safety; it is not necessary for a practical implementation to mimic this list as programs actually execute. We could justify this claim after proving type safety by creating a second operational semantics in which reference-counted values have the form `rc(n) w`. Then, we might show that this second operational semantics executes in an identical (modulo the pseudonyms), step-by-step fashion to the first.

6.4 An Ordered Type System

Just as linear type systems provide a foundation for managing memory allocated on the heap, *ordered* type systems provide a foundation for managing memory allocated on the stack. The central idea is that by controlling the exchange property, we are able to guarantee that certain values, those values allocated on the stack, are used in a first-in/last-out order.

To formalize this idea, we will organize the store into two parts: a stack, which is a sequence of locations that can be accessed on one end (the "top"), and a heap, which is just like the store described in previous sections of this chapter. Pairs, functions and other objects introduced with unrestricted or linear qualifiers will be allocated on the heap as before. And as before, when a linear pair or function is used, it is deallocated. In contrast, pairs and functions introduced using an ordered qualifier (`ord`) are allocated at the top of the stack. Due to the lack of the exchange property, an ordered object can only be used when it is at the top of the stack. When this happens, the ordered object is popped off the top of the stack.

$$
\begin{array}{lll}
v ::= & & \textit{values:} \\
 & \ldots & \textit{as before} \\
 & \texttt{rc(n,X) w} & \textit{ref-counted value} \\
E ::= & & \textit{evaluation contexts:} \\
 & \ldots & \textit{as before} \\
 & \texttt{inc(E)} & \textit{inc context} \\
 & \texttt{dec(E,t)} & \textit{dec context} \\
 & \texttt{dec(x,E)} & \textit{dec context}
\end{array}
$$

Evaluation $\boxed{\texttt{(S;t)} \longrightarrow_\beta \texttt{(S';t')}}$

$$\frac{\texttt{(q} \in \texttt{\{un,lin\})}}{\texttt{(S;q <t}_1\texttt{,t}_2\texttt{>)} \longrightarrow_\beta \texttt{(S, x} \mapsto \texttt{q <t}_1\texttt{,t}_2\texttt{>; x)}} \text{(E-PAIR')}$$

$$\texttt{(S;rc <t}_1\texttt{,t}_2\texttt{>)} \longrightarrow_\beta \texttt{(S, x} \mapsto \texttt{rc(1,y) <t}_1\texttt{,t}_2\texttt{>; y)} \quad \text{(E-PAIRRC)}$$

$$\frac{\texttt{S(x) = q <y}_1\texttt{,z}_1\texttt{>} \qquad \texttt{(q} \in \texttt{\{un,lin,rc(1,X)\})}}{\texttt{(S;split x as y,z in t)} \longrightarrow_\beta \texttt{(S} \overset{q}{\sim} \texttt{x;[y} \mapsto \texttt{y}_1\texttt{][z} \mapsto \texttt{z}_1\texttt{]t)}} \text{(E-SPLIT')}$$

$$\frac{\texttt{S(x) = rc(n,X) <y}_1\texttt{,z}_1\texttt{>} \qquad \texttt{(n > 1)} \qquad \texttt{incr(S;y}_1\texttt{,z}_1\texttt{) = (S';y}'_1\texttt{,z}'_1\texttt{)}}{\texttt{(S;split x as y,z in t)} \longrightarrow_\beta \texttt{((S'} \overset{q}{\sim} \texttt{x);[y} \mapsto \texttt{y}'_1\texttt{][z} \mapsto \texttt{z}'_1\texttt{]t)}} \text{(E-SPLITRC)}$$

$$\frac{\texttt{(q} \in \texttt{\{un,lin\})}}{\texttt{(S;q} \lambda\texttt{y:T.t)} \longrightarrow_\beta \texttt{(S, x} \mapsto \texttt{q} \lambda\texttt{y:T.t; x)}} \text{(E-FUN')}$$

$$\texttt{(S;rc} \lambda\texttt{y:T.t)} \longrightarrow_\beta \texttt{(S, x} \mapsto \texttt{rc(1,z)} \lambda\texttt{y:T.t; z)} \quad \text{(E-FUNRC)}$$

$$\frac{\texttt{S(x}_1\texttt{) = q} \lambda\texttt{y:T.t} \qquad \texttt{(q} \in \texttt{\{un,lin,rc(1,X)\})}}{\texttt{(S;x}_1\ \texttt{x}_2\texttt{)} \longrightarrow_\beta \texttt{(S} \overset{q}{\sim} \texttt{x}_1\texttt{;[y} \mapsto \texttt{x}_2\texttt{]t)}} \text{(E-APP')}$$

$$\frac{\texttt{S(x}_1\texttt{) = rc(n,X)} \lambda\texttt{y:T.t} \qquad \texttt{(n > 1 and X = } FV(\lambda\texttt{y:T.t})\texttt{)} \qquad \texttt{incr(S;X) = (S';X')}}{\texttt{(S;x}_1\ \texttt{x}_2\texttt{)} \longrightarrow_\beta \texttt{(S} \overset{q}{\sim} \texttt{x}_1\texttt{;[X} \mapsto \texttt{X'][y} \mapsto \texttt{x}_2\texttt{]t)}} \text{(E-APPRC)}$$

$$\frac{\texttt{(x}_i \in \texttt{X)} \qquad \texttt{S=S}_1\texttt{, x} \mapsto \texttt{rc(n,X) w, S}_2 \qquad \texttt{S'=S}_1\texttt{, x} \mapsto \texttt{rc(n+1, (x}_{n+1}\texttt{,X)) w, S}_2}{\texttt{(S;inc(x}_i\texttt{))} \longrightarrow_\beta \texttt{(S';lin <x}_i\texttt{,x}_{n+1}\texttt{>)}} \text{(E-INC)}$$

$$\frac{\texttt{(x}_i \in \texttt{X)} \qquad \texttt{(n > 1)} \qquad \texttt{S=S}_1\texttt{, x} \mapsto \texttt{rc(n,X) w, S}_2 \qquad \texttt{S'=S}_1\texttt{, x} \mapsto \texttt{rc(n-1,X-x}_i\texttt{) w, S}_2}{\texttt{(S;dec(x}_i\texttt{,x}_f\texttt{))} \longrightarrow_\beta \texttt{(S';un ())}} \text{(E-DEC1)}$$

$$\frac{\texttt{S=S}_1\texttt{, x} \mapsto \texttt{rc(1,x}_i\texttt{) w, S}_2 \qquad \texttt{S'=S}_1\texttt{, x} \mapsto \texttt{lin w, S}_2}{\texttt{(S;dec(x}_i\texttt{,x}_f\texttt{))} \longrightarrow_\beta \texttt{(S';x}_f\ \texttt{x)}} \text{(E-DEC2)}$$

Figure 6-17: Linear lambda calculus: Reference counting operational semantics

Syntax

q	::=	*qualifiers:*
	ord	*ordered*
	lin	*linear*
	un	*unrestricted*
t	::=	*terms:*
	x	*variable*
	q b	*Boolean*
	if t then t else t	*conditional*
	q <x, y>	*pair*
	split t as x, y in t	*split*
	q λx:T.t	*abstraction*
	x y	*application*
	let x = t in t	*sequencing*
P	::=	*pretypes:*
	Bool	*booleans*
	T*T	*pairs*
	T→T	*functions*
T	::=	*types:*
	q P	*qualified pretype*
Γ	::=	*contexts:*
	∅	*empty context*
	Γ, x:T	*term variable binding*

Figure 6-18: Ordered lambda calculus: Syntax

Syntax

The overall structure and mechanics of the ordered type system are very similar to the linear type system developed in previous sections. Figure 6-18 presents the syntax for the simply typed system. One key change from our linear type system is that we have introduced an explicit sequencing operation `let x = t1 in t2` that first evaluates the term t_1, binds the result to `x`, and then continues with the evaluation of t_2. This sequencing construct gives programmers explicit control over the order of evaluation of terms, which is crucial now that we are introducing data that must be used in a particular order. Terms that normally can contain multiple nested subexpressions such as pair introduction and function application are syntactically restricted so that their primary subterms are variables and the order of evaluation is clear. To understand why this restriction is important for the ordered type system to come, see exercise 6.4.2.

The other main addition is a new qualifier `ord` that marks data allocated on the stack. Ordered assumptions will be tracked in the type checking context Γ like other assumptions. However, they will not be subject to the exchange property. Moreover, the order that they appear in Γ mirrors the order that they will appear on the stack, with the rightmost position in Γ representing the top of the stack.

Context Split $\boxed{\Gamma = \Gamma_1 \circ \Gamma_2}$

$$\frac{\Gamma =_2 \Gamma_1 \circ \Gamma_2}{\Gamma = \Gamma_1 \circ \Gamma_2} \text{ (M-TOP)}$$

$$\emptyset =_1 \emptyset \circ \emptyset \text{ (M-EMPTY)}$$

$$\frac{\Gamma =_1 \Gamma_1 \circ \Gamma_2}{\Gamma, \mathtt{x{:}ord\ P} =_1 (\Gamma_1, \mathtt{x{:}ord\ P}) \circ \Gamma_2} \text{ (M-ORD1)}$$

$$\frac{\Gamma =_2 \Gamma_1 \circ \Gamma_2}{\Gamma, \mathtt{x{:}ord\ P} =_2 \Gamma_1 \circ (\Gamma_2, \mathtt{x{:}ord\ P})} \text{ (M-ORD2)}$$

$$\frac{\Gamma =_1 \Gamma_1 \circ \Gamma_2}{\Gamma =_2 \Gamma_1 \circ \Gamma_2} \text{ (M-1TO2)}$$

$$\frac{\Gamma =_{1,2} \Gamma_1 \circ \Gamma_2}{\Gamma, \mathtt{x{:}lin\ P} =_{1,2} (\Gamma_1, \mathtt{x{:}lin\ P}) \circ \Gamma_2} \text{ (M-LINA)}$$

$$\frac{\Gamma =_{1,2} \Gamma_1 \circ \Gamma_2}{\Gamma, \mathtt{x{:}lin\ P} =_{1,2} \Gamma_1 \circ (\Gamma_2, \mathtt{x{:}lin\ P})} \text{ (M-LINB)}$$

$$\frac{\Gamma =_{1,2} \Gamma_1 \circ \Gamma_2}{\Gamma, \mathtt{x{:}un\ P} =_{1,2} (\Gamma_1, \mathtt{x{:}un\ P}) \circ (\Gamma_2, \mathtt{x{:}un\ P})} \text{ (M-UN)}$$

Figure 6-19: Ordered Context Manipulation Rules

Typing

The first step in the development of the type system is the definition of rules for splitting contexts with ordered assumptions ($\Gamma = \Gamma_1 \circ \Gamma_2$). As before, we will allow unrestricted assumptions to be used in all subexpressions, and linear assumptions to be used in exactly one subexpression. Ordered assumptions should be used in exactly one subexpression, and must be used in the order in which they appear. Therefore, some (nondeterministically chosen) sequence of ordered assumptions taken from the left-hand side of Γ will be placed in Γ_1 and the remaining ordered assumptions will be placed in Γ_2. The context Γ_2 will be used by the first subexpression to be evaluated (since the top of the stack is at the right) and Γ_1 will be used by the second subexpression to be evaluated. Formally, we define the "=" relation in terms of two subsidiary relations, "$=_1$," which places ordered assumptions in Γ_1, and "$=_2$," which places ordered assumptions in Γ_2.

The second step is to determine the containment rules for ordered data structures. Previously, we saw that if an unrestricted object can contain a linear object, a programmer can write functions that duplicate or discard linear objects, thereby violating the central invariants of the system. A similar situation arises if linear or unrestricted objects can contain stack objects; in either case, the stack object might be used out of order, after it has been popped off the stack. The qualifier relation $\mathtt{q}_1 \sqsubseteq \mathtt{q}_2$, which specifies that $\mathtt{ord} \sqsubseteq \mathtt{lin} \sqsubseteq \mathtt{un}$, ensures such problems do not arise.

6.4.1 EXERCISE [★,RECOMMENDED]: Specify the appropriate relation for poly-

Typing $\boxed{\Gamma \vdash t : T}$

$$\frac{\mathtt{un}(\Gamma_1, \Gamma_2)}{\Gamma_1, x{:}T, \Gamma_2 \vdash x : T} \quad \text{(T-OVAR)}$$

$$\frac{\mathtt{un}(\Gamma)}{\Gamma \vdash q\ b : q\ \mathtt{Bool}} \quad \text{(T-OBOOL)}$$

$$\frac{\Gamma_2 \vdash t_1 : q\ \mathtt{Bool} \qquad \Gamma_1 \vdash t_2 : T \qquad \Gamma_1 \vdash t_3 : T}{\Gamma_1 \circ \Gamma_2 \vdash \mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3 : T} \quad \text{(T-OIF)}$$

$$\frac{\Gamma_1 \vdash x_1 : T_1 \qquad \Gamma_2 \vdash x_2 : T_2 \qquad q(T_1) \qquad q(T_2)}{\Gamma_1 \circ \Gamma_2 \vdash q\ \langle x_1, x_2 \rangle : q\ (T_1 * T_2)} \quad \text{(T-OPAIR)}$$

$$\frac{\Gamma_2 \vdash t_1 : q\ (T_1 * T_2) \qquad \Gamma_1, x{:}T_1, y{:}T_2 \vdash t_2 : T}{\Gamma_1 \circ \Gamma_2 \vdash \mathtt{split}\ t_1\ \mathtt{as}\ x, y\ \mathtt{in}\ t_2 : T} \quad \text{(T-OSPLIT)}$$

$$\frac{q(\Gamma) \qquad \Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash q\ \lambda x{:}T_1.t_2 : q\ T_1 \to T_2} \quad \text{(T-OABS)}$$

$$\frac{\Gamma_1 \vdash x_1 : q\ T_{11} \to T_{12} \qquad \Gamma_2 \vdash x_2 : T_{11}}{\Gamma_1 \circ \Gamma_2 \vdash x_1\ x_2 : T_{12}} \quad \text{(T-OAPP)}$$

$$\frac{\Gamma_2 \vdash t_1 : T_1 \qquad \Gamma_1, x{:}T_1 \vdash t_2 : T_2}{\Gamma_1 \circ \Gamma_2 \vdash \mathtt{let}\ x = t_1\ \mathtt{in}\ t_2 : T_2} \quad \text{(T-OLET)}$$

Figure 6-20: Ordered lambda calculus: Typing

morphic qualifiers in this ordered type system. □

The typing rules for the ordered lambda calculus appear in Figure 6-20. For the most part, the containment rules and context splitting rules encapsulate the tricky elements of the type system. Let's examine the rules for functions to see how they do it. The rule for introducing functions (T-OABS) looks ordinary: It adds an assumption x to the right-hand side of the context for the purposes of checking the function body. If the function has ordered type and the argument has ordered type, this rule states that any argument passed to the function will be allocated on the top of the stack. Moreover, ordered variables in the function closure will appear deeper on the stack than the argument. In the rule for function application (T-OAPP), the context is split, with the right hand side (top of the stack) being used in formation of the argument and the left-hand side being used in formation of the function. So indeed, when the function is called, its argument will appear at the top of the stack, above the objects in the function closure. The rules for booleans and pairs are designed in a similar fashion.

As a simple example, consider the following function. It takes a boolean and a pair as an argument, allocated sequentially at the top of the stack. The boolean is at the very top of the stack and the integer pair is next (the top of the stack is to the left). If the boolean is true, it leaves the components of the pair in the same order as they appeared on input and otherwise, it swaps their order.

```
lamda x:ord (ord (int * int) * bool).
  split x as p,b in
  if b then
    p
  else
    split p as i1,i2 in
    ord <i2,i1>
```

6.4.2 EXERCISE [★,RECOMMENDED]: Write a program that demonstrates what can happen if the syntax of pair formation is changed to allow programmers to write nested subexpressions (ie: we allow `ord <`e_1`,` e_2`>` rather than `ord <x,y>`)? □

The stack typing discipline that arises from our ordered type system is not too flexible as it does not allow programmers to use objects allocated deep on the stack; all uses must occur at the top of the stack. However, this is exactly the sort of typing discipline required to check that the operation of stack-based virtual machines such as the JVM is safe.

6.4.3 EXERCISE [★★★,↛]: Look up the definition of the Java Virtual Machine Language (Yellin (Lindholm and Yellin, 1997) provides a good reference) and design an ordered type system to keep track of the machine operand stack. □

Another related application of this ordered type system involves type-preserving compilation of functional languages via the *continuation-passing style* (CPS) transform. CPS decomposes ordinary high-level functions into low-level CPS functions that explicitly take a *continuation* (the function's return address, itself a function) as an argument and explicitly invoke (call) the continuation to return, the return value being the argument of the continuation. For example, we can recode the function above in CPS as follows.

```
type cont = ord (ord (int * int) → ans)

lamda x:ord (cont * ord (bool * ord (int * int))).
  split x as cont,x in
  split x as p,b in
  if b then
    cont p
  else
    split p as i1,i2 in
    cont (ord <i2,i1>)
```

We had to make two main changes to the function. First, we split the incoming argument into two parts: the actual argument and the continuation.

Second, rather than returning the newly allocated pair, we call the continuation. We assume the type `ans` is the type of the final result of the computation.

Since the translation changes the calling convention for functions, the translation must also change function application. Each function call with the form `f x` will now have the form `f ord <k, x>` where `k` is the current continuation.

6.4.4 EXERCISE [★★★★,↛]: Research the continuation-passing style transformation in the literature. Define a type-preserving, continuation-passing style translation from the ordinary, simply-typed lambda calculus to the ordered lambda calculus that allocates continuations on the stack. State and prove a theorem that shows your translation this type-preserving. A basis for the solution can be found in research on ordered type theory by Polakow and Pfenning (Polakow and Pfenning, 2000). □

Operational Semantics

To define the operational semantics for our new ordered type system, we will divide our previous stores into two parts, a heap `H` and a stack `K`. Both are just a list of bindings as stores were before (see Figure 6-21 for a summary).

Rather than rewrite our operational semantics for the new ordered (stack-allocated) elements, we simply alter the definition of our store manipulation functions. First, we need to say what it means to add a binding to the store. This is straightforward: unrestricted and linear bindings are added to the heap and ordered bindings are added to the stack.

$$
\begin{array}{lcl}
(\mathtt{H};\mathtt{K}),\mathtt{x} \mapsto \mathtt{ord}\,\mathtt{w} & = & (\mathtt{H};\mathtt{K},\mathtt{x} \mapsto \mathtt{ord}\,\mathtt{w}) \\
(\mathtt{H};\mathtt{K}),\mathtt{x} \mapsto \mathtt{lin}\,\mathtt{w} & = & (\mathtt{H},\mathtt{x} \mapsto \mathtt{lin}\,\mathtt{w};\mathtt{K}) \\
(\mathtt{H};\mathtt{K}),\mathtt{x} \mapsto \mathtt{un}\,\mathtt{w} & = & (\mathtt{H},\mathtt{x} \mapsto \mathtt{un}\,\mathtt{w};\mathtt{K})
\end{array}
$$

Second, we need to specify how to remove a binding from the store.

$$
\begin{array}{lcl}
(\mathtt{H};\mathtt{K}_1,\mathtt{x} \mapsto \mathtt{v},\mathtt{K}_2) \overset{\mathtt{ord}}{\sim} \mathtt{x} & = & \mathtt{H};\mathtt{K}_1,\mathtt{K}_2 \\
(\mathtt{H}_1,\mathtt{x} \mapsto \mathtt{v},\mathtt{H}_2;\mathtt{K}) \overset{\mathtt{lin}}{\sim} \mathtt{x} & = & \mathtt{H}_1,\mathtt{H}_2;\mathtt{K} \\
(\mathtt{H};\mathtt{K}) \overset{\mathtt{un}}{\sim} \mathtt{x} & = & \mathtt{H};\mathtt{K}
\end{array}
$$

With these simple changes, the evaluation rules from previous sections can be reused essentially unchanged. We only need to add the the evaluation context for sequencing (`let x = E in t`) and its evaluation rule.

S	::=	*stores:*		$\mathtt{H}, \mathtt{x} \mapsto \mathtt{v}$	*heap binding*
	$\mathtt{H};\mathtt{K}$	*complete store*	K	::=	*stack:*
H	::=	*heap:*		$\emptyset$	*empty context*
	$\emptyset$	*empty context*		$\mathtt{K}, \mathtt{x} \mapsto \mathtt{v}$	*stack binding*

Figure 6-21: Ordered Lambda Calculus: Operational Semantics

$$(\mathtt{S};\mathtt{let}\ \mathtt{x} = \mathtt{x}_1\ \mathtt{in}\ \mathtt{t}_2) \longrightarrow_\beta (\mathtt{S};[\mathtt{x} \mapsto \mathtt{x}_1]\mathtt{t}_1) \qquad \text{(E-LET)}$$

In all cases but one, deallocation of an ordered object will occur at the top of the stack. In each of these cases, the ordered deallocation operation could be rewritten as follows and implemented as an efficient stack pointer increment.

$$(\mathtt{H};\mathtt{K}, \mathtt{x} \mapsto \mathtt{v}) \overset{\mathtt{ord}}{\sim} \mathtt{x} \quad = \quad \mathtt{H};\mathtt{K}$$

The single anomalous case is the case for application of an ordered function to an ordered argument. In this case, the stack is organized with the ordered variables in the function appearing lowest on the stack, the ordered function pointer appearing immediately above that and the ordered argument appearing on top. However, the ordered function pointer is used (and deallocated) immediately upon application, whereas the ordered argument may be used at some later point in the function body. Hence, to call an ordered function with an ordered argument, a low-level implementation would have to first load the function pointer from its position on the stack, then shift the argument down the stack so it occupies the space directly above the variables in the ordered function's closure, and finally, begin to run the ordered function's code.

6.4.5 EXERCISE [★★★]: Define new typing rules and a new operational semantics specific to the case when an ordered function is applied to an ordered argument. Come up with a system that avoids having to "shift" the argument down the stack when the function is called. □

6.4.6 EXERCISE [★★,↛]: It would be nice if we could access objects deep on the stack. Define the static and dynamic semantics of an operation that copies the data n locations from the top of the stack into the heap (for any abitrary, but statically known integer n). □

6.5 Further Applications

Memory management applications make good motivation for substructural type systems and provides a concrete framework for studying their properties. However, substructural types systems, and their power to control the number and order of uses of data and operations, have found many applications outside of this domain. In the following paragraphs, we informally discuss a few of them.

Controlling Temporal Resources

We have studied several ways that substructural type systems can be used to control physical resources such as memory and files. What about controlling the temporal resources? Amazingly, substructural type systems can play a role here as well: Careful crafting of a language with an *affine* type system, where values are used at most once, can ensure that computations execute in polynomial time.

To be begin, we will allow our polynomial time language to contain affine booleans, pairs and (non-recursive) functions. In addition, to make things interesting, we will add affine lists to our language, which have constructors `nil` and `cons` and a special iterator to recurse over objects with list type. Such iterators have the following form.

```
iter (stop ⇒ t1 | x with y ⇒ t2)
```

If $\mathtt{t}_1$ and $\mathtt{t}_2$ have type `T`, our iterator defines a function from lists to objects with type `T`. Operationally, the iterator does a case to see whether its input list is `nil` or `cons(hd,tl)` and executes the corresponding branch. More specifically we can define the operation of iterators using two simple rules.[3]

$$\mathtt{iter\ (stop} \Rightarrow \mathtt{t}_1 \mid \mathtt{hd\ with\ rest} \Rightarrow \mathtt{t}_2\mathtt{)\ nil} \longrightarrow_\beta \mathtt{t}_1 \quad \text{(E-IterNil)}$$

$$\frac{\mathtt{iter\ (stop} \Rightarrow \mathtt{t}_1 \mid \mathtt{hd\ with\ rest} \Rightarrow \mathtt{t}_2\mathtt{)\ v}_2 \longrightarrow_\beta \mathtt{*v}_2'}{\mathtt{iter\ (stop} \Rightarrow \mathtt{t}_1 \mid \mathtt{hd\ with\ rest} \Rightarrow \mathtt{t}_2\mathtt{)\ cons(v}_1\mathtt{,v}_2\mathtt{)} \longrightarrow_\beta [\mathtt{hd} \mapsto \mathtt{v}_1][\mathtt{rest} \mapsto \mathtt{v}_2']\mathtt{t}_2} \quad \text{(E-IterCons)}$$

In the second rule, the iterator is invoked inductively on `tl`, giving the result $\mathtt{v}_2'$, which is used in term $\mathtt{t}_2$.

The `append` function below illustrates the use of iterators.

3. Since we are not interested in memory management here, we have simplified our operational semantics from previous parts of this chapter by deleting the explicit store and using substitution instead. The operational judgment has the form $\mathtt{t} \longrightarrow_\beta \mathtt{t}'$ and, in general, is defined similarly to the operational systems in *Types and Programming Languages.*

```
val append : T list→T list→T list =
  iter (
    stop ⇒ λ(l:T list).l
  | hd with rest ⇒ λ(l:T list).cons(hd,rest l))
```

When applied to a list $\texttt{l}_1$, the iterator builds up a function that expects a second list $\texttt{l}_2$ and concatenates $\texttt{l}_2$ to the end of $\texttt{l}_1$. Clearly, `append` is a polynomial time function, a linear-time one in fact, but it is straightforward to write exponential time algorithms in the language as we have defined it so far. For instance:

```
val double : T list→T list =
  iter (
    stop ⇒ nil
  | hd with rest ⇒ cons(hd,cons(hd,rest)))

val exp : T list→T list =
  iter (
    stop ⇒ nil
  | hd with rest ⇒ double (cons(hd,rest)))
```

The key problem here is that it is trivial to write iterators like `double` that increase the size of their arguments. After constructing such an iterator, we can use it as the inner loop of another iterator, like `exp`, and cause an exponential blow-up in our running time. However, this is not the only problem. Higher-order functions make it even easier to construct exponential-time algorithms:

```
val compose =
  λ(fg:(T list→T list) * (T list→T list)).
    λ(x:T list).
      split fg as f,g in f (g x)

val junk : T

val exp₂ : T list→T list→T list =
  iter (
    stop ⇒ λ(l:T list).cons(junk,l)
  | hd with rest ⇒ λ(l:T list).compose <rest,rest> l)
```

Fortunately, a substructural type system can be used to eliminate both problems by allowing us to define a class of *non-size-increasing* functions and by preventing the construction of troublesome higher-order functions, such as $\texttt{exp}_2$.

The first step is to demand that all user-defined objects have affine type. They can be used zero or one times, but not more. This restriction immediately rules out programs such as $\texttt{exp}_2$. System defined operators like cons can be used many times.

The next step is to put mechanisms in place to prevent iterators from increasing the size of their inputs. This can be achieved by altering the cons constructor so that it can only be applied when it has access to a special resource with type R.

```
operator cons : (R,T,T list) → T list
```

There is no constructor for resources with type R so they cannot be generated out of thin air; we can only apply cons as many times as we have resources. We also adapt the syntax for iterators as follows.

```
iter (
  stop ⇒ t1
| hd with tl and r ⇒ t2)
```

Inside the second clause of the iterator, we are only granted a single resource (r) with which to allocate data. Consequently, we can allocate at most one cons cell in t_2. This provides us with the power to rebuild a list of the same size, but we cannot write a function such as double that doubles the length of the list or exp that causes an exponential increase in size. To ensure that a single resource from an outer scope does not percolate inside the iterator and get reused on each iteration of the loop, we require that iterators be closed, mirroring the containment rules for recursive functions defined in earlier sections of this chapter.

Although restricted to polynomial time, our language permits us to write many useful functions in a convenient fashion. For instance, we can still write append much like we did before. The resource we acquire from destructing the list during iteration can be used to rebuild the list later.

```
val append : T list → T list → T list =
  iter (
    stop ⇒ λ(l:T list).l
  | hd with rest and r ⇒ λ(l:T list). cons(r,hd,rest l))
```

We can also write double if our input list comes with appropriate credits, in the form of unused resources.

```
val double : (T*R) list → T list =
  iter (
    stop ⇒ nil
  | hd with rest and r1 ⇒
     split hd as x,r2 in cons(r1,hd,cons(r2,hd,rest)))
```

Fortunately, we will never be able to write `exp`, unless, of course, we are given an exponential number of credits in the size of the input list. In that case, our function `exp` would still only run in linear time with respect to our overall input (list and resources included).

The proof that all (first-order) functions we can define in this language run in polynomial time uses some substantial domain theory that lies outside the scope of this book. However, the avid reader should see Section 6.6 for references to the literature where these proofs can be found.

Compiler Optimizations

Many compiler optimizations are enabled when we know that there will be *at most one use* or *at least one use* of a function, expression or data structure. If there is at most one use of an object then we say that object has *affine* type. If there is at least one use then we say the object has *relevant* (or *strict*) type. Here are just a few optimizations that usage information can enable.

- *Floating in bindings.* Consider the expression `let x = e in (λy....x...)`. Assume the computation `e` has no effects, is it a good idea to float the binding inside the lambda and create the new expression `λy.let x = e in (...x...)`? The answer depends on how many times the resulting function is used. If it is used at most once, the optimization is probably a good one: we may avoid computing `e` and will never compute it more than once.

- *Inlining expressions.* In the example above, if we have the further information that `x` itself is used at most once inside the body of the function, then we might want to substitute the expression `e` for `x`. This may give rise to further local optimizations at the site where `e` is used. Moreover, if it turns out that `e` is used zero times (as opposed to one time) we will have saved ourselves the trouble of computing it.

- *Thunk update avoidance.* In lazy functional languages such as Haskell, evaluation of function parameters is delayed until the parameter is actually used in the function body. In order to avoid recomputing the value of the parameter each time it is used, implementers make each parameter a *thunk* — a reference that may either hold the computation that needs to be run or the value itself. The first time the thunk is used, the computation will be run and will produce the necessary result. In general, this result is stored back in the thunk for all future uses of the parameter. However, if the compiler can determine that the data structure is used as most once, this thunk update can be avoided.

- *Eagerness.* Again in lazy functional languages, if we can guarantee that an object is used at least once, then we can evaluate it right away and avoid creating a thunk altogether.

The optimizations described above may be implemented in two phases. The first phase is a program analysis that may be implemented as affine and/or relevant type inference. After the analysis phase, the compiler uses the information to transform programs. Formulating compiler optimizations as type inference followed by type-directed translation has a number of advantages over other techniques. First, the language of types can be used to communicate optimization information across modular boundaries. This can facilitate the process of scaling intra-procedural optimizations to inter-procedural optimizations. Second, the type information derived in one optimization pass can be maintained and propagated to future optimization passes or into the back end of the compiler where it can be used to generate Typed Assembly Language or Proof-Carrying Code, as discussed in Chapters 8 and 7.

6.6 Notes

Substructural logics are very old and date back to at least Orlov (Orlov, 1928), who axiomatized the implicational fragment of relevant logic. Somewhat later, Moh (Moh, 1950) and Church (Church, 1951) provided alternative axiomatizations of the relevant logic now known as R. In the same time period, Church was developing his theory of the lambda calculus at Princeton University, and his λI calculus (Church, 1941), which dissallowed abstraction over variables that did not appear free in the body of the term, was the first substructural lambda calculus. Lambek (Lambek, 1958) introduced the first "ordered logic," and used it to reason about natural language sentence structure. More recently, Girard (Girard, 1987) developed linear logic, which gives control over both contraction and weakening, and yet provides the full power of intuitionistic logic through the unrestricted modality "!". O'Hearn and Pym (O'Hearn and Pym, 1999) show that the logic of bunched implications provides another way to recapture the power of intuitionistic logic while giving control over the structural rules.

For a comprehensive account of the history of substructural logics, please see Došen (Došen and Schroeder-Heister, 1993), who is credited with coining the phrase "substructural logic," or Restall (Restall, 2001). Restall's textbook on substructural logics (Restall, 2000) provides good starting point to those looking to study the technical details of either the proof theory or model theory for these logics.

John Reynolds pioneered the study of substructural type systems for programming languages with his development of syntactic control of interference (Reynolds, 1978, 1989), which prevents two references from being bound to the same variable and thereby facilitates reasoning about Algol programs. Later, Girard's development of linear logic inspired many researchers to develop functional languages with linear types. One of the main applications of these new type systems was to control effects and enable in-place update of arrays in pure functional languages.

Lafont (Lafont, 1988) was the first to develop a linear abstract machine in which linear values could be deallocated after use. He was soon followed by many other researchers, including Baker (Baker, 1992) who informally showed how to compile Lisp into a linear assembly language in which all allocation, deallocation and pointer manipulation is completely explicit, yet safe. Another influential piece of work is due to Chirimar, Gunter and Riecke (Chirimar, Gunter, and Riecke, 1996) who developed an interpretation of linear logic based on reference counting. Their reference counting scheme is slightly different than the one proposed here due to the differences between linear logic and the (non-logical) linear types that make up our system. Turner and Wadler (Turner and Wadler, 1999) summarize two computational interpretations that arise directly through the Curry-Howard isomorphism from Girard's linear logic. They differ from the account given in this chapter as neither account has both shared, useable data structures and deallocation. These two features, which are essential if one is to build a practical programming language, appear incompatible with a type system derived directly from linear logic and its single unrestricted modality.

The development of practical linear type systems with two classes of type, one linear and one unrestricted, began with Wadler's work (Wadler, 1990) in the early nineties. The presentation given in this chapter is inspired by work from Wansbrough and Peyton Jones (Wansbrough and Peyton Jones, 1999) and Walker and Watkins (Walker and Watkins, 2001a). Wansbrough and Peyton Jones included qualifier subtyping and bounded parametric polymorphism in their system in addition to many of the features described here. However, the parameterization of our system according to the combination of `un`, `lin`, and `rc` qualifiers and containment rules is novel. The idea of formulating the system with a generic context splitting operator was taken from Cervesato and Pfenning's presentation of Linear LF (Cervesato and Pfenning, 2002).

The algorithmic type system described in section 6-5 solves what is commonly known in the linear logic programming and theorem proving literature, as the *resource management problem*. Many of the ideas for the current presentation came from work by Cervasato and Pfenning (Cervesato, Ho-

das, and Pfenning, 2000), who solve the more general problem that arises when linear logic's additive connectives are considered. Hofmann takes a related approach when solving the type inference problem for a linearly-typed functional language (Hofmann, 1997a).

The ordered type system developed here is also novel. However, it was derived from Polakow and Pfenning's ordered logic (Polakow and Pfenning, 1999), in the same way that the practical linear type systems mentioned above emerged from linear logic. The closest related type system (as opposed to logic) that we are aware of is Petersen's ordered lambda calculus (Petersen, Harper, Crary, and Pfenning, 2003a); this latter system does not classify types using qualifiers as we have done, but rather uses a single "mobility" modality.

Analysis and reasoning about the time and space complexity of programs has always been an important part of computer science. However, the use of programming language technology, and type systems in particular, to automatically constrain the complexity of programs is somewhat more recent. For instance, Bellantoni and Cook (Bellantoni and Cook, 1992) and Leivant (Leivant, 1993) developed predicative systems that control the use and complexity of recursive functions. It is possible to write all, and only, the polynomial-time functions in their system. However, it is not generally possible to compose functions and therefore many "obviously" polynomial-time algorithms cannot be coded naturally in their system. Girard (Girard, 1998), Hofmann (Hofmann, 2000, 1999), and Bellantoni et al. (Bellantoni, Niggl, and Schwichtenberg, 2000) show how linear type systems can be used to alleviate some of these difficulties. The material presented in this chapter is derived from Hofmann's work.

One of the most successful and extensive applications of substructural type systems in programming practice can be found in the Concurrent Clean programming language (N ocker, Smetsers, van Eekelen, and Plasmeijer, 1991). Clean is a commercially developed, pure functional programming language. It uses *uniqueness types* (Barendsen and Smetsers, 1993), which are a variant of linear types, and strictness annotations (N ocker and Smetsers, 1993) to help support concurrency, I/O and in-place update of arrays. The implementation is fast and is fully supported by a wide range of program development tools including an Integrated Development Environment for project management and GUI libraries, all developed in Clean itself.

Substructural type systems have also found gainful employment in the intermediate languages of the Glaskow Haskell Compiler. For instance, Turner et al. (Turner, Wadler, and Mossin, 1995) and Wansbrough and Peyton Jones (Wansbrough and Peyton Jones, 1999) showed how to use affine types and affine type inference to optimize programs as discussed earlier in this chapter. They

also use extensive strictness analysis to avoid thunk creation.

Recently, researchers have begun to investigate ways to combine substructural type systems with dependent types and effect systems such as those described in Chapters 4 and 5. The combination of both dependent and substructural types provides a very powerful tool for enforcing safe memory management and more general resource-usage protocols. For instance, DeLine and Fähndrich developed Vault (DeLine and Fähndrich, 2001; Fähndrich and DeLine, 2002), a programming language that uses static capabilities (Walker, Crary, and Morrisett, 2000b) (a hybrid form of linear types and effects) to enforce a variety of invariants in Microsoft Windows device drivers including locking protocols, memory management protocols and others. Cyclone (Jim, Morrisett, Grossman, Hicks, Cheney, and Wang, 2002; Grossman, Morrisett, Jim, Hicks, Wang, and Cheney, 2002b), a completely type-safe substitute for C, also uses linear types and effects to grant programmers fine-grained control over memory allocation and deallocation. In each of these cases, the authors do not stick to the pure linear types described here. Instead, they add coercions to the language to allow linearly-typed objects to be temporarily aliased in certain contexts, following a long line of research on this topic (Wadler, 1990; Odersky, 1992; Kobayashi, 1999; Smith, Walker, and Morrisett, 2000a; Aspinall and Hofmann, 2002).

Part IV

Low-Level Languages

7 *Proof-Carrying Code*

By George Necula

In this chapter and the one that follows, we explore a number of techniques that can be used to extend high-level typing ideas to programs written in low-level languages. This problem is interesting for a number of reasons. It seems natural to assume that low-level programs obtained by compiling well-typed high-level programs should have the same semantics and similar typing properties. Yet, it is not immediately clear how to write a type-checker for low-level programs.

Type checking is a convenient way to ensure that a program has certain semantic properties, such as memory safety. Type checking has gained acceptance as a major component of the security infrastructure in distributed systems in which code is shared between hosts that do not trust each other. Notable examples are the Java Virtual Machine (JVM) (Lindholm and Yellin, 1997) and the Microsoft Common Language Infrastructure (MS-CLI), both of which feature type checkers for intermediate languages used for the distribution of code in distributed systems. Such distribution schemes rely on low-level languages in order to delegate to the untrusted code producer most or all of the compilation effort and to reduce or eliminate the need for a trusted compiler on the receiving side. This strategy also has the potential to give the code producer some flexibility in the choice of the high-level language in which the development of the code takes place.

The distribution of untrusted mobile-code is the main scenario that we will use to guide our design choices in this chapter. We refer to a system that receives and executes untrusted code as a *host* and to the untrusted code itself as an *agent*. Hosts want to have strong guarantees about the safety of the agent's execution. Probably the most basic guarantee is memory safety, which ensures that the agent does not attempt to dereference memory addresses to which it has not been given explicit access. But often hosts need

to have stronger guarantees, such as bounded execution and resource usage, or proper use of the provided APIs. We shall start to address the memory safety aspect first and then show how the techniques that we develop can be applied to more complex safety polices.

It is worth pointing out that, in the absence of type checking, alternative mechanisms for ensuring the memory safety of untrusted code are based on coarse-grained memory protection, enforced either by hardware or by software. Hardware memory protection is used when the untrusted code is run in a separate address space. A software equivalent is Software Fault Isolation (SFI) (Wahbe, Lucco, Anderson, and Graham, 1993), in which the untrusted code is instrumented with run-time checks that prevent memory accesses outside a pre-configured memory range. In both of these cases, the communication between the untrusted code and the host system is expensive because it must involve copying of data into the untrusted-code address space.

Most of the challenges in typing low-level languages arise because we cannot abstract details of how high-level features are implemented. In JVM and MS-CLI, these problems are attenuated by keeping in the intermediate language the most troublesome high-level features, such as exceptions and dynamic dispatch.

In this chapter we examine instead an extreme situation, in which the untrusted code is written in assembly language. Correspondingly, the type system must be more expressive than would be necessary for similar programs written in a high-level language. In essence, a type checker for a low-level language verifies not only the well-typedness of the corresponding high-level constructs but also that the construct has been compiled correctly. Most of the mechanisms that we'll need can be expressed using types. **Note: cross-reference to TAL**. But since such a large part of low-level type checking is closer to program verification than type checking, we use in this chapter a method based on logic. Besides being an instructive alternative presentation of type systems and type checking, this approach will allow us to move seamlessly beyond type checking and into checking more complex properties of the untrusted code.

7.1 Overview of Proof Carrying Code

Proof-Carrying Code (PCC) (Necula, 1997; Necula and Lee, 1996) is a *general* mechanism that allows the host to check *quickly* and *easily* that the agent has certain safety properties. The key technical detail that makes PCC powerful is a requirement that the agent producer cooperates with the host by attaching to the agent code an "explanation" of why the code complies with the

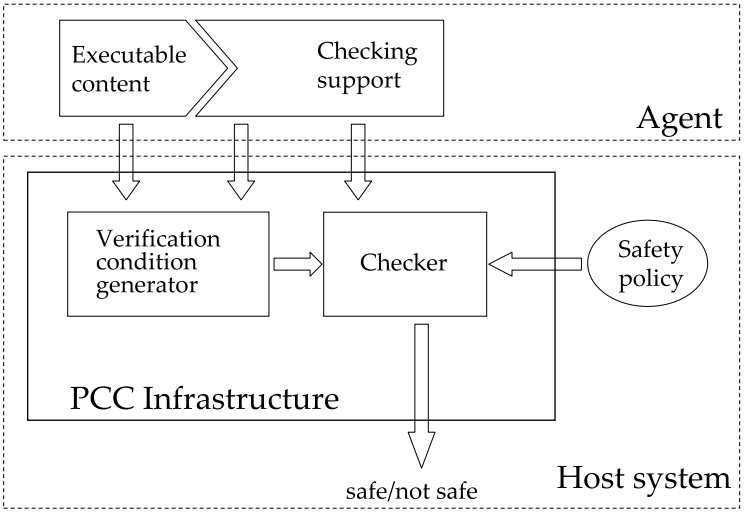


Figure 7-1 The high-level structure of the PCC architecture.

safety policy. Then all that the host has to do to ensure the safe execution of the agent is to define a framework in which the "explanation" must be conducted, along with a simple yet sufficiently strong mechanism for checking that (a) the explanation is acceptable (i.e., is within the established framework), that (b) the explanation pertains to the safety policy that the host wishes to enforce, and (c) that the explanation matches the actual code of the agent.

There are a number of possible forms of explanations each with its own advantages and disadvantages. Safety explanations must be precise and comprehensive, just like formal proofs. In fact, in this chapter, the explanations are going to be formal proofs encoded in such a way that they can be checked easily and reliably by a simple proof checker.

A high-level view of the architecture of a PCC system is shown in Figure 7-1. The agent contains, in addition to its executable content, checking-support data that allows the PCC infrastructure resident on the receiving host to check the safety of the agent. The PCC infrastructure is composed of two main modules. The verification-condition generator (VCGen) scans the executable content of the agent and checks directly simple syntactic conditions (e.g. that direct jumps are within the code boundary). Each time VCGen encounters an instruction whose execution could violate the safety policy, it asks the Checker module to verify that the dangerous instruction executes

safely in the actual current context. VCGen can be quite simple because it relies on the Checker to verify complex safety requirements. There are some cases, however, when VCGen might have to understand complex invariants of the agent code in order to follow its control and data flow. For example, VCGen must understand the loop structure of the agent in order to avoid scanning the loop body an unbounded number of times. Also, VCGen must be able to understand even obscure control flow, as in the presence of indirect jumps or function pointers. In such situations, VCGen relies on *code annotations* that are part of the checking support and are packaged with the agent. This puts most of the burden of handling the complex control-flow issues on the agent producer and keeps the VCGen simple.

The Checker module verifies for VCGen that all dangerous instructions are used in a safe context. The Checker module that we describe in this chapter requires that VCGen formulates the safety precondition of the dangerous instruction as a formula in a logic. We refer to these formulas as the *verification conditions*. Then the Checker simply expects to find in the checking-support data packaged with the agent a formal proof that the safety precondition is met. If the Checker verifies the validity of the verification-condition proofs for all dangerous instructions that VCGen identifies, then we have successfully verified that the execution of the code is guaranteed to be safe.

The PCC infrastructure that we describe here can be customized to check various safety policies. The "Safety Policy" element in Figure 7-1 is a collection of configuration data the specifies the precise logic that VCGen uses to encode the verification conditions, along with the trusted proof rules that can be used in the safety proofs supplied by the agent producer. It is important to separate the safety policy configuration data from the rest of the infrastructure both for conceptual and for engineering reasons. This architecture allows the infrastructure to work with multiple safety policies, without changing its implementation.

An Example Agent

In the rest of this chapter we explore the design and implementation details of the PCC infrastructure. The infrastructure can be configured to check many safety policies. In the example that we use here we check a simple type-safety policy for an agent written in a generic assembly language. The agent is a function that adds all the elements in a list containing either integers or pairs of integers. If this agent were written in OCaml, its source code might be as shown Figure 7-2.

In order to write the agent in assembly language we must decide what is the representation strategy for lists and for the `maybepair` type. We repre-

```
type maybepair = Int of int | Pair of int * int
let rec sum(acc : int, x : maybepair list) =
  match x with
  | nil → acc
  | Int i :: tail → sum(acc + i, tail)
  | Pair (l, r) :: tail → sum (acc + l + r, tail)
```

Figure 7-2 The OCaml source version of the example agent

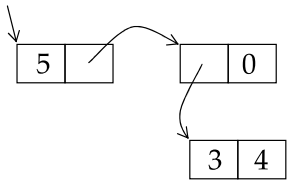


Figure 7-3 The concrete representation of the list `[Int 2; Pair (3, 4)]`

sent a list is as either the value 0 (for the empty list), or a pointer to a two-word memory area. The first word of the memory area contains a list element and the second element contains the tail of the list. In order to represent an element of type `maybepair` in an economical way we ensure that any element of kind `Pair`(x, y) is an even-valued pointer to a two-word memory area containing x and y. We represent an element of kind `Int` x as the integer $2x + 1$ (to ensure that it is odd and thus distinguishable from a pair). For example, the representation of the list `[Int 2; Pair (3, 4)]` has the concrete representation shown in Figure 7-3.

In our examples we will use the simple subset of a generic assembly language shown below. The expressions e contain arithmetic and logic operations involving constants and registers.

`set` r_x `to` e	assign the result of e to register r_x
`load` r_x `from` e	load r_x from address e
`store` e `to` e'	store the result of e to address e'
`goto` L	jump to a label L
`if` e `goto` L	branch to label L if e is true
`return`	return from the current function

Given our representation strategy and the choice of assembly language instructions, the code for the agent is shown Figure 7-4. On entry to this code fragment registers r_x and r_{acc} contain the value of the formal arguments x and `acc` respectively. The code fragment also uses temporary registers r_t

```
1  sum:                                          ; r_x : maybepair list
2  Loop:
3          if r_x ≠ 0 goto LCons                 ; Is r_x empty?
4          set r_R to r_acc
5          return
6  LCons:  load r_t from r_x                     ; Load the first data
7          if even(r_t) goto LPair
8          set r_t to r_t div 2
9          set r_acc to r_acc + r_t
10         goto LTail
11 LPair:  load r_s from r_t                     ; Get the first pair element
12         set r_acc to r_acc + r_s
13         load r_t from r_t + 4                 ; and the second element
14         set r_acc to r_acc + r_t
15 LTail:  load r_x from r_x + 4
16         goto Loop
```

Figure 7-4 The assembly language code for the function shown in Figure 7-2.

and r_s. To simplify somewhat the handling of the return instruction we use the convention that the return value is always contained in the register r_R.

The safety policy in this case requires that all memory reads be from pointers that are either non-null lists, in which case we can read either the first or the second field of a list cell, or from pointers that refer to elements of the `Pair` kind. In the case of a memory write, the safety policy constrains the values that can be written to various addresses as follows: in the first word of a list cell we can write either an odd value or an even value that is a pointer to an element of `Pair` kind, and in the second word of a list cell we can write either zero or a pointer to some list cell. There are no constraints on what we can write to the elements of a `Pair`.

The safety policy specifies not only requirements on the agent behavior but can also specify assumptions that the agent can make about the context of the execution. In the case of our agent, the safety policy might specify that the contents of the register r_x on entry is either zero or a pointer to a list cell. Also, the safety policy can allow the agent to assume that the value read from the first word of a list cell is either odd or otherwise a pointer to a `Pair` cell. Similarly, the agent can assume that the value it reads from the second word of a cell is either null or else a pointer to a list cell. We formalize this safety policy in the next section.

7.2 Formalizing the Safety Policy

At the core of a safety policy is a list of instructions whose execution may violate safety. For each such instruction, the safety policy specifies what is the verification condition that guarantees its safe execution. In the variant of PCC described here, the instructions that are handled specially are the memory operations along with the function calls and returns. This choice is hard coded in the verification-condition generator. However, the specific verification condition for each of these instructions is customizable. In such an implementation we can control very precisely what memory locations can be read, what memory locations can be written and what can be written into them, what functions we call and in what context, and in what context we return from a function. This turns out to be sufficient for a very large class of safety policies.

The customizable elements of the safety policy are the following:

- A language of symbolic expressions and formulas that can be used to express verification conditions.
- A set of function preconditions and postconditions for all functions that form the interface between the host and the agent.
- A set of proof rules for verification conditions.

In the rest of this section we describe in turn these elements.

The Syntax of the Logic

For this presentation we use a first-order language of symbolic expressions and formulas, as shown below:

Formulas	F	::=	$\texttt{true} \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid F_1 \Rightarrow F_2 \mid \forall x.F \mid \exists x.F$
			$\mid \texttt{addr}\ E_a \mid E_1 = E_2 \mid E_1 \neq E_2 \mid f\ E_1 \ldots E_n$
Expressions	E	::=	$x \mid \texttt{sel}\ E_m\ E_a \mid \texttt{upd}\ E_m\ E_a\ E_v \mid f\ E_1 \ldots E_n$

We consider here only the subset of logical connectives that we need for examples. In practice, a full complement of connectives is used. The formula $(\texttt{addr}\ E_a)$ is produced by VCGen as a verification condition for a memory read or a memory write to address E_a. This formula holds whenever E_a denotes a valid address. Formulas can also be constructed using a set of formula constructors that are specific to each safety policy.

The language of expressions contains variables and a number of constructors that includes integer numerals and arithmetic operators and can also be extended by safety policies. A notable expression construct is $(\texttt{sel}\ E_m\ E_a)$

that is used to denote the contents of memory address denoted by E_a in memory state E_m. The construct (`upd` E_m E_a E_v) denotes a new memory state that is obtained by storing the value E_v at address E_a in memory state E_m. For example, the contents of the address c in a memory that is obtained after writing the value 1 at address a and value 2 at address b in memory state m can be written:

```
sel (upd (upd m a 1) b 2) c
```

A safety policy extends the syntax of the logic by defining new expression and formula constructors. In particular, for our example agent we add constructors for encoding types and a predicate constructor for encoding the typing judgment:

Word types	W	::=	`int` | `ptr`{S} | `list` W | {x | F(x)}
Structure types	S	::=	W | W;S
Formulas	F	::=	... | E : W | `listinv` E_m

We distinguish among types the *word types*, whose values fit in a machine register or memory word. Pointers can point to an area of memory containing a sequence of words. The word type {x | F(x)} contains all those values for which the formula F is true (this type is sometimes called a comprehension or set type. **Note: reference to TAPL?**). The typing constructor ":" is written in infix notation. We also add the `listinv` formula constructor that will be used to state that the contents of the memory satisfies the representation invariant for our lists of pairs.

Using these constructors we can write the low-level version of the ML typing judgment x : `maybepair list` as

```
x : list {y | (even y) ⇒ y : ptr{int; int}}
```

In the rest of this section we use the abbreviation `mp_list` for the type `maybepair list`. Notice that we have built-in the recursive type of lists in our logic, in order to avoid the need for recursion at the level of the logic, but we choose to express the union and tuple types explicitly.

7.2.1 EXERCISE [RECOMMENDED, ★]: The singleton type is a type populated by a single value. Write the formula corresponding to the assertion that x has type singleton for the value v. Show also how you can write using our language of types the singleton type for the value v. □

7.2.2 EXERCISE [RECOMMENDED, ★]: Consider an alternative representation for the `maybepair` type. A value of this type is a pointer to a tagged memory

area containing a tag word followed either by another word encoding an `Int` if the tag value if 0, or by two words encoding a `Pair` if the tag value is 1. Write the formula corresponding to the assertion that `x` has this representation. □

The Preconditions and Postconditions

A PCC safety policy contains preconditions and postconditions for the functions that form the interface between the agent and the host. These are either functions defined by the agent and invoked by the host or library functions exported by the host for use by the agent. These preconditions and postconditions are expressed as logic formulas that use a number of formula and expression constructors specific to the safety policy.

Function preconditions and postconditions at the level of the assembly language are expressed in terms of argument and return registers, and thus specify also the calling convention. For verification purposes, we model the memory as a pseudo-register $\mathtt{r}_M$.

The function precondition and postcondition for our agent are:

$$\begin{array}{lcl} \mathtt{Pre}_{\mathtt{sum}} & = & \mathtt{r}_x : \mathtt{mp\_list} \wedge \mathtt{listinv}\ \mathtt{r}_M \\ \mathtt{Post}_{\mathtt{sum}} & = & \mathtt{listinv}\ \mathtt{r}_M \end{array}$$

The safety policy requires that the memory state be well-typed after the agent returns and allows the agent to assume that the memory state is well-typed when the host invokes it. Notice that we do not specify constraints on the integer arguments and results. This reflects our decision that any value whatsoever can be used as an integer.

7.2.3 EXERCISE [RECOMMENDED, ★]: Write the precondition and postcondition of a function of OCaml type `(int * int) → int list → int list`. The first argument is represented as a pointer to two integers. Consider that the return value is placed in register $\mathtt{r}_R$. □

7.2.4 EXERCISE [RECOMMENDED, ★]: Consider a function that takes in register $\mathtt{r}_1$ a pointer to a sequence of integer lists. The length of the sequence is passed in register $\mathtt{r}_2$. The function does not return anything. Write the precondition and postcondition for this function. □

Technically, the preconditions and postconditions are not well-formed formulas because they refer to registers. We can obtain formulas from them once we have a substitution of register names with expressions. We shall see later how this works out in detail.

$$\frac{F_1 \quad F_2}{F_1 \wedge F_2}\ (\text{ANDI}) \qquad \frac{F_1 \wedge F_2}{F_1}\ (\text{ANDEL}) \qquad \frac{F_1 \wedge F_2}{F_2}\ (\text{ANDER})$$

$$\begin{array}{c} F_1 \\ \vdots \\ F_2 \end{array} \Big/ \frac{}{F_1 \Rightarrow F_2}\ (\text{IMPI}) \qquad \frac{F_1 \Rightarrow F_2 \quad F_1}{F_2}\ (\text{IMPE})$$

$$\frac{A = A'}{\texttt{sel}\ (\texttt{upd}\ M\ A\ V)\ A' = V}\ (\text{MEM0}) \qquad \frac{A \neq A'}{\texttt{sel}\ (\texttt{upd}\ M\ A\ V)\ A' = \texttt{sel}\ M\ A'}\ (\text{MEM1})$$

Figure 7-5: Built-in proof rules

The Proof Rules

The last part of the safety policy is a set of proof rules that can be used to reason about formulas in our logic in general and about verification conditions in particular. In Figure 7-5 we show, in natural deduction form, a selection of the derivation rules for the first-order logical connectives. We show the conjunction introduction (ANDI) and the two conjunction eliminations (ANDEL, ANDER), and the similar rules for implication. We also have two rules (MEM0 and MEM1) that allow us to reason about the `sel` and `upd` constructors for memory expressions. These two rules should not be necessary for most type-based safety policies because in those cases all we care to know about the contents of a memory location is its type, not its value.

Note that in this set of base proof rules we do not yet specify when we can prove that `addr` holds. This is the prerogative of the safety-policy specific rules that we describe next.

Each safety policy can extend the built-in proof rules with new rules. In fact, such an extension is necessary if the safety policy uses formula constructors beyond the built-in ones. The rules specific to our safety policy are shown in Figure 7-6. We have rules that are used for reasoning about the type constructors: lists (NIL, CONS), set types (SET) and pointers to sequences (THIS and NEXT). These rules are similar to corresponding rules from other formulation of type systems with recursive types and tuples. **Note: cross-reference other chapters**. Next come two rules that are used to reason about

$$\frac{}{0 : \texttt{list}\ W}\ (\textsc{nil}) \qquad \frac{E : \texttt{list}\ W \qquad E \neq 0}{E : \texttt{ptr}\{W; \texttt{list}\ W\}}\ (\textsc{cons}) \qquad \frac{E : \{y \mid F(y)\}}{F(E)}\ (\textsc{set}) \qquad \frac{E : \texttt{ptr}\{W; S\}}{E : \texttt{ptr}\{W\}}\ (\textsc{this})$$

$$\frac{E : \texttt{ptr}\{W; S\}}{E + 4 : \texttt{ptr}\{S\}}\ (\textsc{next}) \qquad \frac{A : \texttt{ptr}\{W\} \qquad \texttt{listinv}\ M}{(\texttt{sel}\ M\ A) : W}\ (\textsc{sel})$$

$$\frac{\texttt{listinv}\ M \qquad A : \texttt{ptr}\{W\} \qquad V : W}{\texttt{listinv}\ (\texttt{upd}\ M\ A\ V)}\ (\textsc{upd}) \qquad \frac{A : \texttt{ptr}\{W\}}{\texttt{addr}\ A}\ (\textsc{ptraddr})$$

Figure 7-6: The proof rules specific to the example safety policy

the typing properties of reading and writing from pointers. The rule SEL says that the location referenced by a pointer to a word type in a well-typed memory state has the given word type. And the rule UPD is used to prove that a well-typed write preserves well-typedness of memory. These rules are similar to corresponding rules for reference types. **Note: cross-reference**.

Notice that these proof rules are exposing more concrete implementation details than the corresponding source-level rules. For example, the CONS rule specifies that a list cell is represented as a pointer to a pair of words, of which the first one stores the data and the second the tail of the list.

Finally, the PTRADDR rule relates the safety-policy specific formula constructors with the built-in `addr` memory safety formula constructor. This rule says that addresses that can be proved to have pointer type in our type system are valid addresses. And since this is the only rule whose conclusion uses the `addr` constructor, the safety policy is essentially restricting memory accesses to such addresses.

7.2.5 EXERCISE [RECOMMENDED, ★★]: Add a new `array` type constructor to our safety policy and write the proof rules for its usage. An array is represented as a pointer to a memory area that contains the number of elements in the array in the first word and then the array elements in order. Consider first the case where each element is a word type (as in OCaml), and then the case when each element can be a structure (as in C). □

We show here just a few of the rules for a simple safety policy. A safety policy for a full language can easily have over one hundred proof rules. For

example, the Touchstone (Colby, Lee, Necula, Blau, Plesko, and Cline, 2000) implementation of PCC for the Java type system has about 150 proof rules.

7.3 Verification-Condition Generation

So far we have showed how to set up the safety policy and we need now to describe a method to enforce it. An analogous situation in the realm of high-level type systems is when we have setup a type system, with a language of types and a set of typing rules, and we need to design a type checker for it. A type checker must scan the code and must know what typing rule to apply at each point in the code. Our PCC infrastructure accomplishes a similar task but this time the scanning of the code is separated from the decision of what safety policy proof rules to apply. The scanning is done by the verification-condition generator, which also identifies *what* must be checked for each instruction. *How* the check is performed is decided by the Checker module, with considerable help from the proof that accompanies the code. In a regular type checker there is no pressing need to separate code scanning from the construction of the typing derivation because the scanning process is often simple and the structure of the typing derivation follows closely that of the code. This is not true for low-level type checking. In fact, programs written in assembly language have very little structure.

To illustrate some of the difficulties of type checking low-level code, consider the following fragment of code written in ML, where x is a variable of type Tlist and the variable t occurs in the expression e also with type Tlist:

```
match x with
  _ :: t → e
```

A type checker for ML parses this code, constructs an abstract syntax tree (AST) and then it can verify its well-typedness in a relatively simple manner by traversing the AST. This is possible because the `match` expression packages in one construction all the elements that are needed for type checking: the expression to be matched, the patterns with the variables they define, and the bodies of the cases.

Consider now one particular compilation of this code fragment, as shown below:

```
set r_t to r_x
set r_t to r_t + 4
if r_x = 0 goto LNil
load r_t from r_t
...
```

We assume that the variable x is allocated to register r_x and that the expression e is compiled with the assumption that, on entry, the variable t is

allocated to the register $\mathtt{r_t}$. We observe that the code for compiling the `match` construct is spread over several non-contiguous instructions mixed with the instructions that implement the cases themselves. This is due to the intrinsically sequential nature of assembly language. It would be hard to implement a type checker for assembly language that identifies the code for the `match` by recognizing patterns, as a source-level type checker does. Furthermore, such a type checker would be sensitive to code generation and optimization choices.

Another difficulty is that some high-level operations are split into several small operations. For example the extraction of the tail of the list is separated into the computation of an address in register $\mathtt{r_t}$ and a memory load. We cannot check one of the two instructions in isolation of the other because they both can be used in other contexts as well. Furthermore, it is not sufficient to type check the addition $\mathtt{r_t} + 4$ as we would do in a high-level language (i.e., verify that both operands have compatible arithmetic types). Instead we need to remember that we added the constant 4 to the contents of register $\mathtt{r_x}$, so that when we reach the load instruction we can determine that we are loading the second word from a list cell. Additionally, our type-checking algorithm has to be path sensitive because the outcomes of conditional expressions sometimes determine the type of values. In our example, if the conditional falls through then we know that $\mathtt{r_x}$ points to a list cell and therefore that $\mathtt{r_t}$ points to the second element in a list cell. If, however, the conditional jumps to `LNil` then we cannot even assign a type to $\mathtt{r_t}$ after the addition.

Yet another complication with assembly language is that, unlike in high-level languages, we cannot count on a variable having a single type throughout its scope. In assembly language the registers play the role of variables and since there is a finite number of them, compilers reuse them aggressively to hold different data at different program points. In our example, the register $\mathtt{r_t}$ is used before the load to hold both a pointer to a memory location containing a list and after the load instruction to hold a list. We must thus keep different types for registers for different program points.

There are a number of approaches for overcoming these difficulties. All of them do maintain different types for registers at different program points but differ on how they handle the dependency on conditionals and the splitting of high-level operations into several instructions. At one extreme is the Java Bytecode Verifier (Lindholm and Yellin, 1997) which typechecks programs written in the Java Virtual Machine Language (JVML). The JVML is relatively high-level and maintains complicated operations bundled in high-level constructs. For instance, in JVML you cannot separate the address computation from the memory access itself. In the context of our example this means that the addition and the load instruction would be expressed as one bytecode in-

struction. The JVML is designed such that the outcome of conditionals does not matter for typechecking. For example, array-bounds checks and pointer null-checks are bundled with the memory access in high-level bytecode instructions. This approach simplifies the type-checking problem but has the disadvantage that the agent producer cannot really do much optimization. Also this approach puts more burden on the code receiver for compiling and optimizing the code.

Another approach is Typed Assembly Language (TAL), described in the previous chapter, where a more sophisticated type system is used to keep track of the intermediate result of unbundled instructions. **Note: can TAL as described in the previous chapter give a type to r_t before the check?** But even in TAL some low-level instructions are treated as bundles for the purpose of verification. Examples are memory allocation and array accesses.

Here we are going to describe a type checking method that can overcome the difficulties described above. The method is based on *symbolic evaluation* and it was originally used in the context of program verification. The method is powerful enough to verify full correctness of a program, not just its well-typedness, which will come in handy when we consider safety policies beyond type safety.

Symbolic Evaluation

In order to introduce symbolic evaluation consider the code fragment from above but without the conditional.

```
set r_t to r_x
set r_t to r_t + 4
load r_t from r_t
```

This fragment does exhibit the problems due to reuse of registers with different types and the splitting of high-level operations into low-level instructions. We have already observed that it is more important to remember the effect of the addition instruction than it is to type check it immediately as we see it. In fact, we are going to postpone all checking as much as possible and are going to focus on "remembering" the effect of instructions instead. Observe that if we allow arbitrary complex operands in our instructions, we can rewrite the above code sequence as follows:

```
load r_t from r_x + 4
```

Now, when we have removed all intermediate computations the address computation is now bundled with the memory access, and we can actually perform the usual pattern matching to recognize what typing rule to apply. Symbolic evaluation is a technique that has the effect of collecting the results of intermediate computations to create the final result as a complex

expression whose meaning is equivalent to the entire computation. A symbolic evaluator is an interpreter that maintains for each register a symbolic expression. We will use the symbol σ to range over *symbolic states*, which are mappings from register names to symbolic expressions. The symbolic state is initialized with a distinct fresh variable for each register, to model our complete lack of information about the initial values of the registers. For our example involving the registers $\mathtt{r}_t$ and $\mathtt{r}_x$ the initial symbolic state is

$$\sigma_0 = \{\mathtt{r}_t = t, \mathtt{r}_x = x\}$$

where t and x are distinct fresh variables. Technically, this symbolic state says that at the given program point the following invariant holds:

$$\exists t.\exists x.\mathtt{r}_t = t \wedge \mathtt{r}_x = x$$

The symbolic evaluator proceeds forward to interpret the instructions and modifies the symbolic state as specified by the instruction. We show below the sequence of symbolic states as the symbolic evaluator proceeds to interpret our sequence of instructions.

$\sigma = \{\mathtt{r}_t = t, \mathtt{r}_x = x\}$

`set` $\mathtt{r}_t$ `to` $\mathtt{r}_x$

$\sigma = \{\mathtt{r}_t = x, \mathtt{r}_x = x\}$

`set` $\mathtt{r}_t$ `to` $\mathtt{r}_t + 4$

$\sigma = \{\mathtt{r}_t = x + 4, \mathtt{r}_x = x\}$

`load` $\mathtt{r}_t$ `from` $\mathtt{r}_t$

When the instruction "`set` $\mathtt{r}_t$ `to` $\mathtt{r}_x$" is processed, the symbolic evaluator looks up the value of $\mathtt{r}_x$ in the current symbolic state and then sets $\mathtt{r}_t$ to that value. Notice how at the time the load instruction is processed the symbolic evaluator can figure out that the address being accessed is $x + 4$.

In order to handle memory reads and writes we use a pseudo-register $\mathtt{r}_M$ and the `sel` and `upd` constructors introduced in Section 7.2. Assuming that the initial symbolic value of the memory is m, then the symbolic state after the load instruction in the previous example is:

$$\mathtt{sel}\ m\ (x + 4)$$

For memory loads and writes the symbolic evaluator also emits the required verification conditions using the `addr` constructor. For example, the verification condition for the load instruction would be $\mathtt{addr}\ (x + 4)$.

Another element of interest is the handling of conditionals. In order to allow for path sensitive checking the symbolic evaluator maintains, in addition to the symbolic state, a list of assumptions about the state. These assumptions are simply formulas involving the same existentially quantified

variables that the symbolic state uses. As the symbolic evaluator follows the branches of a conditional, it extends the list of assumptions with formulas that capture the outcome of the conditional expression.

If we now add back the conditional instruction in our example, the symbolic state and the set of assumptions (initially A) at each point are shown below:

$$\sigma = \{r_t = t, r_x = x, r_M = m\}, A$$

```
set r_t to r_x
```

$$\sigma = \{r_t = x, r_x = x, r_M = m\}, A$$

```
set r_t to r_t + 4
```

$$\sigma = \{r_t = x + 4, r_x = x, r_M = m\}, A$$

```
if r_x = 0 goto LNil
```

$$\sigma = \{r_t = x + 4, r_x = x, r_M = m\}, A \wedge x \neq 0$$

```
load r_t from r_t
```

$$\sigma = \{r_t = \texttt{sel}\ m\ (x + 4), r_x = x, r_M = m\}, A \wedge x \neq 0$$

```
...
```

LNil: $\sigma = \{r_t = x + 4, r_x = x, r_M = m\}, A \wedge x = 0$

The symbolic state immediately before the load instruction essentially states that the following invariant holds at that point:

$$\exists t.\exists x.r_t = x \wedge r_x = x + 4 \wedge r_M = m \wedge A \wedge x \neq 0$$

This means that the Checker module would have to check the following verification condition for the load instruction:

$$\forall t.\forall x.(r_t = x \wedge r_x = x + 4 \wedge r_M = m \wedge A \wedge x \neq 0) \Rightarrow \texttt{addr}\ (x + 4)$$

7.3.1 EXERCISE [RECOMMENDED, ★]: Consider the following two code fragments. The one on the right has been obtained from the one on the left by performing a few simple local optimizations. First, we did register allocations, by renaming register r_a, r_b, r_c, and r_d to r_1, r_2, r_3 and r_4 respectively. Then we removed the dead instruction from line 1. We performed copy propagation followed by common subexpression elimination in line 5. Finally, we performed instruction scheduling by moving the instruction from line 3 to be the last in the block.

```
1 set r_a to 2                    set r_1 to r_2 + 1
2 set r_a to r_b + 1              set r_4 to r_1
3 set r_c to r_a + 2              set r_3 to r_1 + 2
4 set r_d to 1
5 set r_d to r_b + r_d
```

Show that the result of symbolic evaluation at the end of the two basic

blocks is identical if you start with symbolic states $\{r_b = b\}$ and $\{r_2 = b\}$ respectively. This suggests that symbolic evaluation is insensitive to some common optimizations. □

7.3.2 EXERCISE [RECOMMENDED, ★]: Now consider the code fragment shown in the left of Exercise 7.3.1 and add the instruction "`set` r_a `to 3`" immediately before line 5. In this case it is not correct to perform common-subexpression elimination. Show now that the result of symbolic evaluation is different for the modified code fragment and the transformed code from Exercise 7.3.1. This suggests that symbolic evaluation can be use to verify the result of compiler optimizations. This technique is in fact so powerful that it can be used to verify most optimizations that the GNU C compiler performs (Necula, 2000). □

Before we can give a complete formal definition of the VCGen, we must consider what happens in the cases when the symbolic evaluator should not follow directly the control-flow of the program. Two such cases are for loops (when following the control-flow would make VCGen loop forever) and for functions (when it is desirable to scan the body of a function only once). In order to handle those cases, VCGen needs some assistance from the agent producer, in the form of code annotations.

The role of program annotations

The VCGen module attempts to execute the untrusted program symbolically in order to signal all potentially unsafe operations. To make this execution possible in finite time and without the need for conservative approximations on the part of VCGen, we require that the program be annotated with invariant predicates. At least one such invariant must be specified for each cycle in the program's control-flow graph. An easy but conservative way to enforce such a constraint is to require an invariant annotation for every backward branch target.

The agent code shown in Figure 7-4 has one loop whose body starts at the label `Loop`. There must be one invariant annotation somewhere in that loop. Let us say that the agent producer places the following invariant at label `Loop`:

$$\texttt{Loop: INV} = r_x : \texttt{mp\_list} \wedge \texttt{listinv}\ r_M$$

The invariant annotation says that whenever the execution reaches the label `Loop` the contents of register r_x is a list. It also says that the contents of the memory satisfies the representation invariants of lists. Just like the preconditions and postconditions the invariants can refer to register names.

A valid question at this point is who discovers this annotation and how. There are several possibilities. First, annotations can be inserted by hand by the programmer. This is the only alternative when the agent code is programmed directly in assembly language or when the programmer wants to hand-optimize the output of a compiler. It is true that this method does not scale well, but it is nevertheless a feature of PCC that the code receiver does not care whether the code is produced by a trusted compiler and will gladly accept code that was written or optimized by hand.

Another possibility is that the annotations can be produced automatically by a certifying compiler. For our simple type safety policy the only annotations that are necessary consist of type declarations for the live registers at that point. See (Necula, 1998) for more details.

Finally, note that the invariant annotations are required but cannot be trusted to be correct as they originate from the same possibly untrusted source as the code itself. Nevertheless, VCGen can still use them safely, as described in the next section.

7.3.1 The verification-condition generator

Now we have all the elements necessary to describe the verification-condition generator for the case of one function whose precondition and postcondition are specified by the safety policy. We will assume that each invariant annotation occupies one instruction slot, even though in practice they are stored in a separate region of the agent. Let *Inv* be the partial mapping from program counters to invariant predicates. If $i \in Dom(Inv)$ then there is an invariant Inv_i at program counter i. Next, for a more uniform treatment of functions and loops, we will consider that the first instruction in each agent function is an invariant annotation with the precondition predicate. In our example, this means that $Inv_1 = r_x : \texttt{mp\_list} \wedge \texttt{listinv}\ r_M$. This, along with the loop invariant (with the same predicate) at index 2 are all the loop invariants in our example. Thus, $Dom(Inv) = \{1, 2\}$, and the first few lines of our agent example are modified as follows:

```
1 sum:    INV r_x : mp_list ∧ listinv r_M
2 Loop:   INV r_x : mp_list ∧ listinv r_M
3         if r_x ≠ 0 goto LCons                    ; list is empty
```

Given a symbolic state σ and an expression e that contains references to register names, we write $(\sigma\ e)$ to denote the result of substituting the register names in e with the expressions given by σ. We extend this notation to formulas F that refer to register names (e.g., function preconditions or post-

conditions, or loop invariants). We also write $\sigma[r \leftarrow e]$ to denote a symbolic state that is as σ but with register r mapped to e.

We write Π_i to denote the instruction (or annotation) at the program counter i.

The core of the verification-condition generator is a symbolic evaluation function *SE* that given a value i for the program counter and a symbolic state σ, produces a formula that captures all of the verification conditions from the given program counter until the next return instruction or invariant. The definition of the *SE* function is shown below:

$$SE(i,\sigma) = \begin{cases} SE(i+1, \sigma[r \leftarrow \sigma\ e]) & \text{if } \Pi_i = \texttt{set}\ r\ \texttt{to}\ e \\ (\sigma\ e) \Rightarrow SE(L,\sigma) \wedge & \text{if } \Pi_i = \texttt{if}\ \ e\ \texttt{goto}\ \ L \\ \quad (\texttt{not}\ (\sigma\ e)) \Rightarrow SE(i+1,\sigma) & \\ \texttt{addr}\ (\sigma\ a) \wedge & \text{if } \Pi_i = \texttt{load}\ r\ \texttt{from}\ a \\ \quad SE(i+1, \sigma[r \leftarrow (\sigma\ (\texttt{sel}\ r_M\ a))]) & \\ \texttt{addr}\ (\sigma\ a) \wedge & \text{if } \Pi_i = \texttt{store}\ e\ \texttt{at}\ a \\ \quad SE(i+1, \sigma[r_M \leftarrow (\sigma\ (\texttt{upd}\ r_M\ a\ e))] & \\ \sigma\ \mathit{Post} & \text{if } \Pi_i = \texttt{return} \\ \sigma\ I & \text{if } \Pi_i = \mathit{Inv}\ I \end{cases}$$

Symbolic evaluation is defined by cases analysis of the instruction contained at a give program counter. Symbolic evaluation is undefined for values of the program counter that do not contain a valid instruction. In the case of a `set` instruction, the symbolic evaluator substitutes the current symbolic state into the right-hand side of the instruction and then uses the result as the new value of the destination register. Then the symbolic evaluator continues with the following instruction. For a conditional, the symbolic evaluator adds the proper assumption about the outcome of the conditional expression. Memory operations are handled like assignments but with the generation of additional verification conditions.

When either the return instruction or an invariant is encountered, the symbolic evaluator stops with a predicate obtained by substituting the current symbolic state into the postcondition or the invariant formula. The symbolic evaluator also ensures (using a simple check not shown here) that each loop in the code has at least one invariant annotation. This ensures the termination of the *SE* function.

What remains to be shown is how the verification-condition generator uses the *SE* function. For each invariant in the code, VCGen starts a symbolic evaluation with a symbolic state initialized with distinct variables. Assuming that the set of registers is $\{r_1, \ldots, r_n\}$, we define the *global verification con-*

1: Generate fresh values $r_M = m_0$, $r_R = r_0$, $r_x = x_0$, $r_{acc} = acc_0$, $r_t = t_0$ and $r_s = s_0$

1: Assume Invariant $\underline{x_0 : \texttt{mp\_list}}$

$\underline{\texttt{listinv}\ m_0}$

2: Invariant $\boxed{x_0 : \texttt{mp\_list}}$

$\boxed{\texttt{listinv}\ m_0}$

2: Generate fresh values $r_M = m_0$, $r_R = r_0$, $r_x = x_0$, $r_{acc} = acc_0$, $r_t = t_0$ and $r_s = s_0$

2: Assume Invariant $\underline{x_0 : \texttt{mp\_list}}$

$\underline{\texttt{listinv}\ m_0}$

3: Branch 3 taken $\underline{x_0 \neq 0}$

6: Check load $\boxed{\texttt{addr}\ x_0}$

7: Branch 7 taken $\underline{\texttt{even}\ (\texttt{sel}\ m_0\ x_0)}$

11: Check load $\boxed{\texttt{addr}\ (\texttt{sel}\ m_0\ x_0)}$

13: Check load $\boxed{\texttt{addr}\ ((\texttt{sel}\ m_0\ x_0) + 4)}$

15: Check load $\boxed{\texttt{addr}\ (x_0 + 4)}$

16: Goto Loop

2: Invariant $\boxed{(\texttt{sel}\ m_0\ (x_0 + 4)) : \texttt{mp\_list}}$

$\boxed{\texttt{listinv}\ m_0}$

7: Branch 7 not taken $\underline{\texttt{odd}\ (\texttt{sel}\ m_0\ x_0)}$

10: Goto LTail

15: Check load $\boxed{\texttt{addr}\ (x_0 + 4)}$

16: Goto Loop

2: Invariant $\boxed{(\texttt{sel}\ m_0\ (x_0 + 4)) : \texttt{mp\_list}}$

$\boxed{\texttt{listinv}\ m_0}$

3: Branch 3 not taken $\underline{x_0 = 0}$

5: Return $\boxed{\texttt{listinv}\ m_0}$

Figure 7-7 The sequence of actions taken by VCGen. We show on the left the program points and a brief description of each action. Some actions result in extending the stack of assumptions that the Checker is allowed to make. These assumptions are shown underlined and with an indentation level that encodes the position in the stack of each assumption. Thus an assumption at a given indentation level implicitly discards all previously occurring assumptions at the same or larger indentation level. Finally, we show right-justified and boxed the checking goals submitted to the Checker.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x_0 : \texttt{mp\_list} \quad x_0 \neq 0}}{x_0 : \texttt{ptr}\{\texttt{maybepair};\texttt{mp\_list}\}}\,\text{CONS}}{\texttt{listinv}\ m_0 \qquad x_0 : \texttt{ptr}\{\texttt{maybepair}\}}\,\text{THIS}}{(\texttt{sel}\ m_0\ x_0) : \texttt{maybepair}}\,\text{SEL}}{\texttt{even}\,(\texttt{sel}\ m_0\ x_0) \Rightarrow (\texttt{sel}\ m_0\ x_0) : \texttt{ptr}\{\texttt{int};\texttt{int}\}}\,\text{SET} \qquad \texttt{even}\,(\texttt{sel}\ m_0\ x_0)}{(\texttt{sel}\ m_0\ x_0) : \texttt{ptr}\{\texttt{int};\texttt{int}\}}\,\text{IMPE}}{(\texttt{sel}\ m_0\ x_0) : \texttt{ptr}\{\texttt{int}\}}\,\text{THIS}}{\texttt{addr}\,(\texttt{sel}\ m_0\ x_0)}\,\text{RD}$$

Figure 7-8 The proof of the verification condition `addr (sel` m_0 x_0`)` using the assumptions x_0 : `mp_list`, `listinv` m_0 and `even (sel` m_0 x_0`)`.

dition VC as follows:

$$\begin{aligned} VC \quad = \quad & \textstyle\bigwedge_{i \in Dom(Inv)} \forall x_1 \ldots x_n. \quad \sigma_0\, Inv_i \Rightarrow SE(i+1, \sigma_0) \\ & \text{where } \sigma_0 = \{r_1 = x_1, \ldots, r_n = x_n\} \end{aligned}$$

Essentially the VCGen evaluates symbolically every path in the program that connects two invariants or an invariant and a return instruction. In Figure 7-7 we show the operation of the VCGen algorithm on the agent code from Figure 7-4 (after we have added the invariant annotations for the precondition and the loop, as explained at the beginning of this section). There are two invariants (in lines 1 and 2) and for each one we generate fresh new variables for registers, we assume that the invariant holds, and then we start the symbolic evaluator. For the first invariant the symbolic evaluator when starting in line 2 encounters an invariant and terminates.

Every boxed formula shown flushed right in Figure 7-7 is a verification condition that VCGen produces and the Checker module has to verify for some arbitrary values of the initial variables.

Notice that the invariant formulas are used both as assumptions and as verification conditions. There is a strong similarity between the role of invariants and predicates in a proof by induction. In the latter case the predicate is assumed to hold and with this assumption we must prove that it holds for a larger value in a well founded order. This effectively ensures that the invariant formulas are preserved through an arbitrary execution from one invariant point to another.

Let us consider now how one would prove the verification conditions. The first interesting one is the `addr` from line 6. Let

$$\texttt{maybepair} \stackrel{\text{def}}{=} \{y \mid \texttt{even}(y) \Rightarrow y : \texttt{ptr}\{\texttt{int}; \texttt{int}\}\}$$

To construct its proof we first derive x_0 : `ptr {maybepair;mp_list}` using the rule CONS with the assumptions x_0 : `mp_list` and $x_0 \neq 0$. Then we can derive `addr` x_0 using the rule RD.

A more interesting case is that of proving `addr (sel` m_0 x_0`)` from the assumptions x_0 : `mp_list`, `listinv` m_0 and `even (sel` m_0 x_0`)`. This proof is shown in Figure 7-8.

7.3.3 EXERCISE [★★]: Construct the the proof of the verification condition corresponding to the loop invariant from line 2. You must prove that `sel` m_0 (x_0+4) : `mp_list` from the assumptions x_0 : `mp_list`, `listinv` m_0 and `even (sel` m_0 x_0`)`. □

7.3.4 EXERCISE [★★★]: Notice that we have to prove that `sel` m_0 x_0 : `ptr{int}` several times as a step in proving several of the verification conditions from Figure 7-7. Show how you can add an invariant to the program to achieve the effect of proving this fact only once. □

We have been arguing that symbolic evaluation is just an alternative method for type checking, with additional benefits for checking more complex safety policies. Since there is a simple type checker at the source level for our type system, it seems reasonable to wonder whether we could hope to build automatically the proofs of these verification conditions. This is indeed possible for such type-based safety policies. Consider for instance how the proof shown in Figure 7-8 could be constructed through a goal-directed manner. The goal is an `addr` formula, and we observe that only the RD among our safety policy specific rules (shown in Figure 7-6) has a conclusion that matches the goal. The subgoal now is `(sel` m_0 x_0`) : ptr{int}`. In order to prove that the result of reading from a memory location has a certain type, we must prove that the memory is well-typed and the address has some pointer type. When we try to prove that x_0 has a pointer type we find among the assumptions that x_0 : `mp_list`. The remaining steps can be easily constructed by a theorem prover that knows the details of the type system. This general strategy was used successfully to construct a simple theorem prover that can build automatically and efficiently proofs of verification conditions for the entire Java type safety policy (Colby, Lee, Necula, Blau, Plesko, and Cline, 2000).

7.3.5 EXERCISE [★★★]: Extend the verification-condition generator approach described here to handle a new function call instruction `call L`, where L is a label that is considered the start of a function. For each such function there is a precondition and a postcondition. Make the simplifying assumption that the `call` instruction saves the return address and a set of callee-save registers on a special stack that cannot be manipulated directly by the program. A `ret` instruction always returns to the last return address saved on the stack and also restores the callee-save registers. □

7.3.6 EXERCISE [★★]: It is sometimes useful to use more kinds of annotations in addition to the loop invariants. For example, the agent producer might know that a certain point in the code is not reachable, as is the case for the label `L1` in the code fragment shown below:

```
        call myexit
L1:     UNREACHABLE
        ...
```

In such a case it is useful to add an annotation `UNREACHABLE` to signal to the symbolic evaluator that it can stop the evaluation at that point. Show how can you change the symbolic evaluator to handle these annotation without allowing the agent producer to "lie" about reachability of code. □

7.3.7 EXERCISE [★★]: Extend the symbolic evaluator to handle the indirect jump instruction `jump at` *e*, where *e* must evaluate to a valid program counter. Indirect jumps are often use to implement efficiently `switch` statements, in which case the destination address is one of a statically-known set of labels. Assume that immediately after the indirect jump instruction there is an annotation of the form `JUMPDEST(L1, L2)` to declare that the destination address is one of `L1` or `L2`. □

7.3.8 EXERCISE [★★★★]: Extend the symbolic evaluator to handle stack frames. The basic idea is that there is a dedicated stack pointer register r_{SP} that always points at the last used stack word. This register can only be incremented or decremented by a constant amount. You can ignore stack overflow issues. The stack frame for a function has a fixed size that is declared with an annotation. The only accesses to it are through the stack pointer register along with a constant offset. The key insight is that since there is no aliasing to the stack frame slots they can be treated as pseudo registers. Make sure you handle properly the overlapping of stack frames at function calls. □

This completes our simplified account of the operation of VCGen. Note that the VCGen defined here constructs a global verification condition that it then passes to the Checker module. This approach, while natural and easy to

describe, turns out to be too wasteful. For large examples on the order of millions of instructions it is quite common for this monolithic formula to require hundreds of megabytes for storage, slowing down the checking process considerably.In some of the largest examples not even a virtual address space of 1Gb could hold the whole formula. A high-level type checker that would construct an explicit typing derivation would be just as wasteful. A more efficient VCGen architecture passes to the Checker module each verification condition as it is produced. After the checker validates it, the verification condition is discarded and the symbolic evaluation resumes. This optimization might not seem interesting from a scientific point of view but it is illustrative of a number of purely engineering details that must be addressed to make PCC scalable.

7.4 Soundness Proof

In this section we prove that the type checking technique presented so far is sound, in the sense that "well-typed programs cannot go wrong". More precisely, we prove that if the global verification condition for a program is provable using the proof rules given by the safety policy, then the program is guaranteed to execute without violating memory safety. The method we use is similar to that used for type systems for high-level languages. We define formally the operational semantics of the assembly language, along with the notion of "going wrong". A little bit more difficult is to formalize the notion of well-typed programs. In high-level type systems there is a direct connection between the typing derivations and the abstract-syntax tree of the program. In our case, the connection is indirect: we first use a verification-condition generator and then we exhibit a derivation of the global verification condition using the safety policy proof rules. In order to reflect this staging in the operation of our type checker we split the soundness proof into a proof of soundness of the set of safety policy rules and a proof of soundness of the VCGen algorithm.

Soundness of the Safety Policy

The ultimate goal of our safety policy is to provide memory safety. In order to prove that our typing rules enforce memory safety we must first define the semantics of the expression and formula constructors that we have defined.

The semantic domain for the expressions is the set of integers[1] , except for

1. A more accurate model would use integers modulo 2^{32} in order to reflect the finite range of integers that are representable as machine words.

the memory expressions that we model using partial maps from integers to integers.

Next we observe that the typing formulas involving pointer types and the `listinv` formulas have a well-defined meaning only in a given context that assigns types to addresses. Without such a context we cannot possibly establish, for example, whether or not the formula "$8 : \texttt{ptr}\,\{\texttt{int}\}$" holds. The necessary context is a mapping $\mathcal{M}$ from a *valid* address to the word type of the value stored at that address. Since we do not consider allocation or deallocation our type system ensures that the mapping $\mathcal{M}$ remains constant throughout the execution of the program.

We write $\models_{\mathcal{M}} F$ when the formula F holds in the memory typing $\mathcal{M}$. A few of the most interesting cases from the definition of $\models_{\mathcal{M}}$ are shown below:

$$
\begin{array}{lll}
\models_{\mathcal{M}} F_1 \wedge F_2 & \texttt{iff} & \models_{\mathcal{M}} F_1 \text{ and } \models_{\mathcal{M}} F_2 \\
\models_{\mathcal{M}} F_1 \Rightarrow F_2 & \texttt{iff} & \text{whenever } \models_{\mathcal{M}} F_1 \text{ then } \models_{\mathcal{M}} F_2 \\
\models_{\mathcal{M}} \forall x.F(x) & \texttt{iff} & \forall e \in \mathbb{Z}.\, \models_{\mathcal{M}} F(e) \\
 & & \\
\models_{\mathcal{M}} a : \texttt{int} & \texttt{iff} & a \in \mathbb{Z} \\
\models_{\mathcal{M}} a : \texttt{list}\ W & \texttt{iff} & a = 0 \vee (\mathcal{M}(a) = W \wedge \mathcal{M}(a+4) = \texttt{list}\ W) \\
\models_{\mathcal{M}} a : \texttt{ptr}\,\{S\} & \texttt{iff} & \forall i.0 \leq i < |S| \Rightarrow \mathcal{M}(a + 4 * i) = S_i \\
\models_{\mathcal{M}} a : \{y \mid F(y)\} & \texttt{iff} & \models_{\mathcal{M}} F(a) \\
 & & \\
\models_{\mathcal{M}} \texttt{listinv}\ m & \texttt{iff} & \forall a \in \mathit{Dom}(\mathcal{M}).\, \models_{\mathcal{M}} m\ a : \mathcal{M}(a) \\
\models_{\mathcal{M}} \texttt{addr}\ a & \texttt{iff} & a \in \mathit{Dom}(\mathcal{M})
\end{array}
$$

In the above definition we used the notation $|S|$ for the length of a sequence of word types S, and S_i for the i^{th} element of the sequence.

With these definitions we can now start to prove the soundness of the derivation rules. Given a rule with variables $x_1, \ldots, x_n$, premises $H_1, \ldots, H_m$ and conclusion C, we must prove

$$\models_{\mathcal{M}} \forall x_1.\forall x_2.\ldots.\forall x_n.(H_1 \wedge \cdots \wedge H_m) \Rightarrow C$$

For example, the soundness of rule SEL requires proving the following fact:

$$\models_{\mathcal{M}} \forall a.\forall W.\forall m.(a : \texttt{ptr}\,\{W\}) \wedge (\texttt{listinv}\ m) \Rightarrow (\texttt{sel}\ m\ a) : W$$

From the first assumption we derive that $\mathcal{M}(a) = W$. From the second assumption we derive that $\models_{\mathcal{M}} m\ a : W$ and since $\models_{\mathcal{M}} (\texttt{sel}\ m\ a) = m\ a$ we obtain the desired conclusion.

7.4.1 EXERCISE [RECOMMENDED, ★]: Prove that the rules `cons` and `next` are sound. □

7.4.2 EXERCISE [★★]: Prove the soundness of the remaining rules shown in Figure 7-6. □

$$(i,\rho) \rightsquigarrow \begin{cases} (i+1, \rho[r_d \leftarrow \rho\ e]), & \textit{if} \quad \Pi_i = \texttt{set}\ r_d\ \texttt{to}\ e \\ (i+1, \rho[r_d \leftarrow \rho\ (\texttt{sel}\ r_M\ e)]), & \textit{if} \quad \Pi_i = \texttt{load}\ r_d\ \texttt{from}\ e \\ & \quad \textit{and} \quad \rho\ e \in \mathcal{A}\texttt{ddr} \\ (i+1, \rho[r_M \leftarrow \rho\ (\texttt{upd}\ r_M\ e_2\ e_1)]), & \textit{if} \quad \Pi_i = \texttt{write}\ e_1\ \texttt{to}\ e_2 \\ & \quad \textit{and} \quad \rho\ e_2 \in \mathcal{A}\texttt{ddr} \\ (L, \rho), & \textit{if} \quad \Pi_i = \texttt{if}\ e\ \texttt{goto}\ L \\ & \quad \textit{and} \quad \rho\ e \\ (i+1, \rho), & \textit{if} \quad \Pi_i = \texttt{if}\ e\ \texttt{goto}\ L \\ & \quad \textit{and} \quad \rho\ (\texttt{not}\ e) \\ (i+1, \rho), & \textit{if} \quad \Pi_i = \texttt{INV}\ I \end{cases}$$

Figure 7-9 The abstract machine for the soundness proof.

An Operational Semantics for Assembly Language

Next we formalize an operational semantics for the assembly language. We model the execution state as a mapping ρ from register names to values. Just like in the previous chapter the domain of values is $\mathbb{Z}$, except for the r_M register which takes as values partial mappings from $\mathbb{Z}$ to $\mathbb{Z}$. Since we do not consider allocation and deallocation the domain of the memory mapping does not change. Let $\mathcal{A}\texttt{ddr}$ be that domain. The operational semantics is defined only for programs whose memory accesses are only to addresses in the $\mathcal{A}\texttt{ddr}$ domain.

We write $(\rho\ e)$ for the result of evaluating in the register state ρ the expression e, which can refer to register names. We write $\rho[r_r \leftarrow v]$ for the new register state obtained after setting register r_r to value v in state ρ.

The operational semantics is defined in Figure 7-9 in the form of a small-step transition relation $(i, \rho) \rightsquigarrow (i, \rho')$ from a given program counter and register state to another such pair. Notice that the transition relation is defined for memory operations only if the referenced addresses aer valid.

We follow the usual convention and leave the transition relation undefined for those states where the execution is deemed unsafe. For instance, the transition relation is not defined if the program counter is outside the code area or if it points to an unrecognized instruction. More importantly, the transition relation is not defined if a memory access is attempted at an invalid address.

Soundness of Verification-Condition Generation

The soundness theorem for VCGen states that if the verification condition holds, and all addresses that are in $Dom(\mathcal{M})$ are valid addresses (i.e., they belong to $\mathcal{A}\mathtt{ddr}$), then the execution starting at the beginning of the agent in a state that satisfies the precondition will make progress either forever or until it reaches a return instruction in a state that satisfies the postcondition. What this theorem rules out is the possibility that the execution gets stuck either because it tries to execute an instruction at an invalid program counter or it tries to dereference an invalid address. The formal statement of the theorem is the following:

7.4.3 THEOREM [SOUNDNESS OF VCGEN]: Let ρ_1 be a state such that $\models_{\mathcal{M}} \rho_1$ *Pre*. If $Dom(\mathcal{M}) \subseteq \mathcal{A}\mathtt{ddr}$ and if $\models_{\mathcal{M}} VC$ then the execution starting at $(1, \rho_1)$ can make progress either forever or until it reaches a `return` instruction in state ρ, in which case $\models_{\mathcal{M}} \rho$ *Post*. □

We prove by induction on the number of execution steps that either we have reached the return instruction or else we can make further progress. As in all proofs by induction the most delicate issue is the choice of the induction hypothesis. Informally, our induction hypothesis is that for each execution state there is a "corresponding" state of the symbolic evaluator.

In order to express the notion of correspondence we must consider the differences between the concrete execution states ρ (mapping register names to values) and the symbolic evaluation states σ (mapping register names to symbolic expressions that use expression constructors and variables). To bridge these two notions of states we need a mapping τ from variables that appear in σ to values. For a symbolic expression e that contains variables, we write $\tau\ e$ for the result of replacing the variables in e as specified by τ and evaluating the result. Consequently, we write $\tau \circ \sigma$ for a mapping from register names to values that maps each register name $\mathtt{r}_i$ to the value $\tau\ (\sigma\ \mathtt{r}_i)$. Thus $\tau \circ \sigma$ is a concrete execution state.

The main relationship that we impose between ρ and σ is that there exists a mapping τ such that $\rho = \tau \circ \sigma$. The full induction hypothesis relates these states with the program counter i and is defined as follows:

$$IH(i, \rho, \sigma, \tau) \stackrel{\text{def}}{=} \rho = \tau \circ \sigma \wedge \models_{\mathcal{M}} \tau\ (SE(i, \sigma))$$

The core of the soundness proof is the following lemma:

7.4.4 THEOREM [PROGRESS]: Let Π be a program such that $\models_{\mathcal{M}} VC$ and $Dom(\mathcal{M}) \subseteq \mathcal{A}\mathtt{ddr}$. For any execution state (i, ρ) and σ and τ such that $IH(i, \rho, \sigma, \tau)$ then either:

- $\Pi_i = \texttt{return}$, and $\models_{\mathcal{M}} \rho\ \mathit{Post}$, or
- there exist new states ρ', σ' and a mapping τ' such that $(i, \rho) \rightarrow (i', \rho')$ and $\mathit{IH}(i', \rho', \sigma', \tau')$.

□

Proof: The proof is by case analysis on the current instruction. Since we have that $\models \tau\ \mathit{SE}(i, \sigma)$ we know that the program counter is valid and Π_i is a valid instruction. We show here only the most interesting cases.

Case: $\Pi_i = \texttt{return}$. In this case $\mathit{SE}(i, \sigma) = \sigma\ \mathit{Post}$ and from $\models_{\mathcal{M}} \tau\ \mathit{SE}(i, \sigma)$ along with $\rho = \tau \circ \sigma$ we can infer that $\models_{\mathcal{M}} \rho\ \mathit{Post}$.

Case: $\Pi_i = \texttt{load}\ r_d\ \texttt{from}\ e$. In this case $\mathit{SE}(i, \sigma) = \texttt{addr}\ (\sigma\ e) \wedge \mathit{SE}(i + 1, \sigma[r_d \leftarrow \sigma\ (\texttt{sel}\ r_{\mathcal{M}}\ e)])$. Let $\sigma' = \sigma[r_d \leftarrow \sigma\ (\texttt{sel}\ r_{\mathcal{M}}\ e)]$, $\rho' = \rho[r_d \leftarrow \rho\ (\texttt{sel}\ r_{\mathcal{M}}\ e)]$, $i' = i + 1$ and $\tau' = \tau$. In order to prove progress, we must prove $\rho\ e \in \mathcal{A}\mathtt{ddr}$. The induction hypothesis $\mathit{IH}(i, \rho, \sigma, \tau)$ ensures that $\models_{\mathcal{M}} (\texttt{addr}\ (\tau(\sigma\ e)))$, which in turn means that $(\rho\ e) \in \mathit{Dom}(\mathcal{M})$. Since we require that the memory typing be defined only on valid addresses we obtain the progress condition.

Next we have to prove that the induction hypothesis is preserved. The only interesting part of this proof is that $\tau' \circ \sigma' = \rho'$, which in turn requires proving that $\tau(\sigma\ (\texttt{sel}\ r_{\mathcal{M}}\ e)) = \rho\ (\texttt{sel}\ r_{\mathcal{M}}\ e)$. This follows from $\tau \circ \sigma = \rho$.

Case: $\Pi_i = \texttt{INV}\ I$. In this case $\mathit{SE}(i, \sigma) = \sigma\ I$. We know that $\models_{\mathcal{M}} \tau(\sigma\ I)$ and therefore $\models_{\mathcal{M}} \rho\ I$. The execution can always make progress for an invariant instruction and we must choose $i' = i + 1$ and $\rho' = \rho$. We know that $\models_{\mathcal{M}} \mathit{VC}$ and hence

$$\models_{\mathcal{M}} \forall x_1 \ldots . \forall x_n . \sigma_0\ I \Rightarrow \mathit{SE}(i + 1, \sigma_0)$$

where $\sigma_0 = \{r_1 = x_1, \ldots, r_n = x_n\}$. We choose $\sigma' = \sigma_0$ and τ' as follows:

$$\tau' = \{x_1 = \rho\ r_1, \ldots, x_n = \rho\ r_n\}$$

This ensures that $\rho = \tau' \circ \sigma'$ and also that $\models_{\mathcal{M}} \tau'\ \mathit{SE}(i + 1, \sigma')$, which completes this case of the proof.

□

7.4.5 EXERCISE [RECOMMENDED, ★]: Finish the proof of Theorem 7.4.4 by proving the remaining cases (assignment, conditional branch and memory write). □

The progress theorem constitutes the inductive case of the proof of the soundness theorem 7.4.3.

7.4.6 EXERCISE [★]: Prove Theorem 7.4.3. □

7.5 The Representation and Checking of Proofs

In previous sections we showed how verification-condition generation can be used to verify certain properties of low-level code. The soundness theorem states that VCGen constructs a valid verification condition for an agent program only if the agent meets the safety policy. One way to verify the validity of the verification condition is to witness a derivation using a sound system of proof rules. In PCC such a derivation must be attached to the untrusted code so that the Checker module can find and check it. For this to work properly in practice we need a framework for encoding proofs of logical formulas so that they are relatively compact and easy to check. We would like to have a framework and not just one proof checker for a given logic because the set of axioms and inference rules are likely to change frequently, as we adapt PCC to different safety policies. We would like to be able to adapt proof checking to other safety policies with as few changes to the infrastructure as possible. In this section we present a logical framework derived from the Edinburgh Logical Framework (Harper, Honsell, and Plotkin, 1993b), along with associated proof representation and proof checking algorithms, that have the following desirable properties:

- The framework can be used to encode judgments and derivations from a wide variety of logics, including first-order and higher-order logics.
- The implementation of the proof checker is parameterized by a high-level description of the logic. This allows a unique implementation of the proof checker to be used with many logics and safety policies.
- The proof checker performs a directed, one-pass inspection of the proof object, without having to perform search. This leads to a simple implementation of the proof checker that is easy to trust and install in existing extensible systems.
- Even though the proof representation is detailed, it is also compact.

The above desiderata are important not only for proof-carrying code but for any application where proofs are represented and manipulated explicitly. One such application is a proof-generating theorem prover. A theorem prover that generates an explicit proof object for each successfully proved predicate enables a distrustful user to verify the validity of the proved theorem by checking the proof object. This effectively eliminates the need to trust the soundness of the theorem prover at the relatively small expense of having to trust a much simpler proof checker.

The first impulse when designing efficient proof representation and validation algorithms is to specialize them to a given logic or a class of related logics. For example, we might define the representation and validation algorithms by cases, with one case for each proof rule in the logic. This approach has the major disadvantage that a new representation and validation algorithm has to be designed and implemented for each logic. To make matters worse, the size of such proof checking implementations grow with the number of proof rules in the logic. We would prefer instead to use general algorithms that are parameterized by a high-level description of the particular logic of interest.

We choose the Edinburgh Logical Framework (LF) (Harper, Honsell, and Plotkin, 1993b) as the starting point in our quest for efficient proof manipulation algorithms because it scores very high on the first three of the four desirable properties listed above. Edinburgh LF is a simple variant of λ-calculus with the property that, if a predicate is represented as an LF type then any LF expression of that type is a proof of that predicate. Thus, the simple logic-independent LF type-checking algorithm can be used for checking proofs.

The Edinburgh Logical Framework

The Edinburgh Logical Framework (also referred to as LF) has been introduced by Harper, Honsell and Plotkin (Harper, Honsell, and Plotkin, 1993b) as a metalanguage for high-level specification of logics. LF provides natural support for the management of binding operators and of the hypothetical and schematic judgments through LF bound variables. Consider for example, the usual formulation of the implication introduction rule IMPI in first-order logic, shown in Figure 7-5. This rule is hypothetical because the proof of the right-hand side of the implication can use the assumption that the left-hand side holds. However, there is a side condition requiring that this assumption not be used elsewhere in the proof. As we shall see below, LF can represent this side condition in a natural way by representing the assumption as a local variable bound in the proof of the right side of the implication. The fact that these techniques are supported directly by the logical framework is a crucial factor for the succinct formalization of proofs.

The LF type theory is a language with entities at three levels: objects, types and kinds, whose abstract syntax is shown below:

Kinds	K	$::=$	$\texttt{Type} \mid \Pi x{:}A.K$
Types	A	$::=$	$a \mid A\,M \mid \Pi x{:}A_1.A_2$
Objects	M	$::=$	$x \mid c \mid M_1 M_2 \mid \lambda x{:}A.M$

Types are used to classify objects and similarly, kinds are used to classify

$$
\begin{array}{lcl}
\iota & : & \texttt{Type} \\
o & : & \texttt{Type} \\
w & : & \texttt{Type} \\
s & : & \texttt{Type} \\
 & & \\
\texttt{zero} & : & \iota \\
\texttt{sel} & : & \iota \to \iota \to \iota \\
\texttt{upd} & : & \iota \to \iota \to \iota \to \iota \\
 & & \\
\texttt{int} & : & w \\
\texttt{list} & : & w \to w \\
\texttt{seq2} & : & w \to s \to s \\
\texttt{seq1} & : & w \to s \\
\texttt{ptr} & : & s \to w \\
\texttt{settype} & : & (\iota \to o) \to w
\end{array}
$$

(a)

$$
\begin{array}{lcl}
\texttt{true} & : & o \\
\texttt{and} & : & o \to o \to o \\
\texttt{impl} & : & o \to o \to o \\
\texttt{all} & : & (\iota \to o) \to o \\
\texttt{eq} & : & \iota \to \iota \to o \\
\texttt{neq} & : & \iota \to \iota \to o \\
\texttt{addr} & : & \iota \to o \\
\texttt{hastype} & : & \iota \to w \to o \\
\texttt{ge} & : & \iota \to \iota \to o
\end{array}
$$

(b)

Figure 7-10 The LF signature Σ that constitutes the representation of the syntax first-order predicate logic with equality and subscripted variables. Both expression (a) and predicate constructors (b) are shown.

types. The type $\Pi x : A.B$ is a dependent function type with x bound in B. In the special case when x does not occur in B, we use the more familiar notation $A \to B$. Also, `Type` is the base kind, a is a type constant and c is an object constant.

The encoding of a logic in LF is described by an *LF signature* Σ that contains declarations for a set of LF type constants and object constants corresponding to the syntactic formula constructors and to the proof rules. For a more concrete discussion, I describe in this section the LF representation of the safety policy that we have developed for our example agent.

The syntax of the logic is described by the declarations shown in Figure 7-10. This signature defines an LF type constant for each kind of syntactic entity within the logic: expressions (ι), formulas (o), word types (w), and structure types (s). Then, there is an LF constant declaration for each syntactic constructor. The LF type of such a constant describes the arity of the constructor, the types of the arguments and the type of the constructed value. There are two syntactic constructors that are worth explaining. The `settype` construc-

$$
\begin{array}{lcl}
\texttt{pf} & : & o \to \texttt{Type} \\
 & & \\
\texttt{truei} & : & \texttt{pf}\ \texttt{true} \\
\texttt{andi} & : & \Pi p{:}o.\Pi r{:}o.\texttt{pf}\ p \to \texttt{pf}\ r \to \texttt{pf}\ (\texttt{and}\ p\ r) \\
\texttt{andel} & : & \Pi p{:}o.\Pi r{:}o.\texttt{pf}\ (\texttt{and}\ p\ r) \to \texttt{pf}\ p \\
\texttt{ander} & : & \Pi p{:}o.\Pi r{:}o.\texttt{pf}\ (\texttt{and}\ p\ r) \to \texttt{pf}\ r \\
\texttt{impi} & : & \Pi p{:}o.\Pi r{:}o.(\texttt{pf}\ p \to \texttt{pf}\ r) \to \texttt{pf}\ (\texttt{impl}\ p\ r) \\
\texttt{impe} & : & \Pi p{:}o.\Pi r{:}o.\texttt{pf}\ (\texttt{impl}\ p\ r) \to \texttt{pf}\ p \to \texttt{pf}\ r \\
\texttt{alli} & : & \Pi p{:}\iota \to o.(\Pi v{:}\iota.\texttt{pf}\ (p\ v)) \to \texttt{pf}\ (\texttt{all}\ p) \\
\texttt{alle} & : & \Pi p{:}\iota \to o.\Pi e{:}\iota.\texttt{pf}\ (\texttt{all}\ p) \to \texttt{pf}\ (p\ e) \\
\texttt{mem0} & : & \Pi m{:}\iota.\Pi a{:}\iota.\Pi v{:}\iota.\Pi a'{:}\iota.\texttt{pf}\ (\texttt{eq}\ a\ a') \to \texttt{pf}\ (\texttt{eq}\ (\texttt{sel}\ (\texttt{upd}\ m\ a\ v)\ a')\ v) \\
\texttt{mem1} & : & \Pi m{:}\iota.\Pi a{:}\iota.\Pi v{:}\iota.\Pi a'{:}\iota. \\
 & & \qquad \texttt{pf}\ (\texttt{neq}\ a\ a') \to \texttt{pf}\ (\texttt{eq}\ (\texttt{sel}\ (\texttt{upd}\ m\ a\ v)\ a')\ (\texttt{sel}\ m\ a')) \\
 & & \\
\texttt{cons} & : & \Pi E{:}\iota.\Pi W{:}w. \\
 & & \texttt{pf}\ (\texttt{hastype}\ E\ (\texttt{list}\ W)) \to \texttt{pf}\ (\texttt{neq}\ E\ \texttt{zero}) \to \\
 & & \texttt{pf}\ (\texttt{hastype}\ E\ (\texttt{ptr}\ (\texttt{seq2}\ W\ (\texttt{seq1}\ (\texttt{list}\ W))))). \\
\texttt{set} & : & \Pi E{:}\iota.\Pi F{:}\iota \to o.\texttt{pf}\ (\texttt{hastype}\ E\ (\texttt{settype}\ F)) \to \texttt{pf}\ (F\ E).
\end{array}
$$

Figure 7-11 The LF signature Σ that constitutes the representation of the axiomatization of safety policy proof rules.

tor, used to represent word types of the form $\{y \mid F(y)\}$, has one argument, the function F from expressions to formulas. The other similar case is the `all` constructor used to encode universally quantified formulas. In both of these cases we are representing a binding in the object logic (i.e., the logic that is being represented) with a binding in LF. The major advantages of this representation is that α-equivalence and β-reduction in the object logic are supported implicitly by the similar mechanisms in LF. This representation strategy is called *higher-order representation* and is essential for a concise representation of logics with biding constructs.

The LF representation function $\ulcorner\cdot\urcorner$ is defined inductively on the structure of expressions, types and formulas. For example, we have the following two representations:

$$
\begin{array}{rcl}
\ulcorner P \Rightarrow (P \wedge P)\urcorner & = & \texttt{imp}\ \ulcorner P\urcorner\ (\texttt{and}\ \ulcorner P\urcorner\ \ulcorner P\urcorner) \\
\ulcorner \forall x.\texttt{addr}\ x\urcorner & = & \texttt{all}\ (\lambda x : \iota.\texttt{addr}\ x)
\end{array}
$$

7.5.1 EXERCISE [★]: Write the LF representation of the predicate $\forall a.a : \texttt{ptr}\,\{\texttt{int}\} \Rightarrow \texttt{addr}\,a$ □

The strategy for representing proofs in LF is to define a type family "pf" indexed by representation of formulas. Then, we represent the proof of "F" as an LF expression having type "pf F". This representation strategy is called "judgments as types and derivations as objects" and was first used in the work of Harper, Honsell and Plotkin (Harper, Honsell, and Plotkin, 1993b). Note that the dependent types of LF allow us to encode not only that an expression encodes a proof but also whose proof it is.

One can view the axioms and inference rules as proof constructors. This justifies representing the axioms and inference rules in a manner similar to the syntactic constructors, by means of LF constants. The signature shown in Figure 7-11 contains a fragment of the proof constructors required for the proof rules shown in Figure 7-5 (for first-order logic) and Figure 7-6 (for our safety policy). Note how the dependent types of LF can define precisely the meaning of each rule. For example, the declaration of the constant "andi" says that, in order to construct the proof of a conjunction of two predicates, one can apply the constant "andi" to four arguments, the first two being the two conjuncts and the other two being the representations of proofs of the conjuncts respectively.

The LF representation function $\ulcorner\cdot\urcorner$ is extended to derivations and is defined recursively on the derivation, as shown in the following examples (the letters $\mathcal{D}$ are used to name sub-derivations):

$$\ulcorner\, \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2\\ F_1 & F_2\end{array}}{F1 \wedge F_2}\, \urcorner = \texttt{andi}\ \ulcorner F_1 \urcorner\ \ulcorner F_2 \urcorner\ \ulcorner \mathcal{D}_1 \urcorner\ \ulcorner \mathcal{D}_2 \urcorner$$

$$\ulcorner\, \dfrac{\begin{array}{c}[F_1]^u\\ \vdots\ \mathcal{D}^u\\ F_2\end{array}}{F_1 \Rightarrow F_2}\, \urcorner = \texttt{impi}\ \ulcorner F_1 \urcorner\ \ulcorner F_2 \urcorner\ (\lambda u{:}\texttt{pf}\ \ulcorner F_1 \urcorner.\ulcorner \mathcal{D}^u \urcorner)$$

In the representation of the implication introduction proof rule, the letter u is the name of the assumption that F_1 holds. Note how the representation encodes the constraint that this assumptions must be local to the proof of F_2.

To conclude the presentation of the LF representation, consider the proof of the formula "$F \Rightarrow (F \wedge F)$". The LF representation of this proof is shown in Figure 7-12.

[h]

$$M = \texttt{impi}\ \ulcorner F \urcorner\ (\texttt{and}\ \ulcorner F \urcorner\ \ulcorner F \urcorner)\ (\lambda x{:}\texttt{pf}\ \ulcorner F \urcorner.\texttt{andi}\ \ulcorner F \urcorner\ \ulcorner F \urcorner\ x\ x)$$

Figure 7-12 The LF representation of the proof by implication introduction followed by conjunction introduction of the predicate $F \Rightarrow (F \wedge F)$.

7.5.2 EXERCISE [★]: Write the LF representation of the proof of the formula $\forall a.a : \texttt{ptr}\,\{\texttt{int}\} \Rightarrow \texttt{addr}\ a$, using the proof rules from our safety policy. □

7.5.1 The LF Type System

The main advantage of using LF for proof representation is that proof validity can be checked by a simple type-checking algorithm. That is, to check that the LF object M is the representation of a valid proof of the predicate F we use the LF typing rules (to be presented below) to verify that M has type $\texttt{pf}\ \ulcorner F \urcorner$ in the context of the signature Σ declaring the valid proof rules.

Type checking in the LF type system is defined by means of four judgments described below:

$\Gamma \vdash^{LF} A : K$	A is a valid type of kind K
$\Gamma \vdash^{LF} M : A$	M is a valid object of type A
$A \equiv_\beta B$	A is β-equivalent to B
$M \equiv_\beta N$	M is β-equivalent to N

where Γ is a typing context assigning types to LF variables. These typing judgment are with respect to a given signature Σ.

The derivation rules for the LF typing judgments are shown in Figure 7-13. For the β-equivalence judgments we omit the rules that define it to be an equivalence and a congruence.

As an example of how LF type checking is used to perform proof checking, consider the proof representation M shown in Figure 7-12. It is easy to verify, given the LF typing rules and the declaration of the constants involved, that this proof has the LF type "$\texttt{pf}\ (\texttt{imp}\ulcorner F \urcorner\ (\texttt{and}\ulcorner F \urcorner\ \ulcorner F \urcorner))$". The adequacy of LF type checking for proof checking in the logic under consideration is stated formally in the Theorems 7.5.3 and 7.5.4 below. These theorems follow immediately from more general version that are proved in (Harper, Honsell, and Plotkin, 1993b). From their proofs it is evident that they continue to hold if the logic is extended with new expression and predicate constructors.

7.5.3 THEOREM: **(Adequacy of Syntax Representation.)**

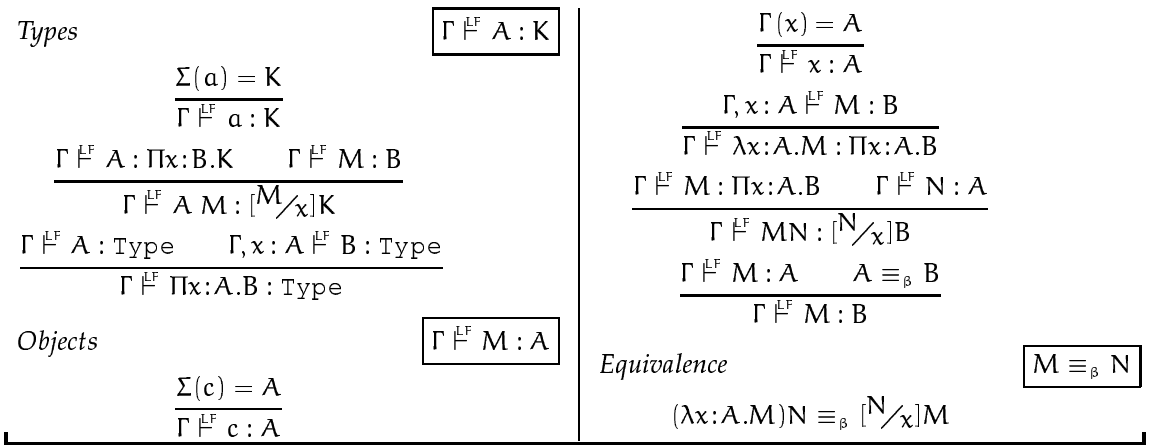


Figure 7-13: The LF Type System

1. If E is a closed expression, then $\cdot \vdash^{LF} \ulcorner E \urcorner : \iota$. If M is a closed LF object such that $\cdot \vdash^{LF} M : \iota$, then there exists an expression E such that $\ulcorner E \urcorner \equiv_\beta M$.

2. If W is a word-type, then $\cdot \vdash^{LF} \ulcorner W \urcorner : w$. If M is a closed LF object such that $\cdot \vdash^{LF} M : w$, then there exists a word type W such that $\ulcorner W \urcorner \equiv_\beta M$.

3. If S is a structured type, then $\cdot \vdash^{LF} \ulcorner S \urcorner : s$. If M is a closed LF object such that $\cdot \vdash^{LF} M : s$, then there exists a structured type S such that $\ulcorner S \urcorner \equiv_\beta M$.

4. If F is a closed formula, then $\cdot \vdash^{LF} \ulcorner F \urcorner : o$. If M is a closed LF object such that $\cdot \vdash^{LF} M : o$, then there exists a formula F such that $\ulcorner F \urcorner \equiv_\beta M$.

□

7.5.4 THEOREM: **(Adequacy of Derivation Representation.)**

1. If $\mathcal{D}$ is a derivation of F then $\cdot \vdash^{LF} \ulcorner \mathcal{D} \urcorner : \mathtt{pf}\ \ulcorner F \urcorner$.

2. If M is a closed LF object such that $\cdot \vdash^{LF} M : \mathtt{pf}\ \ulcorner F \urcorner$, then there exists a derivation $\mathcal{D}$ of F such that $\ulcorner \mathcal{D} \urcorner \equiv_\beta M$.

□

In the context of PCC, Theorem 7.5.4(2) says that if the agent producer can exhibit an LF object having the type "$\mathtt{pf}\ \ulcorner VC \urcorner$" then we know that there is a derivation of the verification condition within the logic, which in turn means

that the verification condition is valid and the agent code satisfies the safety policy.

Owing to the simplicity of the LF type system, the implementation of the type checker is simple and easy to trust. Furthermore, because all of the dependencies on the particular object logic are separated in the signature, the implementation of the type checker can be reused directly for proof checking in various first-order or higher-order logics. The only logic-dependent component of the proof checker is the signature, which is usually easy to verify by visual inspection.

Unfortunately, the above-mentioned advantages of LF representation of proofs come at a high price. The typical LF representation of a proof is large, due to a significant amount of redundancy. This fact can already be seen in the proof representation shown in Figure 7-12, where there are six copies of F as opposed to only three in the predicate to be proved. The effect of redundancy observed in practice increases non-linearly with the size of the proofs. Consider for example, the representation of the proof of the n-way conjunction $F \wedge \ldots \wedge F$. Depending on how balanced is the binary tree representing this predicate, the number of copies of F in the proof representation ranges from an expected value of $n \log n$ (when the tree is perfectly balanced) to a worse case value of $n^2/2$ (when the tree degenerates into a list). The redundancy of representation is not only a space problem but also leads to inefficient proof checking, because all of the redundant copies have to be type checked and then checked for equivalence with instances of F from the predicate to be proved.

The proof representation and checking framework presented in the next section is based on the observation that it is possible to retain only the skeleton of an LF representation of a proof and to use a modified LF type-checking algorithm to reconstruct on the fly the missing parts. The resulting *implicit LF* (or LF_i) representation inherits the advantages of the LF representation (i.e., small and logic-independent implementation of the proof checker) without the disadvantages (i.e., large proof sizes and slow proof checking).

Implicit LF

The solution to the redundancy problem is to eliminate the redundant subterms from the proof. In most cases we can eliminate all copies of a given subterm from the proof and rely instead on the copy that exists within the predicate to be proved, which is constructed by the VCGen and is trusted to be well formed. But now the code receiver will be receiving proofs with missing subterms. One possible strategy is for the code receiver to reconstruct the original form of the proof and then to use the simple LF type checking algo-

rithm to validate it. But this does not save proof-checking time and requires significantly more working memory than the size of the incoming LF_i proof. Instead, we modify the LF type-checking algorithm to reconstruct the missing subterms while it performs type checking. One major advantage of this strategy is that terms that are reconstructed based on copies from the verification condition do not need to be type checked themselves.

We will not show the formal details of the type reconstruction algorithm but will show instead how it operated on a simple example. For expository purposes, the missing proof subterms are marked with placeholders, written as $*$. Consider now the proof of the predicate $F \Rightarrow (F \wedge F)$ of Figure 7-12. If we replace all copies of "F" with placeholders we get the following LF_i object:

$$\texttt{impi}\ *_1\ *_2\ (\lambda u{:}*_3.\texttt{andi}\ *_4\ *_5\ u\ u)$$

This implicit proof captures the structure of the proof without any redundant information. The subterms marked with placeholders can be recovered while verifying that the term has type "$\texttt{pf}\ (\texttt{impl}\ulcorner F\urcorner\ (\texttt{and}\ulcorner F\urcorner\ \ulcorner F\urcorner))$", as described below.

Reconstruction starts by recognizing the top-level constructor `impi`. The expected type of the entire term, "$\texttt{pf}\ (\texttt{impl}\ \ulcorner F\urcorner\ (\texttt{and}\ \ulcorner F\urcorner\ \ulcorner F\urcorner))$", is "matched" against the result type of the `impi` constant, as given by the signature Σ. The result of this matching is an instantiation for placeholders 1 and 2 and a residual type-checking constraint for the explicit argument of `impi`, as follows:

$$\begin{array}{rlcl} & *_1 & \equiv & \ulcorner F\urcorner \\ & *_2 & \equiv & \texttt{and}\ulcorner F\urcorner\ulcorner F\urcorner \\ \vdash & (\lambda u{:}*_3.\texttt{andi}\ *_4\ *_5\ u\ u) & : & \texttt{pf}\ulcorner F\urcorner \rightarrow \texttt{pf}\ (\texttt{and}\ulcorner F\urcorner\ulcorner F\urcorner) \end{array}$$

Reconstruction continues with the remaining type-checking constraint. From its type we can recover the value of placeholder 3 and a typing constraint for the body:

$$\begin{array}{rlcl} & *_3 & \equiv & \texttt{pf}\ulcorner F\urcorner \\ u:\texttt{pf}\ulcorner F\urcorner \vdash & \texttt{andi}\ *_4\ *_5\ u\ u & : & \texttt{pf}\ (\texttt{and}\ulcorner F\urcorner\ulcorner F\urcorner) \end{array}$$

Now `andi` is the top-level constant and by matching its result type as declared in the signature with the goal type of the constraint we get the instantiation for placeholders 4 and 5 and two residual typing constraints:

$$\begin{array}{rlcl} & *_4 & \equiv & \ulcorner F\urcorner \\ & *_5 & \equiv & \ulcorner F\urcorner \\ u:\texttt{pf}\ulcorner F\urcorner \vdash & u & : & \texttt{pf}\ulcorner F\urcorner \\ u:\texttt{pf}\ulcorner F\urcorner \vdash & u & : & \texttt{pf}\ulcorner F\urcorner \end{array}$$

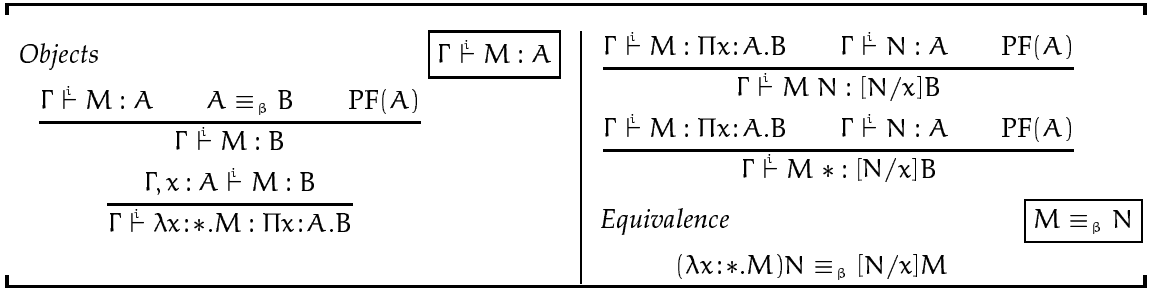


Figure 7-14: The rules that are new in the LF_i type system.

The remaining two constraints are solved by the variable typing rule. Note that this step involves verifying the equivalence of the objects $\ulcorner F \urcorner$ from the assumption and the goal. This concludes the reconstruction and checking of the entire proof. We reconstructed the full representation of the proof by instantiating all placeholders with well-typed LF objects. We know that these instantiations are well-typed because they are ultimately extracted from the original constraint type, which is assumed to contain only well-typed subterms.

The formalization of the reconstruction algorithm described informally above is in two stages. First, we show a variant of the LF type system, called implicit LF or LF_i, that extends LF with placeholders. This type system has the property that all well-typed LF_i terms can be reconstructed to well-typed LF terms. However, unlike the original LF type system, the LF_i type system is not amenable to a direct implementation of deterministic type checking. Instead, we use a separate reconstruction algorithm.

An object M is fully reconstructed, or fully explicit, when it is placeholder free. We write $PF(M)$ to denote this property. We extend this notation to type environments and write $PF(\Gamma)$ to denote that all types assigned in Γ to variables are placeholder free.

The LF_i typing rules are an extension of the LF typing rules with two new typing rules for dealing with implicit abstraction and placeholders, and one new β-equivalence rule dealing with implicit abstraction. These additions are shown in Figure 7-14. The LF_i typing judgment is written $\Gamma \stackrel{i}{\vdash} M : A$.

Note that according to the LF_i type system placeholders cannot occur on a function position, but only as arguments in an application. This restriction allows us to simplify the reconstruction algorithm by avoiding higher-order unification. Note also that several LF_i rules require that the types involved do not contain placeholders. This restriction simplifies greatly the proofs of

soundness of the reconstruction algorithms and does not seem to diminish the effectiveness of the LF_i representation.

A quick analysis of the LF_i typing rules reveals that they are not directly useful for type checking or type inference. The main reason is that type checking an application involves "guessing" appropriate A and N. The type A can sometimes be recovered from the type of the application head, but the term N in an application to a placeholder cannot be found easily in general. This is not a problem for us because we need the LF_i type-system only as a step in proving the correctness of the type-reconstruction algorithm, and not as the basis for an implementation of a type-checking algorithm.

The only property of interest of the LF_i type system is that once we have a typing derivation we can reconstruct the object involved and a corresponding LF typing derivation for it. To make this more precise we introduce the notation $M \nearrow M'$ to denote that M' is a fully-reconstructed version of the implicit object M (i.e., $PF(M')$). This means that M' can be obtained from M by replacing all of its placeholders with fully-explicit LF objects. Note that the reconstruction relation is not a function as there might be several reconstructions of a given implicit object or type.

7.5.5 THEOREM: **Soundness of LF_i typing** If $\Gamma \overset{i}{\vdash} M : A$ and $PF(\Gamma)$, $PF(A)$, then there exists M' such that $M \nearrow M'$ and $\Gamma \overset{LF}{\vdash} M' : A$. □

7.5.6 EXERCISE [★★, ↛]: Prove Theorem 7.5.5 □

7.6 Proof Generation

We have seen that a successfully checked proof of the verification condition guarantees that the verification condition is valid, which in turn guarantees that the code adheres to the safety policy. The PCC infrastructure is simple, easy-to-trust and automatic. But this is only because all the difficult tasks have been delegated to the code and proof producers. The first difficult task, besides writing code that is indeed safe, is to generate the code annotations consisting of loop invariants for all loops and of function specifications for all local functions. The other difficult task is to prove the verification condition produced by the verification-condition generator.

Fortunately there are important situations when both the generation of the annotations and of the proof can be automated. Consider the situation in which there exists a high-level language, perhaps a domain-specific one, in which the safety policy is guaranteed to be satisfied by a combination of static and run-time checks. For example, if the safety policy is memory safety then any memory-safe high-level language can be used. The key insight is that

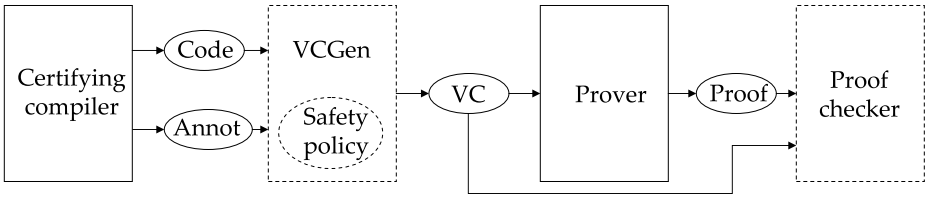


Figure 7-15 The interaction between the untrusted PCC tools (used by the code producer and shown with continuous lines) and the trusted PCC infrastructure (used by the code receiver and shown with interrupted lines).

in these systems the safety policy is guaranteed to hold by the design of the static and run-time checks. In essence, the high-level type checker acts as a theorem prover. All we have to do is to show that a sufficient number and kind of static and run-time checks have been performed.

In the architecture shown in Figure 7-15 the annotations are generated automatically by a *certifying compiler* from high-level language to assembly language. For safety polices that follow closely the high-level type system it is surprisingly easy for a compiler to produce the loop invariants, which are essentially conjunctions of type declarations for the live registers at the given program point. This is information that the compiler can easily maintain and emit.

Before it can generate the required proofs the code producer must pass the annotated code to a local copy of VCGen. The proof itself is generated by a theorem prover customized for the specific safety policy. As discussed in Section 7.3.1, such a theorem prover is little more than a type checker. However, unlike a regular type checker or theorem prover, the PCC theorem prover must generate explicit representation of the proofs. The architecture shown in Figure 7-15 is described in detail in (Necula, 1998).

7.7 PCC Beyond Types

The presentation of PCC so far has focused on type-based safety policies. We have shown that verification condition generation followed by theorem proving can overcome many of the difficulties of type checking programs low-level languages. It should be obvious that we can take the example that we used so far and change the type system by simply changing the proof rules, with no changes to the infrastructure itself. But the machinery we have

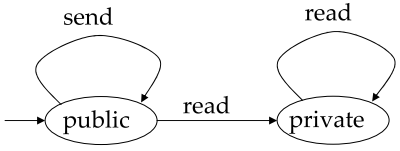


Figure 7-16

constructed in the process can be used to enforce more complex safety polices than are usually associated with types. And we can do this with very few changes, thanks both to the modular design of the infrastructure and to the choice of using the lower-level mechanism of logic rather than committing to a high-level type system. In this section, we explore one such safety policy.

Consider a safety policy that allows access to two host services: *read* the contents of a local file and *send* data over the network. The host wishes to enforce the policy that the agent cannot send data after it has read local files. This is a conservative way to ensure that no local file contents will be leaked. This example is taken from (Schneider, 2000).

This safety policy can be described using the state machine shown in Figure 7-16. Initially the agent is in the `public` state in which it can use both the `send` and the `read` services. However, once it uses the `read` service the agent transitions in the `private` state, in which it can use only the `read` service.

In order to enforce such a safety policy it is sufficient to check that the `send` service cannot be used after the `read` service has been used. At the level of assembly language these services would be most likely implemented as function calls. In that case the privacy safety policy can be implemented as a precondition on the `send` function. Instead of introducing a general mechanism for handling function calls (see Exercise 7.3.5) we use a special-purpose handling of the instructions `call read` and `call send`.

In the presentation of PCC from previous sections there is no element of the state of the computation that reflects whether a certain function has been invoked or not. One way to address this issue is to require that the agent code keep track of its own public/private state at run-time, presumably in a register or a memory location. Then the postcondition of `read` would require that this state element reflect the `private` state and the precondition of `send` would require that the state element reflect the `public` state. This strategy is appropriate when the producer of the agent code wishes to use run-time checking to enforce the safety policy, in which case it would have to prove that the appropriate checks have been inserted. This strategy also

has the benefit of not requiring any changes in the PCC infrastructure.

We are going to pursue another alternative. We will modify VCGen and the symbolic evaluator to keep track of the public/private state. And since we prefer to extend the PCC infrastructure with a general-purpose mechanism rather than a specific policy, the VCGen extension should be able to record any information about the *history* of the execution, not just its public/private state.

For this purpose we extend the symbolic evaluation state with another pseudo-register, called $\mathtt{r}_H$ to store a sequence of interesting events in the past of the computation. The set of symbolic expressions that this register can have are shown below:

$$\begin{array}{llll} \text{Histories} & H & ::= & x \mid \mathtt{event}\ V\ H \\ \text{Events} & V & ::= & \mathtt{init} \mid \mathtt{read} \mid \mathtt{send} \end{array}$$

Additionally we add a number of formulas that we can use for stating properties of the history of execution:

$$\text{Formulas} \quad F ::= \ldots \mid \mathtt{publicState}\ H \mid \mathtt{privateState}\ H$$

As usual when we extend the language of formulas we must also extend the proof rules. For our safety policy we add the following three proof rules:

$$\mathtt{publicState}\ (\mathtt{event}\ \mathtt{init}\ H) \qquad \text{(INIT)}$$

$$\frac{\mathtt{publicState}\ H}{\mathtt{publicState}\ (\mathtt{event}\ \mathtt{send}\ H)} \qquad \text{(SEND)}$$

$$\mathtt{privateState}\ (\mathtt{event}\ \mathtt{read}\ H) \qquad \text{(READ)}$$

The definition of the VCGen and the symbolic evaluator can remain unchanged for the instructions considered so far, except that the $\mathtt{r}_H$ register can be used in loop invariants and function preconditions and postconditions. In particular, for the privacy safety policy the invocations of the `read` and `send` services can be handled in the symbolic evaluator as follows:

$$SE(i,\sigma) = \begin{cases} \ldots & \\ SE(i+1, \sigma[\mathtt{r}_H \leftarrow (\sigma\ (\mathtt{event}\ \mathtt{read}\ \mathtt{r}_H))]) & \text{if } \Pi_i = \mathtt{call}\ \mathtt{read} \\ \mathtt{publicState}\ (\sigma\ \mathtt{r}_H) \wedge & \\ SE(i+1, \sigma[\mathtt{r}_H \leftarrow (\sigma\ (\mathtt{event}\ \mathtt{send}\ \mathtt{r}_H))]) & \text{if } \Pi_i = \mathtt{call}\ \mathtt{send} \end{cases}$$

The symbolic evaluator extends the history state with information about the services that were used. Additionally, the `send` call requires through

its precondition that the history of the computation be consistent with the `public` state of the safety policy. A realistic symbolic evaluator would support a general-purpose function call instruction, in which case the effect of the `read` and `send` functions could be achieved by appropriate function preconditions and postconditions.

7.7.1 EXERCISE [★★,↛]: Add two actions `lock` *e* and `unlock` *e* that can be used to acquire and release a lock that is denoted by the expression *e*. Define a PCC safety policy (extensions to the logic, new proof rules and changes to the symbolic evaluator) that requires correct use of locks: a lock cannot be acquired or released twice in a row, the agent must release all locks upon return. □

7.7.2 EXERCISE [★★, ↛]: The verification-condition generator that we described in Section 7.3.1 cannot enforce a safety policy that allows the agent to "probe" the accessibility of a memory page by attempting a read from an address within that page. This is a common way to check for stack overflow in many systems. Show how you can change the symbolic evaluator to use the history register for the purpose of specifying such a safety policy. □

This example shows how to use PCC for safety policies that go beyond type checking. In fact, PCC is extremely powerful in this sense. Any safety policy that could be enforced by an interpreter using run-time checking could in principle be enforced by PCC. A major advantage of PCC over interpreters is that it can check properties that would be very expensive to check at run time. Consider, for example, how complicated it would be to write an interpreter that enforces at run-time a fine grained memory safety policy. Each memory word would have to be instrumented with information whether it is accessible or not. By comparison, we can PCC along with a strong type system to achieve the same effect, with no run-time penalty.

7.8 Conclusion

Below is a list of the most important ways in which PCC improves over other existing techniques for enforcing safe execution of untrusted code:

- PCC operates at **load time** before the agent code is installed in the host system. This is in contrast with techniques that enforce the safety policy by relying on extensive run-time checking or even interpretation. As a result PCC agents run at native-code speed, which can be ten times faster than interpreted agents (written for example using Java bytecode) or 30% faster than agents whose memory operations are checked at run time.

Additionally, by doing the checking at load time it becomes possible to enforce certain safety policies that are hard or impossible to enforce at run time. For example, by examining the code of the agent and the associated "explanation" PCC can verify that a certain interrupt routine terminates within a given number of instructions executed or that a video frame rendering agent can keep up with a given frame rate. Run-time enforcement of timing properties of such fine granularity is hard.

- PCC is **small**. PCC is simple and small because it has to do a relatively simple task. In particular, PCC does not have to discover on its own whether and why the agent meets the safety policy.
- For the same reason, PCC can operate even on agents expressed in **native-code** form. And because PCC can verify the code after compilation and optimization, the checked code is ready to run without needing an additional interpreter or compiler on the host. This has serious software engineering advantages since it reduces the amount of security critical code and it is also a benefit when the host environment is too small to contain an interpreter or a compiler, such as is the case for many embedded software systems.
- PCC is **general.** All PCC has to do is to verify safety explanations and to match them with the code and the safety policy. By standardizing a language for expressing the explanations and a formalism for expressing the safety policies it is possible to implement a single algorithm that can perform the required check, for any agent code, any valid explanation and a large class of safety policies. In this sense a single implementation of PCC can be used for checking a variety of safety policies.

The combination of benefits that PCC offers is unique among the techniques for safe execution of untrusted code. Previously one had to sacrifice one or more of these benefits because it is impossible to achieve them all in a system that examines just the agent code and has to discover on its own why the code is safe.

The PCC infrastructure is designed to complement a cryptographic authentication infrastructure. While cryptographic techniques such as digital signatures can be used by the host to verify external properties of the agent program, such as freshness and authenticity, or the author's identity, the PCC infrastructure checks internal semantic properties of the code such as what the code does and what it does not do. This enables the host to prevent safety breaches due to either malicious intent (for agents originating from untrusted sources) or due to programming errors (for agents originating from trusted sources).

However, proof-carrying code is not without costs. The most notable challenge to using PCC is the difficulty of producing code annotations and proofs. In some cases, these can be produced automatically based on some high-level language invariants. But in general a human is required to be involved and the more complex the safety policy the more onerous the burden of proof can be expected to be. All that PCC offers in this direction is a way to shift this burden from the code received to the code producer who can be expected to have more computational power, and especially more knowledge of why the code satisfies the safety policy.

Proof-carrying code is a witness to the fact that programming language technology and type theory are the basis of valuable techniques for solving practical engineering problems. However, in the process of applying these techniques for the design of a PCC infrastructure it became necessary to adapt the off-the-shelf techniques in non-trivial ways to the particular application domain. Some of that adaptation can be carried out in a theoretical setting, such as the extension of Edinburgh LF to implicit LF, while other parts involve real engineering.

8 *Typed Assembly Language*

By Greg Morrisett

The principle of *proof-carrying code* (PCC) is that we can eliminate the need to trust a piece of code by requiring a formal, machine-checkable proof that the code has some desired properties. The key insight is that checking a proof is usually quite easy, and can be done with a relatively small (and hence trustworthy) proof-checking engine.

If we are to effectively use PCC to build trustworthy systems, then we must solve two problems:

1. What properties should we require of the code?

2. How do code producers construct a formal proof that their code has the desired properties?

The first question is extremely context and application dependent. It requires that we somehow rule out all "bad" things without unduly restricting "good" things, and make both "bad" and "good" formal. The second question is impossible to solve automatically for arbitrary code and even simple safety properties. So, how are we to take advantage of PCC?

One approach is based on *type-preserving* compilation. The idea here is to focus on some form of type safety as the desired property. The advantage of focusing on type safety is that programmers are willing to do the hard part—construct a proof that some code is type-safe. The way they do this is by writing the code in a high-level language (e.g., Java or ML). If the source they write doesn't type-check, then they rewrite the code until it does. In this repsect, they are engaging in a form of interactive theorem proving.

Once the initial proof is done, then a type-preserving compiler takes over. It simply maps the type-safe source code through a series of successively lower-level intermediate languages to target code. As it transforms the code,

it also (conceptually) transforms the proof. The point is that it is usually much easier to do this sort of transformation than to try and prove type safety directly on the generated machine code.

Of course, such a methodology demands that as part of the compilation process, we design a series of typed intermediate languages, culminating with typed machine code. Intermediate languages such as the Java Virtual Machine Language (JVML) and Microsoft's Common Intermediate Language (CIL) are examples of widely used typed intermediate languages that are targets of certifying compilers for a number of high-level languages, including Java, C#, and ML.

However, both the JVML and CIL are relatively high-level "CISC-like" abstract machines. That is, they have pre-conceived notions of methods and objects which may be incompatible with an efficient encoding for a given language. For instance, the JVM does not support tail-calls, making the implementation of functional languages impractical. The CIL does support tail-calls, but there are other features that it lacks. For instance, arrays are treated as covariant in the type system, and thus require a run-time check upon update.

Of course, there will never be a universal, typed intermediate language (TIL) that is portable and readily supports all possible languages and implementation strategies. Nonetheless, we seek a principled approach to the design of TILs that minimizes the need to add new features and typing rules. In particular, we seek a more "RISC-like" design for type systems that makes it possible to encode high-level language features, and to support a wide variety of optimizations.

8.1 TAL-0: Control-Flow-Safety

We begin our design for a "RISC"-style typed assembly language by focusing on one safety property, known as *control-flow safety*. Informally, we wish to ensure that a program does not jump to arbitrary machine addresses throughout its execution, but rather, it only jumps to a well-defined subset of possible entry points. Control-flow safety is a crucial building block for building dynamic checks into a system. For instance, before performing a system call, such as a file read, we might need to first check that the arguments to the call have the right properties (e.g., the file has been opened for reading, and the destintation buffer is sufficiently large.) Without control-flow safety, a malicious client could jump past these checks and directly into the underlying routine.

A focus on control-flow safety will also let us start with an extremely sim-

r	::=	*registers:*	ι	::=	*instructions:*
	`r1 | r2 | ··· | rk`			$\texttt{mov}\, r_d, v$	
v	::=	*operands:*		$\texttt{add}\, r_d, r_s, v$	
	n	*integer literal*		$\texttt{beq}\, r, v$	
	ℓ	*label or pointer*	I	::=	*instruction sequences:*
	r	*registers*		$\texttt{jmp}\, v$	
				$\iota; I$	

Figure 8-1: Instructions and Operands for TAL-0

ple abstract machine and demonstrate the key ideas of adapting a type system to machine code. In subsequent sections, we will expand this machine and its type system to accomodate more features.

The syntax for our control-flow-safe assembly language, which we will call TAL-0, is given in Figure 8-1. We assume a fixed set of k general-purpose registers and a few representative instructions based on a subset of MIPS assembly language. Intuitively, each instruction uses the value in a source register (r_s) and an operand to compute a value which is placed in the destination register (r_d). In this setting, an operand is either another register, a word-sized immediate integer[1] , or a label. We use "value" to refer to an operand that is not a register.

For our purposes, it is also useful to define instruction sequences (I) as lists of instructions terminated by an explicit, unconditional control transfer (i.e., a jump). Of course, when the assembly code is mapped down to machine code, jumps to adjacent blocks can be eliminated since the code will fall through. We could make the order of instruction sequences and hence fall-throughs explicit, but we prefer a simpler, more uniform assembly language to present the key ideas.

Here is an example TAL-0 code fragment which computes the product of registers `r1` and `r2`, placing the final result in `r3` before jumping to a return address assumed to be in `r4`.

```
prod: mov r3,0; // res := 0
      jmp loop

loop: beq r1,done;   // if a = 0 goto done
      add r3,r2,r3;  // res := res + b
```

1. As we are dealing with an assembly language, we ignore the issue of fitting a full word-sized integer into a single instruction.

```
        add r1,r1,-1;  // a := a - 1
        jmp loop

done:   jmp r4         // return
```

We model evaluation of TAL-0 assembly programs using a rewriting relation between *abstract* machine states. Rather than model the execution of a concrete machine, we are deliberately using a higher-level representation for machine states which maintains certain distinctions. For instance, in a real machine, labels are resolved during program loading to some machine address which is also represented as an integer. In our abstract machine, we will continue to maintain the distinction between labels and arbitrary integers because this will allow us to easily state and then prove our desired safety property—that any control flow instruction can only branch to a valid, labelled entry point. Indeed, our abstract machine will get stuck if we try to transfer control to an integer as opposed to a label. So, the problem of enforcing the safety property now reduces to ensuring that our abstract machine cannot get stuck.

However, to support this level of abstraction, we must worry about the situation where a label is added to an integer, or where a test is done on a label. One possibility is to assume a coercion function `intof` which maps labels to integers, and use `intof`(ℓ) whenever ℓ appears as an operand to an arithmetic instruction. Though a perfectly reasonable approach, there are a number of reasons one might avoid it: First, it violates the abstraction that we are attempting to provide which may lead to subtle information flows. In some security contexts, this could be undesirable. Second, such a coercion would make it harder to prove the equivalence of two program states, where labels are α-converted. In turn, this would restrict an implementation's freedom to re-arrange code or recycle its memory[2] Finally, it simplifies the type system if we simply treat labels as abstract. In particular, if we included such a coercion, we would need some form of subtyping to validate the coercion. However, we emphasize that it is possible to expose labels as machine integers if desired.

The syntax for TAL-0 abstract machines is given in Figure 8-2. An abstract machine state M contains three components: (1) A heap H which is a finite, partial map from labels to heap values (h), (2) a register file R which is a total map from registers to values, and (3) a current instruction sequence I. The instruction sequence is meant to model the sequence of instructions pointed to by the program counter in a concrete machine. One way to think of the

2. The problem can be avoided by assuming a coercion relation between labels and integers that respects the alpha-equivalence class of labels.

R	::=	*register files:*	H	::=	*heaps:*
	$\{r_1 = v_1, \ldots, r_k = v_k\}$			$\{\ell_1 = h_1, \ldots, \ell_m = h_m\}$	
h	::=	*heap values:*	M	::=	*machine states:*
	I	*code*		(H, R, I)	

Figure 8-2: TAL-0 Abstract Machine Syntax

machine's operation is that it pre-fetches sequences of instructions up to the nearest `jmp` whenever a control-transfer is made.

We consider heaps and register files to be equivalent up to re-ordering. We also consider the labels in the domain of H to bind the free occurrences of those labels in the rest of the abstract machine. Finally, we consider abstract machine states to be equivalent up to alpha-conversion of bound labels.

The rewriting rules for TAL-0 are as follows:

$$\frac{H(\hat{R}(v)) = I}{(H, R, \texttt{jmp}\ v) \longrightarrow (H, R, I)} \quad \text{(JMP)}$$

$$(H, R, \texttt{mov}\ r_d, v; I) \longrightarrow (H, R[r_d = \hat{R}(v)], I) \quad \text{(MOV)}$$

$$\frac{R(r_s) = n_1 \qquad \hat{R}(v) = n_2}{(H, R, \texttt{add}\ r_d, r_s, v; I) \longrightarrow (H, R[r_d = n_1 + n_2], I)} \quad \text{(ADD)}$$

$$\frac{R(r) = 0 \qquad H(\hat{R}(v)) = I'}{(H, R, \texttt{beq}\ r, v; I) \longrightarrow (H, R, I')} \quad \text{(BEQ-EQ)}$$

$$\frac{R(r) = n \qquad n \neq 0}{(H, R, \texttt{beq}\ r, v; I) \longrightarrow (H, R, I)} \quad \text{(BEQ-NEQ)}$$

The rules make use of $\hat{R}$ which simply lifts R from operating on registers to operands:

$$\hat{R}(r) = R(r)$$
$$\hat{R}(n) = n$$
$$\hat{R}(\ell) = \ell$$

Notice that for `jmp` and a successful `beq` we simply load a new instruction sequence from the heap and begin executing it. Of course, this assumes that the destination operand evaluates to some label ℓ (as opposed to an integer), and that the heap provides a binding for ℓ. Otherwise, the machine state cannot make the transition and becomes stuck. For instance, if we attempted to evaluate `jmp 42`, then no transition could occur. In other words, if we can

devise a type system that rules out such stuck machine states, then we can be assured that all control transfers must go to properly labelled instruction sequences.

Of course, there are other ways this particular abstract machine can get stuck. For instance, if we attempt to add an integer to a label, or test a label using `beq`, then the machine will get stuck. This reflects our choice to leave labels abstract.

8.1.1 EXERCISE [★ ↛]: Taking H to be a heap that maps the labels `prod`, `loop`, and `done` to their respective instruction sequences above, and taking $R_0 =$ `{r1:2, r2:2, r3:0, r4:exit}` where `exit` is some unspecified label, show that $(H, R_0,$ `jmp prod`$)$ steps to a state $(H, R,$ `jmp r4`$)$ such that $R($`r3`$) = 4$. □

8.1.2 EXERCISE [RECOMMENDED, ★★★↛]: Using your favorite programming language, build an interpreter for the TAL-0 abstract machine. □

8.1.3 EXERCISE [★★★↛]: Formulate a semantics for a concrete machine based on the TAL-0 instruction set. The concrete machine should manipulate only integers and have states of the form (M, R, pc) where M is a memory mapping 32-bit integers to 32-bit integers, R is a register file, and pc holds a 32-bit integer for the next instruction to execute. You should assume an isomorphism `encode` and `decode` that maps instructions to and from distinct integers.

Then, prove that the TAL-0 abstract machine is faithful to the concrete machine by establishing a simulation relation between abstract and concrete machine states, and by showing that this relation is preserved under evaluation. □

8.2 The TAL-0 Type System

The goal of the type system for TAL-0 is to ensure that any well-formed abstract machine M cannot get stuck—that is, there always exists an M' such that $M \longrightarrow M'$. Obviously, our type system is going to have to distinguish labels from integers to ensure that the operands of a control transfer are labels. But we must also ensure that, no matter how many steps are taken by the abstract machine, it never gets into a stuck state (i.e., typing is preserved.) Thus, when we transfer control to a label, we need to know what kinds of values it expects to have in the registers.

To this end, we define our type syntax in Figure 8-3. There are four basic type constructors. Obviously, `int` will be used to classify integer values and `code` types will classify labels. Furthermore, `code`(Γ) will classify those labels which, when jumped to, expect to have values described by Γ in the

τ ::=		*operand types:*	Γ ::=		*register file types:*
	`int`	*word-sized integers*		$\{r_1 : \tau_1, \ldots, r_k : \tau_k\}$	
	`code`(Γ)	*code labels*	Ψ ::=		*heap types:*
	α	*type variables*		$\{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}$	
	$\forall\alpha.\tau$	*universal polymorphic types*			

Figure 8-3: TAL-0 Type Syntax

associated register. Here, Γ is a *register file type*—a total function from registers to types. In this respect, we can think of a label as a continuation which takes a record of values described by Γ as an argument.

We also support universal polymorphism in TAL-0 through the addition of type variables (α) and quantified types ($\forall\alpha.\tau$). As usual, we consider types up to alpha-equivalence of bound type variables. Finally, we consider register file types and heap types to be equivalent up to re-ordering.

With these static constructs in hand, we can now formalize the type system using the inference rules in Figure 8-4. The first judgment, $\Psi \vdash v : \tau$, is used to determine the type of a value. Recall that a value is a register-free operand, so there is no need for a register context in the judgment. Integer literals are given type `int`, whereas labels are given the type assigned to them by the heap type context Ψ. Note that this rule only applies when the label is in the domain of Ψ.

The second judgment, $\Psi; \Gamma \vdash v : \tau$ lifts value typing to operands. A register is given the type assigned by the register file type Γ. In addition, a polymorphic operand can be instantiated with any type, in a fashion similar to ML.

The next judgment, $\Psi \vdash \iota : \Gamma_1 \to \Gamma_2$ is used to check instructions. The notation is meant to suggest that the instruction expects a register file described by Γ_1 on input, and produces a register file described by Γ_2 on output. Note that for the `beq` instruction, we must ensure that the destination operand v is a code pointer that expects a register file described by the same Γ as any subsequent instruction. This ensures that, no matter which way the branch goes, the resulting machine state will be well-formed.

The judgment $\Psi \vdash I :$ `code`(Γ) assigns an instruction sequence I the type `code`(Γ) when the sequence expects to be given a register file described by Γ. In particular, a `jmp` instruction's type is dictated by the type of its operand. The code type for a sequence of instructions is determined from composition. Most importantly, we can generalize the type of an instruction sequence by abstracting any type variables. Note that there is no need to prevent abstrac-

Values $\boxed{\Psi \vdash v : \tau}$

$$\Psi \vdash n : \texttt{int} \quad \text{(S-INT)}$$

$$\Psi \vdash \ell : \Psi(\ell) \quad \text{(S-LAB)}$$

Operands $\boxed{\Psi;\Gamma \vdash v : \tau}$

$$\frac{\Psi \vdash v : \tau}{\Psi;\Gamma \vdash v : \tau} \quad \text{(S-VAL)}$$

$$\Psi;\Gamma \vdash r : \Gamma(r) \quad \text{(S-REG)}$$

$$\frac{\Psi;\Gamma \vdash v : \forall\alpha.\tau}{\Psi;\Gamma \vdash v : \tau[\tau'/\alpha]} \quad \text{(S-INST)}$$

Instructions $\boxed{\Psi \vdash \iota : \Gamma_1 \to \Gamma_2}$

$$\frac{\Psi;\Gamma \vdash v : \tau}{\Psi \vdash \texttt{mov}\ r_d, v : \Gamma \to \Gamma[r_d : \tau]} \quad \text{(S-MOV)}$$

$$\frac{\Psi;\Gamma \vdash r_s : \texttt{int} \qquad \Psi;\Gamma \vdash v : \texttt{int}}{\Psi \vdash \texttt{add}\ r_d, r_s, v : \Gamma \to \Gamma[r_d : \texttt{int}]} \quad \text{(S-ADD)}$$

$$\frac{\Psi;\Gamma \vdash r_s : \texttt{int} \qquad \Psi;\Gamma \vdash v : \texttt{code}(\Gamma)}{\Psi \vdash \texttt{beq}\ r_s, v : \Gamma \to \Gamma} \quad \text{(S-BEQ)}$$

Instruction Sequences $\boxed{\Psi \vdash I : \tau}$

$$\frac{\Psi;\Gamma \vdash v : \texttt{code}(\Gamma)}{\Psi \vdash \texttt{jmp}\ v : \texttt{code}(\Gamma)} \quad \text{(S-JMP)}$$

$$\frac{\Psi \vdash \iota : \Gamma \to \Gamma_2 \qquad \Psi \vdash I : \texttt{code}(\Gamma_2)}{\Psi \vdash \iota; I : \texttt{code}(\Gamma)} \quad \text{(S-SEQ)}$$

$$\frac{\Psi \vdash I : \tau}{\Psi \vdash I : \forall\alpha.\tau} \quad \text{(S-GEN)}$$

Register Files $\boxed{\Psi \vdash R : \Gamma}$

$$\frac{\forall r \in dom(\Gamma).\Psi \vdash R(r) : \tau}{\Psi \vdash R : \Gamma} \quad \text{(S-REGFILE)}$$

Heaps $\boxed{\vdash H : \Psi}$

$$\frac{\forall \ell \in dom(\Psi).\Psi \vdash H(\ell) : \Psi(\ell) \qquad FTV(\Psi(\ell)) = \emptyset}{\vdash H : \Psi} \quad \text{(S-HEAP)}$$

Machine States $\boxed{\vdash M}$

$$\frac{\vdash H : \Psi \qquad \Psi \vdash R : \Gamma \qquad \Psi \vdash I : \texttt{code}(\Gamma)}{\vdash (H, R, I)} \quad \text{(S-MACH)}$$

Figure 8-4: TAL-0 Typing Rules

tion of type variables which occur free in the context Ψ, as is the case with generalization in ML. This is because Ψ will only contain closed types (see below). Thus, generalization is always possible.

The judgment $\Psi \vdash R : \Gamma$ asserts that the register file R is accurately described by Γ, under the assumptions of Ψ. Similarly, the judgment $\vdash H : \Psi$ asserts that the heap H is accurately described by Ψ. Note that this is essentially the same rule as a "letrec" for declarations in a conventional functional language: We get to assume that the labels have their advertised type, and then check for any inconsistencies within their definitions. This allows labels to refer to one another directly. Note also that we require the types in Ψ to be closed. This ensures that generalization remains valid.

Finally, the judgment $\vdash (H, R, I)$ puts the pieces together: We must have some type assignment Ψ that describes the heap H, a register file typing Γ that describes R consistent with Ψ, and I must be a continuation with a pre-

condition of Γ on the register file, under the assumptions of Ψ.

8.2.1 Some Examples and Subtleties

As a simple example of the type system in action, let us revisit the `prod` example:

```
prod: mov r3,0; // res := 0
      jmp loop

loop: beq r1,done;   // if a = 0 goto done
      add r3,r2,r3;  // res := res + b
      add r1,r1,-1;  // a := a - 1
      jmp loop

done: jmp r4         // return
```

Let Γ be the register file type:

$$\texttt{code\{r1,r2,r3:int,r4:}\forall\alpha\texttt{.code\{r1,r2,r3:int,r4:}\alpha\texttt{\}\}}$$

and let Ψ be the label type assignment that maps `prod`, `loop`, and and `done` to $\texttt{code}(\Gamma)$. Let us verify that the instruction sequence I associated with `loop` is indeed well-formed with the type that we have assigned it.

We must show that $\Psi \vdash \mathrm{I} : \texttt{code}(\Gamma)$. It suffices to show that each instruction preserves Γ, since the final `jmp` is to loop, which expects Γ. For the first instruction, we must show $\Psi;\Gamma \vdash \texttt{beq r1, done} : \Gamma$ using the S-BEQ rule:

$$\dfrac{\dfrac{}{\Psi;\Gamma \vdash \texttt{r1} : \Gamma(\texttt{r1}) = \texttt{int}}\,\text{S-REG} \quad \dfrac{\dfrac{}{\Psi \vdash \texttt{done} : \Psi(\texttt{done}) = \texttt{code}(\Gamma)}\,\text{S-LAB}}{\Psi;\Gamma \vdash \texttt{done} : \texttt{code}(\Gamma)}\,\text{S-VAL}}{\Psi \vdash \texttt{beq r1, done} : \Gamma \to \Gamma}$$

Next, we must show that adding `r2` to `r3` preserves Γ:

$$\dfrac{\dfrac{}{\Psi;\Gamma \vdash \texttt{r2} : \Gamma(\texttt{r2}) = \texttt{int}}\,\text{S-REG} \quad \dfrac{}{\Psi;\Gamma \vdash \texttt{r3} : \Gamma(\texttt{r3}) = \texttt{int}}\,\text{S-REG}}{\Psi \vdash \texttt{add r3, r2, r3} : \Gamma \to \Gamma}$$

Then, we must show that subtracting 1 from `r1` preserves Γ:

$$\dfrac{\dfrac{}{\Psi;\Gamma \vdash \texttt{r1} : \Gamma(\texttt{r2}) = \texttt{int}}\,\text{S-REG} \quad \dfrac{\dfrac{}{\Psi \vdash -1 : \texttt{int}}\,\text{S-INT}}{\Psi;\Gamma \vdash -1 : \texttt{int}}\,\text{S-VAL}}{\Psi \vdash \texttt{add r1, r1, } -1 : \Gamma \to \Gamma}$$

Finally, we must show that the `jmp` which terminates the sequence has type $\texttt{code}(\Gamma)$, using the S-JMP rule:

$$\dfrac{\dfrac{\dfrac{}{\Psi;\Gamma \vdash \texttt{loop} : \Psi(\texttt{loop}) = \texttt{code}(\Gamma)}\,\text{S-LAB}}{\Psi;\Gamma \vdash \texttt{loop} : \texttt{code}(\Gamma)}\,\text{S-VAL}}{\Psi \vdash \texttt{jmp loop} : \texttt{code}(\Gamma)}$$

Stringing the sub-proofs together using the S-SEQ rule, we can thus confirm that $\Psi \vdash I : \texttt{code}(\Gamma)$.

Carrying on, we can show that each label's code has the type associated with it. However, there is a major subtlety with the last `jmp r4` instruction and the type that we have assigned `r4` throughout the code. To understand this, consider the following:

```
foo: mov r1,bar
     jmp r1

bar: ...
```

What type can we assign to `bar`? Without the polymorphism, it must be a code type $\texttt{code}(\Gamma)$ such that $\Gamma(\texttt{r1}) = \texttt{code}(\Gamma)$, since the label `bar` will be in register `r1` when we jump to it. But with only simple types (i.e., no subtyping, polymorphism, or recursive types), there is no solution to this equation.

With our support for polymorphism, the problem can be averted. In particular, we can assign `bar` a polymorphic type τ of the form $\forall\alpha.\texttt{code}\{\texttt{r1} : \alpha, \ldots\}$. At the `jmp` instruction, we have a register file context of the form $\Gamma = \{\texttt{r1} : \tau, \ldots\}$ and we must show that $\Gamma \vdash \texttt{r1} : \texttt{code}(\Gamma)$. Using subsumption, we can instantiate the type of `r1`, which is τ, with τ to derive $\Gamma \vdash \texttt{r1} : \texttt{code}\{\texttt{r1} : \tau, \ldots\}$.

This explains why we have used a polymorphic type for `r4` in the `prod` example above. Of course, this problem can be solved in other ways. Clearly, adding recursive types provides a solution to the problem. An alternative is to add some type *Top* which is greater than or equal to all other types, and use this to forget the type of a register as we jump through it. Yet another solution is to treat register file types as partial maps, and provide a form of subtyping that lets you forget the type of a register, thereby making it unusable as an operand until it is assigned a value (e.g., by the `mov` instruction.) This last approach was the one used by the original TAL. We prefer the approach based on polymorphism, because there are many other compelling uses of this feature.

8.2.1 EXERCISE [★ ↛]: Draw the derivation of well-formedness for the instruction sequences associated with `prod` and `done`. □

For instance, polymorphism can also be used to achieve a type for "join-points" in a control-flow graph. Consider the situation where we have two jumps from distinct contexts to the same label:

```
{r1:int,...}
jmp baz

...
{r1:code{...},...}
jmp baz
```

What type should `baz` require of register `r1`? Again, without support for some form of polymorphism or subtyping, we would be forced to make the types the same, in which case this code would be rejected. The problem could be worked around by, for instance, always loading an integer into `r1` before jumping to this label. But that would slow down the program with unnecessary instructions. Fortunately, polymorphism saves us again. In particular, we can assign `baz` a type of the form $\forall\alpha.$`code{r1:`α`,...}`. Then, at the first `jmp`, we would instantiate α to `int`, whereas at the second `jmp`, we would instantiate α with the appropriate `code` type. Of course, the addition of *Top* would also provide a convenient mechanism for typing join points.

One other feature that polymorphism provides which cannot be captured through simple subtyping, is the idea of *callee-saves* registers. A callee-save register is a register whose value should remain the same across a procedure call. If the procedure wishes to use that register, then it is responsible for saving and restoring its value before returning to the caller. Of course, we don't yet have a way to save and restore registers to memory, but a procedure could shuffle values around into different registers.

Suppose we wish to call a procedure, such as `prod`, and guarantee that the register `r5` is preserved across the call. We can accomplish this by requiring the procedure's entry label to have a type of the form:

$\forall\alpha.${`r5:`α`,r4:`$\forall\beta.$`code{r5:`α`,r4:`β`,...},...}`

where `r4` is the register which is meant to hold the return address for the procedure, and "..." does not contain a free occurrence of α. Note that the return address's type specifies that `r5` must have the same type (α) upon return as was originally passed in. Furthermore, the procedure is required to treat `r5` uniformly since its type is abstract. Since there is no way to manufacture values of abstract type, and since we've only passed in one α value,

it must be that if the procedure ever returns, then `r5` has the same value in it as it did upon entry. Note that the procedure is free to move `r5`'s value into some other register, and to use `r5` to hold other values. But before it can return, it must restore the original value.

So, it is clear that polymorphism can play a unifying role in the design of type systems for low-level languages. It provides a way for us to conveniently "forget" types, which is necessary for jumps through registers and join points. But it also provides an ability to capture critical compiler invariants, such as callee-saves registers.

8.2.2 EXERCISE [RECOMMENDED, ★ ↛]: Verify that the instruction sequences associated with the labels `prod` and `done` are well-typed. □

8.2.3 EXERCISE [RECOMMENDED, ★ ↛]: Suppose we change the `done` instruction sequence from `jmp r4` to `jmp r1`. Show that there is no way to prove the resulting code is well-typed. □

8.2.4 EXERCISE [RECOMMENDED, ★★★, ↛]: Reformulate the type system by eliminating type variables and universal polymorphism in favor of a *Top* type and subtyping. Then show how the product example can be typed under your rules. □

8.2.5 EXERCISE [★★★, ↛]: Reformulate the type system by using recursive types in lieu of polymorphism. Then show how the product example can be typed under your rules. □

8.2.6 EXERCISE [★★★★, ↛]: Prove that the approach to callee-saves registers actually preserves values. □

8.2.2 Proof of Type Soundness for TAL-0

We now wish to show that the type system given in the previous section actually enforces our desired safety property. In particular, we wish to show that, given a well-typed machine state M, then M cannot get stuck (i.e., jump or branch to an integer or undefined label.) It suffices to show that a well-typed machine state is not immediately stuck (progress), and that when it steps to a new machine state M', that state is also well-typed (preservation). For then, by induction on the length of an evaluation sequence, we can argue that there is no stuck M' such that $M \longrightarrow^* M'$.

Our first step is to establish a set of substitution lemmas which show that derivations remain possible after substituting types for type variables:

8.2.7 LEMMA [TYPE SUBSTITUTION]: If:

1. $\Psi;\Gamma \vdash v : \tau_1$, then $\Psi;\Gamma[\tau/\alpha] \vdash v : \tau_1[\tau/\alpha]$.

2. $\Psi \vdash \iota : \Gamma_1 \to \Gamma_2$ then $\Psi \vdash \iota : \Gamma_1[\tau/\alpha] \to \Gamma_2[\tau/\alpha]$.

3. $\Psi \vdash I : \tau_1$, then $\Psi \vdash I : \tau_1[\tau/\alpha]$.

4. $\Psi \vdash R : \Gamma$, then $\Psi \vdash R : \Gamma[\tau/\alpha]$. □

The register substitution lemma ensures that typing is preserved when we look up a value in the register file. It corresponds the the value substituion lemma in a soundness proof for a conventional lambda calculus.

8.2.8 LEMMA [REGISTER SUBSTITUTION]: If $\vdash H : \Psi$, $\Psi \vdash R : \Gamma$ and $\Psi;\Gamma \vdash v : \tau$ then $\Psi;\Gamma \vdash \hat{R}(v) : \tau$ □

As usual, we shall need a Canonical Values lemma that tells us what kind of value we have from its type:

8.2.9 LEMMA [CANONICAL VALUES]: If $\vdash H : \Psi$ and $\Psi \vdash v : \tau$ then:

1. If $\tau = \texttt{int}$ then $v = n$ for some n.

2. If $\tau = \texttt{code}(\Gamma)$ then $v = \ell$ for some $\ell \in dom(H)$ and $\Psi \vdash H(\ell) : \texttt{code}(\Gamma)$. □

This extends to operands as follows:

8.2.10 LEMMA [CANONICAL OPERANDS]: If $\vdash H : \Psi$, $\Psi \vdash R : \Gamma$, and $\Psi;\Gamma \vdash v : \tau$ then:

1. If $\tau = \texttt{int}$ then $\hat{R}(v) = n$ for some n.

2. If $\tau = \texttt{code}(\Gamma)$ then $\hat{R}(v) = \ell$ for some $\ell \in dom(H)$ and $\Psi \vdash H(\ell) : \texttt{code}(\Gamma)$. □

8.2.11 THEOREM [SOUNDESS OF TAL-0]: If $\vdash M$, then there exists an M' such that $M \longrightarrow M'$ and $\vdash M'$. □

Proof: Suppose $M = (H, R, I)$ and $\vdash M$. By inversion of the S-MACH rule, there exists a Ψ and Γ such that (a) $\vdash H : \Psi$, (b) $\Psi \vdash R : \Gamma$, and (c) $\Psi \vdash I : \texttt{code}(\Gamma)$. The proof proceeds by induction on I.

case $I = \texttt{jmp}\ v$: From (c) and inversion of S-JMP, we have $\Psi;\Gamma \vdash v : \texttt{code}(\Gamma)$. From the Canonical Operands lemma, we know that there exists an I' such that $H(\hat{R}(v)) = I'$ and $\Psi \vdash I' : \texttt{code}(\Gamma)$. Taking $M' = (H, R, I')$, we can show $M \longrightarrow M'$ via the JMP rule. We must now show $\vdash (H, R, I')$, but this follows immediately.

case $I = \texttt{mov}\ r_d, v; I'$: From inversion of the S-SEQ rule, we have $\Psi \vdash \texttt{mov}\ r_d, v : \Gamma \to \Gamma_2$ and $\Psi \vdash I' : \texttt{code}(\Gamma_2)$ for some Γ_2. Then, by inversion of the S-MOV rule, we have $\Psi; \Gamma \vdash v : \tau$ and $\Gamma_2 = \Gamma[r_d : \tau]$ for some τ. By the Register Substitution lemma, we have $\Psi; \Gamma \vdash \hat{R}(v) : \tau$. Taking $M' = (H, R[r_d = \hat{R}(v)], I')$, we see that $M \longrightarrow M'$ via the MOV rule. From the S-REGFILE rule, we conclude that $\Psi \vdash R[r_d = \hat{R}(v)] : \Gamma[r_d : \tau]$.

case $I = \texttt{add}\ r_d, r_s, v; I'$: From inversion of the S-SEQ rule, we have $\Psi \vdash \texttt{add}\ r_d, r_s, v : \Gamma \to \Gamma_2$ and $\Psi \vdash I' : \texttt{code}(\Gamma_2)$ for some Γ_2. Then, by inversion of the S-ADD rule, we have $\Psi; \Gamma \vdash r_s : \texttt{int}$, $\Psi; \Gamma \vdash v : \texttt{int}$ and $\Gamma_2 = \Gamma[r_d : \texttt{int}]$. From the and Canonical Operand lemma, we know that there exists integers n_1 and n_2 such that $R(r_s) = n_1$ and $\hat{R}(v) = n_2$. Taking $n = n_1 + n_2$ and $M' = (H, R[r_d = n], I')$, we see that $M \longrightarrow M'$ via the ADD rule. By the S-INT and S-VAL rules, $\Psi; \Gamma[r_d : \texttt{int}] \vdash n : \texttt{int}$. Thus, by the S-REGFILE rule, we conclude that $\Psi \vdash R[r_d = n] : \Gamma[r_d : \texttt{int}]$.

case $I = \texttt{beq}\ r_s, v; I'$: From inversion of the S-SEQ rule, we have $\Psi \vdash \texttt{beq}\ r_s, v : \Gamma \to \Gamma_2$ and $\Psi \vdash I' : \texttt{code}(\Gamma_2)$ for some Γ_2. Then, by inversion of the S-BEQ rule, we have $\Psi; \Gamma \vdash r_s : \texttt{int}$, $\Psi; \Gamma \vdash v : \texttt{code}(\Gamma)$ and $\Gamma_2 = \Gamma$. By the Canonical Operands lemma, there exists an ℓ and I_2 such that $\hat{R}(v) = \ell$, $H(\ell) = I_2$, and $\Psi \vdash I_2 : \texttt{code}(\Gamma)$. Also by Canonical Operands, $R(r_s) = n$ for some integer n. If $n = 0$ then $M \longrightarrow (M, R, I_2)$ via BEQ-EQ. If $n \neq 0$ then $M \longrightarrow (M, R, I')$ via BEQ-NEQ. In either case, the well-formedness of the resulting machine state follows from the S-MACH rule. □

8.2.3 Proof Representation and Checking

It is not clear whether type inference for TAL-0 machine states is decidable. That is, given a machine state (H, R, I), does there exist a Ψ and Γ such that $\vdash H : \Psi$, $\Psi \vdash R : \Gamma$, and $\Psi \vdash : \texttt{code}(\Gamma)$? On the one hand, this seems possible since the type system is so simple. On the other hand, the system, as presented, supports polymorphic recursion for which inference is known to be undecidable in the context of the lambda calculus. Furthermore, as we progress to more advanced typing features, the decidability of type reconstruction will surely vanish. Thus, in any practical realization, we must require some help for constructing a proof that the code is indeed type-correct.

In the case of TAL-0, it is sufficient to provide types for the labels (i.e., Ψ). Indeed, it is even possible to omit types for some labels and keep reconstruction decidable. We really only need enough type information to cut each loop in the control-flow graph, or for those labels that are moved into registers. Minimizing the type information is an important goal in any practical system, since the size of the types can often be larger than the code itself!

However, it is desirable to keep the type checker as simple as possible so that we can trust it is properly enforcing the type system.

One way to keep the type checker simple is to modify the syntax so that type reconstruction is entirely syntax directed. By this, we mean simply that for any given term, at most one rule should apply. Furthermore, the checker should not have to "guess" any of the sub-goal components. For TAL-0, this could be accomplished by (a) requiring types on all labels and (b) adding a form of explicit type instantiation to operands (e.g., $v[\tau]$).

An alternative approach is to force the code provider to ship an explicit representation of the complete proof of well-formedness, along with the code, and make sure that the proof and the code have the same instructions, labels, etc. Of course, the proof will tend to be much larger than the code itself, but we can use various techniques to reduce the size of the proof representation (see for instance Necula and Lee, 1998). Such a separation of proofs and code is advantageous because we can ship the *binary* machine code (as opposed to the assembly code), disassemble it, and then compare it against the assembly-level proof. If everything checks out, then we can load the binary and execute it directly. Such an approach was used by Necula's Touchstone compiler, and the Special-J compiler.

In the rest of this paper, we will remain vague about how proofs are to be represented. The key thing to note is that we are not limited in the choice of type constructors by issues of inference. Rather, we will require that the code producer provide us with enough evidence that we can easily reconstruct and check the proof of well-formedness. Therefore, our only limitation will be the incompletenesses of the resulting proof system.

8.2.12 EXERCISE [★★★★, ↛]: Build a type-checker for TAL-0 in your favorite programming language. Assume that you are given as input a set of labels, their associated types, and instruction sequences. Furthermore, assume that operands are augmented with explicit syntax for polymorphic instantiation. □

8.3 TAL-1: Simple Memory-Safety

TAL-0 includes registers and heap-allocated code, but provides no support for allocated *data*. In this section, we will add primitive support for allocated objects that can be shared by reference (i.e., pointer) and extend our safety property to include a notion of object-level memory safety: No memory access should read or write a data object at a given location unless the program has been granted access to that location.

From a typing perspective, the critical issue will be how to accomodate

locations that hold values of different types at different times during the execution of the program. We need such a facility to at least support the construction of compound values, such as tuples, records, datatype constructors, or objects. A high-level language, such as ML, provides mechanisms to allocate and initialize data structures as a single expression. For instance, `{x = 3, y = 4}` is an expression that builds a record with two components. At the assembly level, such high-level compound expressions must be broken into machine-level steps. We must first allocate space for the object, and then initialize the components by storing them in that space. To prevent someone from treating an uninitialized component as if it holds a valid value, we must use a different type. But obviously, once we initialize that component, its type should change to reflect that it is now valid for use.

Already, we have support for storing values of different types in registers. For instance, nothing prevents us from moving a `code` value into a register which currently holds an `int`. However, when we add allocated data objects, we can no longer track the changes easily due to *aliasing*. Let `ptr(`τ`)` denote a pointer to a data object of type τ and consider the following sequence of instructions:

```
      {r1:ptr(code(...))}
1.    mov r3,0            // r3 := 0
2.    st  r3,r1(0)        // Mem[r1] := r3
3.    ld  r4,r1(0)        // r4 := Mem[r1]
4.    jmp r4
```

We assume upon entry that `r1` is a pointer to a data location that contains a code label. The first two instructions overwrite the contents of memory at the location in `r1` with the integer 0. The third instruction loads the value from the location in `r1` and jumps to it. Clearly, this code should be rejected by the type-checker as it violates our control-flow safety property. To ensure this, we might require that the type system update the type of `r1` whenever we store through it. For instance, after the second instruction, the type of `r1` would change from `ptr(code(...))` to `ptr(int)`. Then at instruction four, the code would be rejected because of an attempt to jump to an integer.

Now consider this sequence:

```
      {r1:ptr(code(...)),r2:ptr(code(...))}
1.    mov r3,0            // r3 := 0
2.    st  r3,r1(0)        // Mem[r1] := r3
3.    ld  r4,r2(0)        // r4 := Mem[r2]
4.    jmp r4
```

The code is exactly the same except that instead of loading the value pointed to by `r1`, we load the value pointed to by `r2`. Should this code type-check?

The answer depends on whether or not `r1` and `r2` hold the same value—that is, whether or not they are aliases for the same location. There is no problem when they are not aliases, but when they are, the code behaves the same as in the previous example (i.e., attempts to jump to the integer 0.) It becomes clear that to prevent this problem, whenever we update a memory location with a value of a different type, we must update the types of *all* aliases to that location. To do so, the type system must track whether or not two values are the same and, more generally, whether or not two code labels behave the same.

Of course, there is no complete logic for tracking the equalities of values and computations, but it is possible to construct a powerful type system that allows us to conservatively track equality of *some* values. For instance see the work on *alias types* (Smith, Walker, and Morrisett, 2000b; Walker and Morrisett, 2001; DeLine and Fähndrich, 2001). But all of these systems are, in my opinion, technically daunting. Furthermore, certifying compilers for high-level languages rarely need the complex machinery that they introduce.

Nonetheless, we need some support for (a) allocating and initializing data structures that are to be shared, and (b) stack-allocating procedure frames. Therefore, we will focus on typing principles that try to strike a balance between expressiveness and complexity. After all, our goal is to provide simple but expressive structure for implementing type-safe high level languages on conventional architectures.

In particular, we will use the type system to separate locations into one of two classes: The first class, called *shared pointers*, will support arbitrary aliasing. However, the types of the contents of shared pointers must remain invariant. That is, we can never write a value of a different type into the contents of a shared location. This is the same basic principle that ML-style `ref`s and other high-level languages follow.

The second class of locations, called *unique pointers*, will support updates that change the type of the contents. However, unique pointers cannot be aliased. In particular, we will prevent unique pointers from being copied. Thus, they will behave much the same way as registers.

The combination of unique and shared pointers will provide us with a simple, but relatively flexible framework for dealing with memory. In particular, we will be able to use unique pointers to handle the thorny problem of allocating and initializing shared data structures. We will also be able to use unique pointers to model data structures whose lifetime is controlled by the compiler, such as stack frames.

r	::=		*registers:*		`sfree` n	*free n stack words*
		`r1` | `r2` | $\cdots$ | `rk`	*gp registers*	v ::=		*operands:*
		`sp`	*stack pointer*		r	*registers*
ι	::=		*instructions:*		n	*integer literals*
		$\cdots$	*as in TAL-0*		ℓ	*code or shared data pointers*
		`ld` $r_d, r_s(n)$	*load from memory*		`uptr`(h)	*unique data pointers*
		`st` $r_s, r_d(n)$	*store to memory*	h	::=		*heap values:*
		`malloc` r_d, n	*allocate n heap words*			I	*instruction sequences*
		`commit` r_d	*become shared*			$\langle v_1, \ldots, v_n \rangle$	*tuples*
		`salloc` n	*allocate n stack words*				

Figure 8-5: TAL-1 Syntax Additions

8.3.1 The TAL-1 Extended Abstract Machine

Figure 8-5 gives a set of syntactic extensions to TAL-0 which are used in the definition of TAL-1. We add six new instructions. The `ld` and `st` instructions can be used to load a value from memory into a register, and store a register's value to memory respectively. The effective address for both instructions is calculated as a word-level offset from a base register. The other instructions are non-standard. The `malloc` instruction is used to allocate an object with n words. A (unique) reference to the object is placed in the destination register. Typically, `malloc` will be implemented by the concrete machine using a small sequence of inlined instructions or a procedure call. We abstract from these details here so that our abstract machine can support a wide variety of allocation techniques. The `commit` instruction is used to coerce a unique pointer to a shared pointer. It has no real run-time effect but it makes it easier to state and prove the invariants of the type system.

The `salloc` and `sfree` constructs manipulate a special unique pointer which is held in a distinguished register called `sp` (stack pointer). The instruction `salloc` attempts to grow the stack by n words, whereas `sfree` shrinks the stack by n words. The type system will prevent the stack from underflowing, so in principle, `sfree` could be implemented by a simple arithmetic operation (e.g., `add sp, sp,` n.) Unfortunately, stack overflow will not be captured by this type system. Therefore, we assume that the `salloc` instruction checks for overflow and aborts the computation somehow.

As before, we will model machine states using a triple of a heap, register file, and instruction sequence. And, as before, register files will map registers to word-sized values, while heaps will map labels to heap-values. We extend

heap values to include tuples of word-sized values. Thus, a label can refer to either code or data. We could also use the heap to store unique data values, but this would make it more difficult to prove that the pointers to these values are indeed unique. Instead, we will extend operands with terms of the form $\texttt{uptr}(h)$ and use such a term to represent a unique pointer to a heap value h.

The rewriting rules for the instructions of TAL-1 that overlap with TAL-0 remain largely the same. However, we must prevent unique pointers from being copied. More precisely, we must prevent the situation where we have two references to the same unique data. Note that, for the `add` and `beq` instructions, the use of a unique pointer as an operand will cause the machine to get stuck since the operands must be integers to make progress. However, the `mov` instruction must be changed to prevent copies of unique pointers:

$$\frac{\hat{R}(v) \neq \texttt{uptr}(h)}{(H, R, \texttt{mov}\ r_d, v; I) \longrightarrow (H, R[r_d = v], I)} \quad \text{(MOV-1)}$$

This rule can only fire when the source operand is not a unique pointer.

We must now give the rewriting rules for the new instructions:

$$(H, R, \texttt{malloc}\ r_d, n; I) \longrightarrow (H, R[r_d = \texttt{uptr}\langle m_1, \ldots, m_n\rangle], I) \quad \text{(MALLOC)}$$

$$\frac{r \neq \texttt{sp} \qquad \ell \notin \mathit{dom}(H)}{(H, R[r_d = \texttt{uptr}(h)], \texttt{commit}\ r_d; I) \longrightarrow (H[\ell = h], R[r_d = \ell], I)} \quad \text{(COMMIT)}$$

$$\frac{R(r_s) = \ell \qquad H(\ell) = \langle v_0, \ldots, v_n, \ldots, v_{n+m}\rangle}{(H, R, \texttt{ld}\ r_d, r_s(n); I) \longrightarrow (H, R[r_d = v_n], I)} \quad \text{(LD-S)}$$

$$\frac{R(r_s) = \texttt{uptr}\langle v_0, \ldots, v_n, \ldots, v_{n+m}\rangle}{(H, R, \texttt{ld}\ r_d, r_s(n); I) \longrightarrow (H, R[r_d = v_n], I)} \quad \text{(LD-U)}$$

$$\frac{R(r_d) = \ell \qquad H(\ell) = \langle v_0, \ldots, v_n, \ldots, v_{n+m}\rangle \qquad R(r_s) = v \qquad v \neq \texttt{uptr}(h)}{(H, R, \texttt{st}\ r_s, r_d(n); I) \longrightarrow (H[\ell = \langle v_0, \ldots, v, \ldots, v_{n+m}\rangle], R, I)} \quad \text{(ST-S)}$$

$$\frac{R(r_d) = \texttt{uptr}\langle v_0, \ldots, v_n, \ldots, v_{n+m}\rangle, \qquad R(r_s) = v \qquad v \neq \texttt{uptr}(h)}{(H, R, \texttt{st}\ r_s, r_d(n); I) \longrightarrow (H, R[r_d = \texttt{uptr}\langle v_0, \ldots, v, \ldots, v_{n+m}\rangle], I)} \quad \text{(ST-U)}$$

$$\frac{R(\texttt{sp}) = \texttt{uptr}\langle v_0, \ldots, v_p\rangle \qquad p + n \leq \texttt{MAXSTACK}}{(H, R, \texttt{salloc}\ n) \longrightarrow (H, R[\texttt{sp} = \texttt{uptr}\langle m_1, \ldots, m_n, v_0, \ldots, v_p\rangle])} \quad \text{(SALLOC)}$$

$$\frac{R(\texttt{sp}) = \texttt{uptr}\langle m_1, \ldots, m_n, v_0, \ldots, v_p\rangle}{(H, R, \texttt{salloc}\ n) \longrightarrow (H, R[\texttt{sp} = \texttt{uptr}\langle v_0, \ldots, v_p\rangle])} \quad \text{(SFREE)}$$

The `malloc` instruction places a unique pointer to a tuple of n words in the destination register. We assume that the memory management subsystem

has initialized the tuple with some arbitary integer values $m_1, \ldots, m_n$. Recall that the rewriting rules prevent these unique pointers from being copied. However, the `commit` instruction allows us to move a unique pointer into the heap where it can be shared. As we will see, however, shared pointers must have invariant types, whereas unique pointers' types can change.

The `ld` instruction has two variants, depending upon whether the source register holds a value that is shared or unique. If it is shared, then we must look up the binding in the heap to get the heap value. If it is unique, then the heap value is immediately available. Then, in both cases, the heap value should be a tuple. We extract the n^{th} word and place it in the destination register.

The `st` instruction is the dual and is used to update the nth component of a tuple. Note that the machine gets stuck on an attempt to store a unique pointer, thereby preventing copies from leaking into a data object.

As a simple example of the use of these constructs, consider the following code:

```
copy: {r1:ptr(int,int), r2,r3:int}
      malloc r2,2
      ld r3,r1(0)
      st r3,r2(0)
      ld r3,r1(1)
      st r3,r2(1)
      commit r2
      {r1:ptr(int,int), r2:ptr(int,int), r3:int }
```

The code is meant to do a deep copy of the data structure pointed to by `r1` and place the copy in register `r2`. Suppose that r1 holds a label ℓ_1 and $H(\ell_1) = \langle 3,5\rangle$. After executing the malloc instruction, `r2` will hold a unique pointer to a pair of (arbitrary) integers of the form $\texttt{uptr}\langle m_1, m_2\rangle$. After the `ld` and `st`, the first integer component of `r1` will have been copied into the first component of `r2`. Thus, the contents of ℓ_2 will have changed to $\texttt{uptr}\langle 3, m_2\rangle$. After the second `ld` and `st`, the second component will have been copied, so `r2` will hold the value $\texttt{uptr}\langle 3,5\rangle$. Finally, after the `commit` instruction, `r2` will hold a fresh, shared label ℓ_2 and the heap will have been extended so that it maps ℓ_2 to the heap value $\langle 3,5\rangle$.

Here is an example program which uses `salloc` and `sfree`:

```
foo: {sp : uptr(int), r1 : code{...}}
      salloc 2     // {sp : uptr(int,int,int)}
      st  r1,sp(0) // {sp : uptr(code{...},int,int)}
      sfree 1      // {sp : uptr(int,int)}
```

On input, the stack has one integer element and `r1` has a code pointer. The

τ ::=		*operand types:*
	...	*as in TAL-0*
	$\texttt{ptr}(\sigma)$	*shared data pointers*
	$\texttt{uptr}(\sigma)$	*unique data pointers*
	$\forall\rho.\tau$	*quantification over allocated types*

σ ::=		*allocated types:*
	ϵ	*empty*
	τ	*value type*
	σ_1, σ_2	*adjacent*
	ρ	*allocated type variable*

Figure 8-6: TAL-1 Types

first instruction grows the stack by two words. The second instruction stores the value in `r1` into the top of the stack. The third instruction frees one of the words. Note that `salloc` becomes stuck if we attempt to allocate more than `MAXSTACK` (total) words and that `sfree` becomes stuck if we attempt to shrink the stack by more words than are on the stack. Finally, note that our stacks grow "up" (indexing is positive) whereas the common convention is to have stacks that grow down. The only reason for this choice is that it unifies unique pointers to tuples with stacks. If we wanted to support downward stacks, then we could introduce a new kind of data structure (e.g., `stptr`.)

8.3.1 EXERCISE [RECOMMENDED, ★ ↛]: Show how stack-push and stack-pop instructions can be explained using the primitives provided by TAL-1. Explain how a sequence of pushes or a sequence of pops can be optimized. □

8.3.2 EXERCISE [★★★ ↛]: Modify the abstract machine so that unique pointers are allocated in the heap, just like shared pointers, but are represented as a tagged value of the form $\texttt{uptr}(\ell)$. Then show that the machine maintains the invariant that there is at most one copy of a unique pointer. □

8.4 TAL-1 Changes to the Type System

What changes and additions are needed to the type system to ensure that the new abstract machine won't get stuck? In particular, how do we ensure that when we do a load, the source register contains a data pointer (as opposed to an integer or code label), and the data pointer points to a heap value that has at least as many components as the offset requires? Similarly, how do we ensure that for a store, the destination register is a data pointer to a large enough heap value? And how do we ensure that the stack does not underflow? How do we ensure that we don't try to copy a unique pointer? In short, how do we ensure progress?

Figure 8-6 gives a new set of types for classifying TAL-1 values. The τ types are used to classify values and operands, whereas the σ types are used to

Heap Values $\boxed{\Psi \vdash v : \tau}$

$$\frac{\Psi;\Gamma \vdash v_i : \tau_i}{\Psi \vdash \langle v_1, \ldots, v_n \rangle : \tau_1, \ldots, \tau_n} \quad \text{(S-TUPLE)}$$

Operands $\boxed{\Psi \vdash v : \tau}$

$$\frac{\Psi;\Gamma \vdash h : \sigma}{\Psi;\Gamma \vdash \texttt{uptr}(h) : \texttt{uptr}(\sigma)} \quad \text{(S-UPTR)}$$

Figure 8-7: TAL-1 Typing Rules (Heap Values and Operands)

classify heap-allocated data. We have added three new operand types corresponding to shared pointers ($\texttt{ptr}(\sigma)$), unique pointers ($\texttt{uptr}(\sigma)$), and polymorphism over allocated types ($\forall\rho.\tau$.)

The allocated types (σ) consist of sequences of operand types. The syntax supports nesting structure (i.e., trees) but we implicitly treat adjacency as associative with ϵ as a unit. So, for instance:

$$\texttt{ptr}(\texttt{int}, (\rho, (\texttt{int}, \epsilon))) = \texttt{ptr}((\texttt{int}, \rho), \texttt{int})$$

Allocated types also support variables (ρ) which are useful for abstracting a chunk of memory. That is, α can be used to abstract a single word-sized type, whereas ρ can be used to abstract a type of arbitrary size. As we will see, polymorphism over allocated types is the key to efficient support for procedures.

Figure 8-7 gives the new typing rules. As in TAL-0, we take Γ to be a total map from registers to operand types. We are also assuming that Ψ is a finite partial function from labels to allocated operand types (i.e., `code` or `ptr` types.)

The well-formedness rules for tuples and unique pointers are straightforward. The S-MOV-1 rule defines the new type for the `mov` instruction. It has as a pre-condition that the value being moved should not be a unique pointer.

The typing rule for `malloc` requires a non-negative integer argument, and updates the destination register's type with a unique pointer of an n-tuple of integers. The `commit` instruction expects a unique pointer in the given register, and simply changes its type to a shared pointer.

The `ld` and `st` instructions require two rules each, depending upon whether they are operating on unique pointers. Notice that for the `st` rules, we are not allowed to place a unique pointer into the data structure. Notice also that that when storing into a unique pointer, there is no requirement that the new value have the same type as the old value. In contrast, for shared pointers, the old and new values must have the same type.

The typing rules for `salloc` and `sfree` are relatively straightforward. For `sfree` n we must check that there are at least n *values* on the stack to avoid underflow. Note that the rule does not allow allocated type variables (ρ) to

Instructions $\boxed{\Psi \vdash \iota : \Gamma_1 \to \Gamma_2}$

$$\frac{\Psi;\Gamma \vdash v : \tau \qquad \tau \neq \mathtt{uptr}(\sigma)}{\Psi \vdash \mathtt{mov}\ r_d, v : \Gamma \to \Gamma[r_d : \tau]} \quad \text{(S-MOV-1)}$$

$$\frac{n \geq 0}{\Psi \vdash \mathtt{malloc}\ r_d, n : \Gamma \to \Gamma[r_d : \mathtt{uptr}\langle \mathtt{int}_0, \ldots, \mathtt{int}_n \rangle]} \quad \text{(S-MALLOC)}$$

$$\frac{\Psi;\Gamma \vdash r_d : \mathtt{uptr}(\sigma) \qquad r_d \neq \mathtt{sp}}{\Psi \vdash \mathtt{commit}\ r_d : \Gamma \to \Gamma[r_d : \mathtt{ptr}(\sigma)]} \quad \text{(S-COMMIT)}$$

$$\frac{\Psi;\Gamma \vdash r_s : \mathtt{ptr}(\tau_1, \ldots, \tau_n, \sigma)}{\Psi \vdash \mathtt{ld}\ r_d, r_s(n) : \Gamma \to \Gamma[r_d : \tau_n]} \quad \text{(S-LDS)}$$

$$\frac{\Psi;\Gamma \vdash r_s : \mathtt{uptr}(\tau_1, \ldots, \tau_n, \sigma)}{\Psi \vdash \mathtt{ld}\ r_d, r_s(n) : \Gamma \to \Gamma[r_d : \tau_n]} \quad \text{(S-LDU)}$$

$$\frac{\Psi;\Gamma \vdash r_s : \tau_n \qquad \tau_n \neq \mathtt{uptr}(\sigma') \qquad \Psi;\Gamma \vdash r_d : \mathtt{ptr}(\tau_1, \ldots, \tau_n, \sigma)}{\Psi \vdash \mathtt{st}\ r_s, r_d(n) : \Gamma \to \Gamma} \quad \text{(S-STS)}$$

$$\frac{\Psi;\Gamma \vdash r_s : \tau \qquad \tau \neq \mathtt{uptr}(\sigma') \qquad \Psi;\Gamma \vdash r_d : \mathtt{uptr}(\tau_1, \ldots, \tau_n, \sigma)}{\Psi \vdash \mathtt{st}\ r_s, r_d(n) : \Gamma \to \Gamma[r_d : \mathtt{uptr}(\tau_1, \ldots, \tau, \sigma)]} \quad \text{(S-STU)}$$

$$\frac{\Psi;\Gamma \vdash \mathtt{sp} : \mathtt{uptr}(\sigma) \qquad n \geq 0}{\Psi \vdash \mathtt{salloc}\ n : \Gamma \to \Gamma[\mathtt{sp} : \mathtt{uptr}(\mathtt{int}_1, \ldots, \mathtt{int}_n, \sigma)]} \quad \text{(S-SALLOC)}$$

$$\frac{\Psi;\Gamma \vdash \mathtt{sp} : \mathtt{uptr}(\tau_1, \ldots, \tau_n, \sigma)}{\Psi \vdash \mathtt{sfree}\ n : \Gamma \to \Gamma[\mathtt{sp} : \mathtt{uptr}(\sigma)]} \quad \text{(S-SFREE)}$$

Figure 8-8: TAL-1 Typing Rules (Instructions)

be eliminated. This reflects the fact that, in general, we do not know how many words are occupied by ρ. For instance, ρ could be instantiated with ϵ in which case there are no values. A similar restriction happens for both `ld` and `st`—we must at least know the sizes up through the word component that we are projecting or storing.

To prevent the machine from becoming stuck, `salloc` would ideally check that adding n words to the stack would not exceed `MAXSTACK`. But alas, we cannot always determine the current length of the stack. In particular, if its type contains an allocated variable ρ, then we are in trouble. One way around

this problem is to change the abstract machine so that it does not get stuck upon overflow by defining a transition (e.g., by jumping to a pre-defined label). This would correspond to a machine trap due to an illegal access.

It is possible to extend our soundness proof for TAL-0 to cover the new constructs in TAL-1 and show that a well-formed machine state cannot get stuck, except when the stack overflows. In the proof of soundness, one critical property we must show is that any typing derivation remains valid under an extension of the heap. In particular, if $\vdash H : \Psi$ and $\Psi \vdash h : \tau$, then $\vdash H[\ell = h] : \Psi[\ell : \tau]$. Another critical property is that we would have to show that a given label ℓ that occurs in the heap has exactly one type throughout the execution of the program. In other words, once we have commited a pointer so that it can be shared, its type must remain invariant.

8.4.1 EXERCISE [★★★★ ↛]: Extend the proof of soundness to cover the new features in TAL-1. □

8.5 Compiling to TAL-1

At this point, TAL-1 provides enough mechanism that we can use it as a target language for the compiler of a polymorphic, procedural language with integers, tuples, records, and function pointers (but not yet lexically nested closures.) As a simple example, let us start with the following C code:

```
int prod(int x, int y) {
  int a = 0;
  while (x != 0) {
    a = a + y;
    x = x - 1;
  }
  return a;
}

int fact(int z) {
  if (z != 0) return prod(fact(z-1),z);
  else return 1;
}
```

and show how it may be compiled to TAL-1, complete with typing annotations on the code labels. We assume a calling convention where arguments are passed on the stack, and the return address is passed in `r4`. We also assume that results are returned in register `r1`, and arguments are popped of the stack by the callee. Finally, we assume that registers `r2` and `r3` are freely available as scratch registers.

Let us first translate the `prod` function under these conventions:

```
prod: ∀a,b,c,s.
       code{r1:a,r2:b,r3:c,sp:uptr(int,int,s),
            r4:∀d,e,f.code{r1:int,r2:d,r3:e,r4:f,sp:uptr(s)}}
      ld r2,sp(0) // r2:int, r2 := x
      ld r3,sp(1) // r3:int, r3 := y
      mov r1, 0   // r1:int, a := 0
      jmp loop

loop: ∀s.code{r1,r2,r3:int,sp:uptr(int,int,s),
            r4:∀d,e,f.code{r1:int,r2:d,r3:e,r4:f,sp:uptr(s)}}
      beq r2,done  // if x ↔ 0 goto done
      add r1,r1,r3 // a := a + y
      add r2,r2,-1 // x := x - 1
      jmp loop

done: ∀s.code{r1,r2,r3:int,sp:uptr(int,int,s),
            r4:∀d,e,f.code{r1:int,r2:d,r3:e,r4:f,sp:uptr(s)}}
      sfree 2      // sp:uptr(r)
      jmp r4
```

The code itself is rather straightforward. What is most interesting is the types we have placed on the labels. Note that, upon input to `prod`, `r1`, `r2`, and `r3` can have any types, since we have abstracted their types. Note also that the stack pointer has type `uptr(int,int,s)` and thus has two integers at the front, but can have any sequence of values following, since we have abstracted the tail with an allocated type variable `s`. The return address in `r4` is polymorphic for `r2` and `r3` to allow values of any type in those registers upon return. As discussed earlier, the type of `r4` is abstracted by the return address to allow jumping through that register. Finally, the return address demands that the stack have type `uptr(s)`, reflecting that the callee should pop the arguments before jumping to the return address. Furthermore, the contents of the rest of the stack is guaranteed to be preserved since `s` is abstract.

In general, a source-level procedure that takes arguments of types $\tau_1, \ldots, \tau_n$ and returns a value of type τ would translate to a label with the same type as `prod`'s, except that the stack pointer would have type $\texttt{uptr}(\tau_1, \ldots, \tau_n, s)$, and `r4`'s return code type would expect `r1` to have type τ.

The types for the `loop` and `done` labels are similar to `prod`'s, except that registers `r1`,`r2`, and `r3` must hold integers. The reader is encouraged to check that the resulting code is well-formed according to the typing rules for TAL-1.

Now let us translate the recursive factorial procedure using the same calling conventions:

```
fact: ∀a,b,c,s.
       code{r1:a,r2:b,r3:c,sp:uptr(int,s),
            r4:∀d,e,f.code{r1:int,r2:d,r3:e,r4:f,sp:uptr(s)}}
      ld  r1,sp(0)  // r1:int, r1 := z
      beq r1,retn   // if z = 0 goto retn
      add r2,r1,-1  // r2:int, r2 := z-1
      salloc 2      // sp:uptr(int,int,int,s)
      st  r4,sp(1)  // sp:uptr(int,(∀d,e,f.code{...}),int,s)
      st  r2,sp(0)  // sp:uptr(int,(∀d,e,f.code{...}),int,s)
      mov r4,cont
      jmp fact      // r1 := fact(z-1)

cont: ∀c,s,d,e,f.
       code{r1:int,r2:d,r3:e,r4:f,
            sp:uptr(∀d,e,f.code{...},int,s)}
      ld  r4,sp(0)  // restore original return address
      st  r1,sp(0)  // sp:uptr(int,int,s)
      jmp prod      // tail call prod(fact(z-1),z)

retn: ∀b,c,s.
       code{r1:int,r2:b,r3:c,sp:uptr(int,s),
            r4:∀d,e,f.code{r1:int,r2:d,r3:e,r4:f,sp:uptr(s)}}
      mov  r1,1
      sfree 1       // sp:uptr(s)
      jmp  r4       // return 1
```

The first couple of instructions load the argument from the stack, test if it is zero, and if so, jump to the `retn` block where the argument is popped off the stack and the value 1 is returned. If the argument `z` is non-zero, then we must calculate `z-1`, pass it in a recursive call to `fact`, and then pass the result along with `z` to the `prod` function.

To do the recursive call, we must allocate stack space for the return address (`r4`) and the argument `z-1`, and save those values on the stack. Then we must load a new return address into `r4`, namely `cont` and finally jump to `fact`. When the recursive call returns, control will transfer to the `cont` label. Notice that `cont` expects the stack to hold the original return address and the value of `z`. We restore the original return address by loading it from the stack. We then overwrite the same stack slot with the result of `fact(z-1)`. Finally, we do a tail-call to `prod`.

It is important to recognize that the calling conventions we chose are not specific to the abstract machine. For instance, we could have chosen a con-

vention where the return address is pushed on the stack, or where arguments are passed in registers, introduced callee-saves registers, etc. In contrast, virtual machines such as the JVM and CLR bake the notion of procedure and procedure call into the language. To add support for different calling conventions (e.g., tail-calls, or a tailored convention for leaf procedures) requires additions and changes to the abstract machine and its type system. In contrast, by focusing on a more primitive set of type constructors (e.g., $\forall$, `code`, and `uptr` types), we are able to accomodate many conventions without change to the type system.

8.5.1 EXERCISE [RECOMMENDED, ★★ ↛]: Rewrite the `fact` procedure with a calling convention where the arguments and return address are placed on the stack. Include typing annotations on the labels and convince yourself that the code type-checks. □

8.6 Some Real World Issues

Clearly, TAL-1 provides the mechanisms needed to implement very simple tuple or record-like data structures, as well as local stack allocation. Furthermore, the operations of the machine, with the exception of `malloc` and `commit`, have a one-to-one correspondence with the operations of a typical RISC-style machine. However, `commit` is, at the concrete machine level, a no-op and could be pushed into the proof representation. And abstracting `malloc` insulates us from details of the memory management runtime.

There are, of course, a number of simple extensions that could be made to make the type system a little more useful. For instance, we could annotate primitive memory type components with flags to control whether that component supports read-only, write-only, or read-write access.

8.6.1 EXERCISE [★↛]: Assuming you added type qualifiers for read-only and write-only access to tuple components. How would you change the abstract machine so that you captured their intended meaning? □

We could also add support for subtyping in a number of ways. For instance, we could take: $\texttt{ptr}(\sigma, \sigma') \leq \texttt{ptr}(\sigma)$. That is, we can safely forget the tail of a sequence of values, for both `ptr` and `uptr` types. We can also consider a read-write component to be a subtype of a read-only or a write-only component. For shared, read-only components, we can support covariant deep subtyping, and for write-only components, we can have contra-variant subtyping. Interestingly, it is sound to have covariant subtyping on read-write components of unique pointers. All of these extensions were supported in some form for TALx86.

An issue not addressed here is support for primitive values of sizes less than a machine word (e.g., a `char` or `short`). But this too is relatively easy to accomodate. The key thing is that we need some function from operand types to their sizes so that we can determine whether or not a `ld` or `st` is used at the right offset. A slightly more troublesome problem is the issue of alignment. On many architectures (e.g., the MIPS, SPARC, and Alpha), primitive datatypes must be naturally aligned. For instance, a 64-bit value (e.g., a `double`) should be placed on a double-word boundary. Of course, the compiler can arrange to insert padding to ensure this property, if we assume that `malloc` places objects on maximally aligned boundaries. Still, we might need to add a well-formedness judgment to memory types to ensure that they respect alignment constraints.

The approach to typing the stack is powerful enough to accomodate standard procedure calls for a C-like language, but cannot handle nested procedures, even if they are "downward-only" as in Pascal. To support this, we would, in general, need some form of static pointers back into the stack (or display.) The STAL type system supports a limited form of such pointers which also provides the mechanisms needed to implement exceptions (Morrisett, Crary, Glew, and Walker, 2002). These extensions were used in the TALx86 implementation. An alternative approach, based on intersection types, is suggested by Crary's TALT (Crary, 2003) which has the advantage that it better supports stack-allocated objects. Yet another approach is to integrate support for *regions* into the type system in the style of Cyclone (Grossman, Morrisett, Jim, Hicks, Wang, and Cheney, 2002b) or one of the other region-based systems described in Chapter 5.

The original TAL used a different mechanism, based on initialization flags and subtyping, to support shared-object initialization. More recently, the work of Petersen, Harper, Crary, and Pfenning, 2003b provides an approach to initialization based on a "fuse" calculus. The approach based on initialization flags has the advantage that uninitialized objects are first class, which is useful in some contexts, such as "tail-allocation" (Minamide, 1998). Neither the approach we suggest here based on unique pointers, nor the fuse calculus supports this. Furthermore, neither unique pointers nor the fuse calculus supports the initialization of circular data structures, which is important when building recursive closures (see the next section.) These concerns motivated the design of alias types (Smith, Walker, and Morrisett, 2000b) which handles all of these concerns and more, and which were implemented in TALx86. Recently, Ahmed and Walker, 2003 have suggested yet another approach based on an embedding of the logic of bunched implications within a type system.

It is possible to add a `free` instruction to TAL-1 which takes a unique

pointer and returns it to the runtime for re-use. In some sense, we can think of `free` as the ultimate type-changing operation, for it is simply a way to recycle memory so that it can later be used to hold values of a different type. Unfortunately, it is not so easy to provide a `free` operation for shared pointers.

8.6.2 EXERCISE [RECOMMENDED, ★↛]: Given an example program that could "go wrong" if we allowed `free` to operate on shared pointers. □

As noted earlier, the type system is closed under extensions to the heap, but not necessarily a shrinking heap. It can be shown that if a location is not reachable from the registers (or from reachable code) then a heap value can be safely thrown away. But discovering that this property is true requires more than a single machine instruction. Indeed, it requires a run-time examination of the pointer-graph to determine the unreachable objects.

Therefore, with the introduction of shared data pointers, we are essentially buying into the need for a garbage collector to effectively recycle memory. Of course, to support accurate garbage collection requires that we provide enough information that the collector can determine pointers from other data values at run-time. Furthermore, the collector requires knowledge of the size of heap objects. Finally, many collectors require a number of subtle properties to hold (e.g., no pointers to the interior of heap objects) before they can be invoked. Capturing all of these constraints in a useful type system is still somewhat of a challenge.

TALx86 and the proposed TALT use a *conservative* collector to recycle memory. Conservative collectors do not require precise information about which objects are pointers. However, they tend to have leaks, since they sometimes think an integer is a pointer to an unreachable object. Like other collectors, a number of invariants must hold in order for the collection to be sound. The TALT system formalizes these constraints as part of its type system.

Another possibility is to integrate the Capability types which provide a general support for regions at the TAL level (Walker, Crary, and Morrisett, 2000a). With this type system, it is possible to code up a copying collector within the language as suggested by Wang and Appel, 2001. However, doing an efficient copying collector requires a bit more technical machinery. Some of these issues are resolved by Monnier, Saha, and Shao, 2001.

Finally, we remark that at this point, of course, a paper and pencil proof of the soundness of a system that incorporates these extensions becomes quite large (and tedious) and we are therefore likely to make mistakes. To avoid such pitfalls, we would be wise to encode the abstract machine and type system in some formal system where the proof can be verified. For example, Crary encodes his TALT abstract machine and typing rules using

the LF framework (Crary, 2003) whereas Hamid et al. have done this using Coq (Hamid, Shao, Trifonov, Monnier, and Ni, 2002). Another approach championed by Appel and Felty, 2000 is called Foundational Proof Carrying Code, whereby the types are semantically constructed using higher-order logic in such a way that they are by definition sound with respect to the (concrete) machine's semantics.

8.7 Scaling to Other Language Features

TAL-1 only supports simple tuple or record-like data structures. Thus, it is insufficient for compiling real-world high-level languages which provide data abstraction mechanisms such as closures, algebraic datatypes, objects, and/or arrays.

8.7.1 Simple Objects and Closures

Support for closures and simple forms of objects is readily accomodated by adding *existential* abstraction for both operand and allocated types:

$$\tau \quad ::= \quad \ldots \mid \exists\alpha.\tau \mid \exists\rho.\tau$$

The rules for introducing and eliminating existentials on operands are extremely simple:

$$\frac{\Psi;\Gamma \vdash v : \tau[\tau'/\alpha]}{\Psi;\Gamma \vdash v : \exists\alpha.\tau} \qquad \text{(S-PACK)}$$

$$\frac{\Psi;\Gamma \vdash v : \exists\alpha.\tau \qquad \alpha \notin FTV(\Gamma)}{\Psi;\Gamma \vdash v : \tau} \qquad \text{(S-UNPACK)}$$

(There are two similar rules for existentials that abstract allocated types.) The first rule allows us to abstract a common type for some components of a data structure. For instance, if r has type

```
ptr(code{r1:int,r2:int,...},int)
```

then we can use the S-PACK rule to treat the value as if it has type

```
∃α.ptr(code{r1:α,r2:int,...},α).
```

Now, such a value can only be used by eliminating the existential using the S-UNPACK rule. However, we are required to continue treating α as abstract and distinct from any other type that may be in our context.

As suggested by Pierce and Turner, 1993, we can use existentials to encode very simple forms of objects. For example, consider an object interface that looks like this, written in a Java-style:

```
interface Point {
  int getX();
  int getY();
}
```

and consider two classes that implement this interface:

```
class C1 implements Point {
  int x = 0, y = 0;
  int getX() { return x; }
  int getY() { return y; }
}

class C2 implements Point {
  int x = 0, y = 0 , n = 0;
  int getX() { n++; return x; };
  int getY() { n++; return y; }
}
```

We can think of objects as pairs of a method table and an instance variable frame. The methods take the instance variable frame as an implicit "self" argument. For instance, the `C1` class would have an instance frame that holds two integers, whereas the `C2` class would have an instance frame that holds three integers. Thus, at an intermediate language level, `C1`'s get operations would have type:

$$\texttt{ptr(int,int)} \rightarrow \texttt{int}$$

while `C2`'s operations would have type:

$$\texttt{ptr(int,int,int)} \rightarrow \texttt{int}$$

When we build a `C1` object or a `C2` object, we need to hide the type of the instance frame so that only the methods can gain access to and manipulate the values of the instance variables. We also need to hide the type of the instance frames so that we can give the objects a common type. We can achieve this by using an existential to abstract the type of the instance frame and pairing the methods with the instance frame. At an intermediate level, the type of a `Point` object would thus be something like:

$$\exists\alpha.\texttt{ptr(ptr}(\alpha \rightarrow \texttt{int}, \alpha \rightarrow \texttt{int}), \alpha)$$

Note that any for any value with this type, we cannot directly access the instance variables because their type is abstract (i.e., α). However, once such an object is unpacked, we can project out a method and the instance frame,

and pass the frame to the method because the methods expect an α value as an argument.

We remark that closures are simple forms of objects where the instance frame holds the environment, and there is a single method for applying the closure to its arguments. Thus, with the simple addition of existential types, we have the ability to encode the primary features of modern languages, notably closures and objects. Indeed, the original TAL paper (Morrisett, Walker, Crary, and Glew, 1999) showed how a polymorphic, core functional language could be mapped to a version of typed assembly with support for existentials. This translation was based on previous work of Minamide, Morrisett, and Harper, 1996.

Of course, in a more realistic implementation, we might represent objects without the level of indirection on instance variables, and instead of passing only the instance variabls to methods, we usually pass the whole object to a method. With the addition of recursive types ($\mu\alpha.\tau$) and the usual isomorphism ($\mu\alpha.\tau = \tau[\mu\alpha.\tau/\alpha]$), this becomes possible:

$$\exists\rho.\mu\alpha.\texttt{ptr}(\texttt{ptr}(\alpha \to \texttt{int}, \alpha \to \texttt{int}), \rho)$$

Notice that in the above encoding, we have abstracted an allocated type (ρ) which is used to describe the rest of the object after the method pointer. Furthermore, the methods in the method table expect to take values of type α which is isomorphic to the type of the (unpacked) object. That is, the methods take in the whole object (including the method table) instead of just the instance variables.

This last encoding of objects is closely based on ideas of Bruce, 1995. There are many other potential encodings (see Bruce, Cardelli, and Pierce, 1997 for a nice overview.) In short, it is possible to draw upon the wealth of literature on object and closure encodings to find a small set of re-usable type constructors, such as F-bounded existentials, to provide your typed assembly langauge with enough power to support compilation of modern object-oriented or functional languages, without baking in a particular object model. Again, this contrasts with the JVM and CLR which fix on a single object model, and therefore provide poor support for encoding langauges outside that model.

One problem with these object encodings is that they do not readily support a form of "down-casting" where we perform a run-time test to determine whether an object implements a given interface. Such operations are common in languages such as Java, where the lack of parametric polymorphism and the erroneous addition of covariant arrays requires dynamic type tests. In general, dynamic type tests can be accomplished by using some form of representation types (Crary, Weirich, and Morrisett, 1998), but these encodings are relatively heavyweight and do not support the actual represen-

tations used by implementations. Glew, 1999 suggested and implemented with TALx86, a form of representation types that better supports practical implementations.

8.7.2 Arrays, Arithmetic, and Dependent Types

In TAL-1, we are restricted to using *constant* offsets to access the data componenents of an object. Similarly, we can only allocate objects whose size is known at compile time. Thus, we cannot directly encode arrays.

The simplest way to add arrays is to revert to high-level, CISC-like instructions. In particular, we could imagine adding a primitive `newarray` r_d, r_i, r_s which would allocate an array with r_i elements, and initialize all of the components with the value in r_s, returning a pointer to the array in register r_d. To read components out of an array, we might have an operation `ldarr` r_d, r_s, r_i which would load the r_i^{th} component of array r_s, placing the result in r_d. Dually, the operation `starr` r_s, r_d, r_i would store the value in r_s into the r_i^{th} component of array r_d.

To ensure type safety for the `ldarr` and `starr` operations, we would need to check that the element offset r_i did not exceed the number of elements in the array and jump to an exception handler if this constraint is not met. In turn, this would demand that we (implicitly) maintain the size of the array somewhere. For instance, we could represent an array with n elements as a tuple of $n + 1$ components with the size in the first slot.

8.7.1 EXERCISE [RECOMMENDED, ★★→]: Extend the TAL-1 abstract machine with rewriting rules for arrays as described above and provide typing rules for the new instructions. □

The advantage of the approach sketched above is that it leaves the type systm simple. However, it has a number of drawbacks: First, there is no way to eliminate the check that an offset is in bounds, even if we know and can prove that this is the case. Second, this approach requires that we maintain the size of the array at runtime, even though the size might be statically apparent. Third, the real machine-level operations that make up the primitive subscript and update operations would not be subject to low-level optimizations (e.g., instruction scheduling, strength reduction, and induction variable elimination.)

An alternative approach based on *dependent types* was suggested by Xi and Harper, 2001 and implemented (to some degree) in TALx86. The key idea behind the approach is to first add a form of *compile-time* expressions to the type system:

$$e ::= n \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid i \mid \cdots$$

A compile-time expression is made up of constants (n), arithmetic operations, and compile-time integer variables (i). We then allow type constructors to depend upon (i.e., be indexed by) a compile time expression. For instance, the type arr(τ,30 + 12) would classify those arrays that have 42 components, each of which is a value of type τ. Similarly, the type int(36) would classify those integer values that are equal to 36—in other words, int(36) is a singleton type.

To support integers whose value is unknown, or arrays whose number of elements are unknown, we can use a suitably quantified compile-time variable. For instance, the type ∃i.int(i) would classify any integer value, and the type ∃i.arr(τ,i*2) would classify arrays of τ values with an even number of components. More importantly, the type

$$\forall i_1, i_2.\texttt{code}\{\texttt{r1:int}(i_1),\texttt{r2:arr(T,}i_1),\texttt{r3:int}(i_2)\}$$

would classify code that expects r2 to hold an array with i_1 elements, r1 to hold an integer equal to i_1, and r3 to hold some other integer equal to i_2. Thus, we can use this limited form of dependent types to track an important relation between two values—that one register holds the number of elements in the array pointed to by another register.

To support the elimination of array bounds checks, we need to go beyond equality relations and track refinements of values as we perform tests. For instance, in a context with the code type above, if we wanted to use r3 to index into array r2, it should be sufficient to check that the value in r3 is greater than or equal to 0, and less than r1[3] :

```
sub:  ∀i1,i2.code{r1:int(i1),r2:arr(T,i1),r3:int(i2), ...}
  blt r3,L          // if r3 < 0 goto L
  sub rt,r3,r1      // if r3 >= r1 goto L
  bge rt,L          //
  ldarr rd,r2,r3    // rd := r2[r3]
```

In other words, the ldarr operation above should type-check since we are in a context where r2 is an array of size i_1, r3 is an integer equal to i_2, and the predicate $(i_2 \geq 0) \wedge (i_2 < i_1)$ is true. To support this validation, DTAL checked instructions under a typing context of the form (Γ;P) where P was a predicate that was assumed to be true. Tests, such as the blt r3, ERROR instruction, would typically add conjuncts to the context's predicate.

For example, type-checking for the fragment above would proceed as follows:

```
1. sub:  ∀i1,i2.(...; true)
```

3. In practice, this can be determined using a single unsigned comparison.

```
2.   blt r3,L      // ({...}; true ^ (i2 >= 0))
3.   sub rt,r3,r1 // ({...,rt:int(i2-i1)};(i2 >= 0))
4.   bge rt,L      // ({...,rt:int(i2-i1)};(i2 >= 0)^(i2-i1 < 0))
5.   ldarr rd,r2,r3
```

After checking line 2, the context's predicate (`true`) has been refined by adding the conjunct i_2 `>= 0` since `r3` has type `int(`i_2`)` and the test can only fall through when `r3` is greater than or equal to zero. At line 3, `rt` is given the type `int(`i_2-i_1`)` since `r1` has type `int(`i_1`)` and `r3` has type `int(`i_2`)` and the operation places `r3 - r1` into `rt`. Then, at line 4, the test adds the conjunct i_1-i_1 `< 0` to the fall-through continuation's context. Finally, at line 5, we are able to satisfy the pre-condition of the `ldarr` construct since we are in a context where the index lies between 0 and the size of the array.

In DTAL, the predicates used for refinements were restricted to linear inequalities so as to keep type-checking decidable. But in general, we could add support for arbitrary predicates, and simply require that the code produce provide an explicit proof that the current (inferred) predicate implied the necessary pre-conditions. In other words, we can fall back to a very general proof-carrying code framework.

However, as the designer of the type system, we would still be responsible for providing the code producer a set of sound (and relatively complete) inference rules for proving these relations. Though this is possible, it is nowhere as easy as the simple proofs that we have presented here.

8.8 Conclusions

Type systems for low-level code, including compiler intermediate languages and target languages, are an exciting area of research. In part, this is because the "human constraint" is lifted since the typing annotations are produced and consumed by machines instead of humans. That is, we do not have to worry about the type system being too complicated for the average programmer, or that it requires too many typing annotations. These concerns often dominate the design of type systems for high-level languages. Of course, it is still important to keep the design as simple and orthogonal as possible so that we can construct proofs of soundness and have confidence in the implementation. Ideally, proofs should be carried out in a machine-checked environment.

Low-level languages also present new challenges to type system designers. For instance, the issues of initialization and memory recycling are of little concern in high-level languages, since these details are meant to be handled by the compiler and run-time system. Yet, the nitty-gritty details of the run-

time system are crucial for the proper functioning of the system.

PART V

Programming in the Large

9 *Design Issues in Advanced Module Systems*

By Robert Harper and Benjamin C. Pierce

A programming language intended for large-scale software development must provide some means of breaking large programs into parts of manageable size, commonly known as modules. The division into modules is chosen to reflect natural divisions of labor within a program, minimizing redundancy and maximizing opportunities for re-use.

The literature on modularity is extensive, covering both *methodology*—how best to decompose programs into modules to satisfy a variety of desirable engineering characteristics—and *mechanisms* used to support modular programming. In this chapter, we focus on the latter, laying out a set of core requirements and design issues and developing linguistic mechanisms for addressing them. The heart of our story is the "advanced module systems" found in present-day dialects of ML, but the discussion touches on modularity features from a range of other popular languages such as C, Modula, and Java.

The presentation emphasizes type systems for modularity grounded in the framework of *Types and Programming Languages*. To keep the discussion focused on basic concepts and issues and avoid type-theoretic technicalities, our presentation is informal. However, the material will be easier to follow for readers with some familiarity with basic concepts of subtyping (TAPL Chapter 15), universal polymorphism (TAPL Chapter 23), existential polymorphism and abstract types (TAPL Chapter 24), and type operators (TAPL Chapter 29). Some more advanced typing features will be mentioned in passing, but prior acquaintance with these features will not be assumed; these include recursive types (TAPL Chapters 20 and 21), bounded quantification (TAPL Chapters 26 and 28), dependent types (Chapter 4 of this volume) and singletons (Chapter 10).

The chapter begins in §9.1 by developing a basic suite of modularity mech-

anisms, such as can be found in almost every language used for large-scale programming—modules and interfaces, namespace management, separate compilation, inter-module type checking, and principal interfaces. §9.4 introduces the central concept of the *phase distinction* and the distinction between *first-* and *second-class* module systems. §9.5 discusses *abstract data types*. Abstract types arise by *sealing* a module with an interface that selectively suppresses the definitions of its type components. Data abstraction raises a number of important technical issues, including *representation independence* and the *avoidance problem*. §9.6 extends the module language with nested hierarchies of modules. §9.7 discusses two alternative mechanisms for representing *families of interfaces*—explicitly *parameterized interfaces* and the less familiar but more flexible idea of *fibered interfaces*, which allow any submodule in an interface to be considered *a posteriori* as the "index" of an interface family. §9.8 extends this discussion to *families of modules* defined by *functors* and raises the issue of *coherence*. We compare two approaches to the coherence problem—*sharing by construction*, which is based on parameterized interfaces, and *sharing by specification*, based on fibered interfaces—and explain why the latter scales well while the former does not. We then discuss the pragmatic motivations for module families in more depth, exploring several classes of situations in which functors arise naturally. The section closes with a discussion of *generative* and *applicative* functors. §9.9 briefly describes three more advanced topics in module system design: first-class modules, in which modules can be treated as ordinary values, higher-order modules, in which functors are treated on the same footing as other modules, and recursive modules, which permit self-reference. §9.10 relates the modularity concepts of this chapter to the mechanisms provided by several well-known languages. §9.11 closes the chapter with historical remarks and additional suggestions for further reading.

9.1 Basic Modularity

Informally, a *module* is a collection of components that may include procedure or function definitions, variable declarations, type definitions, and initialization code—the specifics vary from one language to another. A *program* consists of a collection of bindings of module names to modules, with a specified root module as the main entry point of the system. As a rule, it is desirable that the modules that make up a program be developed and maintained independently of one another. This permits sharing of modules among programs, reduces the time required to rebuild a large program after a small change, and allows programmers to check for type errors during

early stages of development.

One module in a program may refer to another by using the latter's name as an *external reference*. The occurrences of external references between modules determine a dependency ordering in which the referring module depends on the module to which it refers. The job of a linker is to compose a complete program by resolving external references. The linker creates module bindings for each of the external references in the partial program under construction until no unresolved references remain.

To support separate compilation the dependency of one module on another is mediated by an *interface* that describes the externally visible components of a module. To support separate compilation an interface must be sufficiently expressive as to enable clients of a module to be compiled without reference to its implementation. This information typically includes type declarations for procedures and variables and the definitions of type identifiers.

In practice most languages support modularity through a mixture of linguistic and extra-linguistic mechanisms. For example, modules are often organized as files, and module naming conventions are often tied to file system naming conventions. To avoid such complications, we will concentrate our attention on a module language that emphasizes the central concepts, relegating its realization in specific languages and development environments to informal discussions in §9.10.

Syntax

We will employ a notation for modules and interfaces that is loosely based on ML. We consider the *module language* to be constructed in terms of some underlying *core language*, whose details we do not care too much about. The principal point of contact between the module and core language consists of references to components of modules from within core language expressions. To account for the type definitions found in interfaces, the core language's notion of type equivalence must be enriched to expand such definitions as necessary to ensure that type components are synonymous with their definitions.

The grammar given in Figure 9-1 defines the syntax of a basic module system that we will enrich as further ideas are developed. We use the metavariables `x` and `y` to range over term identifiers, `s`, `t`, and `u` to range over terms, `X` and `Y` to range over type identifiers, `S`, `T`, and `U` to range over types, `m` and `n` to range over module identifiers, `M` and `N` to range over module expressions,

P	::=	*programs:*	CD	::=	*component declarations:*
	$B_1 \ldots B_n$	*binding sequence*		type X [> X] = T	*type declaration*
B	::=	*bindings:*		val x [> x] : T	*value declaration*
	module m [: I] = M	*module binding*	T	::= ...	*types:*
	interface J = I	*interface binding*		m.X	*type selection*
M	::=	*modules:*	t	::= ...	*terms:*
	m	*module variable*		m.x	*value selection*
	mod { $CB_1, \ldots, CB_n$ }	*basic module*	Γ	::=	*typing contexts:*
I	::=	*interfaces:*		∅	*empty*
	J	*interface identifier*		Γ, D	*declaration*
	int { $CD_1, \ldots, CD_n$ }	*basic interface*	D	::=	*declarations:*
CB	::=	*component bindings:*		m : I	*module declaration*
	type X [> X] = T	*type binding*			
	val x [> x] = t	*value binding*			

Figure 9-1: Basic Module Syntax

and I and J to range over interfaces and interface identifiers.[1]

A *program* consists of a sequence of *bindings* each of which has one of two forms, a *module binding* or an *interface binding*. A module binding binds a module variable to a module expression, perhaps with a specified interface. An interface binding binds an interface identifier to an interface. The scope of a binding in a program is the remainder of the program following that binding.

Interface bindings are used simply to give names to interfaces: a bound interface identifier is considered to be synonymous with its binding. Consequently, we frequently elide the distinction between an interface identifier and the interface to which it is bound. Thus, if I is an interface identifier, we refer to "the interface I" when we mean "the interface bound to I."

A *module expression* is either a *basic module* or a *module variable*. A *basic module* consists of a sequence of component bindings, which are either *type bindings* or *value bindings*. A type binding is a labeled binding of a type vari-

1. We are departing slightly from TAPL's metavariable conventions here. In TAPL, lowercase identifiers were used consistently for term-level expressions and variables, and uppercase identifiers for type-level expressions and variables. Here, we are using M and N for module-level expressions and m and n for module-level variables. Also, we will use I and J to denote both interfaces and interface identifiers. No confusion results from this overlap, since in any case we regard an interface variable as just an abbreviation for its definition.

able to a type expression. A value binding binds a run-time entity to a value identifier. These entities may include procedures, classes, objects, mutable reference cells, and other structures from the core language.

Each component binding has both a *label*, which is underlined, and a *variable*, which is not. The variable governs references to that binding *within* the module; the label governs reference from *outside* of the module. For this reason the label is sometimes called the *external name* of the component, and the variable its *internal name*. The use of a label from outside of a module to designate one of its components is called an *external reference*; the use of a variable from inside the module to designate a preceding binding is called an *external reference*. An external reference to a module bound to the variable `m` has the form `m.X`, where `X` is the label of a type component of `m`, or `m.x`, where `x` is the label of a value component.

Internal names are bound variables whose scope is the rest of the module in which they occur. As usual, internal names may be renamed consistently within their scope without affecting the meaning of the module. In contrast the external name of a component cannot be renamed without affecting the rest of the program in which it occurs. The distinction between external and internal names is necessary for both conceptual and technical reasons (detailed in §9.6). However, in most cases it is not important to emphasize the distinction, so we take the liberty of providing a single name for each component binding with the understanding that it plays a dual role as both the external and internal name of that component.

A *basic interface* consists of a sequence of *component declarations*, either a type declaration or a value declaration. A *type declaration* is a labeled type binding, with the same syntactic form as a type binding in a module. (For the moment, we ignore the possibility of indicating just the presence of a type binding while suppressing its definition. This is the basis for data abstraction, which we introduce in §9.5.) A *value declaration* defines the type of an identifier, but does not give its actual binding. As with bindings, we usually assign a single name to each declaration with the understanding that it serves as both the internal and external name of that component. (Strictly speaking, there is no fundamental need for an internal name for a value binding, but we maintain the distinction for the sake of uniformity.)

Examples

Here is an example of a module binding:

```
module m = mod {
  type X = Nat
  val x = 5
```

```
}
```

The module bound to `m` consists of one type binding and one value binding. These components of `m` are designated by `m.X` and `m.x`, which are, respectively, core language type and value expressions.

Here is a more interesting module binding:

```
module n = mod {
  type X = λW:*. W × W
  val f = λy:X(Nat). plus y.1 y.2
}
```

First, the right-hand side of its type binding has kind $*\to*$ (i.e., this module is exporting a type *operator*). Second, the right-hand side of its term binding uses the previously bound type operator `X`. This illustrates the impact of the module language on core-language type checking: in order to check that the core-language expression `λy:X(Nat). plus y.1 y.2` is well typed, we need to use the module-level information that `X` is definitionally equal to `λW:*. W × W`.

The interface binding

```
interface I = int {
  type X = Nat
  val x : X
}
```

introduces the interface identifier `I`, which is bound to the given interface. This interface describes the module bound to `m` above, in a sense to be made precise shortly. Similarly, the interface binding

```
interface J = int {
  type X = λW:*. W × W
  val f : X(Nat) → Nat
}
```

binds `J` to an interface corresponding to the module `n`.

9.2 Type Checking and Evaluation of Modules

To keep the discussion from bogging down in formalities, we describe type checking and evaluation throughout the chapter in English prose rather than giving precise, formal definitions. For readers interested in a more technical treatment, §9.11 offers a number of pointers into the literature.

Type Checking

Interfaces are used to describe modules. If the module `M` is accurately described by the interface `I`, then we say that `M` *implements* `I`. This relation may be defined in one of two ways. The direct method simply defines a correspondence between a module and any interface that it may implement. An indirect method, which is applicable only in certain cases, is to associate with each module `M` its *principal interface*, the "most precise" (least in the subtyping ordering on interfaces) interface implemented by `M`. We will start by defining the implementation relation directly and later discuss conditions under which it may be reduced to subtyping.

We say that a basic module `M` *implements* a basic interface `I` if `M` contains at least the type and value components specified by `I`, up to type equivalence. More precisely, each type component declared in `I` must be bound in `M` with the same kind and equivalent definition. Moreover, each value component declared in `I` must be matched by a value binding in `M` whose type is equivalent to that specified in `I`. The type equivalence relation here is inherited from the core language, enriched to include the expansion of definitions introduced by type bindings in modules and interfaces.

When a module binding specifies an interface, the type checker ensures that its right-hand side implements this interface. For example, the following bindings are well-formed because the module bound to `m` implements the interface `I`.

```
interface I = int {
  type T = Int
  type U = Int × Int
  val x : U
}
module m : I = mod {
  type T = Int
  type U = T × T
  val x : T × T = (3,4)
}
```

Since the binding for `m` must implement the interface `I`, the interface `I` determines the type and value components of `m`. Here, since `I` provides definitions for the types `T` and `U` and declares the value `x`, it follows that `m.T` and `m.U` are valid type expressions (equal, respectively, to `Int` and `Int × Int`), and `m.x` is a valid value expression (of type `m.T × m.T`, which is equivalent to `m.U`).

To account for external references during type checking, each module variable is assigned an interface by a *typing context*. This assignment amounts to

an assumption that the binding of that variable will be to a module that implements that interface. The manner in which interface assignments to module variables are stated and checked is discussed in §9.3.

Interface Matching

Since interfaces are descriptions of modules, it is natural to consider a subtyping relation between interfaces, called *interface matching* and written `I<:J`. As usual, an interface `I` may be considered a subtype of an interface `J` only if any module implementing `I` also implements `J`; this is just a statement of the subsumption principle for type systems with subtyping. Said differently, if `I` is a subtype of `J`, then `I` expresses stronger requirements on an implementation of it than does `J`. When `I<:J` we say that `I` *matches* the interface `J`, and that `I` is the *candidate* and `J` the *target* of the matching relationship.

There is room for variation in how the interface matching relation is defined, subject only to the requirement that it validate subsumption. There are two well-known styles of interface matching, which we call *structural* and *nominal*. Structural matching is based entirely on the requirements imposed by the interface, without requiring any explicit declaration or imposing any constraint on how the interface is defined. Nominal matching is based on the explicit declaration of subtyping relationships among interfaces. Such declarations are often tied to a naming mechanism for modules and interfaces, which gives rise to the terminology. (This distinction exactly mirrors the distinction between structural and nominal subtype relations discussed in TAPL §19.3.)

Structural matching affords greater flexibility, since it does not require any forethought by the programmmer to specify that one interface subsumes another. Structural matching does not preclude "accidental" matches that may not have been intended by the programmer; this is at once a strength and a weakness. Nominal matching sacrifices flexibility for simplicity by requiring that the programmer explicitly state any matching relations that may hold among interfaces, as long as this is consistent with subsumption. Nominal matching precludes accidental implementation relationships, but requires that any intended ones be explicitly stated.

The definition of structural matching is guided by purely semantic considerations: it is the largest pre-order on interfaces that validates the subsumption principle. That is, `I<:J` iff every module implementing `I` also implements `J`. This is ensured by the following requirements:

1. Every type declaration in `J` must be matched by a corresponding type declaration in `I`. Moreover, their definitions must be equivalent, taking into account the preceding type declarations in `I`.

2. Every value declaration in `J` must be matched by a corresponding value declaration in `I`. Moreover, the type in `I` must be a subtype of that in `J`, taking account of the preceding type declarations in `I`.

These conditions do not impose any ordering requirements, and permit the sub-interface to have components not present in the super-interface. (In the terminology of record subtyping from TAPL Chapter 15, the subtype relation between interfaces permits width subtyping, depth subtyping, and permutation. But, in contrast to record subtyping, permutation must be limited to respect the scoping of internal names for components. For example, a value specification cannot be permuted to precede a type specification on which it depends.) For example, according to this definition the interface

```
interface I = int {
  type T = Int
  type U = T × T
  type V = Int
}
```

matches the interface

```
interface J = int {
  type T = Int
  type U = Int × T
}
```

9.2.1 EXERCISE [★★★★]: How much of the development in the rest of the chapter can be carried out in a nominal setting? □

Principal Interfaces

The *principal interface* of a module, when it exists, is the most precise interface that the module implements. It specifies exactly the properties of the module that may be relied upon in any use of that module. Put in terms of subtyping, the principal interface for a module is the least interface for it in the subtype ordering. If a module `M` has a principal interface, $\mathtt{I}_\mathtt{M}$, then `M` implements an interface `I` exactly when $\mathtt{I}_\mathtt{M}$ is a subtype of `I`. Checking whether a module implements an interface is thereby reduced to checking a subtyping relation between the target interface and the module's principal interface. This reduction is possible in general only if every module expression has a principal interface. Otherwise there is a module expression `M` and an interface `I` such that `M` implements `I`, yet there is no way to express this fact as a subtyping relationship.

Unfortunately not all module languages—very few, in fact—have principal interfaces. Perhaps the greatest impediment to achieving principality is

the avoidance problem, discussed in §9-3. Another impediment is a simple lack of expressive power in the interface language. For example, if subtyping for interfaces is nominal, then the inferred interface for a module will not, in general, be its smallest interface in the declared subtyping hierarchy. One work-around (exemplified by Standard ML) is to draw a distinction between an interface *per se* and its internal representation in a type checker. Every well-formed interface has an internal representation, but some modules may have an internal representation that is not denotable by an interface of the language itself. This creates an unnatural separation between what a particular type checking algorithm knows about a module and what a programmer may state about it using an interface. (See §9.3 for further discussion of the disadvantages of this approach.) An alternative approach is to require that the programmer specify an interface for every module, which is then deemed to be the least interface for that module (even if it is weaker than necessary). Doing so avoids the need for principal interfaces, but at the expense of verbosity and loss of flexibility (in the case that the specified interface precludes uses of the module that would otherwise be permissible).

Evaluation

Complete programs (those with no free variables) are executed by evaluating each of the module bindings in the order given. Basic modules are evaluated by evaluating each of its component bindings in turn according to the rules of the core language. A *module value* is the basic module expression resulting from evaluating all of its component bindings. We insist on a "by value" binding discipline for module variables; a module variable is bound to the value of its binding. (The reason for this requirement will be explained in §9.5.)

The notion of an *initializer* for a module arises here as a value binding whose right-hand side has a side-effect when evaluated. For example, evaluating the right-hand side of the binding of `f` in

```
module p = mod {
  val f =
    let r = ref 0 in
    λx:Nat. r := plus x !r
}
```

allocates a storage cell and then returns a function whose body uses this cell. This example also illustrates the need to distinguish between module expressions and module values. Each time the expression is evaluated, a new cell will be allocated and a different module value will result.

9.3 Compilation and Linking

The process of evaluating a program may be decomposed into two steps, compilation and execution. For the present purposes, the most important aspect of compilation is type checking, and the most important aspect of execution is linking. We shall not concern ourselves with code generation, which may be thought of as a by-product of type checking, or the execution of compiled code. A key distinction between compilation and execution is that the former may be performed on a module-by-module basis, provided only that we are given the interfaces of the free module variables occurring in a module, whereas the latter is performed on a complete program in which we have at hand the bindings of all of its free module variables. We follow Cardelli (Cardelli, 1997) by modelling linking as a process of binding modules to module variables.

Compilation

To support code re-use and team development, it is important to compile modules independently from one another. To compile a module, it is necessary to have an assignment of an interface to each of its free module variables (external references) provided by the typing context. There are two main methods of obtaining this context, *separate* and *incremental* compilation.[2] The difference is whether the interfaces of free module variables are explicitly given by the programmer (separate compilation) or are inferred by the compiler from the source code of the referenced module (incremental compilation). Both separate and incremental compilation may be supported in the same language. Moreover, both are compatible with *cut-off compilation* (Adams, Tichy, and Weinert, 1994), which reduces the time to re-build a system after changes are made to it. Specifically, if the source code of a module has changed, but its interface has not, then there is no need to recompile modules that depend on it — recompilation may be "cut off" at that point.

In a *separate* compilation system, the programmer states interface assumptions for each of the external references in a module. This is typically achieved by "import" declarations that state such assumptions. The module is compiled relative to these assumptions, independently of whether the implementation of the externally referenced modules is available. This affords maximal flexibility in the order of development of the modules in a program and permits re-use of libraries of previously compiled modules whose source may not be available at all. Separate compilation places a burden on the linker to

2. We caution the reader that this terminology is not standard; these and related phrases are used with a variety of loosely related meanings in the literature.

ensure that the binding of a module variable implements the presumed interface. A subtle point is that two different modules may import the same module, but with a different assumed interface. The linker must ensure that each such assumption is satisfied to ensure safety, or else insist that all imports specify equivalent interfaces. Since most conventional linkers are incapable of verifying typing constraints, it is usually necessary to devise language-specific linkers or to introduce post-linking checks (similar to Java bytecode verification) to ensure type safety.

In an *incremental* compilation system, it is not necessary to specify the interfaces of externally referenced modules. Instead, the compiler consults the implementation of that module to determine its interface, which is used for compiling any module on which it depends. This implies that the implementation of any externally referenced module must be present in order to compile a module that refers to it. This obviously impedes independent development, but avoids the need to ensure that the binding of a module implements the presumed interface, since it does so by explicit construction. Lacking a principal interface, a module system cannot properly support incremental compilation.

Linking

A linker assembles a complete program from a collection of module bindings, called the *linking context*.[3] This is achieved by tracing the external references occurring in the collection of program fragments (starting with a specified root module), and building a sequence of module bindings that is consistent with the occurrences of these references. Whenever an external reference is encountered, its binding is determined by consulting the linking context, and emitted as part of the resulting fully linked program. The external reference is thereby said to be *resolved*. The occurrence of external references constrains the order of the bindings in the fully linked program, but it does not completely determine it. Further constraints on the order of bindings are imposed by initialization code whose side effects constitute an implicit dependency of one module on another.

This motivates the definition of a *dependency relation* among a set of modules, consisting of its *reference dependencies* together with its *initialization dependencies*. Reference dependencies are determined by inspection of the code of a module. If a module `N` contains an external reference `m` to a module `M`, then `N` is said to contain a reference dependency on `M`. Interfaces may also

3. We are talking here about conventional *static linking*. Languages that support *dynamic linking* permit name resolution during execution.

contain reference dependencies on modules, for if an interface `I` contains a reference `m` to a module `M`, then `I` depends on `M` and hence `m` must be bound before `I` can be used. (At this point such dependencies are inessential, because they can only arise in type selections of the form `m.X`, which may be replaced by their definitions. However, once abstract types are introduced in §9.5, such references are not in general eliminable in this way.) Initialization dependencies arise when the evaluation of one module is materially affected by the evaluation of another, even though no reference dependency need exist between them. Initialization dependencies cannot always be determined by inspection; for example, one module may read a file that another writes without either sharing a common reference. Therefore, initialization dependencies must be explicitly specified (by some means not detailed here) to ensure that they are respected by the linker.

Ordinarily, the dependency relation among a collection of modules is required to be *acyclic*, precluding circular dependencies of a module on itself (whether via intermediate modules or not). This is enough to ensure that it is always possible to find a linear ordering of modules consistent with the dependency relation. It is possible to permit circular depenendencies, at the expense of considerable complications in the general case; see §9.9 for further discussion.

The assumption that the linking context uniquely determines the referent of an external reference means that external references are *definite*. Within a given linking context there is exactly one referent for a given reference, thereby ensuring that all modules in a linking context may be identified with their name. Definite references are to be contrasted with the *indefinite* references that arise with parameterized modules and interfaces. Indefinite references raise difficulties related to aliasing, called coherence problems. These issues are discussed further in §9.8.

9.4 Phase Distinction

Most modern programming languages are *statically typed*, meaning that type checking may be performed prior to, and independently of, execution. Statically typed languages maintain a clear separation between the *static (type checking)* and *dynamic (execution)* phases of processing, and are therefore said to respect the *phase distinction*.[4] Languages that do not respect the phase dis-

4. This terminology was introduced by Cardelli (Cardelli, 1988a) in an attempt to relate phases to a universe distinction in type theory. The formulation used here is based on Harper, Mitchell, and Moggi (Harper, Mitchell, and Moggi, 1990a).

tinction are said to be *dependently typed*.[5] Type checking for a dependently typed language requires reasoning about the run-time equivalence of ordinary expressions (sometimes called "symbolic evaluation"). Examples of dependently typed programming languages include Russell (Donahue and Demers, 1985), Pebble (Burstall and Lampson, 1984), and Cayenne (Augustsson, 1998). (See Chapter **??** for further information on dependently typed languages.)

The phase distinction is of particular importance for module systems, because modules are hybrid entities that consist of both static (type) and dynamic (value) components. It is reasonable to ask that a module system preserve respect for the phase distinction—if the core language is statically typed, then so also should be the module system. Confusingly, many modularity constructs closely resemble constructs found in dependently typed languages, suggesting that module systems with these features may violate the phase distinction. For example, a type expression of the form `m.X` appears to be dependent, because the module `m` is part of the type expression `m.X`, and modules generally have both type and value components. However, the type theory of a statically typed module system is carefully arranged to ensure that the phase distinction is preserved.

Respect for the phase distinction is closely intertwined with the distinction between first- and second-class module systems. We say that a particular module expression is a *first-class module expression* if its static part is only determined at run-time; otherwise, we say that it is a *second-class module expression*. A *second-class module system* is one in which all module expressions are second class; a *first-class module system* is one that admits first-class module expressions. We will confine our attention to statically typed core languages, for which there is a useful distinction between first- and second-class module systems.

Two examples will help clarify these definitions. Basic module expressions are second-class since they explicitly exhibit the static parts independently of the dynamic parts.

```
mod {
  type X = Nat
  val f = λx:X. x
  type Y = Bool
}
```

On the other hand, an expression that chooses between two modules based on run-time state is first-class. Consider the following basic module bindings:

5. The natural contrasting phrase is "dynamically typed", but this conflicts with the term's established usage to mean run-time dispatch on tagged data.

```
module m1 = mod {
  type X = Int
  type Y = X→X
  val x = 3
  val f = succ
}
module m2 = mod {
  type X = Bool
  type Y = X→X
  val x = false
  val f = not
}.
```

Let `M` be the (hypothetical) module expression

```
if ... moon is full ... then m1 else m2
```

The binding of the `X` component of `M` is dependent on the outcome of the conditional test, which can only be resolved at execution time. Consequently, the type components of `M` depends on a dynamic condition, rendering `M` first-class. On the other hand, if in such a conditional both branches agree on their common type components, then the conditional remains second-class, since in that case there is no actual dependency of static on dynamic information.

The astute reader will have observed that the conditional module `M` is malformed, for the simple reason that any interface for `M` must reveal the definition of the type `X`, which is manifestly impossible. To assign an interface to a module expression such as `M`, it is necessary to permit hiding of type components in an interface. This is one application of the general concept of data abstraction.

9.5 Abstract Type Components

Up to this point interfaces are *transparent* in that the definitions of their type components are revealed. Transparency is convenient in one sense, but inconvenient in another. If type definitions are transparent, then we may freely divide a program into modules without fear of introducing type checking errors. On the other hand transparency impedes modularity by increasing the interdependencies among modules in a system. This is because the type correctness of one module depends on the type definitions in the other, so that a change in the latter forces reconsideration of the former. In other words transparent interfaces do not support data abstraction. Moreover, the restriction to transparent interfaces rule out non-trivial first-class module expressions such as the conditional, `M`, given above.

To address these concerns it is sufficient to permit an interface to contain *hidden*, or *abstract*, type components. An *abstract type declaration* reveals the existence (and kind) of a type component, without revealing its definition. Abstract type declarations are to be contrasted with *concrete type declarations*, which reveal the definiton. Holding a type component abstract in the interface of a module permits the module to change its definition without affecting the type correctness of any module that may depend on it, a fundamental requirement for modular programming.[6]

The simplest way of using abstract type components is to hold *all* types in an interface abstract. Such interfaces are said to be *opaque*. Often, however, complete opacity is overly restrictive: it is unnatural or inconvenient to hold a type abstract, for example, when it is serving primarily as an abbreviation for another type. A more flexible approach is to permit selective suppression of type definitions according to the needs of the situation: some type components may be abstract, others may be transparent. Such interfaces are said to be *translucent*, with opaque and transparent interfaces being limiting cases in either direction.

While translucency may seem, at first, to be a straightforward enrichment of our notation for interfaces, it is precisely translucency that makes possible a number of significant enrichments of the module language with a minimum of additional machinery. In particular, we will see in §9.7 that families of interfaces come *for free* along with translucency. Interface families are important because they provide the means for stating and controlling the flow of type information within a hierarchy of modules (§9.6 or across instances of a parameterized module (§9.6). Most importantly, translucency provides the foundation for a direct and natural way of ensuring the compatibility of the parameters of a parameterized module. It is remarkable that a single mechanism, translucent interfaces, not only affords flexible type abstraction, but also provides all of the supporting apparatus required for several important extensions to the basic formalism. (As might be expected, this increase in expressiveness goes hand-in-hand with some significant meta-theoretic challenges. At this level, the extension to translucent interfaces is the most significant step in the chapter.)

6. Modules with abstract type declarations are often discussed in connection with existential types—see, e.g., Mitchell and Plotkin (1988c) and TAPL Chapter 24. However, although the similarities are suggestive, the correspondence should be taken with a grain of salt: existentials do not support *dot notation* (i.e., given an existential package `p` with an abstract type component `X`, one cannot just refer to `p.X`; instead, `p` must first be "opened" in some particular lexical scope). For this reason, existentials do not offer a fully satisfactory foundation for module systems. This point is discussed in detail by Cardelli and Leroy (1990a).

CD ::=		*component declarations:*
	type X [> X]	*opaque type*
	type X [> X] = T	*transparent type*

Figure 9-2: Translucent Interface Syntax

Translucent Interfaces

To support translucency we extend the syntax of our language to permit two forms of type declaration, one that reveals the definition, and one that suppresses it, as detailed in Figure 9-2.

For example, the interface

```
interface J = int {
  type X
  type Y = X→Nat
  val c : X
  val f : Y
}
```

specifies the existence of type components named `X` and `Y`, revealing the definition of `Y`, but hiding the definition of `X`.

The subtyping relation for interfaces is generalized to admit "forgetting" of type definitions. An abstract type declaration `type X` in a super-interface may be matched by either an abstract or a concrete type declaration in the subinterface. For example, the interface

```
interface J = int {
  type X = Nat
  type Y = X→Nat
  val c : X
  val f : Y
}
```

is a subinterface of the interface `I` above.

As we noted on page 477, the definitions of type components of an interface are propagated forward when checking whether one interface matches another. So, for example, the interface

```
interface K = int {
  type X = Nat
  type Y = X→Nat
```

```
  val c : Nat
  val f : Nat→X
}
```

matches `J`, and so, perhaps surprisingly, the interface `K` matches the interface `I`.

We say that a module implements a translucent interface if it provides the type components specified in the interface with, where given, bindings equivalent to the specified definitions. During type checking, the definitions of type components of a module are always propagated forward while checking the remainder of the components against the specified interface. For example, the module expression `M` given by

```
mod {
  type X = Nat
  type Y = X→Nat
  val c = 5
  val f = λx:X. succ x
}
```

implements the translucent interface `I` given above.

One consequence of translucency is that interfaces may make sense only within the scope of a module binding. The reason is that interfaces may contain free module variables in the form of a type selection from those variables. When interfaces are transparent, all such type selections are eliminable by simply replacing the selection with its definition as given by the interface of the module variable. However, this is not possible if the referenced interface holds a type component abstract. Consequently, the referring interface makes sense only within the scope of this binding. Consider the following example.

```
interface I = int {
  type X
  val c : X
  val f : X→X
}
module m : I = mod {
  type X = Int
  val c = 0
  val f = succ
}
interface J = int {
  type Y
  val d : m.X
```

```
}
module n : J = mod {
  type Y = m.X
  val d = m.f(m.f(m.c))
}.
```

The interface `J` makes sense only within the scope of the binding of `m`, because the type `m.X` is abstract.

Sealing

By attaching an interface to a module binding we assert that the binding implements the stated interface. For example, the following binding makes such an assertion:

```
module m : I = mod {
  type X = Nat
  type Y = X→Nat
  val c = 5
  val f = λx:X. succ x
}
```

Since the interface `I` (defined in the preceding section) holds the type component `X` abstract, this declaration ensures the definition of `m.X` is suppressed wherever it occurs in the scope of this binding for `m`. On the other hand, since `I` reveals the definition of `Y`, any occurrence of `m.Y` will be equivalent to `m.X→Nat`, as specified in the interface and, therefore, in the module itself.

This example illustrates the use of binding to impose abstraction. The type `m.X` is held abstract within the scope of the variable `m`, which may be as large as the entire program, but may also (as more mechanism is introduced) be limited to a specified portion of the program. This is called a *scoped* abstraction mechanism, because abstraction is tied to the scope of a bound variable.

It is useful to separate abstraction from binding. This is achieved by *sealing* a module `M` with an interface `I`, which is written `M!I`. The grammar of module expressions is extended to permit sealing, and to remove interface assignments from module bindings. Module bindings with ascribed interfaces are now written `module m ! I = M` to emphasize that this is simply shorthand for `module m = M ! I`. Additionally, in anticipation of further developments, we enrich the type and value expression grammar to permit selection from a general module expression. (See Figure 9-3.)

A sealed module `M!I` implements the interface `I` iff the module `M` implements the interface `I`.[7] Though simply stated, the force of this definition is

7. Note the similarity to the term-level *ascription* operator described in TAPL Chapter 11.

M,N ::= ...	*modules:*	T,U ::= ...	*type:*
M ! I		M.X	*type selection*
CB ::= ...	*module components:*	t,u ::= ...	*term:*
module m[>m] = M		M.x	*value selection*

Figure 9-3: Mechanisms for Abstraction

stronger than it may seem. Since our type system for modules includes subsumption, the interface `I` can be weaker than other possible interfaces for `M`. In particular, `I` may suppress the definitions of one or more type components of `M` that are exposed in a stronger interface for it. Since sealing imposes abstraction independently of binding, it is said to be a *scope-free* abstraction mechanism. Sealed modules are evaluated by simply stripping off the seal, and evaluating the underlying module. The idea is to view data abstraction as a static discipline, relevant only during type checking, with no dynamic significance.

Determinacy and Abstraction

Sealing imposes an interface on a module. If sealed module `M!I` implements only interfaces that are matched by `I` (i.e., that are no stronger than `I` in the matching pre-order). In particular, if `I` holds a type component `X` abstract, then this ensures that the definition of `X` is hidden for all possible uses of `M!I`. But in what sense does sealing enforce data abstraction? We wish to ensure that sealing ensures *representation independence* for abstract types. At a minimum this means that if `M` and `N` implement `I`, then we may replace `M!I` by `N!I` in a program without disturbing its type correctness.[8]

Surprisingly, without further restrictions representation independence is not ensured. To see why, it is necessary to reconsider the definition of type equality in the presence of sealing. Suppose that we wish to compare the types `(M!I).X` and `(N!I).X`. Since type equality is an equivalence relation, it must be reflexive. Therefore we have no choice but to consider these two types to be equivalent exactly when `M` and `N` are equivalent. Not only is this undecidale in general, but also it is a clear violation of representation inde-

8. A stronger requirement is that if `M` and `N` are suitably related, then the *behavior* as well as the type correctness of a program is not changed by the replacement. See TAPL §tapl:existentials for further discussion of representation independence for abstract types.

pendence. The only way around this is to preclude formation of such type expressions in the first place by deeming `(M!I).X` ill-formed.

A closely related issue is that type expressions are ordinary regarded as "pure" in the sense that they may be freely replicated or eliminated by substitution for type variables during type checking. But if `M` has computational effects, then replication or elimination of the type expression `M.X` amounts to a replication or elimination of `M`'s effects. In fact, permitting such type expressions without restriction can lead to violations of type safety. The only solution is to limit the use of type selections of the form `M.X` to cases where safety is assured.

The basic module system described in §9.1 requires that all modules be bound to variables, which rules out examples such as these. While making this restriction is sufficient to ensure type safety and representation independence, requiring all modules to be bound to variables before use is both inconvenient and unnecessary. Moreover, doing so would ruin type preservation, which requires that replacement of a variable by its value preserve type correctness. We therefore undertake a systematic analysis of when selection of a type component from a module is permissible.

We will say that a module expression is *determinate* iff it permits selection of its type components. That is, `M.X` is well-formed only if `M` is determinate, and not otherwise. The idea is that formation of `M.X` should be permitted only if it makes sense to refer *at compile time* to "the" type component `X` of `M`. If so, then `M.X` may be freely replicated, eliminated, or compared for equality without fear of violating safety or abstraction. In general, however, the static part of a module expression `M` may not be determined until run-time, so that referring to `M.X` at compile time is senseless. Such a module expression is said to be *indeterminate*.

Which module expressions are determinate, and which are indeterminate? Module values are always determinate. To ensure safety, we must regard any first-class module (i.e., one whose static part is not determined until run-time) as indeterminate. To ensure representation independence, we also regard any sealed module as indeterminate. The first two cases may be thought of as matters of *truth*, whereas the third case is a matter of *knowledge*. Whereas the decision for module values and first-class modules is forced by considerations of safety, sealed modules are *deemed* indeterminate as a design decision.

To help clarify these decisions, let us consider each in turn. First off, a module value is determinate because it gives explicit definitions for all of its type components. Thus, if `M` is a module value, `M.X` may be replaced by the definition of `X` within `M`, which is well-formed provided that `M` is well-formed. Thus, for example, the module expression `M` given by

```
mod {
  type X = Bool
  type Y = X→X
  val x = false
  val f = not
}
```

may be considered to be determinate because the type `M.X` is equivalent to `Bool`, and the type `M.Y` is equivalent to `Bool→Bool`, both of which are well-formed types.

Properly first-class module expressions are inherently indeterminate. For example, recall the conditional module expression, `M`, given in §9.4: **??**

```
if ... moon is full ... then m1 else m2.
```

Since the `X` component of `M` is dependent on the outcome of the run-time test, the identity of "the" type component `X` of `M` is not known during type checking—it could turn out to be `Int` or `Bool`, but we do not know which at compile time.

9.5.1 EXERCISE [★★★]: Develop this example to prove that unrestricted type selection is unsound. More precisely, devise an expression `t` involving unrestricted selection from `M` that incurs a type error. □

Contrast this with the following example, which binds a module variable `m` to `M`:

```
module m = if ... moon is full ... then m1 else m2
```

Although formation of `M.X` does not make sense, it *does* make sense to form `m.X`! This follows from the call-by-value binding discipline for module variables. The variable `m` stands for the *value* of `M`, not for `M` itself. Once `M` has been evaluated, the decision about the identity of its `X` component has been made, and hence we may soundly refer to it. Thus, module variables are determinate.

9.5.2 EXERCISE [★★, RECOMMENDED]: What, specifically, would go wrong if we changed the evaluation of module bindings to call-by-name? □

These decisions are forced for reasons of type safety. Let us turn now to representation independence. As stated earlier, we deem all sealed modules to be indeterminate, regardless of the determinacy of the underlying module. In other words we suspend knowledge of the determinacy or indeterminacy of the underlying implementation of a sealed module in the interest of ensuring representation independence. In effect we "assume the worst" of a sealed

module by treating it as a first-class module, even when it is not. This frees an implementor to choose any (type-correct) implementation for an abstract type, without fear of disturbing the type correctness of any client code. This treatment of sealed modules makes clear that data abstraction is a *protocol* for using a module, rather than a *property* of the module itself.

9.5.3 EXERCISE [★]: Suppose that `M` and `N` are determinate modules implementing an interface `I` with an abstract type component `X`. Find a well-typed term `t` involving `(M!I).X` such that the term `u` obtained by replacing `M!I` by `N!I` is ill-typed, a violation of representation independence. □

Deeming sealed modules indeterminate also sheds light on other issues related to data abstraction. For example, abstract types are often described as "new" or "fresh" in the sense that they are distinct from all other types in a program, regardless of any coincidences of representation. This point of view may be accounted for in our framework by observing that to access the type components of a sealed module, we must first bind it to a module variable. By α-conversion the bound variable may be renamed at will without affecting the meaning of the program. In other words the module variable may be chosen to be distinct from all other module variables in a program. Therefore, a type of the form `m.X`, where `m` is a module variable, is distinct from any other type if `X` is abstract.

9.5.4 EXERCISE [★]: Why would it be bad for two copies of `M!I` to induce interchangeable abstract type components? □

This same observation also accounts for the informal idea that data abstraction ties a type to a particular set of operations that interpret it. Any non-trivial computation with a value of that type must be through these operations. This greatly facilitates maintaining a representation invariant on the data structure, since those and only those, operations may violate it. Moreover, by insisting that sealed modules are indeterminate, we ensure that the operations from two different abstract types are not interchangeable, even if the underlying representation of values of those types are the same.

9.5.5 EXERCISE [★★, RECOMMENDED]: Devise an example of two implementations of an abstract interface that share a common representation type, but differ in the operations used to interpret it. Assuming that these two implementations give rise to the same (but hidden) abstract type, give a program that incurs an error that would otherwise be avoided. □

An important special case arises when the implementation of an abstraction involves private state. In that case two instances of the abstract type

must be kept distinct, even though both the representation type and the code of the associated operations are identical! The following exercise explores one example of what can go wrong.

9.5.6 EXERCISE [★★, RECOMMENDED]: Devise an implementation of a hash table involving state, and show that if two instances of the hash table were to determine equivalent abstract types, then errors can arise that would otherwise be avoidable. □

A moment's thought reveals that a separable module is also determinate. For if `M` implements a fully transparent interface `I`, then every type component `X` of `M` is explicitly defined in `I`, and hence `M.X` may be replaced by its definition in `I`. Is a determinate module also separable? No, but for rather technical reasons. For example, if a module variable `m` is bound to a sealed module `M!I` which holds a type component `X` abstract, then any interface for `m` must also hold `X` abstract. Consequently, `m` is not separable, even though it is determinate. One unfortunate consequence is that tracking type identity across module boundaries is unnecessarily weak. Consider the following situation:

```
interface J =
  int {
    type X
    val x : X
    val f : X→X
  }
module m = (...)!J
module n = m
```

Unfortunately, since the type `X` is abstract in `I`, `m.X` and `n.X` are considered to be different types, even though they are manifestly the same!

This limitation may be easily overcome using a device, called *selfification*, which permits strengthening of the interface of a determinate module to express the self-identity of their components. More precisely, if `M` is a determinate module implementing `I`, then `M` also implements a subinterface `I'` of `I` obtained by replacing every opaque type `X` in `I` by a corresponding transparent declaration that defines `X` to be `M.X`. The result is a fully transparent interface for `M`, rendering it separable. For example, if `M` is a determinate module implementing the interface `J` given by the binding

```
interface J = int {
  type X
  type Y = X→X
},
```

then M also implements the interface J′ given by the binding

```
interface J' = int {
  type X = M.X
  type Y = X→X
}
```

The fully transparent interface for M given by selfification is the *principal* interface of M.

9.5.7 EXERCISE [★]: Why must a module be determinate for selfification to make sense? □

9.5.8 EXERCISE [★★, ↛]: Convince yourself that determinate modules have principal interfaces via selfification. □

9.5.9 EXERCISE [★★, RECOMMENDED]: Show how to use self-ification to ensure that m.X and n.X are equivalent types when n is bound to m and X is abstract in the interface of m. □

The Avoidance Problem

Consider a local module binding construct of the form

```
let module m = M in M'.
```

This expression implements the interface I′ provided that (1) M implements some interface I, and (2) M′ implements I′ under the assumption that m implements I.

At first glance, it would seem reasonable to say that the principal interface for a let expression would simply be the principal interface of its body. But what if the principal interface of the body involves an abstract type component from M? For example, consider the following the module expression, N:

```
let
   module m = M ! I
in
   mod { val z = m.y }
```

where I is the interface

```
int {
  type X
  val y : X
}
```

Clearly, the principal interface of the body of the `let` is `int { val z : m.X }`. But this interface cannot be the type of `N`, because it involves an essential reference to the locally bound module variable `m`. (The analogous problem for the `unpack` form for existential types motivates the scoping restrictions discussed in TAPL §28.7.)

It is tempting to consider `N` to be ill-formed, since it attempts to export the type `m.X` outside of its scope. But this neglects the possibility that `N` has *some* interface that does not involve `m.X`. For example, if the core language includes a subtype relation with a maximal `Top`, then another possible interface for the body of the `let` is `int { val z : Top }`. Indeed, this may even be the principal interface for `N`. In general, the principal interface of a `let` expression of the form `let module m = M in M'` is the least interface (in the subtype ordering) for `M'` that does not involve the bound module variable `m`.

The problem of finding such an interface is called the *avoidance problem*. First reported by Ghelli and Pierce (1998) in the context of System $F_{\leq}$, the avoidance problem is a central design issue for module systems that support data abstraction. Unfortunately, it does not appear to admit a completely satisfactory solution. In some languages (including ML), there exists an interface `I` involving a module variable `m` with more than one minimal super-interface avoiding `m`, none of which is least. In such cases the occurrence of `m` cannot be avoided without losing valuable type information.

9.5.10 EXERCISE [★★★]: Consider the interface `I`

```
int {
  type X = λW:*. m.Z
  type Y = m.Z
}
```

containing a free module variable `m` whose interface has an abstract type component `Z`. Show that `I` has infinitely many super-interfaces that avoid `m` and yet are mutually incomparable in the interface subtype ordering. Assume, for this exercise, that the core language is just F^{ω}, with no subtyping between core-language types. (For [substantial] extra credit, find a similar example where the core language is $F_{\leq}$.) □

What to do? A fallback position is to admit as well formed those `let` expressions for which there is a principal interface avoiding the bound module variable, and to reject all others. The trouble is that there is no simple characterization of which modules admit principal interfaces and which do not. Reliance on a particular algorithm for detecting cases for which a principal interface exists ruins the declarative nature of the type system. An alternative is to require the programmer to specify the interfaces of all `let` expres-

CB ::= ...	*component bindings:*	M ::= ...	*modules:*
`module` m [`>` m] `=` M	*module binding*	M`.`m	*module selection*
CD ::= ...	*component declarations:*		
`module` m [`>` m] `:` I	*module declaration*		

Figure 9-4: Mechanisms for Hierarchy

sions. Rather than solving the problem, this approach simply shifts the burden to the programmer. Another possibility is to prohibit leaving the scope of a module variable whose interface has an abstract type component. This means that all abstract types must be global, rather than local. To soften the blow we may rename locally-declared abstract types to indicate that they are "hidden", relying on a programming convention to avoid using types with such names. Such a convention may be systematically imposed during elaboration of the source language program into internal form. Using this approach, hiding abstract types is handled in much the same manner as type inference, pattern compilation, and overloading resolution (Dreyer, Crary, and Harper, 2003).

9.6 Module Hierarchies

To avoid name clashes it is useful to organize a collection of module bindings into "clusters" of closely related bindings that have more limited dependencies on each other. This may be achieved by permitting module bindings to occur as components of other modules (with the usual distinction between its internal and external names). Correspondingly, we introduce a new form of module expression, the selection of a module component from another module. The additional syntax to support module hierarchies is given in Figure 9-4.

A module that is bound within another is called a *submodule* of the surrounding module. Most of the properties and relations associated with modules are extended recursively to sub-modules. For example, if all of the submodules of a module are determinate, then so is the module itself. Equivalently, if any sub-module is indeterminate (in particular, if it is sealed), then the module itself is indeterminate. Similarly, the implementation relation between modules and interfaces is extended recursively to submodules so that the module

```
module q = mod {
```

```
  module m = mod {
    val x = 5
    val y = 6
  }
  module n = mod {
    val z = 7
  }
}
```

implements the interface

```
interface Q = int {
  module m : int { val x:Nat val y:Nat }
  module n : int { val z:Nat }
}
```

The interface matching relation is extended covariantly to submodules. For example, the interface `Q` above is a subtype of the interface

```
interface Q' = int {
  module m : int { val y:Nat }
}
```

among many others.

Besides simple namespace management, hierarchical modularity is also useful in representing compound abstractions. A familiar example is the dictionary abstraction, which builds on the concept of a linearly ordered type of keys. The layering of dictionaries atop keys is naturally expressed using a module hierarchy.

```
interface Ordered = int {
  type X
  val leq : X × X → Bool
}

interface Dict = int {
  module key : Ordered
  type Dict : *→*
  val new : ∀V. Dict V
  val add : ∀V. Dict V → key.X → V → Dict V
  val member : ∀V. Dict V → key.X → Bool
  val lookup : ∀V. Dict V → key.X → V
}
```

The `Ordered` interface specifies a type equipped with a binary operation that is intended to be a total ordering of that type. The `Dict` interface specifies a sub-module `key` implementing an ordered type.

The types of the operations declared in the interface `Dict` make reference to the type `key.X`, the type of keys. This illustrates the *dependence* of the "rest" of an interface on (the type components of) a preceding sub-module declaration. Strictly speaking, the type selections `key.X` occurring within the interface `Dict` refer to the *internal name* of the sub-module `key`, whereas any selections from a module implementing `Dict` refer to the *external name*, or *label*, of that sub-module. To distinguish these two aspects of the sub-module declaration we may write the `Dict` interface as follows:

```
interface Dict = int {
  module key > k : Ordered
  type Dict : *→*
  val new : ∀V. Dict V
  val add : ∀V. Dict V → k.X → V → Dict V
  val member : ∀V. Dict V → k.X → Bool
  val lookup : ∀V. Dict V → k.X → V
}
```

In most cases it is not necessary to make explicit the distinction between the internal and external name of a sub-module, and we rarely do. However, there are situations in which the distinction is acute, as in the following example. Consider the module expression, `M`,

```
mod {
  type X = Int
  module m =  mod {
    type X = Bool
    val f = λa:X. 3
  }
}.
```

We wish to assign an interface to `M` that specifies `M.m.f` to be a function of type `M.m.X→M.X`. Without distinguishing internal from external names, while holding `M.m.X` and `M.X` abstract, there is no way to write such an interface. The only possible attempt

```
int {
  type X
  module m : int { type X val f : X → X }
},
```

fails because of shadowing of the outer declaration of `X` by the inner one. However, by distinguishing the internal from the external name, we may write the desired interface as follows:

```
int {
```

```
    type X > X'
    module m : int { type X > X" val f : X" → X' }
  }.
```

Since the internal name is a bound variable, it may be renamed at will, thereby avoiding problems of shadowing.

Returning to the `Dict` interface, the declaration of the sub-module `key` indicates that any module implementing `Dict` comes equipped with its own ordered type of keys. At first glance this may seem unnatural, since we do not ordinarily expect a dictionary abstraction to *provide* an ordered type of keys, but rather to *require* one. The distinction is largely a matter of perspective. Even though the `key` sub-module is a component of an implementation of `Dict`, it would ordinarily be obtained "off the shelf" by reference to another module such as the type of integers ordered by magnitude, or the type of strings ordered lexicographically. However, nothing precludes defining the `key` module "in place", for example in the case that there is precisely one dictionary in use in an entire program. Conversely, we would ordinarily expect that the type constructor `Dict` to be constructed as part of the implementation of `Dict`, but this need not be the case. We might, in fact, copy this type from another module, say a generic implementation of balanced binary search trees. Or we may choose to construct a suitable data structure "on the spot". Thus, the components of a module may sometimes play the role of an "argument" to that module, yet at other times play the role of a "result." This is of particular importance when considering families of interfaces and modules.

9.7 Interface Families

To support code re-use it is important to isolate repeated patterns in both modules and interfaces so that we may consolidate what is common to many instances, allowing only the essential differences to vary. This is achieved by introducing *families* of interfaces and modules that isolate the pattern and that may be specialized to recover a specific instance of the pattern. In this section we consider families of interfaces; families of modules are discussed in §9.8.

A good example of the need for interface families is provided by the `Dict` abstraction introduced in the preceding section. An implementation of the `Dict` interface for a an ordered type of keys takes the following form:

```
module dict1 = mod {
  module key = key1
  type Dict = λX::* . ...
```

```
  ...
}
```

Here $\texttt{key}_1$ is some module implementing the interface `Ordered`. The principal interface for `Dict1` specifies the type of keys:

```
interface Dict₁ = int {
  module key : int {
    type Elt = key₁.Elt
    val leq : Elt × Elt → Bool
  }
  type Dict :: *→*
  ...
}
```

We may seal the module $\texttt{dict}_1$ with the interface $\texttt{Dict}_1$ to ensure that the type constructor `Dict` is held abstract. Note that it would *not* make sense to seal $\texttt{dict}_1$ with the interface `Dict`.

9.7.1 EXERCISE [★]: Why would sealing $\texttt{dict}_1$ with `Dict` be senseless? □

Now suppose that we wish to implement a second dictionary whose keys are drawn from the module $\texttt{key}_2$. As matters stand, we have no choice but to replicate the same scenario, replacing $\texttt{key}_1$ by $\texttt{key}_2$ wherever it occurs.

```
interface Dict₂ = int {
  module key : int {
    type Elt = key₂.Elt
    val leq : Elt × Elt → Bool
  }
  type Dict :: *→*
  ...
}
```

Using this interface we could then define $\texttt{dict}_2$ as follows:

```
module dict₂ ! Dict₂ = int {
  module key = key₂
  type Dict :: *→* = ...
  ...
}
```

Doing this makes the code unnecessarily difficult to modify — any change to the interface `Dict` must be replicated for $\texttt{dict}_1$ and $\texttt{dict}_2$ and so forth.

What is needed, of course, is some means of isolating the common pattern as a *family of modules* implementing a corresponding *family of interfaces*, both

indexed by the type of keys. That way we may obtain each dictionary interface and module as an instance of the family at the corresponding ordered type of keys. We turn first to the representation of families of interfaces; we will consider families of modules in the next section.

Representing Families

There are two main representations of families, *parameterization* and *fibration*.[9] Using parameterization we explicitly abstract the type of keys from the `Dict` interface using a form of λ-abstraction.

```
interface DictP = λX :: * .
  int {
    module key : int {
      type Elt = X
      val leq : Elt × Elt → Bool
    }
  type Dict :: *→*
  ...
}
```

Instances are obtained by application, writing

```
interface Dict1 = DictP(key1.Elt)
interface Dict2 = DictP(key2.Elt)
```

to obtain the interfaces $\texttt{Dict}_1$ and $\texttt{Dict}_2$ defined explicitly above.

Using fibration we simply specify the type of keys by "patching" the generic `Dict` interface using a `where` clause, using a form of interface inheritance, as follows:

```
interface Dict1 = Dict where key.Elt = key1.Elt
interface Dict2 = Dict where key.Elt = key2.Elt
```

As with parameterization, the effect of these declarations is the same as the explicit definitions of $\texttt{Dict}_1$ and $\texttt{Dict}_2$ given earlier. Observe that $\texttt{Dict}_1$ and $\texttt{Dict}_2$ are both subtypes of `Dict`, much as interface inheritance gives rise to subtypes in many object-oriented languages.

In both representations the family of interfaces is indexed by a type. While theoretically sufficient, it is pragmatically unfortunate that the indexing type

9. This terminology is borrowed from category theory, which considers two methods for representing families of categories `F` indexed by a category `I`. An *indexed category* is a functor $F : I \to \mathbf{Cat}$ mapping `I` into the "category of categories"—roughly, a function from `I` to categories. A *fibration* is a functor (satisfying some conditions) $p : F \to I$ displaying the index of a family F. Our use of this terminology is analogical, not technically precise.

is separated from its interpretation by operations. For example, since a type can be ordered in several different ways—for example, strings might be ordered lexicographically or by the prefix ordering—it is preferable to maintain the association of a type with its ordering operation. This may be achieved by generalizing type-indexed families to module-indexed families. In the present case this would amount to parameterization or fibration over a module implementing the interface `Ordered`. In parameterized form this would be written

```
interface DictP' = λkey : Ordered .
  int {
    module key = key
    type Dict = ...
    ...
  }
```

with instances

```
interface Dict1 = DictP'(key1)
interface Dict2 = DictP'(key2)
```

In fibered form we would simply write

```
interface Dict1 = Dict where key = key1
interface Dict2 = Dict where key = key2
```

In either case instantiation of an interface family by a module may be viewed as a convenient form of type indexing, since it is only the type components of the instantiating module that affect the result. This is particularly useful in situations where the indexing module contains several type components, possibly nested within sub-modules.

9.7.2 EXERCISE [★★★]: Give a formal definition of the operation `I where m = M`, making explicit any restrictions that must be made for this operation to be sensible. □

Parameterization vs. Fibration

The chief advantage of parameterization over fibering is familiarity. It is natural to consider a family of interfaces indexed over implementations of an interface `I` as a "function" mapping implementations of `I` to interfaces. Representing interface families by parameterization requires a modest enrichment of the syntax to permit λ-abstractions and applications of interfaces, an extension of interface equivalence to account for instantiation by substitution, and an extension of the type system to classify parameterized interfaces

as a kind of function. Fibering, on the other hand, avoids the need for a new form of interface family by exploiting submodule declarations, which are useful for other reasons.

More importantly, the parameterized approach requires the programmer to anticipate the patterns of abstraction and instantiation that may arise in any future use of a given interface. When several (type or module) components are involved, it can be difficult to anticipate which are to be thought of as parameters and which are to be thought of as constructed components of the module. Indeed, the context may dictate one choice in one situation, and another in another. The fibered approach avoids the need to anticipate the future, because it affords a kind of "after the fact" parameterization—any type or module component may be considered to be the "argument" in a given situation without prior arrangement.

9.7.3 EXERCISE [★]: Illustrate this point by defining—in both parameterized and fibered styles—a family of dictionary interfaces indexed by *both* a module of keys and a module providing the type of values in the dictionary. □

Parameterization makes possible another representation of interface families in which the indexing types appear *only* as parameters, and not as components of the resulting interface. Setting aside the objection that doing so pre-judges whether to regard a component as a parameter or result, the advantage of this approach is a superficial simplicity afforded by the absence of type components. However, type components are necessary in any case to support data abstraction (as a type equipped with operations on it), undermining the purported benefit of simplicity. Moreover, requiring all types in an interface to be parameters amounts to requiring that the programmer perform manual separation of a type from its interpretation, a significant practical disadvantage.

Taken in isolation one may argue the advantages and disadvantages of either representation as compared to the other, with neither coming out a clear winner. However, when examined in the larger context of modular programming, a clear advantage for fibration over parameterization emerges. To explain why this is the case, we must first consider families of modules.

9.8 Module Families

Needless to say, the justifications for introducing families of interfaces apply just as well to implementations. Continuing with the example from §9.7, we might well require, in the same program, several different dictionary modules, differing only in the choice of key type. We would then like to abstract

M, F ::= ...		*modules:*	I ::= ...		*interfaces:*
	λ(m:I) N	*functor*		Π (m:I_1) I_2	*functor interface*
	F(M)	*application*			

Figure 9-5: Mechanisms for Functors

the common pattern by forming a *family of modules* indexed by modules satisfying a particular interface.

A natural representation of a family of modules is as a λ-abstraction of a module expression over a module variable of a specified interface. Such an abstraction is called a *parameterized module*, or *functor*.[10] Instances of the family are obtained by functor application.

The syntax required to support functors is given in Figure 9-5. (This grammar permits higher-order functors, but for now we will concentrate on the first-order case, in which only basic modules may be provided as functor arguments. See §9.9 for a discussion the higher-order case.) The metavariables `F` and `G` range over functors.

In §9.7 we noted that it would be useful to define a family of dictionary modules indexed by the type of keys. Using the notation of Figure 9-5, a dictionary functor might be written

```
module dictFun = λ(key:Ordered) mod { ... }
```

where `...` represents some implementation of the dictionary type and operations. The dictionary module $\texttt{dict}_1$ (with interface $\texttt{Dict}_1$, defined on page 498) would then be obtained by applying `dictFun` to the key module $\texttt{key}_1$:

```
module dict₁ = dictFun(key₁)
```

If a functor is a kind of function, then its interface should be something like a function type—e.g., the interface of the `dictFun` functor should be something like this:

```
interface DictFun =
  Ordered →
    int {
      type Dict : *→*
      val new : ∀V. Dict V
```

10. Rod Burstall once remarked that if we do not call the factorial function a "parameterized integer," then we should not call a functor a "parameterized module"!

```
    val add : ∀V. Dict V → key.X → V → Dict V
    val member : ∀V. Dict V → key.X → Bool
    val lookup : ∀V. Dict V → key.X → V
  }
```

However, the arrow notation does not quite give us what we need: it does not express the dependency between the argument and the result of the dictionary functor—i.e., the fact that the module `key` appearing in the result interface is precisely the argument. To capture this dependency, we need an additional form of interface, called a *functor interface*, of the form `Π(m:I)J`. Such an interface binds the module variable `m` in the interface `J`, permitting the dependency of the result interface on the argument to be expressed. The interface `I` is the called the *domain*, and `J` is called the *range*. (The interface `Π(m:I)J` is a form of *dependent function type*; see Chapter 4 for background.)

For example, the type of the dictionary functor given above may be written as follows:

```
interface DictFun =
  Π(key:Ordered)
    int {
      type Dict : *→*
      val new : ∀V. Dict V
      val add : ∀V. Dict V → key.X → V → Dict V
      val member : ∀V. Dict V → key.X → Bool
      val lookup : ∀V. Dict V → key.X → V
    }
```

Instantiating `DictFun` by a module `M` implementing the domain interface `Ordered` yields a module whose type is the instance of the range interface obtained by replacing `key` by `M` throughout. (Note that, since the range interface contains projections from `key`, the argument `M` must be a determinate module.)

9.8.1 EXERCISE [★]: One might guess that a family of modules would have a family of interfaces, but, instead of this, we introduced a new notion of functor interfaces. Why? □

9.8.2 EXERCISE [★]: Note that `DictFun` can be written more concisely in terms of the parameterized interface family `DictP`, as `Π(key:Ordered) DictP(key)`. Can `DictFun` also be expressed in terms of the fibered interface family `Dict`? □

Functor arguments are required to be determinate because the range interface may involve type selections from the domain parameter. Substitution of

the argument for the parameter results in a specialization of the range interface. Rather than use substitution, we may also formulate the typing rule for functor application using subsumption. Just as for ordinary function types, functor interfaces are *contravariant* in the domain and *covariant* in the range. This implies that we may weaken a functor interface by strengthening its domain type. In particular, if `F` is a functor with interface `Π(m:I)J`, and `M` is a determinate module with transparent interface `I′<:I`, then `F` also implements the interface `Π(m:I′)J`. Since `I′` is transparent, any type selection of the form `m.X` in `J` may be replaced by its definition in `I′`, thereby eliminating the dependence of the range interface on the functor argument. This results in an interface of the form `I′→J′`, where `J′<:J`. By covariance `F` implements `I′→J′`, and hence `F(M)` implements `J′`. In effect we've performed the substitution of `M` for `m` in `J` using the types of `F` and `M` alone, rather than by inspecting `M` itself.

9.8.3 EXERCISE [★]: Work out the type checking of the application `dictFun(M)`, where `M` is a determinate implementation of the key interface `Ordered` of your choosing, using the rules just described. □

Coherence

Since families of modules are just functions from modules to modules, one might guess that the language design issues they raise would be just the ones familiar from higher-order functional programming languages. However, a closer look reveals a significant difficulty, called the *coherence problem*, that must be addressed in any practical language with module families. The coherence problem is illustrated by the following example.

Suppose we have defined modules `ab` and `bc`, each providing a function `f` that maps between some input type `In` and some output type `Out`. For `ab`, the input and output types are `A` and `B`; for `bc` they are `B` and `C`.

```
module ab = mod {
  type In = A
  type Out = B
  val f : In → Out =  /* ... some function from A to B */
}

module bc = mod {
  type In = B
  type Out = C
  val f : In → Out = /* ... some function from B to C */
}
```

Since the output type of `ab` is the same as the input type of `bc`, we can write

a third module of the same form that uses the `f` functions of `ab` and `bc` to construct its own `f` mapping from `A` to `C`.

```
module ac = mod {
  type In = A
  type Out = C
  val f = λx:In. bc.f (ab.f x)
}
```

The point to notice here is that the well-typedness of `ac.f` depends on the fact that the result type of `ab.f` is the same as the argument type of `bc.f`.

Now, suppose that we have a *lot* of modules of the same form as `ab` and `bc`, and we want to write many modules like `ac` that "compose" the transformations provided by a pair of existing modules.[11] We would like to write a composition functor that encapsulates, once and for all, the boilerplate involved in building these composite modules. To do this, we first define a generic interface for "transformers"; this will be the type of the inputs and the output of the composition functor.

```
interface Tr =
  int {
    type In
    type Out
    val f : In → Out
  }
```

Note that both `ab` and `bc` implement `Tr`.

A naive first attempt at writing the composition functor itself would be this:

```
module compose =
  λ(m:Tr) λ(n:Tr)
    mod {
      type In = m.In
      type Out = n.Out
      val f = λx:In. n.f (m.f x)
    }
```

11. One real-world domain where this sort of situation arises is in networking protocol toolkits such as FoxNet (Biagioni, Haines, Harper, Lee, Milnes, and Moss, 1994) and Ensemble (van Renesse, Birman, Hayden, Vaysburd, and Karr, 1998): the modules `ab` and `bc` in the example correspond to individual protocol layers or "micro protocols," the functions `f` correspond to the processing performed by each protocol layer, and the input and output types correspond to different packet or message formats. Composite modules like `ac` correspond to protocol stacks such as TCP/IP.

However, this is not well typed: the type expected by `n.f` is `n.In`, while the type returned by `m.f` is `m.Out`, and we have no reason to believe that these are the same. That is, we have failed to express the fact that, for composition to make sense, the argument modules `m` and `n` must be *coherent* in the sense that they *share* this type.

There are two well-known techniques for ensuring coherence, called *sharing by construction* (also *sharing by parameterization* or *Pebble-style sharing*) and *sharing by specification* (or *ML-style sharing*). Sharing by construction was invented by Burstall and Lampson (1984) in their Pebble language and has been explored by many people, famously Jones (1996). Sharing by specification was originated by MacQueen (1984) and is used in the ML module system.

Sharing by construction relies on the technique of fibered interface families introduced in §9.7: the required coherence between the module parameters `m` and `n` is expressed by refining the interface of `n` so that the only modules to which `compose` can legally be applied are those whose `Out` component coincides with the `In` component of `m`.

```
module compose =
  λ(m:Tr) λ(n:Tr where In = m.Out)
    mod {
      type In = m.In
      type Out = n.Out
      val f = λx:In. n.f (m.f x)
    }
```

The type of this functor is

```
Π(m:Tr) Π(n:Tr where In = m.Out)
  Tr where In=m.In and Out=m.Out
```

(writing `and` as a more readable synonym for `where`).

Before we go on, a short digression on notation is in order. The definition of `compose` that we have just given is a little hard to read: since we have defined functors to take just one parameter at a time and used currying to write multiple-argument functors, the inherently symmetric sharing relation between `m.Out` and `n.In` has to be de-symmetrized. We can recover the symmetry in two steps. We begin by *un*-currying `compose`—i.e., we rewrite `compose` into a one-argument functor whose parameter is a module with two sub-modules `m` and `n`:

```
interface TrPair = int {
  module m : Tr module n : Tr where In = m.Out }
```

```
module compose =
  λ(p:TrPair)
    mod {
      type In = p.m.In
      type Out = p.n.Out
      val f = λx:In. p.n.f (p.m.f x)
    }
```

Second, we rewrite the interface `TrPair` in a symmetric way using a new keyword `sharing`:

```
interface TrPair = int {
  module m : Tr
  module n : Tr
  sharing n.In = m.Out
}
```

The interface of `compose` now becomes:

```
Π(p:TrPair) Tr where In=m.In where Out=m.Out
```

Fortunately, `sharing` declarations add no foundational complexity to the language: they are simple syntactic sugar. The `sharing` form just desugars into the primitive `where` form—i.e., the compiler can straightforwardly expand the second definition of `TrPair` into the first. This syntax clarifies the essential intuition that the argument to `main` is not simply a pair of transformer modules, but a *coherent* pair of modules.[12]

The sharing-by-construction style expresses the coherence required by the `compose` functor in a different way: by "factoring out" the shared type as another parameter to `compose`. To achieve this, we first replace the interface `Tr` by an interface family, indexed by the types `In` and `Out`:

```
interface Tr = λTI:*. λTO:*.
                 int {
                   val f : TI → TO
                 }

module ab = mod { val f : A → B = ... }
          : Tr A B

module cb = mod { val f : B → C = ... }
          : Tr B C
```

Similarly, the interface of composable pairs, `TrPair` is parameterized on three types:

12. That is, in category-theoretic terms, not just a product but a pullback.

```
interface TrPair = λTI:*. λTM:*. λTO:*.
                     int {
                       module m : Tr TI TM
                       module n : Tr TM TO
                     }
```

The coherence between `m` and `n` is expressed here by the fact that their interfaces both mention the same (parameter) type `TM`.

Now we can write a `compose` functor

```
module compose =
  λTI:*. λTM:*. λTO:*.
    λ(p : TrPair TI TM TO)
      mod {
        val f = λx:TI. p.n.f (p.m.f x)
      }
```

of type:

```
ΠTI:*. ΠTM:*. ΠTO:*.
  Π(p : TrPair TI TM TO)
    Tr TI TO
```

(We are using a little syntactic sugar here to make the example easier to read: `compose` takes three types and a module as parameters, whereas, strictly speaking, functors can only take modules as parameters. What we've written can be regarded as an abbreviation for a heavier notation where each "bare type" parameter is wrapped in a little module.)

Sharing by construction has an appealing directness, especially for programmers trained in the mental habits of higher-order functional languages. Moreover, since it relies only on abstraction and application for defining interface families, it can be carried out even in rudimentary module systems lacking the translucency required to express fibered interfaces. Unfortunately, it suffers from a defect that makes it difficult to use in practice: it does not scale to deep hierarchies of functors.

To see why, note how the parameterized form of the `compose` functor has to take the "middle type" `TM` as an explicit parameter. This is not so bad in this example, where the hierarchy is shallow. But suppose that, for some reason, we want to write a functor that composes together *four* transformations.

In the fibered style, we can write another interface that packages together two `TrPair`s with an appropriate `sharing` declaration relating the final output of the first with the initial input of the second:

```
interface TrQuad = int {
  module xy : TrPair
```

```
  module zw : TrPair
  sharing zw.m.In = xy.n.Out
}
```

Note how the coherence between the first and second transformations and between the third and fourth is expressed by the two uses of the `TrPair` interface—all that needs to be stated explicitly in `TrQuad` is the coherence of the second and third. The `compose4` functor is equally straightforward to write, using three applications of the original `compose`:

```
module compose4 =
  λ(q:TrQuad)
    let xtheny = compose q.xy in
    let zthenw = compose q.zw in
    let p = mod { module m = xtheny, module n = zthenw } in
    compose (p)
```

In the parameterized style, on the other hand, the `TrQuad` interface is much more awkward:

```
interface TrQuad = λT1:*. λT2:*. λT3:*. λT4:*. λT5:*.
                     int {
                       module xy : TrPair T1 T2 T3
                       module zw : TrPair T3 T4 T5
                     }
```

Note how the "internal" coherence constraints on `xy` and `zw` have "come to the outside" as parameters to `TrQuad`. Now, `compose4` looks like this:

```
module compose4 =
  λT1:*. λT2:*. λT3:*. λT4:*. λT5:*.
    λ(q : TrQuad T1 T2 T3 T4 T5)
      let xtheny = compose T1 T2 T3 q.xy in
      let zthenw = compose T3 T4 T5 q.zw in
      let p = mod { module m = xtheny, module n = zthenw } in
      compose T1 T3 T5 p
```

9.8.4 EXERCISE [★]: Suppose we extended this pattern to write `compose8` (using two applications of `compose4` and one of `compose`). How many type parameters would `compose8` need to take in the parameterized style? What about `compose16`? □

This example shows how sharing by parameterization forced the "plumbing" required to ensure coherence of a functor low in the dependency hierarchy must be recapitulated by every higher-level functor that uses it, by yet higher level functors that use these, etc. Setting up this plumbing (and worse,

maintaining it as the program evolves), quickly becomes impractical except for very shallow hierarchies. This failure of scalability was observed early on by MacQueen (MacQueen, 1984), but has not been widely recognized.

It is important to emphasize that the representation of interface families bears strongly on the method of expressing sharing relationships. If interface families are represented in parameterized form, then sharing by construction is the only available method of ensuring coherence. This is because we must instantiate two families by application to the common types or modules to ensure compatibility. On the other hand, if interface families are represented in fibered form, then either method of ensuring coherence is available, since we may either specialize two or more interfaces with the common component using the `where` construct, or we may specify that they cohere on the common components using `sharing`.

The crucial reason for this is that a fibered interface *is an interface*—it can be instantiated or not, as a given situation demands. This allows decisions about parameterization and sharing to be performed in a natural "post-hoc" manner: since each module internally recapitulates the whole module dependency graph, coherence requirements can be satisfied simply by adding a few equations tying together subgraphs as appropriate.

The Pragmatics of Functors

Since few present-day languages support families of modules, it is worth surveying the practical motivations for including them in a module system. These fall into several categories:

1. Many abstractions are naturally parametric in a type and its associated operations. For example, we illustrated in §9.8 that a dictionary abstraction is naturally parametric in both the type of its keys and the interpretation of that type as a pre-order. While in a given program we may only have only one dictionary with a single ordered key type, it is unarguably useful to obtain this as an instance of a common pattern available in a shared library. From the point of view of the library itself, it cannot provide for all possible instances of the dictionary abstraction other than by parameterizing it with respect to the ordered type of keys. Thus, *functors arise naturally in shared libraries.*

2. Many programs are naturally functorial in nature. For example, the architecture of the FoxNet implementation of TCP/IP (Biagioni, Haines, Harper, Lee, Milnes, and Moss, 1994) and related networking protocols is based on treating each "layer" of a protocol stack as a functor that is parametric in the layers that occur "above" and "below" it in the protocol

hierarchy. To take another example, the SML/NJ compiler for Standard ML implements cross-compilation by defining the central code generation module to be parametric in the target architecture module, so that several code generators can be simultaneously active and share the bulk of the code. Thus, *some program architectures are naturally functorial.*

3. A variety of link-time techniques—based on mechanisms such as "path hacking," class loaders, and tools provided by the programming environment—are commonly used to achieve effects similar to those expressible using functors. For example, partial linking of several object files into a single, further-linkable object file (using `ld -r` in Unix, for example) is nothing but a means of defining a functor whose parameters are the unresolved modules and whose result is the partially linked module constructed by the linker. Because such devices are extra-linguistic in nature, they can be unsafe because the external tools are not aware of typing restrictions. In particular, when used for languages with abstract types, such devices may violate coherence constraints, leading to unsafe code. Thus, *functors codify and formalize certain extra-linguistic programming practices.*

4. The program structuring features found in other languages can often be viewed as particular cases of functors. For example, Haskell encourages the use of type classes to define interpretations of types by operations. One may, for instance, introduce a class of ordered types as those that come equipped with a binary relation on them. Instances of this class are introduced by specifying the (sole) interpretation of a type by a given binary relation. Instances are often conditional on other instance requirements. For example, one may declare the type `Int` to be pre-ordered by the standard magnitude comparison function, and one may declare the product type `(A,B)` to be pre-order lexicographically (say), provided that `A` and `B` are also ordered. In the terminology of this chapter, type classes are simply interfaces and instance declarations are functors mapping zero or more instances of some classes to an instance of a designated class. (This use of functors is limited in that a given type may implement the class of ordered types in at most one way, whereas in general a type may admit many orderings. A benefit of this limitation is that the "functor applications" required to calculate appropriate instances of classes may be performed automatically by the compiler.) Thus, *mechanisms such as Haskell type classes may be viewed as stylized uses of functors.*

Some special cases of these general situations are amenable to treatment by more primitive modularity mechanisms. (This helps explain why the software industry has not yet ground to a halt due to the lack of functors in main-

stream languages!) These mechanisms are often more convenient for specific purposes, even if they may be subsumed by the more general mechanism of functors.

One class of examples concerns a parameterized abstraction, such as dictionaries (parameterized on keys), provided by a library. If we happen to know that any given program using this abstraction will need to instantiate it *just once*, then the abstraction itself need not be functorized. Instead, the dictionary library can simply be a module containing an unresolved external reference to a key module that must be resolved in the linking context of each program that uses it. Specifying the instance is generally achieved by extra-linguistic mechanisms such as modifying a search path or installing a special-purpose loader, but such mechanisms could also be internalized as a part of the module language.

Another important special case is found in object-oriented languages. In these languages, the role of modules is played, instead, by objects. Each object may be regarded as a module with one opaque type component (the type of its internal state); each of the object's methods is an operation on this type: i.e., it maps the object's current state (plus some other parameters) to a new state. (This correspondence between objects and modules is explored in depth in TAPL Chapter 24.) The role of functors is also played by methods, which may take objects as arguments and return them as results—that is, object-oriented languages embody a simple form of first-class module systems. While these mechanisms suffice in many cases, the limitations of objects as a replacement for full-blown modules can be severe. First, they offer no natural way to define a *group* of interrelated abstract types. Second, they provide no way of expressing coherence constraints—i.e., no way to specify that two arguments to a method must be objects *sharing the same internal representation*—leading, for example, to well-known problems with *binary methods* (Bruce et al., 1996).

An often-repeated argument for functors is that they may be used as a replacement for a linking mechanism. The idea is that all inter-module references are to be mediated by a functor—the so-called *fully functorized* style of programming—so as to improve program readability by making explicit all cross-module references. But adopting a fully functorized style amounts to replacing each definite reference in a module by an indefinite reference—the free module variable is λ-abstracted in the functor. A central "linking module" then applies these functors in dependency order to construct the complete system; i.e., the behavior of the linker itself is internalized and made explicit as a module-level program. Experience has shown this to be a bad idea: all this parameterization—most of it unnecessary—gives rise to spurious coherence issues, which must be dealt with by explicitly (and tediously)

`I ::= ...`		*interfaces:*
	Π_G `(m:`I_1`):`I_2	*generative functor*
	Π_A `(m:`I_1`):`I_2	*applicative functor*

Figure 9-6: Mechanisms for Applicative and Generative Functors

decorating module code with numerous sharing declarations, resulting in a net decrease in clarity and readability for most programs.

Functors and Determinacy

When is a functor application determinate? There are two possibilities, depending on whether the functor is *generative* or *applicative*. If generative, each instance of a functor that yields an abstract type "generates" a new abstract type at the point of instantiation. If applicative, there is one abstract type covering all instances of the functor with equivalent arguments. The difference between these two forms of functor is that the application of a generative functor is indeterminate, whereas the application of an applicative functor is determinate. To model both forms of functor we introduce two forms of functor interface, as described in Figure 9-6.

Needless to say, the classification of functors into applicative and generative is not completely arbitrary. If the body of a functor is indeterminate, then the functor can only be regarded as generative. Otherwise, an application of such a functor would be determinate, even though it is essentially a substitution instance of its indeterminate body. Thus a functor may be deemed applicative only if its body is determinate, but we may regard any functor as generative, by neglecting the possible determinacy of its body. Therefore, it is natural to posit that the applicative functor type is a subtype of the corresponding generative functor type.

9.8.5 EXERCISE [★★★]: Show that it is unsound to consider the generative functor type to be a subtype of the applicative. □

Assuming we have both applicative and generative functors at our disposal, when is it appropriate to use one or the other? We will consider several illustrative examples. If a functor implements an abstract type using per-instance state, it should be generative. For example, consider the implementation of a type of symbols implemented using a hash table:

```
interface ST = int {
```

```
  type Symbol
  val str2sym : String → Symbol
  val sym2str : Symbol → String
  val eq : Symbol × Symbol → Bool
}

module stFun ! ΠG ()ST =
  λG () mod {
    type Symbol = Int
    val table : string array =
      Array.new (100, NONE)
    val str2sym =
      λ(s:String) ...
    val sym2str =
      λ(s:Symbol)
        case Array.sub (table, n)
        of   SOME x ⇒ x | NONE ⇒ ...
    val eq =
      λ(n1, n2) = (n1 = n2)
  }

module stOne = stFun ()
module stTwo = stFun ()
```

The two instances, `stOne` and `stTwo` of `stFun` generate distinct abstract types `Symbol`: the type `stOne.Symbol` is distinct from the type `stTwo.Symbol`. Were these two types confused, symbols from one table could be intermixed with symbols from another, leading to incorrect results and run-time exceptions that could be avoided by keeping them apart. In particular, the `NONE` clause in the body of `stFun` can safely be omitted if the functor is generative, but must be included (or run the risk of a match failure) if it is applicative.

A natural example of an applicative functor is one whose argument consists solely of types, and whose result does not involve any effects. For in such a case there is no reason to distinguish abstract types in different instances of the functor. For example, consider a functor `setFun` that takes a type of elements as argument, and yields an abstract type of sets of these elements.

```
interface setFunInt =
  ΠA (m : int { type Elt })
    int {
      type Set
      val insert : m.Elt * Set → Set
      ...
```

```
    }

module setFun ! setFunInt =
  λ(m : int { type Elt })
    mod {
      type Set = ...
      val insert = ...
    }
```

Notice that the functor itself is sealed with an applicative functor type to ensure that the `Set` type in the result is abstract.

One might argue that two applications of `setFun` to a type (packaged as a module) should yield the *same* abstract type of sets, since there is no reason to distinguish them. For example, consider the following declarations:

```
module intMod = mod { type Elt = Int }
module intSet1 = setFun(intMod)
module intSet2 = setFun(intMod)
```

Since `setFun` is of applicative type, we would expect that the type `intSet1.Set` would be equivalent to the type `intSet2.Set`, yet both would be abstract. This is indeed the case, because the application `setFun(intMod)` is determinate and we need only note that `setFun(intMod).Set` is obviously equivalent to itself. On the other hand, if we define

```
module stringMod = mod { type Elt = string }
module stringSet = setFun(stringMod)
```

then `stringSet.Set` differs from `intSet.Set`, as expected, because the arguments differ.

If the argument to a functor consists of some types together with operations on them, but the body is purely functional, then either a generative or applicative interpretation might be used, according to whether we wish to distinguish abstractions that arise from different arguments. For example, a more realistic version of `setFun` above would require that the argument type come equipped with a comparison operation:

```
interface Ordered = int { type Elt val leq : Elt × Elt → bool }
module setFun′ =
  λ(m : Ordered)
    mod {
      type Set = ...
      val insert = ... m.leq ...
    }
  !
```

```
Π_A (m : Ordered)
  int {
    type Set
    val insert : m.Elt * Set → Set
    ...
  }
```

Now consider these two instances of `setFun`:

```
module intLeq ! Ordered where Elt=Int =
  mod { type Elt = Int val leq = <= }
module intSetLeq = setFun'(intLeq)
module intGeq ! Ordered where Elt=Int =
  mod { type Elt = Int val leq = >= }
module intSetGeq = setFun'(intGeq)
```

When sealed applicatively, `setFun'(intLeq).Set` is the *same* type as `setFun'(intGeq).S` even though the interpretations differ! When sealed generatively, however, these types are distinct.

One consequence of restricting an applicative functor to have a determinate body is that neither its body, nor any of its sub-modules, may be sealed! This explains why we sealed the `setFun` functor itself, rather than writing it in the form

```
module setFun =
  λ(m : int { type Elt })
    (mod {
       type Set = ...
       val insert = ...
     } !
     int {
       type Set
       val insert : m.Elt * Set → Set
       ...
     }).
```

Were we to do so, the body would be indeterminate, forcing the functor to be generative.

While sealing the functor itself can be used to impose abstraction on its instances, it cannot be used to impose abstraction within the body of the functor. One way to remedy this deficiency is to distinguish two forms of sealing, *static sealing* and *dynamic sealing*, and two associated forms of indeterminacy, *static indeterminacy* and *dynamic indeterminacy*. The dynamic forms of sealing and indeterminacy are just those considered up to now. The static

forms are added solely to enrich the class of applicative functors. A statically sealed module is statically indeterminate, which ensures representation independence. An applicative functor body is permitted to be statically indeterminate, but not dynamically indeterminate (which would force generativity). The terminology stems from considering that, for an applicative functor, abstraction is imposed *once* when the functor is type-checked, rather than *each time* the functor is applied; the abstraction effect is "static", rather than "dynamic".

9.9 Advanced Topics

First-Class Modules

The framework developed here is compatible with treating *modules as first-class values*, by which we mean that we may readily enrich the language to permit modules to be manipulated as ordinary values in the core language. For example, we may store a module in a data structure, then retrieve it and reconstitute it as a module, without violating representation independence or type safety. We need only ensure that any means of creating a module from a core language computation is considered indeterminate so as to preserve safety and representation independence.

If modules can be treated as first-class values, why not do away with the distinction between the core and module languages entirely? This would seem to simplify the language by eliminating the separation between the core and module languages. However, while it does achieve a simplification at a purely syntactic level, it rather significantly complicates its semantics and, therefore, its type system. Specifically, to be sufficiently expressive, we must enrich the language with dependent types, along with the apparatus of determinacy in order to ensure static typing. For example, Harper and Lillibridge have formulated such a type system (Harper and Lillibridge, 1994), but these complications result in an undecidable type checking problem! Moreover, even if we consolidate the core and module languages, we nevertheless must introduce mechanisms to support separate compilation, themselves a form of module system. In short, a separate module level arises even if you try to avoid it.

Higher-Order Modules

Higher-order modules present some interesting further difficulties. The classic (if somewhat contrived) motivating example is the `apply` functor, defined as follows:

```
module apply =
  λ(f : Π(a:I)J)
    λ(a : I)
      f(a)
module m ! I = ...
module f ! Π(a:I)J = ...
module n = f(m)
module p = apply(f)(m)
```

There should be no difference between the module `n` and the module `p`. Whether this is the case or not depends on whether the functor argument to `apply` is taken to be applicative or generative. If it is applicative, then we may ascribe the following type to `apply`:

```
ΠA (f:ΠA (i:I)J):ΠA (a:I) (J where X=f(a).X).
```

This expresses the dependence of the result type `X` on the two arguments consistent with the definition of `apply`. Indeed, `apply(f)(a).X` is equivalent to `f(a).X`, as desired.

On the other hand the functor argument to `apply` might be taken to be generative, in which case the best typing for `apply` is

```
ΠG (f:ΠG (i:I):J) ΠG (a:I) J
```

We cannot track the definition of `X` in the result, because the application `f(a)` is indeterminate when `f` is generative (even though `f` and `a` are variables). But now `f(a).X` is itself ill-formed, as is `apply(f)(a).X`, so here again there is no significant difference between `n` and `p`.

It has been suggested that there should only be *one* apply functor that covers both cases illustrated above. To do so requires that we employ a form of intersection type (at the level of interfaces) that captures the two forms of behavior just described. An alternative, suggested by MacQueen and Tofte (MacQueen and Tofte, 1994), is to refrain from assigning types to functors, instead typing only their instances when the arguments are known. Unfortunately this approach conflicts with separate compilation, a fundamental design criterion for module systems.

A second issue related to higher-order modules is the form of module equivalence. In the presence of unknown functors (i.e., functors as parameters) it is essential to consider a notion of equality of modules to be used during type checking. For example, we may wish to compare the type expression `f(a).X` with `g(a).X`, where `f` and `g` are unknown applicative functors and `a` is a determinate module of suitable type. These two types are equivalent only if `f` and `g` are equivalent functors. But what do we mean by

equivalence? An important criterion is that we wish to recognize the equivalence of `f(a).X` and `g(a).X` whenever `F(A).X` is equivalent to `G(A).X` for all concrete instances `F`, `G`, and `A` of `f`, `g`, and `a`, respectively.

For example, in a second-class module system only the static parts of a module expression are relevant to type checking, so we expect two (determinate) modules to be equivalent exactly when their static parts are equivalent (up to core language type equivalence). For obvious reasons, this is called *static equivalence*. It is the coarsest equivalence on modules that conservatively extends core language type equivalence. The alternative, *dynamic equivalence*, takes account of the dynamic, as well as static, parts of a module. It is most appropriate for a first-class module system. In general dynamic equivalence is undecidable, so in practice it must be conservatively approximated.

Recursive Modules

The model of linking discussed in §9.1 requires that the dependency relation among modules be acyclic—that there be a linear ordering of the modules consistent with their dependencies. It is natural to consider whether this restriction might be lifted to permit a more general form of "cross-linking" of modules. Since cyclic dependencies amount to (direct or indirect) self-reference, one approach to modelling such a generalization is via *recursive modules* (Crary, Harper, and Puri, 1999a; Russo, 2001). Another approach is to devise a linking formalism that permits cyclic dependencies (Hirschowitz and Leroy, 2002; Flatt and Felleisen, 1998).

Cyclic dependencies raise some significant problems that must be addressed in any satisfactory solution. Most importantly, permitting recursive modules should not disrupt the semantics of the underlying language. Without restriction, cyclic dependencies among modules can introduce a type `A` satisfying the equation $\texttt{A} = \texttt{A} \rightarrow \texttt{int}$, or a value `v` of type `int` satisfying the equation `v=v+1`. In most languages such equations have no solution, and should not be permitted. Another issue is the interaction with effects. Permitting cyclic dependencies conflicts with the need for a linear initialization order consistent with dependencies. Care must be taken to ensure that values are not referenced before they are defined (or, if they are, that such references are caught at run time). Finally, for maximum flexibility, mutually recursive modules should be separately compilable. This requires some form of "forward" declaration to cut cycles in the dependency graph. It also requires a linking formalism that can support mutually cross-referencing modules, even in the presence of type declarations.

9.10 Relation to Existing Languages

The design issues discussed in this chapter are largely motivated by the ML module system. There are two closely related realizations of the ML module system, the Standard ML module system and the Objective Caml module system. Basic modules are called structures, interfaces are called signatures, and functors are so-called in both cases. Both designs provide for hierarchy and parameterization using essential the same mechanisms as described here, and both adopt the approach to sharing by specification described in §9-5. The designs differ significantly in their treatment of separate compilation, the avoidance problem, and higher-order modularity. The two languages are based on rather different foundations. Standard ML is defined by an elaboration relation that constitutes an algorithmic specification of the well-formed programs. Objective Caml lacks a formal definition, but the design follows quite closely a type theory of the general kind considered here.

Standard ML, as officially defined (Milner, Tofte, Harper, and MacQueen, 1997c), permits only first-order, generative functors, provides no support for separate or incremental compilation, and handles the avoidance problem by a technical device that sacrifices principality. To amplify the last point first, the elaboration relation that defines the static semantics of Standard ML relies on an internal notion of "type names" that are generated during elaboration. Hidden abstract types are represented by type names that cannot be designated by any Standard ML type expression, and hence internal "signatures" are not expressible by any signature in the language. Consequently, the Standard ML module system is not principal in the sense defined on page 477. Since the formal definition does not address separate compilation, each implementation provides its own mechanisms. The most widely used implementation, Standard ML of New Jersey (SML/NJ), has a well-developed compilation manager (Blume and Appel, 1999; Blume, 2002) that supports incremental and cut-off compilation. SML/NJ also provides extensions to permit higher-order modularity that rely on elaborate internal representations of functors that cannot be written in any source language signature, and is therefore incompatible with separate compilation. Moscow/ML (Sestoft, 2003) is an implementation of Standard ML based on a type-theoretic interpretation of the language. It provides recursive and first-class structures, and both applicative and generative functors (Dreyer, Crary, and Harper, 2003).

Objective Caml permits higher-order, applicative functors; supports separate and incremental compilation; and handles the avoidance problem by sacrificing principality. Again taking the last point first, Objective Caml rejects certain well-formed programs (in the sense of the underlying type theory of the language) when the implementation does not succeed in weaken-

ing a signature to avoid the occurrence of an abstract type (Dreyer, Crary, and Harper, 2003). The commitment to applicative functors stems from a desire to permit type selections of the form `f(m).X` in sharing specifications. The absence of generative functors weakens the type system significantly when imperative constructs are used, as described in §9-5. No definitive statements can be made about the language, however, but only about its sole implementation.

The Haskell (Peyton Jones, 2003) module system is rather weak, providing only rudimentary namespace management. This deficiency is ameliorated by type classes. Viewed in terms of the framework of this chapter, the Haskell type class system amounts to a stylized use of modules. Polymorphic abstraction is generalized to functor abstraction — expressions take not only types, but associated operations, as arguments. The functor arguments are generated automatically during type inference based on a significant methodological restriction: no type may admit more than one interpretation by a given set of operations. (For example, in conjunction with type classes no type may be partially ordered in more than one way in a given program.) These interpretations are specified by type class declarations that amount to functor definitions. The type checker implicitly instantiates these functors (through a process of backchaining) to determine the required implicit arguments. Experimental designs for richer modularity mechanisms have been proposed in the literature. For example, Jones (Jones, 1996) regards modules as polymorphic records, which forces the programmer to manage explicitly the separation of the static from the dynamic parts of a module.

Flatt and Felleisen's units (1998) provide a form of modularity for Scheme programs (and other languages) that emphasizes separate compilation and recursive linking. Their language does not consider type abstraction or the associated problems of type sharing. In some realizations, units are first-class values, amounting to records in the underlying language (Flatt and Felleisen, 1998). In others, units are used to structure existing C code to provide namespace management and a flexible linking formalism (Reid, Flatt, Stoller, Lepreau, and Eide, 2000).

9.10.1 EXERCISE [★★★]: The C language lacks an internal notion of module, preferring instead to exploit the ambient file system to provide most of the requisite mechanisms. Discuss. □

9.10.2 EXERCISE [★★★]: The Java language also lacks direct analogs of most of the mechanisms we have introduced. However, Java does offer a rich collection of program structuring mechanisms, some of which can be used to achieve effects similar to the ones illustrated here. Discuss. □

9.10.3 EXERCISE [★★★, RECOMMENDED]: The analog of an interface in ML is called a *signature*. Does Standard ML or Objective Caml have principal signatures for modules? □

9.11 History and Further Reading

The development of the linguistic and methodological foundations of data abstraction and modularity dates back to the earliest days of academic computer science. Seminal work by Wirth (1973) and Hoare (1972) (among many others) was influential on the development of languages in the Algol family such as Pascal (Jensen and Wirth, 1975), Modula-2 (Wirth, 1983), CLU (Liskov, 1993), and Modula-3 (Cardelli, Donahue, Jordan, Kalso, and Nelson, 1989). The Lisp family of languages (Guy L. Steele, 1990) influenced the design of ML (Gordon, Milner, and Wadsworth, 1979a), which introduced type inference, polymorphism, and abstract types. This sparked the development of several languages, such as HOPE (Burstall, MacQueen, and Sannella, 1980), Standard ML (Milner, Tofte, Harper, and MacQueen, 1997c), Objective Caml (Leroy, 1996a), and Haskell (Peyton Jones, 2003), founded on these ideas. The ML module system, originally proposed by MacQueen (1984), further developed in the design of Standard ML and Objective Caml, forms the conceptual basis for much of the material presented in this chapter.

The theoretical framework employed in this chapter (and in *Types and Programming Languages*) is the typed λ-calculus. One important topic was to develop type systems to support data abstraction. A fundamental first step was taken by Mitchell and Plotkin (1988b) who related abstract types to second-order existential quantification, extending the connection between type polymorphism and second-order universal quantification discovered by Girard (1972b) and Reynolds (1974b). MacQueen (1986) pointed out that existential types are not adequate for expressing modular structure, suggesting instead a formalism based on dependent types. These initial steps provided the impetus for further research into type systems for modularity with the overall goal of providing the abstraction guarantees afforded by existential types and the flexible modular programming mechanisms afforded by dependent types.

One strand of research focused on enriching the existential framework to support controlled propagation of type sharing information in a program. Three important developments were Harper and Lillibridge's translucent sum types (1994; 1996), Cardelli's "dot notation" (1990b) and Leroy's manifest types (1994a; 1996b), and Stone and Harper's singleton kinds (Stone and Harper, 2000b; Stone, 2000). These type systems support hierarchy and

parameterization with control over the propagation of type sharing relationships, even in the presence of first-class modules.

Another strand focused on developing the mechanisms of dependent types to support higher-order modules. Building on MacQueen's suggestions, Harper and Mitchell proposed a calculus of dependent types suitable for modelling many aspects of the ML module system (Harper and Mitchell, 1993). This framework was further refined by Harper, Mitchell, and Moggi (1990a) to ensure respect for the phase distinction in a fully expressive higher-order module system. This formalism also provided the foundation for compiling modules into typed intermediate languages (Shao, League, and Monnier, 1998). Further work by Russo (Russo, 1999) underscored this point in the setting of a type-theoretic semantics for Standard ML. Shao (Shao, 1999) considered a type system that ensures the existence of principal signatures, at the expense of ruling out some programs that are expressible in ML.

The abstract-type formalisms provided only very weak support for higher-order modules, and the dependent-type formalisms provided no support for abstraction. Leroy introduced applicative functors (1995a) in an effort to enrich the abstract type formalism with richer higher-order constructs, but in the process sacrificed generative type abstraction. A fully comprehensive formalism was introduced by Dreyer, Crary, and Harper+(2003), based on interpreting type abstraction as a *pro forma* computational effect.

A rather different approach to the semantics of modularity is the elaboration framework of The Definition of Standard ML (Milner, Tofte, Harper, and MacQueen, 1997c). The type checking rules for modular programming are given by an algorithm (expressed in inference rule format) for computing an internal representation of the interface of a module. A weakness of this approach is that it lacks any connection with the typed λ-calculus formalisms that form the foundation for the semantics and implementation of programming languages. This deficiency was addressed by Russo (1998), who re-formulated The Definition using constructs from type theory. Harper and Stone (2000a) provided an alternative definition for Standard ML based on a separation between elaboration, which included type inference, overloading resolution, pattern compilation, and semantics, which was based on a foundational type theory for modularity.

10 *Type Definitions*

By Christopher A. Stone

Practical use of any interesting type system often involves large and complex types. It is therefore convenient, and common, to define abbreviations for types of interest. For example, TAPL Chapter 20 put

```
NatList ≝ μX. <nil:Unit, cons:{Nat,X}>,
```

allowing the type of the `cons` function to be written `Nat → NatList → NatList`, rather than the much larger type

```
Nat →
  (μX. <nil:Unit, cons:{Nat,X}>)  →
     (μX. <nil:Unit, cons:{Nat,X}>).
```

This definition of `NatList` was not encoded into the λμ calculus studied in that chapter; λμ had no syntax for giving names to types. Instead, the definition was intended purely as meta-notation.

In any specific application many such definitions may be required, each building on the previous ones. For example, in some very primitive languages the type of natural numbers itself is defined as a recursive sum or variant type, e.g.,

```
Nat ≝ μY.<zero:Unit, succ:Y>.
```

In this case, `Nat → NatList → NatList` would be abbreviating the still larger type

```
μY. <zero:Unit, succ:Y> →
  (μX. <nil:Unit, cons:{μY.<zero:Unit, succ:Y>,X}>)  →
     (μX. <nil:Unit, cons:{μY.<zero:Unit, succ:Y>,X}>).
```

As long as definitions are non-circular they are convenient but inessential syntactic sugar. In principle all symbolic names can be replaced by their definitions and so we can ignore them when reasoning about the language: whenever we write types such as `Nat` → `NatList` → `NatList`, we are "officially" referring to the corresponding expanded types.

However, in some instances type definitions are explicitly part of the language itself. For example, the ML language permits type definitions by the user; the SML Standard Basis Library even specifies that `string` is a predefined synonym for `char vector`. C and C++ allow similar definitions via `typedef`.

Such definitions may be preserved by language implementations rather than immediately being substituted away; after this expansion process, types can be much larger and take more time to manipulate. Techniques such as hash consing (**?**) can ameliorate the time and space problems, but expanded types are also significantly less readable; if a type is originally written using abbreviations, it is often desirable to retain these abbreviations whenever the type is displayed (e.g., when reporting errors during type checking, as discussed in TAPL §11.4). If definitions are retained, we would still like to know that properties such as type safety continue to hold, and that algorithms (e.g., for type checking or code transformations) remain correct.

For simple definitions like `NatList` above, this may be straightforward to verify, but as definitional mechanisms become more sophisticated (e.g., if definitions involve locally-scoped bound variables or appear in module interfaces), reasoning about their effects on the language can be surprisingly difficult.

10.0.1 EXERCISE [★★,RECOMMENDED]: An alternative method for abbreviating types would be to let `Typeof(t)` represent the type of `t`, where `t` can be any term with a unique type. Then, for instance, if a variable `x` were defined to be a record with many fields, we could thereafter say `Typeof(x)` rather than writing out the corresponding large record type. In some contexts this might be more compact than giving a symbolic name to the record type — we might be able to avoid ever writing the record type at all.

In a language with subtyping, few terms may have unique types. In this case, the natural extension would be for `Typeof(t)` to abbreviate the minimal (most-specific) type of the term `t`.

Show that when `Typeof` is combined with subtyping in this fashion, source transformations such as beta-reduction may not preserve well-typedness. □

More commonly, complications arise because of the interaction between definitions and scope. For example, after the ML definition

```
module N = struct
```

```
    type t = int
    let  x : t = 3
end
```

we can use `N.t` as a synonym for `int`. In this case we have a definition not for the simple name `t`, but for the entire projection `N.t`. Moreover, module components are often referenced indirectly so that naively trying to eliminate this definition by replacing `N.t` by `int` will not work. For example, the further ML code

```
module N' = N

module Diag = functor(S : sig
                            type t
                            val x : t
                          end) →
                struct
                  type u = S.t * S.t
                  let  y : u = (S.x, S.x)
                end

module NN = Diag(N)
```

nowhere contains the projection `N.t`, and yet a correct type checker must nevertheless conclude both that `N'.t` is a synonym for `int` (since by definition the components of `N'` are the same as the components of `N`) and that `NN.u` is equal to the type `int` × `int` (since this is guaranteed by the definition of `Diag`). Additionally, the definition for `u` in the functor's result must be remembered for type checking further uses of `Diag`, and yet we cannot even in principle simply substitute this away; its definition depends on `Diag`'s argument.

It therefore seems worthwhile to study type definitions as primitive concepts. The primary focus of this chapter is definitions of types because they have the most significant effect on type equivalence and type checking and hence on language properties such as safety. Very similar approaches can be and have been used, however, to study term-level definitions and their effects upon term equivalence.

The remainder of the chapter presents three approaches to formalizing definitions. Section 10.1 adds *primitive definitions* of type variables to the typing context; the context can either record `X::K` if `X` is an unknown type variable of kind `K`, or else can record `X::K=T` if `X` is further known to be equal to the type `T`. This is essentially the mechanism required to formalize definitions such as `NatList` above.

Section 10.2 formalizes some of the ideas of Chapter 9 by considering a calculus of second-class modules based on *translucent sums*. Again we have a choice between specifying either a kind or else a kind and a definition, but in this system all type definitions appear in module interfaces.

Finally, Section 10.3 drops the distinction between specifying a kind and specifying a kind and a definition by incorporating definitions into the kind system itself. The kind `*` classifies all ordinary types, while the new, more-precise *singleton kind* $\mathcal{S}$`(T)` classifies those ordinary types equivalent to `T`. This generalization allows definitions at any point where a kind is specified. In addition to permitting alternate formulations of the two previous systems (using singletons instead of primitive definitions), in combination with dependent kinds (analogous to the dependent types of Chapter 4), the translucent sums can be largely encoded into a module-free language, using a *phase-splitting* tranformation.

All three systems are described as extensions of F^ω, the higher-order polymorphic lambda calculus; the types and kinds of the base language are shown in Figure 10-1.

The one possibly unfamiliar judgment in this formulation is $\Gamma \vdash \diamond$, which formalizes the notion of Γ being a well-formed context (see TAPL §30.3.18). A typing context is well-formed if all bound variables are distinct, and if each type within the context is well-formed with respect to the preceeding parts of the context. For convenience in working with the metatheory, all judgments require (explicitly or implicitly) their typing context to be well-formed.

10.1 Definitions in the Typing Context

The simplest approach to adding definitions is simply to extend a language such as F^ω with defined type variables, formalizing definitions such as those for `Nat` and `NatList` above. The language $F^\omega_{\mathtt{let}}$ defined in Figure 10-2 extends F^ω with primitive type definitions. The syntax of contexts is broadened to permit defined type variables; the new rule Q-DEF equates type variables and their definitions. Equivalence of well-formed types therefore depends upon definitions in the typing context. In $F^\omega_{\mathtt{let}}$ we can prove

```
X::*=Int ⊢ Int→X ≡ X→Int :: *
```

but not

```
X::*=Bool ⊢ Int→X ≡ X→Int :: *
```

or

```
X::* ⊢ Int→X ≡ X→Int :: *.
```

F^ω

$$
\begin{array}{lll}
\mathtt{T} ::= & & \textit{types:} \\
& \mathtt{Int}, \mathtt{Bool}, \ldots & \textit{base types} \\
& \mathtt{X} & \textit{type variable} \\
& \mathtt{T} \to \mathtt{T} & \textit{type of functions} \\
& \forall \mathtt{X} :: \mathtt{K} . \mathtt{T} & \textit{universal type} \\
& \lambda \mathtt{X} :: \mathtt{K} . \mathtt{T} & \textit{type operator abstraction} \\
& \mathtt{T}\ \mathtt{T} & \textit{type operator application} \\
\mathtt{K} ::= & & \textit{kinds:} \\
& * & \textit{kind of proper types} \\
& \mathtt{K} \Rightarrow \mathtt{K} & \textit{kind of type operators}
\end{array}
$$

Context Validity $\boxed{\Gamma \vdash \diamond}$

$$\frac{}{\cdot \vdash \diamond} \quad \text{(CTX-EMPTY)}$$

$$\frac{\Gamma \vdash \mathtt{T} :: * \qquad \mathtt{x} \notin \mathit{dom}(\Gamma)}{\Gamma, \mathtt{x} : \mathtt{T} \vdash \diamond} \quad \text{(CTX-TYPE)}$$

$$\frac{\Gamma \vdash \diamond \qquad \mathtt{X} \notin \mathit{dom}(\Gamma)}{\Gamma, \mathtt{X} :: \mathtt{K} \vdash \diamond} \quad \text{(CTX-KIND)}$$

Kinding $\boxed{\Gamma \vdash \mathtt{T} :: \mathtt{K}}$

$$\frac{\mathtt{X} :: \mathtt{K} \in \Gamma \qquad \Gamma \vdash \diamond}{\Gamma \vdash \mathtt{X} :: \mathtt{K}} \quad \text{(K-VAR)}$$

$$\frac{\Gamma \vdash \diamond \qquad \mathtt{T} \in \{\mathtt{Int}, \mathtt{Bool}, \ldots\}}{\Gamma \vdash \mathtt{T} :: *} \quad \text{(K-BASE)}$$

$$\frac{\Gamma, \mathtt{X} :: \mathtt{K}_1 \vdash \mathtt{T}_2 :: \mathtt{K}_2}{\Gamma \vdash \lambda \mathtt{X} :: \mathtt{K}_1 . \mathtt{T}_2 :: \mathtt{K}_1 \Rightarrow \mathtt{K}_2} \quad \text{(K-ABS)}$$

$$\frac{\Gamma \vdash \mathtt{T}_1 :: \mathtt{K}_{11} \Rightarrow \mathtt{K}_{12} \qquad \Gamma \vdash \mathtt{T}_2 :: \mathtt{K}_{11}}{\Gamma \vdash \mathtt{T}_1\ \mathtt{T}_2 :: \mathtt{K}_{12}} \quad \text{(K-APP)}$$

$$\frac{\Gamma \vdash \mathtt{T}_1 :: * \qquad \Gamma \vdash \mathtt{T}_2 :: *}{\Gamma \vdash \mathtt{T}_1 \to \mathtt{T}_2 :: *} \quad \text{(K-ARROW)}$$

$$\frac{\Gamma, \mathtt{X} :: \mathtt{K}_1 \vdash \mathtt{T}_2 :: *}{\Gamma \vdash \forall \mathtt{X} :: \mathtt{K}_1 . \mathtt{T}_2 :: *} \quad \text{(K-ALL)}$$

Type Equivalence $\boxed{\Gamma \vdash \mathtt{S} \equiv \mathtt{T} :: \mathtt{K}}$

$$\frac{\Gamma \vdash \mathtt{T} :: \mathtt{K}}{\Gamma \vdash \mathtt{T} \equiv \mathtt{T} :: \mathtt{K}} \quad \text{(Q-REFL)}$$

$$\frac{\Gamma \vdash \mathtt{T} \equiv \mathtt{S} :: \mathtt{K}}{\Gamma \vdash \mathtt{S} \equiv \mathtt{T} :: \mathtt{K}} \quad \text{(Q-SYM)}$$

$$\frac{\Gamma \vdash \mathtt{S} \equiv \mathtt{U} :: \mathtt{K} \qquad \Gamma \vdash \mathtt{U} \equiv \mathtt{T} :: \mathtt{K}}{\Gamma \vdash \mathtt{S} \equiv \mathtt{T} :: \mathtt{K}} \quad \text{(Q-TRANS)}$$

$$\frac{\Gamma \vdash \mathtt{S}_1 \equiv \mathtt{T}_1 :: * \qquad \Gamma \vdash \mathtt{S}_2 \equiv \mathtt{T}_2 :: *}{\Gamma \vdash \mathtt{S}_1 \to \mathtt{S}_2 \equiv \mathtt{T}_1 \to \mathtt{T}_2 :: *} \quad \text{(Q-ARROW)}$$

$$\frac{\Gamma, \mathtt{X} :: \mathtt{K}_1 \vdash \mathtt{S}_2 \equiv \mathtt{T}_2 :: *}{\Gamma \vdash \forall \mathtt{X} :: \mathtt{K}_1 . \mathtt{S}_2 \equiv \forall \mathtt{X} :: \mathtt{K}_1 . \mathtt{T}_2 :: *} \quad \text{(Q-FORALL)}$$

$$\frac{\Gamma, \mathtt{X} :: \mathtt{K}_1 \vdash \mathtt{S}_2 \equiv \mathtt{T}_2 :: \mathtt{K}_2}{\Gamma \vdash \lambda \mathtt{X} :: \mathtt{K}_1 . \mathtt{S}_2 \equiv \lambda \mathtt{X} :: \mathtt{K}_1 . \mathtt{T}_2 :: \mathtt{K}_1 \Rightarrow \mathtt{K}_2} \quad \text{(Q-ABS)}$$

$$\frac{\Gamma \vdash \mathtt{S}_1 \equiv \mathtt{T}_1 :: \mathtt{K}_{11} \Rightarrow \mathtt{K}_{12} \qquad \Gamma \vdash \mathtt{S}_2 \equiv \mathtt{T}_2 :: \mathtt{K}_{11}}{\Gamma \vdash \mathtt{S}_1\ \mathtt{S}_2 \equiv \mathtt{T}_1\ \mathtt{T}_2 :: \mathtt{K}_{12}} \quad \text{(Q-APP)}$$

$$\frac{\Gamma, \mathtt{X} :: \mathtt{K}_{11} \vdash \mathtt{S}_{12} \equiv \mathtt{T}_{12} :: \mathtt{K}_{12} \qquad \Gamma \vdash \mathtt{S}_2 \equiv \mathtt{T}_2 :: \mathtt{K}_{11}}{\Gamma \vdash (\lambda \mathtt{X} :: \mathtt{K}_{11} . \mathtt{S}_{12})\, \mathtt{S}_2 \equiv [\mathtt{X} \mapsto \mathtt{T}_2]\mathtt{T}_{12} :: \mathtt{K}_{12}} \quad \text{(Q-BETA)}$$

Figure 10-1: Types and Kinds of F^ω

$F^{\omega}_{\mathtt{let}}$ *Extends* F^{ω}

New syntactic forms

$\Gamma ::= \ldots$ *contexts:*
$\Gamma, \mathtt{X::K=T}$ *open-scope definition*

$\mathtt{t} ::= \ldots$ *terms:*
$\mathtt{let\ X = T\ in\ t}$ *closed-scope definition*

Type Equivalence $\boxed{\Gamma \vdash \mathtt{S} \equiv \mathtt{T} :: \mathtt{K}}$

$$\frac{\mathtt{X::K=T} \in \Gamma \qquad \Gamma \vdash \diamond}{\Gamma \vdash \mathtt{X} \equiv \mathtt{T} :: \mathtt{K}} \quad \text{(Q-DEF)}$$

Kinding $\boxed{\Gamma \vdash \mathtt{T} :: \mathtt{K}}$

$$\frac{\mathtt{X::K=T} \in \Gamma \qquad \Gamma \vdash \diamond}{\Gamma \vdash \mathtt{X} :: \mathtt{K}} \quad \text{(K-VARDEF)}$$

Context Validity $\boxed{\Gamma \vdash \diamond}$

$$\frac{\Gamma \vdash \mathtt{T} :: \mathtt{K} \qquad \mathtt{X} \notin \mathit{dom}(\Gamma)}{\Gamma, \mathtt{X::K=T} \vdash \diamond} \quad \text{(CTX-DEF)}$$

Typing rules $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\Gamma \vdash \mathtt{T}_1 :: \mathtt{K}_1 \qquad \mathtt{X} \notin \mathit{FV}(\mathtt{T}_2) \qquad \Gamma, \mathtt{X::K_1=T_1} \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \mathtt{let\ X=T_1\ in\ t}_2 : \mathtt{T}_2} \quad \text{(T-TLET)}$$

Evaluation rules $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$\mathtt{let\ X::K=T\ in\ t} \longrightarrow [\mathtt{X} \mapsto \mathtt{T}]\mathtt{t} \quad \text{(E-TLET)}$$

Figure 10-2: Adding definitions to the context

This is an immediate difference from F^{ω}, where type equivalence can be determined by looking only at the two types involved.

Context validity is extended by the rule CTX-DEF, which requires that definitions make sense in the preceeding context. Consequently, type definitions in well-formed contexts are never circular, e.g., if $\mathtt{X::K=T} \in \Gamma$ and $\Gamma \vdash \diamond$ then X is not free in T.[1] The non-circularity restriction ensures that definitions could in principle all by replaced by their expanded forms.

The new kinding rule K-VARDEF looks up the kind of a defined type variable, paralleling the F^{ω} rule K-TVAR for type variables without definitions.

Definitions in the context can be considered ambient and usable anywhere, and hence are *open-scope*. We can also use this mechanism to describe the typing of primitive *closed-scope* (local) type definitions; the syntax of $F^{\omega}_{\mathtt{let}}$ includes a `let` form whose type checking rule T-TLET puts the definition into the context for use while type checking a term. Thus, for example, the code `let X=Int in (λx:X.x+1)(4)` is well-typed; a proof appears in Figure 10-3, where the omitted leaf proofs are uninteresting context-validity checks.

10.1.1 EXERCISE [★★]: The premise $\mathtt{X} \notin \mathit{FV}(\mathtt{T}_2)$ in T-TLET ensures that the local variable X is used only within the scope where it is defined. Show that the type system would be unsound if this requirement were omitted. □

1. The non-circularity requirements for context validity would not prevent T itself from being a recursive type as in TAPL Chapter 20.

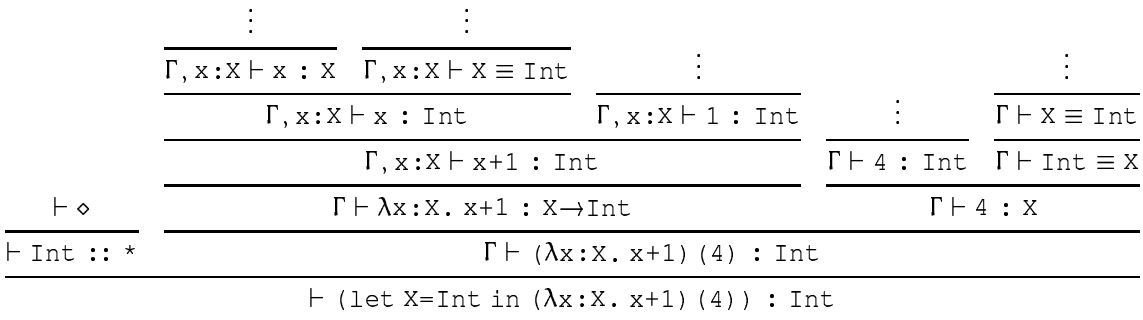


Figure 10-3: Typing of `let X=Int in (λx:X.x+1) (4)`**, using** $\Gamma \stackrel{\text{def}}{=} \texttt{X::*=Int}$

10.1.2 EXERCISE [★★★]: What changes to the simply-typed calculus $\lambda_{\_}\{\rightarrow\}$ would be necessary, if one wanted to add primitive definitions of type variables? □

10.1.3 EXERCISE [★★★★]: If a program involving top-level term and type definitions were divided up into separate files (compilation units) then an $F^{\omega}_{\texttt{let}}$ typing context might be a good starting point for describing the interfaces of these compilation units; type definitions must appear in the interface if they are to be useable in separately-compiled code. Formalize a language with primitive compilation units (sequences of top-level definitions), generalize interfaces to describing imports and exports, and define what it means to link compilation units together. □

Deciding Equivalence

The hardest part of type checking in $F^{\omega}_{\texttt{let}}$, as in F^{ω}, is deciding type equivalence. It does suffice to replace all defined variables by their definitions and then perform equivalence testing as in standard F^{ω} by comparing the expanded types' beta-normal forms. Proving that this algorithm is not just sound but complete is not completely immediate, however. More generally it is useful to know that expanding definitions and reducing applications can be interleaved, e.g., that a definition could be first beta-reduced and then substituted, but that eventually the same normal form appears.

We therefore apply techniques similar to those discussed TAPL Chapter 30 for the confluence and normalization of beta-reduction for F^{ω}. The corresponding notion of reduction in $F^{\omega}_{\texttt{let}}$ is more complex, as reduction includes

Type Reduction $\boxed{\Gamma \vdash S \rightsquigarrow T}$

$$\frac{}{\Gamma_1, X::K=T, \Gamma_2 \vdash X \rightsquigarrow T} \text{(R-DEF)}$$

$$\frac{\Gamma \vdash S_1 \rightsquigarrow T_1}{\Gamma \vdash S_1 \to T_2 \rightsquigarrow T_1 \to T_2} \text{(R-ARROW1)}$$

$$\frac{\Gamma \vdash S_2 \rightsquigarrow T_2}{\Gamma \vdash T_1 \to S_2 \rightsquigarrow T_1 \to T_2} \text{(R-ARROW2)}$$

$$\frac{X \notin dom(\Gamma) \quad \Gamma, X::K_1 \vdash S_2 \rightsquigarrow T_2}{\Gamma \vdash \forall X::K_1 . S_2 \rightsquigarrow \forall X::K_1 . T_2} \text{(R-ALL)}$$

$$\frac{X \notin dom(\Gamma) \quad \Gamma, X::K_1 \vdash S_2 \rightsquigarrow T_2}{\Gamma \vdash \lambda X::K_1 . S_2 \rightsquigarrow \lambda X::K_1 . T_2} \text{(R-ABS)}$$

$$\frac{\Gamma \vdash S_1 \rightsquigarrow T_1}{\Gamma \vdash S_1\ T_2 \rightsquigarrow T_1\ T_2} \text{(R-APP1)}$$

$$\frac{\Gamma \vdash S_2 \rightsquigarrow T_2}{\Gamma \vdash T_1\ S_2 \rightsquigarrow T_1\ T_2} \text{(R-APP2)}$$

$$\Gamma \vdash (\lambda X::K_{11} . S_{12})\ S_2 \rightsquigarrow [X \mapsto S_2]S_{12} \quad \text{(R-APPABS)}$$

Figure 10-4: Type Reduction with Definitions

the possibility of expanding a definition and hence depends upon the context.

Specifically, the reduction steps for bringing a type closer to normal form are the standard beta-reduction rule

$$\Gamma \vdash (\lambda X::K_{11} . T_{12})T_2 \rightsquigarrow [X \mapsto T_2]T_{12}$$

and a definition expansion step, often referred to as delta-reduction[2]

$$\ldots, X::K=T, \ldots \vdash X \rightsquigarrow T$$

along with other rules to allow these operations to be applied inside larger types. The full definition of $F^{\omega}_{\mathtt{let}}$ type reduction is shown in Figure 10-4.

It is convenient to define $\rightsquigarrow_\beta$ to be the restriction of this definition to all the rules except R-DEF (i.e., to be beta-reduction, which in any typing context is exactly the same as beta-reduction in F^{ω}) and $\rightsquigarrow_\delta$ to be the restriction to all rules except for R-APPABS (i.e., to be delta-reduction). Note that if $\Gamma \vdash S \rightsquigarrow T$ then either $\Gamma \vdash S \rightsquigarrow_\beta T$ or $\Gamma \vdash S \rightsquigarrow_\delta T$ since every proof must use exactly one of the two axioms.

As in F^{ω}, we wish to show that that reduction is confluent, that provably equivalent types have a common reduct, and finally that reduction is strongly normalizing (guaranteed to terminate). Therefore, we can determine

2. Some authors (e.g., Barendregt, 1984) prefer to use the name delta-reduction for execution of built-in primitive operators such as addition, e.g., replacing `3+4` by `7`.

Parallel Reduction $\boxed{\Gamma \vdash S \Rrightarrow T}$

$$\frac{}{\Gamma \vdash S \Rrightarrow S} \quad \text{(QR-Refl)}$$

$$\frac{}{\Gamma_1, X::K=T, \Gamma_2 \vdash X \Rrightarrow T} \quad \text{(QR-Def)}$$

$$\frac{\Gamma \vdash S_1 \Rrightarrow T_1 \qquad \Gamma \vdash S_2 \Rrightarrow T_2}{\Gamma \vdash S_1 \to S_2 \Rrightarrow T_1 \to T_2} \quad \text{(QR-Arrow)}$$

$$\frac{X \notin dom(\Gamma) \qquad \Gamma, X::K_1 \vdash S_2 \Rrightarrow T_2}{\Gamma \vdash \forall X::K_1 . S_2 \Rrightarrow \forall X::K_1 . T_2} \quad \text{(QR-All)}$$

$$\frac{X \notin dom(\Gamma) \qquad \Gamma, X::K_1 \vdash S_2 \Rrightarrow T_2}{\Gamma \vdash \lambda X::K_1 . S_2 \Rrightarrow \lambda X::K_1 . T_2} \quad \text{(QR-Abs)}$$

$$\frac{\Gamma \vdash S_1 \Rrightarrow T_1 \qquad \Gamma \vdash S_2 \Rrightarrow T_2}{\Gamma \vdash S_1\ S_2 \Rrightarrow T_1\ T_2} \quad \text{(QR-App)}$$

$$\frac{X \notin dom(\Gamma) \qquad \Gamma, X::K_{11} \vdash S_{12} \Rrightarrow T_{12} \qquad \Gamma \vdash S_2 \Rrightarrow T_2}{\Gamma \vdash (\lambda X::K_{11} . S_{12})\ S_2 \Rrightarrow [X \mapsto T_2]T_{12}} \quad \text{(QR-Beta)}$$

Figure 10-5: Parallel Reduction with Definitions

equivalence by reducing both terms in any way that is convenient, be guaranteed that reduction terminates with normal forms, and then (since confluence implies uniqueness of normal forms) compare the two results for equality up to the names of bound variables.

To prove confluence we can extend the basic reduction rules to a parallel reduction relation in which function applications and definition expansions may occur in independent subterms during a single step, as shown in Figure 10-5.

For all of the reduction relations, we can define the symmetric, transitive, closure (e.g., $\Gamma \vdash S \leadsto_\delta^* T$ or $\Gamma \vdash S \Rrightarrow^* T$) to hold if with Γ constant there is a finite sequence of types beginning with S and ending with T such each pair of successive types is related.

Given this definition, we can show that reduction and parallel reduction share the same transitive closure:

10.1.4 Proposition: $\Gamma \vdash S \leadsto^* T$ if and only if $\Gamma \vdash S \Rrightarrow^* T$. □

Proof: It suffices to show that $\Gamma \vdash S \leadsto T$ implies $\Gamma \vdash S \Rrightarrow T$ and that $\Gamma \vdash S \Rrightarrow T$ implies $\Gamma \vdash S \leadsto^* T$. Both of these follow by induction on the proof of the premise. □

Before proving the confluence of parallel reduction, we begin with a series of useful properties of $F^{\omega}_{\mathtt{let}}$ and parallel reduction. Proofs in the following three propositions all follow by induction on the derivations of their first assumptions.

10.1.5 Proposition [Weakening for $F^{\omega}_{\mathtt{let}}$]: 1. If $\Gamma_1, \Gamma_3 \vdash T :: K$ and $\Gamma_1, \Gamma_2, \Gamma_3 \vdash \diamond$ then $\Gamma_1, \Gamma_2, \Gamma_3 \vdash T :: K$.

2. If $\Gamma_1, \Gamma_3 \vdash \mathtt{S} \equiv \mathtt{T} :: \mathtt{K}$ and $\Gamma_1, \Gamma_2, \Gamma_3 \vdash \diamond$ then $\Gamma_1, \Gamma_2, \Gamma_2 \vdash \mathtt{S} \equiv \mathtt{T} :: \mathtt{K}$.

□

10.1.6 PROPOSITION [SUBSTITUTION]: 1. If $\Gamma_1, \mathtt{X}::\mathtt{K}, \Gamma_2 \vdash \mathcal{J}$ for any judgment form $\mathcal{J}$ and $\Gamma_1 \vdash \mathtt{T} :: \mathtt{K}$ then then $\Gamma_1, [\mathtt{X} \mapsto \mathtt{T}]\Gamma_2 \vdash [\mathtt{X} \mapsto \mathtt{T}]\mathcal{J}$.

2. If $\Gamma_1, \mathtt{X}::\mathtt{K}=\mathtt{S}, \Gamma_2 \vdash \mathcal{J}$ for any judgment form $\mathcal{J}$ and $\Gamma_1 \vdash \mathtt{S} \equiv \mathtt{T} :: \mathtt{K}$ then then $\Gamma_1, [\mathtt{X} \mapsto \mathtt{T}]\Gamma_2 \vdash [\mathtt{X} \mapsto \mathtt{T}]\mathcal{J}$.

□

10.1.7 PROPOSITION: 1. If $\Gamma \vdash \mathtt{T} :: \mathtt{K}$ then $\Gamma \vdash \diamond$.

2. If $\Gamma \vdash \mathtt{S} \equiv \mathtt{T} :: \mathtt{K}$ then $\Gamma \vdash \mathtt{S} :: \mathtt{K}$ and $\Gamma \vdash \mathtt{T} :: \mathtt{K}$.

3. If $\Gamma \vdash \mathtt{T} :: \mathtt{K}$ then $FV(\mathtt{T}) \subseteq dom(\Gamma)$.

4. If $\Gamma_1, \Gamma_2 \vdash \diamond$ then $dom(\Gamma_1) \cap dom(\Gamma_2) = \emptyset$.

□

Next, we show that applying parallel reduction to a well-formed type yields a provably equivalent type.

10.1.8 PROPOSITION: If $\Gamma \vdash \mathtt{S} :: \mathtt{K}$ and $\Gamma \vdash \mathtt{S} \Rrightarrow \mathtt{T}$ then $\Gamma \vdash \mathtt{S} \equiv \mathtt{T} :: \mathtt{K}$ (and hence by Proposition 10.1.7 we have $\Gamma \vdash \mathtt{T} :: \mathtt{K}$). □

Proof: By induction on the proof of $\Gamma \vdash \mathtt{S} \Rrightarrow \mathtt{T}$ □

Next, using a preliminary weakening lemma, we show that parallel reduction is preserved both under substitutions of a single term, and under substitutions of a term and its reduct.

10.1.9 LEMMA [WEAKENING FOR REDUCTION]: 1. If $\Gamma_1, \Gamma_3 \vdash \mathtt{S}_1 \rightsquigarrow \mathtt{S}_2$ then $\Gamma_1, \Gamma_2, \Gamma_3 \vdash \mathtt{S}_1 \rightsquigarrow \mathtt{S}_2$.

2. If $\Gamma_1, \Gamma_3 \vdash \mathtt{S}_1 \Rrightarrow \mathtt{S}_2$ then $\Gamma_1, \Gamma_2, \Gamma_3 \vdash \mathtt{S}_1 \Rrightarrow \mathtt{S}_2$.

□

Proof: By induction on the proof of the assumed reduction, observing that the restrictions on variables for cases such as R-ABS can always be satisfied by renaming bound variables. □

10.1.10 LEMMA: If $\Gamma \vdash \mathtt{S}_1 \Rrightarrow \mathtt{S}_2$ then $\Gamma \vdash [\mathtt{Y} \mapsto \mathtt{S}_1]\mathtt{T} \Rrightarrow [\mathtt{Y} \mapsto \mathtt{S}_2]\mathtt{T}$. □

Proof: By induction on the structure of $\mathtt{T}$. □

10.1.11 LEMMA: If $\Gamma_1 \vdash \mathtt{S}_1 \Rrightarrow \mathtt{S}_2$ and $\Gamma_1, \mathtt{Y}::\mathtt{K}, \Gamma_2 \vdash \mathtt{T}_1 \Rrightarrow \mathtt{T}_2$ where $\Gamma_1, \mathtt{Y}::\mathtt{K}, \Gamma_2 \vdash \diamond$ then $\Gamma_1, [\mathtt{Y} \mapsto \mathtt{S}_2]\Gamma_2 \vdash [\mathtt{Y} \mapsto \mathtt{S}_1]\mathtt{T}_1 \Rrightarrow [\mathtt{Y} \mapsto \mathtt{S}_2]\mathtt{T}_2$. □

Proof: We proceed by induction on the derivation of the assumption $\Gamma_1, \mathtt{Y}::\mathtt{K}, \Gamma_2 \vdash \mathtt{T}_1 \Rrightarrow \mathtt{T}_2$, and cases on the last rule used.

Case QR-REFL: $\mathtt{T}_1 = \mathtt{T}_2$.

If the derivation is simply a use QR-REFL, then the result follows by Lemma 10.1.10 and repeated use of Lemma 10.1.9 to obtain $\Gamma_1, [\mathtt{Y} \mapsto \mathtt{S}_2]\Gamma_2 \vdash \mathtt{S}_1 \Rrightarrow \mathtt{S}_2$.

Case Q-DEF: $\mathtt{T}_1 = \mathtt{X}$ and $\mathtt{T}_2$ is the definition of $\mathtt{X}$ in the context.

There are two subcases. $\mathtt{X}$ cannot be equal to $\mathtt{Y}$ (since $\mathtt{X}$ has a definition and by assumption $\mathtt{Y}$ does not) and so $\mathtt{X}$ is defined in either Γ_1 or Γ_2.

If $\mathtt{X}$ is defined in Γ_1 then neither it nor (since $\mathtt{Y} \notin \mathit{dom}(\Gamma_1)$) its definition $\mathtt{T}_2$ can contain $\mathtt{Y}$. Then $[\mathtt{Y} \mapsto \mathtt{S}_2]\mathtt{T}_2 = \mathtt{T}_2$, so that $\Gamma_1, [\mathtt{Y} \mapsto \mathtt{S}_2]\Gamma_2 \vdash [\mathtt{Y} \mapsto \mathtt{S}_1]\mathtt{T}_1 \Rrightarrow [\mathtt{Y} \mapsto \mathtt{S}_2]\mathtt{T}_2$ is still an instance of definitional expansion.

Otherwise, $\mathtt{X}$ is defined in Γ_2. In this case, its definition in $\Gamma_1, [\mathtt{Y} \mapsto \mathtt{S}_2]\Gamma_2$ is $[\mathtt{Y} \mapsto \mathtt{S}_2]\mathtt{T}_2$, and hence $\Gamma_1, [\mathtt{Y} \mapsto \mathtt{S}_2]\Gamma_2 \vdash [\mathtt{Y} \mapsto \mathtt{S}_1]\mathtt{T}_1 \Rrightarrow [\mathtt{Y} \mapsto \mathtt{S}_2]\mathtt{T}_2$ is also an instance of definitional expansion.

The remaining cases follow as for F^ω. □

At this point we can show the confluence of the parallel reduction relation.

10.1.12 PROPOSITION [CONFLUENCE OF PARALLEL REDUCTION]: If $\Gamma \vdash \mathtt{S} \Rrightarrow \mathtt{T}_1$ and $\Gamma \vdash \mathtt{S} \Rrightarrow \mathtt{T}_2$ then there exists $\mathtt{U}$ such that $\Gamma \vdash \mathtt{T}_1 \Rrightarrow \mathtt{U}$ and $\Gamma \vdash \mathtt{T}_2 \Rrightarrow \mathtt{U}$. □

Proof: Essentially the same as the proof for confluence of parallel reduction F^ω in TAPL §30.3. There is one new case: $\mathtt{S}$ is a variable $\mathtt{X}$ with a definition $\mathtt{T}$ in the context. However, the only possible reducts would be $\Gamma \vdash \mathtt{X} \Rrightarrow \mathtt{X}$ and $\Gamma \vdash \mathtt{X} \Rrightarrow \mathtt{T}$, which have $\mathtt{T}$ as a common reduct. □

Finally, we show that parallel reduction characterizes provable equivalence.

10.1.13 PROPOSITION: If $\Gamma \vdash \mathtt{S} :: \mathtt{K}$ and $\Gamma \vdash \mathtt{T} :: \mathtt{K}$ then $\Gamma \vdash \mathtt{S} \equiv \mathtt{T} :: \mathtt{K}$ if and only if there exists $\mathtt{U}$ such that $\Gamma \vdash \mathtt{S} \Rrightarrow^* \mathtt{U}$ and $\Gamma \vdash \mathtt{T} \Rrightarrow^* \mathtt{U}$. □

Proof: The "if" direction follows easily. By repeated use of Proposition 10.1.8 and transitivity of equivalence we have $\Gamma \vdash \mathtt{S} \equiv \mathtt{U} :: \mathtt{K}$ and $\Gamma \vdash \mathtt{T} \equiv \mathtt{U} :: \mathtt{K}$, so by symmetry and transitivity we have $\Gamma \vdash \mathtt{S} \equiv \mathtt{T} :: \mathtt{K}$.

The "only if" direction follows by induction on the proof that $\Gamma \vdash \mathtt{S} \equiv \mathtt{T} :: \mathtt{K}$, analogous to the proof of Proposition 30.3.10 in TAPL §30.3. □

10.1.14 COROLLARY: If $\Gamma \vdash \mathtt{S} :: \mathtt{K}$ and $\Gamma \vdash \mathtt{T} :: \mathtt{K}$ then $\Gamma \vdash \mathtt{S} \equiv \mathtt{T} :: \mathtt{K}$ if and only if there exists $\mathtt{U}$ such that $\Gamma \vdash \mathtt{S} \rightsquigarrow^* \mathtt{U}$ and $\Gamma \vdash \mathtt{T} \rightsquigarrow^* \mathtt{U}$. □

Consequently, we have shown that type equivalence can be determined by finding a common normal form, since by conflucence normal forms are unique. We therefore turn our attention to showing that reduction is strongly

$$
\begin{array}{lcl}
\delta size(\Gamma,\ \mathtt{Int}) & \stackrel{\text{def}}{=} & 0 \\
\delta size((\Gamma_1, \mathtt{X::K}), \Gamma_2),\ \mathtt{X}) & \stackrel{\text{def}}{=} & 0 \\
\delta size((\Gamma_1, \mathtt{X::K=T}, \Gamma_2),\ \mathtt{X}) & \stackrel{\text{def}}{=} & 1 + \delta size(\Gamma_1,\ \mathtt{T}) \\
\delta size(\Gamma,\ \mathtt{T_1 \to T_2}) & \stackrel{\text{def}}{=} & \delta size(\Gamma,\ \mathtt{T_1}) + \delta size(\Gamma,\ \mathtt{T_2}) \\
\delta size(\Gamma,\ \mathtt{T_1\ T_2}) & \stackrel{\text{def}}{=} & \delta size(\Gamma,\ \mathtt{T_1}) + \delta size(\Gamma,\ \mathtt{T_2}) \\
\delta size(\Gamma,\ \lambda \mathtt{X::K_1\ .\ T_2}) & \stackrel{\text{def}}{=} & \delta size(\Gamma,\ \mathtt{T_2}) \\
\delta size(\Gamma,\ \forall \mathtt{X::K_1\ .\ T_2}) & \stackrel{\text{def}}{=} & \delta size(\Gamma,\ \mathtt{T_2})
\end{array}
$$

Figure 10-6: Bounding Definition Reductions

normalizing, so that we can choose any convenient reduction sequence to find a normal form.

First, delta-reduction by itself is strongly normalizing; intuitively, well-formed contexts forbid circular definitions, and so given any type there can be only finitely many definitions to expand. More formally, we can define a measure $\delta size(\Gamma,\ \mathtt{T})$ which is an upper bound on the number of definition-expansion steps one can take starting at $\mathtt{T}$. The definition of $\delta size(\Gamma,\ \mathtt{T})$ appears in Figure 10-6.

10.1.15 PROPOSITION: 1. If $\Gamma \vdash \mathtt{T} :: \mathtt{K}$ then $\delta size(\Gamma,\ \mathtt{T})$ is well-defined and ≥ 0.

2. If $\Gamma_1 \vdash \mathtt{T} :: \mathtt{K}$ and $\Gamma_1, \Gamma_2 \vdash \diamond$ then $\delta size(\Gamma_1,\ \mathtt{T}) = \delta size((\Gamma_1, \Gamma_2),\ \mathtt{T})$.

3. If $\Gamma \vdash \mathtt{S} :: \mathtt{K}$ and $\Gamma \vdash \mathtt{S} \leadsto_\delta \mathtt{T}$ then $\delta size(\Gamma,\ \mathtt{S}) > \delta size(\Gamma,\ \mathtt{T})$.

□

Proof: 1. By induction on the kinding proof.

2. By induction on the kinding proof.

3. By induction on the proof of $\Gamma \vdash \mathtt{S} \Rrightarrow_\delta \mathtt{T}$, using part 2 for the case where $\mathtt{S}$ is a variable and $\mathtt{T}$ is its definition.

□

10.1.16 COROLLARY [STRONG NORMALIZATION OF DELTA-REDUCTION]: If $\Gamma \vdash \mathtt{T_1} :: \mathtt{K}$ then there is no infinite sequence $\Gamma \vdash \mathtt{T_1} \leadsto_\delta \mathtt{T_2} \leadsto_\delta \mathtt{T_3} \leadsto_\delta \cdots$. □

To show strong normalization of full reduction, we use an auxiliary functions that immediately computes the delta-normal form of a term by expanding all definitions. Figure 10-7 defines $|\mathtt{T}|_\Gamma$, which is the result of expanding away all the definitions in Γ used by variables in $\mathtt{T}$, and defines $|\Gamma|$, which is the result of expanding away and erasing all definitions appearing in Γ.

$$
\begin{array}{rcl}
|\cdot| & \stackrel{\text{def}}{=} & \cdot \\
|\Gamma, x:T| & \stackrel{\text{def}}{=} & |\Gamma|, x \in |T|_\Gamma \\
|\Gamma, X::K| & \stackrel{\text{def}}{=} & |\Gamma|, X \in K \\
|\Gamma, X::K=T| & \stackrel{\text{def}}{=} & |\Gamma|, X \in K \\
\\
|\mathtt{Int}|_\Gamma & \stackrel{\text{def}}{=} & \mathtt{Int} \\
|X|_\Gamma & \stackrel{\text{def}}{=} & \begin{cases} |T|_{\Gamma_1} & \text{if } \Gamma = \Gamma_1, X::K=T, \Gamma_2 \\ X & \text{if } \Gamma = \Gamma_1, X::K, \Gamma_2 \end{cases} \\
|T_1 \to T_2|_\Gamma & \stackrel{\text{def}}{=} & (|T_1|_\Gamma) \to (|T_2|_\Gamma) \\
|\lambda X::K_1 . T_2|_\Gamma & \stackrel{\text{def}}{=} & \lambda X::K_1 . (|T_2|_{\Gamma, X::K_1}) \\
|\forall X::K_1 . T_2|_\Gamma & \stackrel{\text{def}}{=} & \forall X::K_1 . (|T_2|_{\Gamma, X::K_1}) \\
|T_1\ T_2|_\Gamma & \stackrel{\text{def}}{=} & (|T_1|_\Gamma)\ (|T_2|_\Gamma)
\end{array}
$$

Figure 10-7: Full Definition Expansion for Typing Contests and Types

10.1.17 PROPOSITION: If $\Gamma \vdash T :: K$ then $|T|_\Gamma$ is well defined and $\Gamma \vdash T \rightsquigarrow_\delta^* |T|_\Gamma$ (and hence by Proposition 10.1.8 we have $\Gamma \vdash |T|_\Gamma :: K$). □

Proof: By induction on the first given derivation. □

10.1.18 PROPOSITION [SUBSTITUTION FOR EXPANSIONS]: If $\Gamma, X::K_{11} \vdash S_{12} :: K_{12}$ and $\Gamma \vdash S_2 :: K_{11}$ then $|[X \mapsto S_2]S_{12}|_\Gamma = [X \mapsto (|S_2|_\Gamma)](|S_{12}|_{\Gamma, X::K_{11}})$. □

Proof: By induction on the structure of S_{12}. □

10.1.19 PROPOSITION: 1. If $\Gamma \vdash S :: K$ and $\Gamma \vdash S \rightsquigarrow_\delta T$ then $|S|_\Gamma = |T|_\Gamma$.

2. If $\Gamma \vdash S :: K$ and $\Gamma \vdash S \rightsquigarrow_\beta T$ then $|S|_\Gamma \rightsquigarrow_\beta |T|_\Gamma$.

□

Proof: Each case follows by induction on the proof that S reduces to T. □

Now we can relate well-typedness in $F^\omega_{\mathtt{let}}$ with well-typedness in F^ω. To distinguish the two systems, we use the symbol $\vdash^\omega$ to denote a proof not using K-TVARDEF and without definitions in any typing contexts, and hence a proof that could be performed in ordinary F^ω.

10.1.20 PROPOSITION: 1. If $\Gamma \vdash \diamond$ then $|\Gamma| \vdash^\omega \diamond$.

2. If $\Gamma \vdash T :: K$ then $|\Gamma| \vdash^\omega |T|_\Gamma :: K$.

3. If $\Gamma \vdash \mathtt{S} \equiv \mathtt{T} :: \mathtt{K}$ then $|\Gamma| \vdash^{\omega} |\mathtt{S}|_{\Gamma} \equiv |\mathtt{T}|_{\Gamma} :: \mathtt{K}$.

□

Proof: By simultaneous induction on the proof of the assumptions. □

Thus we can show that general reduction is strongly normalizing:

10.1.21 PROPOSITION [STRONG NORMALIZATION FOR REDUCTION]: If $\Gamma \vdash \mathtt{T}_0 :: \mathtt{K}$ then there is no infinite sequence $\Gamma \vdash \mathtt{T}_0 \rightsquigarrow \mathtt{T}_1 \rightsquigarrow \mathtt{T}_2 \rightsquigarrow \cdots$. □

Proof: Suppose such a sequence existed. By the strong normalization of delta-reduction, infinitely many of these steps must be $\rightsquigarrow_{\beta}$. Therefore, by Proposition 10.1.19 the sequence $|\mathtt{T}_0|_{\Gamma}$, $|\mathtt{T}_1|_{\Gamma}$, ... must be an infinite sequence of $\rightsquigarrow_{\beta}$ steps (interspersed with equalities where the original sequence used $\rightsquigarrow_{\delta}$). But by Proposition 10.1.20 we know that $\Gamma \vdash^{\omega} |\mathtt{T}_0|_{\Gamma} :: \mathtt{K}$ and so this would be an infinite beta-reduction in pure F^{ω}, a contradiction. □

10.1.22 COROLLARY: Equivalence of well-formed types is decidable. □

Improving Efficiency

If explicit definitions are being used for the purposes of keeping types small, finding normal forms can be an expensive way to determine type equivalence. For example, if we have definitions

```
Pair = λY::*.(Y × Y)
List = λY::*. (μX. <nil:Unit, cons:{Nat,X}>)
```

we would like to be able to determine that `List(List(Pair(Int)))` and `List(List(Int × Int))` are equivalent without expanding them to their common (but noticeably larger) normal form. Although for arbitrary types we may not be able to avoid doing the work of full normalization, in practice code reuses the same defined names, and so heuristics to avoid full normalization can often help.

One approach to avoiding full normalization involves simultaneous reduction and comparison of the types using weak head reduction, as discussed in Chapter 2. Instead of fully normalizing the types, only the "outermost" applications or definitions are reduced. If the resulting types turn out to have the same shape, then corresponding sub-components of the types can be recursively compared. Conversely, if the two types are weak head normalized but fail to have the same structure then the types are not equivalent and the algorithm can short-circuit and report inequivalence.

Figure 10-8 presents an algorithmic version of equivalence in this fashion. Weak head reduction and normalization are deterministic (i.e., given Γ and

S there is at most one T such that $\Gamma \vdash S \Downarrow T$) and for any structural type equivalence judgment there is at at most one rule that could be used with that conclusion, and the premises are all determined by the conclusion. Hence, given a desired judgment, proof search is deterministic.

The weak head normalization relation $\Gamma \vdash T_1 \Downarrow T_n$ simply requires there be a finite sequence of types $T_1, \ldots, T_n$ with $n \geq 1$ such that each weak head reduces to the next, and such that T_n is weak head normal.

The structural equivalence judgment $\Gamma \vdash T_1 \leftrightarrow T_2$ implements equivalence for head-normal types only; T_1 and T_2 must have the same shape and their subcomponents must be algorithmically equivalent.

Finally, the algorithmic type equivalence judgment $\Gamma \vdash T_1 \Leftrightarrow T_2$ holds if the weak head normal forms of T_1 and T_2 are structurally equivalent. For well-formed types, this judgment corresponds to the declarative specification of equivalence:

10.1.23 THEOREM: If $\Gamma \vdash T_1 :: K$ and $\Gamma \vdash T_2 :: K$ then $\Gamma \vdash T_1 \equiv T_2 :: K$ if and only if $\Gamma \vdash\!\blacktriangleright T_1 \Leftrightarrow T_2$. Furthermore, if both types have kind K then the judgment $\Gamma \vdash\!\blacktriangleright T_1 \Leftrightarrow T_2$ is decidable (i.e., deterministic proof search process must always terminate with success or failure). □

10.1.24 EXERCISE [★★★]: Prove Theorem 10.1.23. □

This approach could be further refined; an implementation might check for α-equivalence even during head normalizations. Thus, a request to compare `List(T1)` with `List(T2)` could directly check whether T_1 and T_2 are equivalent rather than expanding the definition of `List`.

One must still be careful in trying to optimize, though, since the addition of definitions alters usual properties of type equivalence. For example, in ordinary F^ω, the equivalence $X\ T_1 \equiv X\ T_2$ holds if and only if $T_1 \equiv T_2$. In the system with definitions, however, we can prove

```
X::(*⇒*)=(λY::*.Int) ⊢ X Int ≡ X Bool
```

despite `Int` and `Bool` being inequivalent, because both applications are provably equal to `Int`. Therefore, although comparing $X\ T_1$ with $X\ T_2$ by showing that T_1 and T_2 are equivalent may often be faster than expanding out a definition for X, if the arguments are inequivalent we may need to consider the expansion anyway.[3]

3. The presence of definitions has consequences for unification as well (e.g., in implementations of ML type inference): the most general substitution required to make $X\ T_1$ and $X\ T_2$ equal may not require T_1 and T_2 to unify.

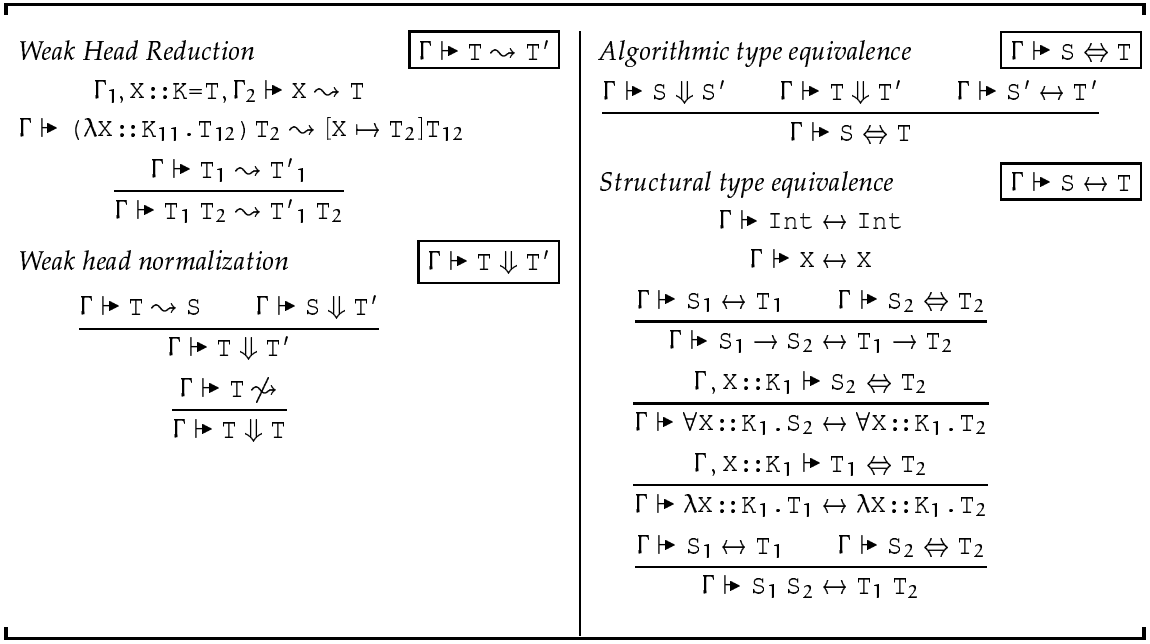


Figure 10-8: Algorithmic Equivalence with Definitions

One might think to special-case just variables like `X` above whose definitions completely ignore their arguments, but similar behavior can arise more generally:

10.1.25 EXERCISE [★★, RECOMMENDED]: Find a typing context and pairwise inequivalent T_1, T_2, and T_3 such that $X\ T_1 \equiv X\ T_2$ but $X\ T_2 \not\equiv X\ T_3$ (and so `X` cannot completely ignore its argument). □

If the simultaneous comparison process finds no short-cuts, it will do work equivalent to entirely normalizing and comparing the two types. It may still be more memory-efficient, however. Full normal forms are not explicitly computed and stored; when two subcomponents of the types are found to be equal their reduced forms can be immediately discarded, freeing up memory for the rest of the comparison.

10.1.26 EXERCISE [★★★, ↛]: Extend the `fullomega` checker to include definitions, and make type equivalence checking as efficient as possible. □

10.2 Definitions in Modules

In the presence of modules, type definitions are often permitted to appear within interfaces. The most interesting aspect of the theory of ML-style module systems involves tracking information about the types involved. As discussed earlier, type components in modules may have definitions that are not syntactically apparent.

One line of research in formalizing the type theory of ML-like module systems (as discussed in Chapter 9) led to the calculi known as Translucent Sums (Harper and Lillibridge, 1994) and Manifest Types (Leroy, 1994b). These similar systems largely correspond to the module systems of Revised Standard ML (Milner, Tofte, Harper, and MacQueen, 1997b) and (with some extensions (Leroy, 1995b)) of Objective Caml.

The Language $\lambda^{(\!|\,|\!)}$

Figure 10-9 defines a minimalist language $\lambda^{(\!|\,|\!)}$ in this tradition with second-class higher-order modules (i.e., modules are not first-class values able to be passed to term-level functions, and similarly interfaces are not types). Despite being vastly simpler than any module system usable in practice, $\lambda^{(\!|\,|\!)}$ is still complex enough to demonstrate many of the issues discussed in Chapter 9.

In ML, modules can contain any combination of named value, type, and sub-module components in any order. $\lambda^{(\!|\,|\!)}$ instead builds up modules starting with two primitives: modules that contain a single unnamed term, written $(\!|\mathtt{t}|\!)$, and modules that contain a single unnamed type, written $(\!|\mathtt{T::K}|\!)$. The contents of primitive modules can be extracted by using the `!` operator.

For each sort of module, there are corresponding interfaces. The interface $(\!|\mathtt{T}|\!)$ classifies primitive modules containing a value of type `t`, while the *opaque* interface $(\!|\mathtt{K}|\!)$ classifies modules containing a type of kind `K`. Modules containing types may also have a *transparent* interface $(\!|\mathtt{K=T}|\!)$ if they contain just the type `T` (or a provably equivalent type) of kind `K`.

More complex modules can be created by using the module-level pairing operator $(\!|\cdot,\cdot|\!)$. The projection operators `.1` and `.2` then access the sub-modules within such a pair.

Interfaces of module-pairs are given by specifying the interfaces of the two submodules. However, in order to permit specifications such as "a module containing an abstract type and a term of that type", these interfaces are allowed to be dependent, similar to the dependent types of Chapter 4. The interface $\Sigma\mathtt{m}:\mathtt{I}_1.\mathtt{I}_2$ classifies pairs whose first component is a module of interface $\mathtt{I}_1$ and whose second component is a module of interface $\mathtt{I}_2$, where

the latter interface may refer to the contents of the first component by the name m.[4]

For example, consider again the module N, defined by

```
module N = struct
              type t = int
              let  x = 3
           end.
```

This is in essence a module containing a single type and a single term, and hence can be encoded into $\lambda^{()}$ as

```
⟪ ⟪Int::*⟫, ⟪3⟫ ⟫.
```

This module satisfies the very precise interface

```
Σm:⟪*=Int⟫. ⟪Int⟫,
```

which describes it as containing the type Int and an integer value. This interface is completely equivalent to

```
Σm:⟪*=Int⟫. ⟪!m⟫,
```

which is an interface satisfied by modules containing the type Int and a value of that same type.

The encoding of n further matches the strictly more abstract (less informative) interface

```
Σm:⟪*⟫. ⟪!m⟫
```

specifying only that the module contains some type and a value of that type.

Parameterized modules, or functors, are simply module-level functions. Thus, for example, the Diag functor defined above can be desugared into

```
λm:(Σm′:⟪*⟫. ⟪!m′⟫).
                ⟪ ⟪!m.1 × !m.1::*⟫, ⟪{!m.2,!m.2}⟫ ⟫
```

The argument of this functor is required to be a module pair containing a type and a value of that type (in sub-modules); it then returns a module pair containing a pair type and a pair value (in sub-modules). By convention, first and second projections bind most tightly, followed by applications, !, and finally the binary operators such as × . Thus the type being returned by this functor is equivalent to (!(m.1)) × (!(m.1)).

4. In the vocabulary of Chapter 9, m is an internal name for the first component of the module pair, while the external names of the components are always 1 and 2.

Interfaces for functors are also dependent, because the types in the functor's result may depend upon the types contained in the functor's argument value. The interface $\Pi \mathtt{m}:\mathtt{I}_1.\mathtt{I}_2$ classifies functors that require an argument satisfying $\mathtt{I}_1$ and which return a result satisfying $\mathtt{I}_2$, where the types within $\mathtt{I}_2$ can refer to the argument value as `m`. Thus, one possible most-precise interface describing the desugared `Diag` functor would be

```
Πm:(Σm′:⦇*⦈. ⦇!m′⦈).
          (Σm″:⦇*=!m.1 × !m.1⦈. ⦇!m″⦈).
```

In both $\Sigma \mathtt{m}:\mathtt{I}_1.\mathtt{I}_2$ and $\Pi \mathtt{m}:\mathtt{I}_1.\mathtt{I}_2$ the variable `m` is bound in $\mathtt{I}_2$. In those cases where `m` does not appear in $\mathtt{I}_2$ we can omit mention of the dependent variable, writing non-dependent pair interfaces as $\mathtt{I}_1 \times \mathtt{I}_2$, and non-dependent functor interfaces as $\mathtt{I}_1 \to \mathtt{I}_2$.

The last sort of module expression is the sealing operation `M :> I`, corresponding to the generative or strong sealing of Chapter 9. This takes a module `M` and hides all information about that module except what is explicitly mentioned in the interface `I`. This is used for information-hiding purposes, in order to create abstract (opaque) types.

The syntax separates out a syntactic set of modules which are *determinate* (have type components that are predictable at compile-time) in the sense of Chapter 9. Only these are allowed in types. Note that some non-values such as `⦇⦇Int::*⦈, ⦇Bool::*⦈⦈.1` are considered syntactically determinate and so can appear within types. This results from the need for modules such as `m.1` to appear within types and the simultaneous desire to keep the syntax of types closed under substitutions replacing module variables by fully-evaluated module values such as `⦇⦇Int::*⦈, ⦇Bool::*⦈⦈`.[5]

Typing and Evaluation Rules

The static semantics of $\lambda^{()}$ appears in Figures 10-9 and 10-10. The judgment $\Gamma \vdash \mathtt{I}$ defines the well-formedness of interfaces, which requires all types in the interface be well-formed. More interesting is the subinterface relation $\Gamma \vdash \mathtt{I} <: \mathtt{I}'$, which is nontrivial even though $\lambda^{()}$ has no subtyping relation. The key rule here is SI-FORGET, which specifies that a module with a transparent interface can be used as if it had the corresponding opaque interface; type definitions in interfaces may be neglected when not relevant. Otherwise, subtyping for interfaces describing terms and for transparent interfaces coincides with interface equivalence, while rules SI-PI and SI-SIGMA correspond

5. An alternative would be to redefine substitution to include reduction of any new projections introduced into types (Lillibridge, 1997).

$\lambda^{(\!|\,|\!)}$ *extends* F^{ω}

Syntax

Γ	::=	...	*contexts:*
		$\mathtt{m}:\mathtt{I}$	*module variable*
$\mathtt{W}$	::=		*determinate modules:*
		$\mathtt{m}$	*variable*
		$(\!|\mathtt{v}|\!)$	*term module*
		$(\!|\mathtt{T}::\mathtt{K}|\!)$	*type module*
		$(\!|\mathtt{W},\mathtt{W}|\!)$	*pairing*
		$\mathtt{W}.1$	*first projection*
		$\mathtt{W}.2$	*second projection*
		$\lambda\mathtt{m}:\mathtt{I}.\mathtt{M}$	*functor*
$\mathtt{M}$	::=		*modules:*
		$\mathtt{W}$	*determinates*
		$(\!|\mathtt{t}|\!)$	*term module*
		$(\!|\mathtt{M},\mathtt{M}|\!)$	*pairing*
		$\mathtt{M}.1$	*first projection*
		$\mathtt{M}.2$	*second projection*
		$\mathtt{M}\,\mathtt{M}$	*application*
		$\mathtt{M}$:> $\mathtt{I}$	*interface ascription*
		let $\mathtt{m}$=$\mathtt{M}$ in $\mathtt{M}$	*local module*
$\mathtt{I}$	::=		*interfaces:*
		$(\!|\mathtt{T}|\!)$	*term interface*
		$(\!|\mathtt{K}|\!)$	*opaque interface*
		$(\!|\mathtt{K}=\mathtt{T}|\!)$	*transparent interface*
		$\Sigma\mathtt{m}:\mathtt{I}_1.\mathtt{I}_2$	*pair interface*
		$\Pi\mathtt{m}:\mathtt{I}_1.\mathtt{I}_2$	*functor interface*
$\mathtt{t}$	::=	...	*terms:*
		$!\mathtt{M}$	*module projection*
		let $\mathtt{m}$=$\mathtt{M}$ in $\mathtt{t}$	*local module*
$\mathtt{T}$	::=	...	*types:*
		$!\mathtt{W}$	*module projection*

Derived Forms

$\mathtt{I}_1 \times \mathtt{I}_2 \stackrel{\text{def}}{=} \Sigma\mathtt{m}:\mathtt{I}_1.\mathtt{I}_2 \quad (\mathtt{m} \notin FV(\mathtt{I}_2))$

$\mathtt{I}_1 \to \mathtt{I}_2 \stackrel{\text{def}}{=} \Pi\mathtt{m}:\mathtt{I}_1.\mathtt{I}_2 \quad (\mathtt{m} \notin FV(\mathtt{I}_2))$

Interface Validity $\boxed{\Gamma \vdash \mathtt{I}}$

$$\frac{\Gamma \vdash \mathtt{T} :: *}{\Gamma \vdash (\!|\mathtt{T}|\!)} \quad \text{(I-Term)}$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash (\!|\mathtt{K}|\!)} \quad \text{(I-Opaque)}$$

$$\frac{\Gamma \vdash \mathtt{T} :: \mathtt{K}}{\Gamma \vdash (\!|\mathtt{K}=\mathtt{T}|\!)} \quad \text{(I-Transp)}$$

$$\frac{\Gamma \vdash \mathtt{I}_1 \qquad \Gamma, \mathtt{m}:\mathtt{I}_1 \vdash \mathtt{I}_2}{\Gamma \vdash \Sigma\mathtt{m}:\mathtt{I}_1.\mathtt{I}_2} \quad \text{(I-Pair)}$$

$$\frac{\Gamma \vdash \mathtt{I}_1 \qquad \Gamma, \mathtt{m}:\mathtt{I}_1 \vdash \mathtt{I}_2}{\Gamma \vdash \Pi\mathtt{m}:\mathtt{I}_1.\mathtt{I}_2} \quad \text{(I-Functor)}$$

Subinterface $\boxed{\Gamma \vdash \mathtt{I} <: \mathtt{I}'}$

$$\frac{\Gamma \vdash \mathtt{T} \equiv \mathtt{T}'}{\Gamma \vdash (\!|\mathtt{T}|\!) <: (\!|\mathtt{T}'|\!)} \quad \text{(SI-Term)}$$

$$\frac{\Gamma \vdash \mathtt{T} \equiv \mathtt{T}'}{\Gamma \vdash (\!|\mathtt{K}=\mathtt{T}|\!) <: (\!|\mathtt{K}=\mathtt{T}'|\!)} \quad \text{(SI-Transp)}$$

$$\Gamma \vdash (\!|\mathtt{K}=\mathtt{T}|\!) <: (\!|\mathtt{K}|\!) \quad \text{(SI-Forget)}$$

$$\Gamma \vdash (\!|\mathtt{K}|\!) <: (\!|\mathtt{K}|\!) \quad \text{(SI-Opaque)}$$

$$\frac{\Gamma \vdash \mathtt{I}_{21} <: \mathtt{I}_{11} \qquad \Gamma, \mathtt{M}::\mathtt{I}_{21} \vdash \mathtt{I}_{12} <: \mathtt{I}_{22}}{\Gamma \vdash \Pi\mathtt{m}:\mathtt{I}_{11}.\mathtt{I}_{12} <: \Pi\mathtt{m}:\mathtt{I}_{21}.\mathtt{I}_{22}} \quad \text{(SI-Pi)}$$

$$\frac{\Gamma \vdash \mathtt{I}_{11} <: \mathtt{I}_{21} \qquad \Gamma, \mathtt{M}::\mathtt{I}_{11} \vdash \mathtt{I}_{12} <: \mathtt{I}_{22}}{\Gamma \vdash \Sigma\mathtt{m}:\mathtt{I}_{11}.\mathtt{I}_{12} <: \Sigma\mathtt{m}:\mathtt{I}_{21}.\mathtt{I}_{22}} \quad \text{(SI-Sigma)}$$

Figure 10-9: Syntax and Static Semantics for $\lambda^{(\!|\,|\!)}$

Kinding $\boxed{\Gamma \vdash T :: K}$

$$\frac{\Gamma \vdash W : (\!|K|\!)}{\Gamma \vdash !W :: K} \quad \text{(K-MPROJ)}$$

Type Equivalence $\boxed{\Gamma \vdash S \equiv T}$

$$\frac{\Gamma \vdash W : (\!|K=T|\!)}{\Gamma \vdash !W \equiv T} \quad \text{(Q-MPROJ)}$$

Well-Formed Modules $\boxed{\Gamma \vdash M : I}$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash (\!|t|\!) : (\!|T|\!)} \quad \text{(M-TERM)}$$

$$\frac{\Gamma \vdash T :: K}{\Gamma \vdash (\!|T::K|\!) : (\!|K=T|\!)} \quad \text{(M-TYPE)}$$

$$\frac{\Gamma \vdash M : I' \qquad \Gamma \vdash I' <: I}{\Gamma \vdash M : I} \quad \text{(M-SUB)}$$

$$\frac{\Gamma \vdash \diamond \qquad m{:}I \in \Gamma}{\Gamma \vdash m : I} \quad \text{(M-VAR)}$$

$$\frac{\Gamma, m{:}I_1 \vdash M_2 : I_2}{\Gamma \vdash \lambda m{:}I_1.M_2 : \Pi m{:}I_1.I_2} \quad \text{(M-ABS)}$$

$$\frac{\Gamma \vdash M_1 : I_1 \qquad \Gamma \vdash M_2 : I_2}{\Gamma \vdash (\!|M_1, M_2|\!) : I_1 \times I_2} \quad \text{(M-PAIR)}$$

$$\frac{\Gamma \vdash M : \Sigma m{:}I_1.I_2}{\Gamma \vdash M.1 : I_1} \quad \text{(M-FST)}$$

$$\frac{\Gamma \vdash M : I_1 \times I_2}{\Gamma \vdash M.2 : I_2} \quad \text{(M-SND)}$$

$$\frac{\Gamma \vdash M_1 : I_1 \to I_2 \qquad \Gamma \vdash M_2 : I_1}{\Gamma \vdash M_1\, M_2 : I_2} \quad \text{(M-APPLY)}$$

$$\frac{\Gamma \vdash W : (\!|K|\!)}{\Gamma \vdash W : (\!|K=!W|\!)} \quad \text{(M-SELF)}$$

$$\frac{\Gamma \vdash M : \Sigma m{:}I_1.I_2 \qquad \Gamma \vdash M.1 : I'_1}{\Gamma \vdash M : \Sigma m{:}I'_1.I_2} \quad \text{(M-SELF-PAIR1)}$$

$$\frac{\Gamma \vdash M : \Sigma m{:}I_1.I_2 \qquad \Gamma \vdash M.2 : I'_2}{\Gamma \vdash M : I_1 \times I'_2} \quad \text{(M-SELF-PAIR2)}$$

$$\frac{\Gamma \vdash M : I}{\Gamma \vdash (M :> I) : I} \quad \text{(M-SEAL)}$$

$$\frac{\Gamma \vdash M_1 : I_1 \qquad \Gamma, m{:}I_1 \vdash M_2 : I \qquad \Gamma \vdash I}{\Gamma \vdash \mathtt{let}\ m{=}M_1\ \mathtt{in}\ M_2 : I} \quad \text{(M-MOD-LET)}$$

Typing $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \vdash M : (\!|T|\!)}{\Gamma \vdash !M : T} \quad \text{(T-MOD-PROJ)}$$

$$\frac{\Gamma \vdash M : I \qquad \Gamma, m{:}I \vdash t : T \qquad \Gamma \vdash T}{\Gamma \vdash \mathtt{let}\ m{=}M\ \mathtt{in}\ t : T} \quad \text{(T-MOD-LET)}$$

Context Validity $\boxed{\Gamma \vdash \diamond}$

$$\frac{\Gamma \vdash I \qquad m \notin dom(\Gamma)}{\Gamma, m{:}I \vdash \diamond} \quad \text{(CTX-DEF)}$$

Figure 10-10: Static Semantics for $\lambda^{(\!|\,|\!)}$, Continued

Module Evaluation $\boxed{M \longrightarrow M'}$

$$\frac{t \longrightarrow t'}{\langle\!|t|\!\rangle \longrightarrow \langle\!|t'|\!\rangle} \quad \text{(E-TERM)}$$

$$\frac{M \longrightarrow M'}{!M \longrightarrow !M'} \quad \text{(E-MPROJ)}$$

$$!\langle\!|v|\!\rangle \longrightarrow v \quad \text{(E-MPROJV)}$$

$$\frac{M_1 \longrightarrow M_1'}{\langle\!|M_1, M_2|\!\rangle \longrightarrow \langle\!|M_1', M_2|\!\rangle} \quad \text{(E-MPAIR1)}$$

$$\frac{M_2 \longrightarrow M_2'}{\langle\!|W_1, M_2|\!\rangle \longrightarrow \langle\!|W_1, M_2|\!\rangle} \quad \text{(E-MPAIR2)}$$

$$\langle\!|W_1, W_2|\!\rangle.1 \longrightarrow W_1 \quad \text{(E-MPAIRBETA1)}$$

$$\langle\!|W_1, W_2|\!\rangle.2 \longrightarrow W_2 \quad \text{(E-MPAIRBETA2)}$$

$$M_1 :> I_2 \longrightarrow M_1 \quad \text{(E-MSEAL)}$$

$$\frac{M_1 \longrightarrow M'_1}{M_1\, M_2 \longrightarrow M'_1\, M_2} \quad \text{(E-MAPP1)}$$

$$\frac{M_2 \longrightarrow M'_2}{W\, M_2 \longrightarrow W\, M'_2} \quad \text{(E-MAPP2)}$$

$$(\lambda m{:}I_{11}.M_{12})\, W_2 \longrightarrow [s \mapsto W_2]M_{12} \quad \text{(E-MAPPABS)}$$

Term Evaluation $\boxed{t \longrightarrow t'}$

$$\frac{M \longrightarrow M'}{\texttt{let m=M in t} \longrightarrow \texttt{let m=M' in t}} \quad \text{(E-MLET)}$$

$$\texttt{let m=W in t} \longrightarrow [m \mapsto W]t \quad \text{(E-MLETV)}$$

Figure 10-11: Dynamic Semantics for $\lambda^{(\!|\,|\!)}$

to the usual contravariant and covariant subtyping rules for the types of pairs and functions.

10.2.1 EXERCISE [★]: Find a syntactically different (but equivalent) precise signature for the `Diag` functor above, and a strictly less-precise signature also satisfied by `Diag`. □

10.2.2 EXERCISE [★,RECOMMENDED]: The language $\lambda^{(\!|\,|\!)}$ has a sub-interface relation, but no subtyping. Suppose we added this, e.g., with `Int <: Top`. How should the interfaces $\langle\!|$`Int`$|\!\rangle$ and $\langle\!|$`Top`$|\!\rangle$ be related? How about $\langle\!|$`*=Int`$|\!\rangle$ and $\langle\!|$`*=Top`$|\!\rangle$? □

$\lambda^{(\!|\,|\!)}$ adds a single kinding rule K-MPROJ, stating that we can project from a syntactically determinate module `W` to obtain a type as long as `W` is a module whose interface guarantees that it contains a type. In this rule we need only check that `W` has an opaque interface of the form $\langle\!|$`K`$|\!\rangle$ because by subsumption and SI-FORGET, any module with a transparent interface also has an opaque interface.

Type equivalence is extended as in $F^{\omega}_{\texttt{let}}$, but the new rule Q-MPROJ looks for type definitions that occur in transparent interfaces rather than for type definitions directly in the context.

The rules M-TERM and M-TYPE give precise interfaces to the two sorts of primitive modules; these interfaces can be weakened by subsumption as specified in M-SUB.

Next, the rules M-VAR through M-APPLY should be relatively familiar from systems with dependent types (see Chapter 4). The only surprise might be the requirement of non-dependent interfaces in the rules M-SND and M-APPLY. In many systems dependencies in these rules can be handled via substitution, but substitution of general modules for module variables may lead to ill-formed types as only syntactically determinate modules may appear in types.

The rule M-SELF is justified by the following observation: assume a module value $\mathtt{W}$ satisfies the interface $(\!|\mathtt{K}|\!)$. Now consider the interface $(\!|\mathtt{K}\mathtt{=}\mathtt{!}\mathtt{W}|\!)$; this is the interface of a module containing a single type, which is the same type as that contained in $\mathtt{W}$. Clearly $\mathtt{W}$ itself satisfies this description and hence ought to have this latter interface. M-SELF ensures that this is always provable. The rules M-SELF1 and M-SELF2 are similar, and allow the M-SELF rule to be applied to submodules of a larger module. For example, by using all three rules we can conclude

$$\mathtt{m} : (\!|\mathtt{*}|\!) \times (\!|\mathtt{*}|\!) \vdash \mathtt{m} : (\!|\, (\!|\mathtt{*}\mathtt{=}\mathtt{!}\mathtt{m}\mathtt{.}\mathtt{1}|\!)\mathtt{,}\ (\!|\mathtt{*}\mathtt{=}\mathtt{!}\mathtt{m}\mathtt{.}\mathtt{2}|\!)\,|\!)\mathtt{,}$$

i.e., that if $\mathtt{m}$ is a module containing two types, then it satisfies a interface that requires exactly the first type in $\mathtt{m}$ and the second type in $\mathtt{m}$.

In the presence of the SELF rules, the more usual dependent typing rules are admissible for determinate modules (Lillibridge, 1997):

$$\frac{\Gamma \vdash \mathtt{W} : \Sigma \mathtt{m}\mathtt{:}\mathtt{I}_1\mathtt{.}\mathtt{I}_2}{\Gamma \vdash \mathtt{W}\mathtt{.}\mathtt{2} : [\mathtt{m} \mapsto \mathtt{W}\mathtt{.}\mathtt{1}]\mathtt{I}_2} \qquad \text{(M-SNDW)}$$

$$\frac{\Gamma \vdash \mathtt{M}_1 : \Pi \mathtt{m}\mathtt{:}\mathtt{I}_1\mathtt{.}\mathtt{I}_2 \qquad \Gamma \vdash \mathtt{W}_2 : \mathtt{I}_1}{\Gamma \vdash \mathtt{M}_1\ \mathtt{W}_2 : [\mathtt{m} \mapsto \mathtt{W}_2]\mathtt{I}_2} \qquad \text{(M-APPLYW)}$$

For example, suppose that we had

$$\mathtt{M}_1 : \Pi \mathtt{m}\mathtt{:}(\!|\mathtt{*}|\!)\mathtt{.}\ (\!|\mathtt{*}\mathtt{=}\mathtt{!}\mathtt{m} \times \mathtt{!}\mathtt{m}|\!)$$

and we have

$$\mathtt{W}_2 : (\!|\mathtt{*}|\!)\mathtt{.}$$

Since the interface of if $\mathtt{M}_1$ is dependent we cannot directly use M-APPLY to type check the application $\mathtt{M}_1\ \mathtt{W}_2$, but we can show, using SI-PI, SI-FORGET, SI-OPAQUE, and M-SUB, that $\mathtt{M}_1$ satisfies the strictly less precise interface

$$\Pi \mathtt{m}\mathtt{:}(\!|\mathtt{*}\mathtt{=}\mathtt{!}\mathtt{W}_2|\!)\mathtt{.}\ (\!|\mathtt{*}\mathtt{=}\mathtt{!}\mathtt{m} \times \mathtt{!}\mathtt{m}|\!)$$

and hence satisfies the equivalent interface

$$\Pi \mathtt{m} : (\!|\, * = !\mathtt{W}_2 \,|\!) \,.\; (\!|\, * = !\mathtt{W}_2 \times !\mathtt{W}_2 \,|\!)$$

i.e., we have that

$$\mathtt{M}_1 \;:\; (\!|\, * = !\mathtt{W}_2 \,|\!) \to (\!|\, * = !\mathtt{W}_2 \times !\mathtt{W}_2 \,|\!).$$

Now by M-SELF we have

$$\mathtt{W}_2 \;:\; (\!|\, * = !\mathtt{W}_2 \,|\!)$$

so the premises of M-APPLY are now satisfied and we obtain

$$\mathtt{M}_1\, \mathtt{W}_2 \;:\; (\!|\, * = !\mathtt{W}_2 \times !\mathtt{W}_2 \,|\!)$$

exactly as the admissible rule M-APPLYW predicts.

Finally, the M-SEAL rule is an explicit form of subsumption corresponding to the implicit subsumption of M-SUB, forcing the type system to hide all information that is not mentioned in the specified interface. In the definition

```
let m = ((|Int::*|),(|3|)) in ...
```

the variable `m` has the very precise interface $(\!|\, * = \mathtt{Int} \,|\!) \times (\!|\, \mathtt{Int} \,|\!)$, allowing expressions such as `(!m.1) + 3` to type check. In contrast, if sealing were used as in the definition

```
let m = ((|Int::*|),(|3|)) :> Σm:(|*|).(|!m|) in ...
```

then in the scope of this definition we must treat `m` as having only the interface $\Sigma \mathtt{m} : (\!|\, * \,|\!) \,.\, (\!|\, !\mathtt{m} \,|\!)$, even though clearly at run-time it will contain the type `Int` and the integer `3`. Any code using `m` must respect abstraction, treating `!m.2` not as an integer, but as a value of the unspecified type `!m.1`.

The evaluation relation for modules looks very much like the evaluation relation for the simply-typed lambda calculus with pairs. The one rule of note is E-MSEAL. Although the strong sealing operation is "generative" and affects type checking by mimicking the creation of a "fresh" abstract type where the sealing operation is performed, once the program begins running it has no observable effects and hence can be discarded.[6]

Type Equivalence and Type Checking

The translucent sums calculus suffers from the avoidance problem discussed in Chapter 9, and hence fails to have principal (most-specific up to equivalence) interfaces.

6. At run-time, values of an abstract stack type really are linked lists, or arrays, or whatever the underlying representation happens to be.

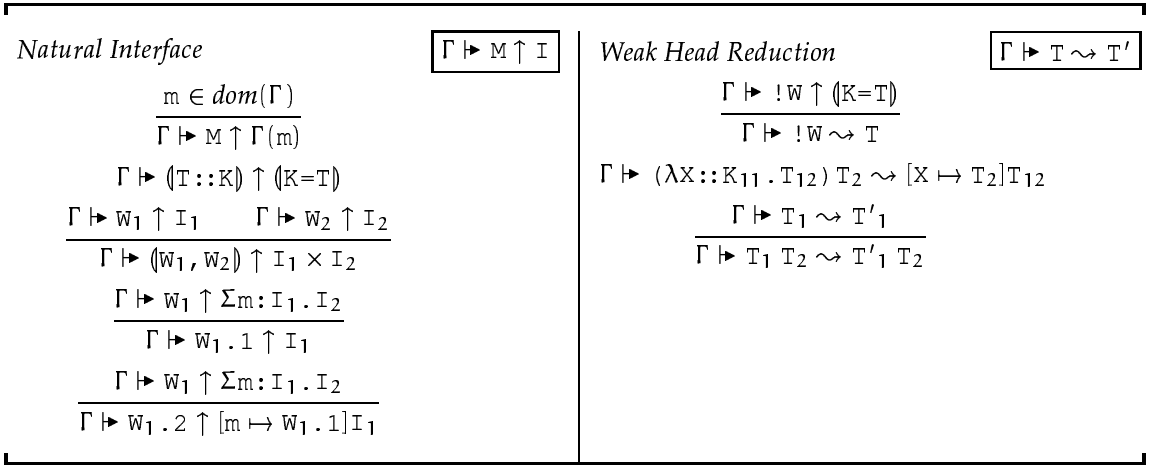


Figure 10-12: Algorithmic Equivalence for $\lambda^{(\!|\,|\!)}$

10.2.3 EXERCISE: ★★★ Find a $\lambda^{(\!|\,|\!)}$ module that has no most-precise signature. (Hint: see Chapter 9.) □

This makes type checking difficult, though in practice programs can often be restricted to subsets guaranteed to have principal interfaces (Leroy, 1996c; Harper and Stone, 2000b).

10.2.4 EXERCISE: ★★★ Consider the following restricted grammar for modules:

```
P ::= m | P.1 | P.2
M ::= P | (|t|) | (|T::K|) | ( P, P ) | λm:I.M | P P | P :> I
```

This enforces programming in a "named form". Instead of writing an expression

```
let m = (λm::(|*|). m) ( ((|Int|), (|Bool|)).2 )
in ...
```

we must give names to all intermediate module computations, e.g.,

```
let m1 = λm::(|*|). m
in let m2 =  (|Int::*|)
   in let m3 = (|Bool::*|)
      in let m4 = (|m2, m3|)
         in let m5 = m4.2
            in let m = m1 m5
               in ... .
```

Show that if in this restricted language, every module has a principal signature. (For simplicity you may assume that every term has a unique type.) □

Even in the absence of principal signatures, however, type *equivalence* remains decidable. A similar algorithm to that for $F^{\omega}_{\mathtt{let}}$ will work, except that now it is not type variables that have definitions, but general modules of the form `!W`. For example, if `m : (|*|) × (|*=Int|)` then `!m.2` is a path whose definition is clearly `Int`. More interestingly, if

```
m : (Σm′::(|*|).(|*=!m′|)),
```

i.e, if `m` is a module containing two equal types, then the path `!m.2` can be thought of as having a definition, namely as being defined to be the first component `!m.1`. This can be formalized through the notion of the *natural interface* of a module value, which would be the principal interface in the absence of the rules M-SELF, M-SELF1, and M-SELF2. The SELF rules collectively allow more equations to be added to the interfaces of module values, but they never introduce any "new" information, and hence are irrelevant when trying to detect definitions. See Figure 10-12 for definition of natural interfaces and the changes to head-reduction for types.

10.2.5 EXERCISE [★]: Verify that according to the definition of weak head reduction shown in Figure 10-12 we have `m : (Σm′:(|*|).(|*=!m′|)) ↦ !m.2 ⇝ !m.1` but that in the same context `!m.1` is weak head normal. □

The proof that this algorithm is correct is much more involved than the proof for $F^{\omega}_{\mathtt{let}}$, however, because the type equivalence relation depends much more closely on all the other judgment forms, particularly well-formedness of modules via Q-MPROJ (Lillibridge, 1997).

10.3 Singleton Kinds

In $F^{\omega}_{\mathtt{let}}$, definitions were introduced with a new sort of context entry. Where previously contexts recorded the kinds of type variables, they now had the option of specifying a specific definition. The introduction of definitions in the translucent sum calculus was quite similar, except the choice between kind and kind-with-definition was in module interfaces.

In general, wherever a language normally requires the specification of a kind, we could add the alternative of either specifying the kind or specifying both the kind and a specific type. For example, we could extend F^{ω} with a new sort of polymorphic abstraction

$$\lambda X::K_1=T_1.t_2,$$

that is to be applied only to the argument T_1.

How might these be used? Recall that closed-scope term-level definitions are often treated as derived forms for an application, namely

$$\texttt{let}\ x=t_1\ \texttt{in}\ t_2 \stackrel{\text{def}}{=} (\lambda x:T_1.t_2)\ t_1$$

where $t_1 : T_1$. A similar construct can be used at the level of types:

$$\texttt{let}\ X=T_1\ \texttt{in}\ T_2 \stackrel{\text{def}}{=} (\lambda X::K_1.T_2)\ T_1$$

where K_1 is the kind of the type T_1.

However, a type definition used in a term does not similarly correspond to an application of System F polymorphism as one might expect. Although

```
let X=Int in (λx:X.x+1)(4)
```

is perfectly reasonable, the polymorphic instantiation

```
(λX::*. (λx:X.x+1)(4))[Int]
```

is ill-typed, because its sub-term

```
λX::*. (λx:X.x+1)(4)
```

is ill-typed.

By using restricted type abstractions. however, we can do slightly better. The derived form becomes

$$\texttt{let}\ X=T_1\ \texttt{in}\ t_2 \stackrel{\text{def}}{=} (\lambda X::K_1=T_1.t_2)(T_1)$$

where K_1 is the kind of T_1. The constraint on the function argument — namely that the value passed in for X will be T_1 — is enough to type check the function body. Then in the application, we must confirm that the actual type argument is indeed equal to T_1. This definition does duplicate the type T_1, but this can still be significantly smaller than $[X \mapsto T_1]t_2$.

Minamide, Morrisett, and Harper, 1996 use exactly this approach for the purposes of polymorphic closure conversion. The goal was to turn both free term variables and free type variables of functions are turned into arguments; this was necessary in the context of a type-passing interpretation of polymorphism (Tarditi, Morrisett, Cheng, Stone, Harper, and Lee, 1996) where types are computed and analyzed at run time. By using the above restricted polymorphic abstractions, they were able to preserve well-formedness when pulling types out of function bodies.

This extension has rather specialized uses, but rather than trying to predict exactly where definitions are needed, a more general approach is to allow definitions at all points where type variables are given kinds. One natural formulation of this approach augments the kinds themselves to include definitions.

This leads to the introduction of *singleton kinds*. The singleton kind $\mathcal{S}(\texttt{T})$ classifies exactly those types of kind $\texttt{*}$ provably equivalent to $\texttt{T}$. For example, the kind $\mathcal{S}(\texttt{int} \times \texttt{int})$ classifies all types that are provably equivalent to the type of pairs of integers; there is up to equivalence exactly one such type. Then instead of choosing between a kind specification $\texttt{Y::*}$ or a kind-and-definition specification $\texttt{Y::*=int}$, we specify either $\texttt{Y::*}$ or $\texttt{Y::}\mathcal{S}(\texttt{int})$. Figure 10-13 defines $\lambda^{\mathcal{S}}$, an alternate variant of F^{ω} including singleton kinds. (In contrast to the previous systems in this chapter, Figure 10-13 shows the type and kind judgments in full rather than just the changes to F^{ω}, which are pervasive.)

The types of $\lambda^{\mathcal{S}}$ have been chosen to include ordinary types, type operators, and (to permit expressiveness similar to that of $\lambda^{()}$), pairs of types. The addition of singletons allows kinds to refer to types, and thus it is natural to permit the kinds classifying type operators or type pairs to be dependent. The kind $\Sigma\texttt{X::K}_1\texttt{.K}_2$ classifies pairs whose first component has kind $\texttt{K}_1$ and second component has kind $\texttt{K}_2$, where $\texttt{K}_2$ can use the type variable $\texttt{X}$ to refer to the value of the first component. In the case where $\texttt{K}_2$ does not mention $\texttt{X}$, we can abbreviate this kind to $\texttt{K}_1 \times \texttt{K}_2$, as usual. Thus, for example, we give the pair of types (which is a collection of two types, not a single type of a pair of values)

```
{Int,Int}
```

the kind

```
* × *
```

stating simply that it is a pair of types, or give it the very precise kind

$\mathcal{S}(\texttt{Int}) \times \mathcal{S}(\texttt{Int})$

i.e., that we have a pair of types whose components are both equal to $\texttt{Int}$, or somewhere in-between such as

$\Sigma\texttt{X::*}.\ \mathcal{S}(\texttt{X})$,

i.e., that we have a pair of types whose first component has kind $\texttt{*}$, and whose second component is the same as the first, or as

$\mathcal{S}(\texttt{Int}) \times \texttt{*}$,

$\lambda^{\mathcal{S}}$ *extends* F^{ω}

Syntax

$\mathtt{K}$ ::=		*kinds:*
	$*$	*Kind of types*
	$\mathcal{S}(\mathtt{T})$	*Singleton kind*
	$\Pi \mathtt{X}::\mathtt{K}_1.\mathtt{K}_2$	*Kind of type functions*
	$\Sigma \mathtt{X}::\mathtt{K}_1.\mathtt{K}_2$	*Kind of type pairs*
$\mathtt{T}$ ::=	...	*types:*
	$\{\mathtt{T},\mathtt{T}\}$	*Pair of types*
	$\pi_1\ \mathtt{T}$	*First projection*
	$\pi_2\ \mathtt{T}$	*Second projection*

Derived Forms

$\mathtt{K}_1 \times \mathtt{K}_2 \stackrel{\text{def}}{=} \Sigma \mathtt{X}::\mathtt{K}_1.\mathtt{K}_2 \quad (\mathtt{X} \notin FV(\mathtt{K}_2))$

$\mathtt{K}_1 \Rightarrow \mathtt{K}_2 \stackrel{\text{def}}{=} \Pi \mathtt{X}::\mathtt{K}_1.\mathtt{K}_2 \quad (\mathtt{X} \notin FV(\mathtt{K}_2))$

Kind validity $\boxed{\Gamma \vdash \mathtt{K}}$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash *} \quad \text{(WK-*)}$$

$$\frac{\Gamma \vdash \mathtt{T} :: *}{\Gamma \vdash \mathcal{S}(\mathtt{T})} \quad \text{(WK-SING)}$$

$$\frac{\Gamma, \mathtt{X}::\mathtt{K}_1 \vdash \mathtt{K}_2}{\Gamma \vdash \Pi \mathtt{X}::\mathtt{K}_1.\mathtt{K}_2} \quad \text{(WK-PI)}$$

$$\frac{\Gamma, \mathtt{X}::\mathtt{K}_1 \vdash \mathtt{K}_2}{\Gamma \vdash \Sigma \mathtt{X}::\mathtt{K}_1.\mathtt{K}_2} \quad \text{(WK-SIGMA)}$$

Subkinding $\boxed{\Gamma \vdash \mathtt{K} <: \mathtt{L}}$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash * <: *} \quad \text{(SK-*)}$$

$$\frac{\Gamma \vdash \mathtt{S} \equiv \mathtt{T} :: *}{\Gamma \vdash \mathcal{S}(\mathtt{S}) <: \mathcal{S}(\mathtt{T})} \quad \text{(SK-SING)}$$

$$\frac{\Gamma \vdash \mathtt{T} :: *}{\Gamma \vdash \mathcal{S}(\mathtt{T}) <: *} \quad \text{(SK-FORGET)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{L}_1 <: \mathtt{K}_1 \\ \Gamma, \mathtt{X}::\mathtt{L}_1 \vdash \mathtt{K}_2 <: \mathtt{L}_2 \\ \Gamma \vdash \Pi \mathtt{X}::\mathtt{K}_1.\mathtt{K}_2\end{array}}{\Gamma \vdash \Pi \mathtt{X}::\mathtt{K}_1.\mathtt{K}_2 <: \Pi \mathtt{X}::\mathtt{L}_1.\mathtt{L}_2} \quad \text{(SK-PI)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{K}_1 <: \mathtt{L}_1 \\ \Gamma, \mathtt{X}::\mathtt{K}_1 \vdash \mathtt{K}_2 <: \mathtt{L}_2 \\ \Gamma \vdash \Pi \mathtt{X}::\mathtt{L}_1.\mathtt{L}_2\end{array}}{\Gamma \vdash \Sigma \mathtt{X}::\mathtt{K}_{11}.\mathtt{K}_{12} <: \Sigma \mathtt{X}::\mathtt{K}_{21}.\mathtt{K}_{22}} \quad \text{(SK-SIGMA)}$$

Kind Equivalence $\boxed{\Gamma \vdash \mathtt{K} \equiv \mathtt{L}}$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash * \equiv *} \quad \text{(QK-*)}$$

$$\frac{\Gamma \vdash \mathtt{S} \equiv \mathtt{T} :: *}{\Gamma \vdash \mathcal{S}(\mathtt{S}) \equiv \mathcal{S}(\mathtt{T})} \quad \text{(QK-SING)}$$

$$\frac{\Gamma \vdash \mathtt{K}_1 \equiv \mathtt{L}_1 \qquad \Gamma, \mathtt{X}::\mathtt{K}_1 \vdash \mathtt{K}_2 \equiv \mathtt{K}_2}{\Gamma \vdash \Pi \mathtt{X}::\mathtt{K}_1.\mathtt{K}_2 \equiv \Pi \mathtt{X}::\mathtt{L}_1.\mathtt{L}_2} \quad \text{(QK-PI)}$$

$$\frac{\Gamma \vdash \mathtt{K}_1 <: \mathtt{L}_1 \qquad \Gamma, \mathtt{X}::\mathtt{K}_1 \vdash \mathtt{K}_2 \equiv \mathtt{K}_2}{\Gamma \vdash \Sigma \mathtt{X}::\mathtt{K}_1.\mathtt{K}_2 \equiv \Sigma \mathtt{X}::\mathtt{L}_1.\mathtt{L}_2} \quad \text{(QK-SIGMA)}$$

Figure 10-13: Singleton Kinds

Kinding rules $\boxed{\Gamma \vdash T :: K}$

$$\frac{\Gamma \vdash \diamond \qquad T \in \{\mathtt{Int}, \mathtt{Bool}, \ldots\}}{\Gamma \vdash T :: *} \quad \text{(K-Base)}$$

$$\frac{X :: K \in \Gamma \qquad \Gamma \vdash \diamond}{\Gamma \vdash X :: K} \quad \text{(K-Var)}$$

$$\frac{\Gamma \vdash T :: *}{\Gamma \vdash T :: \mathcal{S}(T)} \quad \text{(K-SIntro)}$$

$$\frac{\Gamma, X :: K_1 \vdash T_2 :: K_2}{\Gamma \vdash \lambda X :: K_1 . T_2 :: \Pi X :: K_1 . K_2} \quad \text{(K-Abs)}$$

$$\frac{\Gamma \vdash T_1 :: \Pi X :: K_1 . K_2 \qquad \Gamma \vdash T_2 :: K_1}{\Gamma \vdash T_1\ T_2 :: [X \mapsto T_2]K_2} \quad \text{(K-App)}$$

$$\frac{\Gamma, X :: K_1 \vdash T_2 :: *}{\Gamma \vdash \forall X :: K_1 . T_2 :: *} \quad \text{(K-All)}$$

$$\frac{\Gamma \vdash \Sigma X :: K_1 . K_2 \qquad \Gamma \vdash T_1 :: K_1 \qquad \Gamma \vdash T_2 :: [X \mapsto T_1]K_2}{\Gamma \vdash \{T_1, T_2\} :: \Sigma X :: K_1 . K_2} \quad \text{(K-Pair)}$$

$$\frac{\Gamma \vdash T :: \Sigma X :: K_1 . K_2}{\Gamma \vdash \pi_1\ T :: K_1} \quad \text{(K-Fst)}$$

$$\frac{\Gamma \vdash T :: \Sigma X :: K_1 . K_2}{\Gamma \vdash \pi_2\ T :: [X \mapsto \pi_1\ T]K_2} \quad \text{(K-Snd)}$$

$$\frac{\Gamma, X :: K_1 \vdash T(X) :: K_2 \qquad X \notin FV(T)}{\Gamma \vdash T :: \Pi X :: K_1 . K_2} \quad \text{(K-Fn-Eta)}$$

$$\frac{\Gamma \vdash \pi_1\ T :: K_1 \qquad \Gamma \vdash \pi_2\ T :: K_2}{\Gamma \vdash T :: K_1 \times K_2} \quad \text{(K-Pair-Eta)}$$

$$\frac{\Gamma \vdash T :: K_1 \qquad \Gamma \vdash K_1 <: K_2}{\Gamma \vdash T :: K_2} \quad \text{(K-Sub)}$$

Equivalence rules $\boxed{\Gamma \vdash S \equiv T :: K}$

$$\frac{\Gamma \vdash T :: K}{\Gamma \vdash T \equiv T :: K} \quad \text{(Q-Refl)}$$

$$\frac{\Gamma \vdash T \equiv S :: K}{\Gamma \vdash S \equiv T :: K} \quad \text{(Q-Sym)}$$

$$\frac{\Gamma \vdash S \equiv U :: K \qquad \Gamma \vdash U \equiv T :: K}{\Gamma \vdash S \equiv T :: K} \quad \text{(Q-Trans)}$$

$$\frac{\Gamma \vdash S_1 \equiv T_1 :: \Pi X :: K_1 . K_2 \qquad \Gamma \vdash S_2 \equiv T_2 :: K_1}{\Gamma \vdash S_1\ S_2 \equiv T_1\ T_2 :: [X \mapsto S_1]K_2} \quad \text{(Q-App)}$$

$$\frac{\Gamma \vdash S \equiv T :: \Sigma X :: K_1 . K_2}{\Gamma \vdash \pi_1\ S \equiv \pi_1\ T :: K_1} \quad \text{(Q-Fst)}$$

$$\frac{\Gamma \vdash S \equiv T :: \Sigma X :: K_1 . K_2}{\Gamma \vdash \pi_2\ S \equiv \pi_2\ T :: [X \mapsto \pi_1\ S]K_2} \quad \text{(Q-Snd)}$$

$$\frac{\Gamma \vdash \Sigma X :: K_1 . K_2 \qquad \Gamma \vdash S_1 \equiv T_1 :: K_1 \qquad \Gamma \vdash S_2 \equiv T_2 :: [X \mapsto S_1]K_2}{\Gamma \vdash \{S_1, S_2\} \equiv \{T_1, T_2\} :: \Sigma X :: K_1 . K_2} \quad \text{(Q-Pair)}$$

$$\frac{\Gamma \vdash K_1 \equiv K_2 \qquad \Gamma, X :: K_1 \vdash S_2 \equiv T_2 :: K_2}{\Gamma \vdash \lambda X :: K_1 . S_2 \equiv \lambda X :: K_2 . T_2 :: \Pi X :: K_1 . K_2} \quad \text{(Q-Abs)}$$

$$\frac{\Gamma \vdash K_1 \equiv K_2 \qquad \Gamma, X :: K_1 \vdash T_1 \equiv T_2 :: *}{\Gamma \vdash \forall X :: K_1 . T_1 \equiv \forall X :: K_2 . T_2 :: *} \quad \text{(Q-All)}$$

$$\frac{\Gamma \vdash S :: \mathcal{S}(T)}{\Gamma \vdash S \equiv T :: \mathcal{S}(S)} \quad \text{(Q-Elim)}$$

$$\frac{\Gamma \vdash \Sigma X :: K_1 . K_2 \qquad \Gamma \vdash \pi_1 S \equiv \pi_1 T :: K_1 \qquad \Gamma \vdash \pi_2 S \equiv \pi_2 T :: [X \mapsto \pi_1\ S]K_2}{\Gamma \vdash S \equiv T :: \Sigma X :: K_1 . K_2} \quad \text{(Q-Pair-Ext)}$$

$$\frac{\Gamma, X :: K_1 \vdash S\ X \equiv T\ X :: K_2}{\Gamma \vdash S \equiv T :: \Pi X :: K_1 . K_2} \quad \text{(Q-Fn-Ext)}$$

$$\frac{\Gamma \vdash S \equiv T :: L \qquad \Gamma \vdash L <: K}{\Gamma \vdash S \equiv T :: K} \quad \text{(Q-Sub)}$$

Figure 10-13: Singleton Kinds, continued

i.e., that the first type is `Int` and the second is some proper type, and so on.

Similarly, the kind $\Pi \mathtt{X}::\mathtt{K}_1 . \mathtt{K}_2$ is the kind of type operators which take an argument `X` of kind $\mathtt{K}_1$ and return a result of kind $\mathtt{K}_2$, where $\mathtt{K}_2$ can depend on the particular argument type `X`. Again, if $\mathtt{K}_2$ does not mention `X` then the kind can be written $\mathtt{K}_1 \Rightarrow \mathtt{K}_2$.

Thus, possible kinds for the identity function $\lambda \mathtt{X}::* . \mathtt{X}$ on ordinary types will include the familiar

$$* \Rightarrow *$$

as well as the very precise judgment

$$\Pi \mathtt{X}::* . \mathcal{S}(\mathtt{X}),$$

stating that given any type `X` of kind `*` the function returns a result equal to `X`. There are infinitely many other possibilities as well, such as

$$\mathcal{S}(\mathtt{Int}) \Rightarrow \mathcal{S}(\mathtt{Int})$$

i.e., that the function can be applied to the type `Int`, and if so it will return the same type `Int`.

Every type of kind $\mathcal{S}(\mathtt{T})$ is, by the definition of singleton kinds, also a proper type of kind `*`. This induces a *subkinding* relation, i.e., $\mathcal{S}(\mathtt{T}) <: *$ for any `T`. Subkinding is lifted to the kinds of functions and of pairs in the normal way; e.g., function kinds are contravariant in their argument and covariant in their result. Subkinding between singleton kinds coincides with equivalence.

10.3.1 EXERCISE [★]: The language $\lambda^{\mathcal{S}}$ has subkinding but not subtyping. If subtyping were added, e.g., with `Int <: Top`, then how should the kinds $\mathcal{S}(\mathtt{Int})$ and $\mathcal{S}(\mathtt{Top})$ be related? □

The well-formedness rules for kinds are mostly the familiar rules for a dependently-typed (or, in this case, dependently-kinded) lambda calculus with functions and pairs. Four rules stand out for special consideration, however. Rule K-SUB is a subsumption rule that makes use of the subkinding rule; a type with a more-precise kind can also be used as a type with a less-precise kind. Rule K-SINTRO is the introduction rule for singleton kinds. The rule allows a well-formed type of kind `*` to be given a more precise singleton kind.

Rule K-PAIR-ETA has essentially the same purpose as Rules M-SELF-PAIR1 and M-SELF-PAIR2 do in the translucent sum calculus, while K-FN-ETA serves a similar purpose for kinds of type operators. In most type systems K-PAIR-ETA

and K-FN-ETA would be admissible, but here they allow more precise typings.

For example, suppose $\mathtt{Y}$ is a pair of types, that is, $\mathtt{Y} :: * \times *$. Now consider the kind

$$\mathcal{S}(\pi_1\ \mathtt{Y}) \times \mathcal{S}(\pi_2\ \mathtt{Y}),$$

i.e., the kind of pairs of types whose first component is equal to the first component of $\mathtt{Y}$, and whose second component is equal to the second component of $\mathtt{Y}$. Regardless of whether the language includes eta-equivalence for types, we would expect $\mathtt{Y}$ itself to satisfy this latter kind. Rules K-PAIR-ETA and K-SINTRO allow us to prove this conclusion:

$$\dfrac{\dfrac{\dfrac{}{\mathtt{Y} :: * \times * \vdash \pi_1\ \mathtt{Y} :: *}}{\mathtt{Y} :: * \times * \vdash \pi_1\ \mathtt{Y} :: \mathcal{S}(\pi_1\ \mathtt{Y})} \quad \dfrac{\dfrac{}{\mathtt{Y} :: * \times * \vdash \pi_2\ \mathtt{Y} :: *}}{\mathtt{Y} :: * \times * \vdash \pi_2\ \mathtt{Y} :: \mathcal{S}(\pi_2\ \mathtt{Y})}}{\mathtt{Y} :: * \times * \vdash \mathtt{Y} :: \mathcal{S}(\pi_1\ \mathtt{Y}) \times \mathcal{S}(\pi_2\ \mathtt{Y})}$$

Similarly, assume $\mathtt{Z} :: * \Rightarrow *$. Then the kind $\Pi \mathtt{X} :: *.\ \mathtt{S}(\mathtt{Z}\ \mathtt{X})$ classifies all type operators that, when given a type argument $\mathtt{X}$, yield the same result as $\mathtt{Z}$ does when given $\mathtt{X}$. Again, $\mathtt{Z}$ itself has this property and rule K-FN-ETA allows us to prove it.

Rules Q-REFL through Q-ALL are standard rules for a lambda calculus with dependencies; they insure that definitional equivalence is an equivalence relation, and moreover a congruence.

Singletons arise explicitly in Rule Q-ELIM, the elimination rule for singleton kinds:

$$\dfrac{\Gamma \vdash \mathtt{S} :: \mathcal{S}(\mathtt{T})}{\Gamma \vdash \mathtt{S} \equiv \mathtt{T} :: \mathcal{S}(\mathtt{S})}$$

Rules Q-FN-EXT and Q-PAIR-EXT yield extensionality: componentwise-equivalent pairs are equivalent and and pointwise-equivalent functions are equivalent.

Finally, we have a subsumption rule for equivalence, Q-SUB, corresponding to the subsumption rule for typing.

The addition of singleton kinds has more consequences for equivalence than might appear at first. An attentive reader may have noticed that Figure 10-13 is missing both the beta-reduction rule for function applications and the two standard rules for reducing projections from pairs. A surprising fact about the language with singletons, noticed by Aspinall, 1995, is that elimination rules for functions and pairs do not have to be included in the

definition of the system; they are admissible (i.e., if the premises are provable using the above rules then so is the conclusion):

$$\frac{\Gamma,\ X::K_1 \vdash T_{12} :: K_{12} \qquad \Gamma \vdash T_2 :: K_{12}}{\Gamma \vdash (\lambda X::K_{11}.T_{12})\,T_2 \equiv [X \mapsto T_2]T_{12} :: [X \mapsto T_2]K_{12}} \quad \text{(Q-AppAbs)}$$

$$\frac{\Gamma \vdash T_1 :: K_1 \qquad \Gamma \vdash T_2 :: K_2}{\Gamma \vdash \pi_1\ \{T_1, T_2\} \equiv T_1 :: K_1} \quad \text{(Q-Beta-Fst)}$$

$$\frac{\Gamma \vdash T_1 :: K_1 \qquad \Gamma \vdash T_2 :: K_2}{\Gamma \vdash \pi_2\ \{T_1, T_2\} \equiv T_2 :: K_2} \quad \text{(Q-Beta-Snd)}$$

Similarly, using the extensionality rules we can derive the standard eta-equivalence rules:

$$\frac{\Gamma \vdash T :: \Pi X::K_1.K_2 \qquad X \notin FV(T)}{\Gamma \vdash T \equiv (\lambda X::K_1.T\,X) :: \Pi X::K_1.K_2} \quad \text{(Q-Eta-Fn)}$$

$$\frac{\Gamma \vdash T :: \Sigma X::K_1.K_2}{\Gamma \vdash T \equiv \{\pi_1\ T,\ \pi_2\ T\} :: \Sigma X::K_1.K_2} \quad \text{(Q-Eta-Pair)}$$

More importantly (since we could have added beta or eta-equivalence had they not been derivable), the kind at which two types can determine whether or not they are equivalent. Types do not have unique kinds, and a pair of types can be equivalent at one kind but not another.

For example, consider the identity function on types, $\lambda X::*.X$, and the constantly-`Int` function $\lambda X::*.\mathtt{Int}$. There is no way to prove the judgment

$$(\lambda X::*.X) \equiv (\lambda X::*.\mathtt{Int}) :: (*\Rightarrow *).$$

However, by subsumption both functions also have the kind $\mathcal{S}(\mathtt{Int})\Rightarrow *$, and at this kind we can prove

$$\vdash (\lambda X::*.X) \equiv (\lambda X::*.\mathtt{Int}) :: (\mathcal{S}(\mathtt{Int})\Rightarrow *).$$

Viewed as functions that will only be applied to the argument `Int`, the two functions are guaranteed to return the same result, namely `Int`, in all cases. By extensionality, then, the two functions are equivalent at kind $\mathcal{S}(\mathtt{Int})\Rightarrow *$.

10.3.2 Exercise [★★]: Prove in full that this equivalence holds. (You may use the admissible rules.) □

Using this result and Rule Q-App we can further show that

$$\mathtt{Y} :: (\mathtt{S}(\mathtt{Int})\Rightarrow *)\Rightarrow * \vdash \mathtt{Y}(\lambda X::*.X) \equiv \mathtt{Y}(\lambda X::*.\mathtt{Int}) :: *.$$

In this equivalence judgment, both sides are normal with respect to both beta and eta-reduction. Further, `Y` itself has no obvious definition that can be expanded away. The fact that the two types are nevertheless provably equivalent suggests that techniques of expanding definitions and reducing applications used in $F^{\omega}_{\mathtt{let}}$ cannot be so easily applied here.

Singletons at Higher Kinds

In $F^{\omega}_{\mathtt{let}}$ the context could contain definitions for type operators, e.g., `Y :: (*⇒*) = (λX::*.X→X)`. We cannot directly represent this definition using a singleton kind as `Y :: S (λX::*.X→X)` because the kind `S (T)` is well-formed only for types `T` of kind `*`.

Fortunately, using extensionality more general singletons are definable. For example, one can show that the kind

```
Π X::*.S (X→X)
```

classifies all type operators that are equivalent to `λX::*.X→X` at kind `*⇒*`.

More generally, whenever `T :: K` holds we can define the kind of types equivalent to `T` at kind `K` as a derived form, written `S (T :: K)`. Again the kind classifier is crucial, since by the examples above the function `λX::*.X` should have kind

```
S ((λX::*.Int) :: S(Int)⇒*)
```

but not kind

```
S ((λX::*.Int) :: *⇒*).
```

Figure 10-14 defines labeled singleton kinds `S (T::K)` by induction on the size of the classifying kind `K`.

The sizes of kinds are defined in Figure 10-15. An easy inductive proof shows that substitutions have no effect on the size of kinds, and hence the labeled singleton kinds in Figure 10-14 are well-defined.

These labeled singletons behave exactly as one would expect. To show this, we start with some basic facts about $\lambda^{\mathcal{S}}$.

10.3.3 PROPOSITION [WEAKENING FOR $\lambda^{\mathcal{S}}$]: If $\Gamma_1, \Gamma_3 \vdash \mathcal{J}$ for any $\lambda^{\mathcal{S}}$ judgment form $\mathcal{J}$ and $\Gamma_1, \Gamma_2, \Gamma_3 \vdash \diamond$ then $\Gamma_1, \Gamma_2, \Gamma_3 \vdash \mathcal{J}$. □

10.3.4 PROPOSITION: If $\Gamma \vdash \mathtt{T} :: \mathtt{K}$ then $FV(\mathtt{T}) \cup FV(\mathtt{K}) \subseteq dom(\Gamma)$. □

10.3.5 PROPOSITION: 1. If $\Gamma \vdash \mathtt{K}$ then $\Gamma \vdash \diamond$.

2. If $\Gamma \vdash \mathtt{T} :: \mathtt{K}$ then $\Gamma \vdash \mathtt{K}$.

$$
\begin{array}{rcll}
\mathcal{S}(\mathtt{T} :: *) & \stackrel{\text{def}}{=} & \mathcal{S}(\mathtt{T}) & \\
\mathcal{S}(\mathtt{T} :: \mathcal{S}(\mathtt{T}')) & \stackrel{\text{def}}{=} & \mathcal{S}(\mathtt{T}) & \\
\mathcal{S}(\mathtt{T} :: \Pi \mathtt{X}::\mathtt{K}_1.\mathtt{K}_2) & \stackrel{\text{def}}{=} & \Pi \mathtt{X}::\mathtt{K}_1.\mathcal{S}(\mathtt{T}\ \mathtt{X} :: \mathtt{K}_2) & \text{where } \mathtt{X} \notin FV(\mathtt{T}) \\
\mathcal{S}(\mathtt{T} :: \Sigma \mathtt{X}::\mathtt{K}_1.\mathtt{K}_2) & \stackrel{\text{def}}{=} & \mathcal{S}(\pi_1\ \mathtt{T} :: \mathtt{K}_1) \times \mathcal{S}(\pi_2\ \mathtt{T} :: [\mathtt{X} \mapsto \pi_1\ \mathtt{T}]\mathtt{K}_2) &
\end{array}
$$

Figure 10-14: Labeled Singleton Kinds

$$
\begin{array}{rcl}
\mathit{size}(*) & \stackrel{\text{def}}{=} & 1 \\
\mathit{size}(\mathcal{S}(\mathtt{T})) & \stackrel{\text{def}}{=} & 2 \\
\mathit{size}(\Sigma \mathtt{X}::\mathtt{K}_1.\mathtt{K}_2) & \stackrel{\text{def}}{=} & 1 + \mathit{size}(\mathtt{K}_1) + \mathit{size}(\mathtt{K}_2) \\
\mathit{size}(\Pi \mathtt{X}::\mathtt{K}_1.\mathtt{K}_2) & \stackrel{\text{def}}{=} & 1 + \mathit{size}(\mathtt{K}_1) + \mathit{size}(\mathtt{K}_2)
\end{array}
$$

Figure 10-15: Kind Sizes

3. If $\Gamma \vdash \mathtt{S} \equiv \mathtt{T} :: \mathtt{K}$ then $\Gamma \vdash \mathtt{S} :: \mathtt{K}$ and $\Gamma \vdash \mathtt{T} :: \mathtt{K}$.

□

At this point we can consider properties of the labeled singletons themselves.

10.3.6 PROPOSITION: $[\mathtt{X} \mapsto \mathtt{S}](\mathcal{S}(\mathtt{T} :: \mathtt{K})) = \mathcal{S}([\mathtt{X} \mapsto \mathtt{S}]\mathtt{T} :: [\mathtt{X} \mapsto \mathtt{S}]\mathtt{K})$. □

Proof: By induction on the size of $\mathtt{K}$. □

10.3.7 PROPOSITION: 1. If $\Gamma \vdash \mathtt{T} :: \mathtt{K}$ then $\Gamma \vdash \mathtt{T} :: \mathcal{S}(\mathtt{T}::\mathtt{K})$.

2. If $\Gamma \vdash \mathtt{S} :: \mathcal{S}(\mathtt{T} :: \mathtt{K})$ and $\Gamma \vdash \mathtt{T} :: \mathtt{K}$ then $\Gamma \vdash \mathtt{S} \equiv \mathtt{T} :: \mathtt{K}$.

3. If $\Gamma \vdash \mathtt{S} \equiv \mathtt{T} :: \mathtt{K}$ and $\Gamma \vdash \mathtt{K} <: \mathtt{L}$ then $\Gamma \vdash \mathcal{S}(\mathtt{S}::\mathtt{K}) <: \mathcal{S}(\mathtt{T}::\mathtt{K})$.

4. If $\Gamma \vdash \mathtt{T} :: \mathtt{K}$ then $\Gamma \vdash \mathcal{S}(\mathtt{T}::\mathtt{K}) <: \mathcal{S}(\mathtt{T}::\mathcal{S}(\mathtt{T}::\mathtt{K}))$.

□

Proof: We show the proof of just the first two parts.

1. By induction on the size of $\mathtt{K}$. Assume $\Gamma \vdash \mathtt{T} :: \mathtt{K}$.

   Case: $\mathtt{K} = *$, so $\mathcal{S}(\mathtt{T}::\mathtt{K}) = \mathcal{S}(\mathtt{T})$.

   Then $\Gamma \vdash \mathtt{S} :: \mathcal{S}(\mathtt{T})$ by Rule K-SINTRO.

Case: $\mathtt{K} = \mathcal{S}(\mathtt{S})$, so $\mathcal{S}(\mathtt{T}::\mathtt{K}) = \mathcal{S}(\mathtt{T})$.

By Proposition 10.3.5 $\Gamma \vdash \mathcal{S}(\mathtt{S})$, so by inversion of Rule WK-SING we have $\Gamma \vdash \mathtt{S} :: *$. Therefore $\Gamma \vdash \mathcal{S}(\mathtt{S}) <: *$, and so by K-SUB we have $\Gamma \vdash \mathtt{T} :: *$ and hence $\Gamma \vdash \mathtt{T} :: \mathcal{S}(\mathtt{T})$ by K-SINTRO.

Case: $\mathtt{K} = \Pi \mathtt{X}::\mathtt{K}_1 . \mathtt{K}_2$, so $\mathcal{S}(\mathtt{T}::\mathtt{K}) = \Pi \mathtt{X}::\mathtt{K}_1 . \mathcal{S}(\mathtt{T}\,\mathtt{X} :: \mathtt{K}_2)$.

By Proposition 10.3.5 and inversion we have $\Gamma, \mathtt{X}::\mathtt{K}_1 \vdash \diamond$, so by Proposition 10.3.3 and K-APP, $\Gamma, \mathtt{X}::\mathtt{K}_1 \vdash \mathtt{T}\,\mathtt{X} :: \mathtt{K}_2$. By the inductive hypothesis, $\Gamma, \mathtt{X}::\mathtt{K}_1 \vdash \mathtt{T}\,\mathtt{X} :: \mathcal{S}(\mathtt{T}\,\mathtt{X} :: \mathtt{K}_2)$. Thus by Rule K-FN-ETA we have $\Gamma \vdash \mathtt{T} :: \Pi \mathtt{X}::\mathtt{K}_1 . \mathcal{S}(\mathtt{T}\,\mathtt{X} :: \mathtt{K}_2)$ as desired.

Case: $\mathtt{K} = \Sigma \mathtt{X}::\mathtt{K}_1 . \mathtt{K}_2$, so $\mathcal{S}(\mathtt{T}::\mathtt{K}) = \mathcal{S}(\pi_1\,\mathtt{T} :: \mathtt{K}_1) \times \mathcal{S}(\pi_2\,\mathtt{T} :: [\mathtt{X} \mapsto \pi_1\,\mathtt{T}]\mathtt{K}_2)$

By K-FST and K-SND and the inductive hypothesis, we have $\Gamma \vdash \pi_1\,\mathtt{T} :: \mathcal{S}(\pi_1\,\mathtt{T} :: \mathtt{K}_1)$ and $\Gamma \vdash \pi_2\,\mathtt{T} :: \mathcal{S}(\pi_2\,\mathtt{T} :: [\mathtt{X} \mapsto \pi_1\,\mathtt{T}]\mathtt{K}_2)$. Therefore, by Rule K-PAIR-ETA the desired result follows.

2. By induction on the size of $\mathtt{K}$. Assume $\Gamma \vdash \mathtt{S} :: \mathcal{S}(\mathtt{T} :: \mathtt{K})$ and $\Gamma \vdash \mathtt{T} :: \mathtt{K}$.

   Case: $\mathtt{K} = *$, so $\mathcal{S}(\mathtt{T}::\mathtt{K}) = \mathcal{S}(\mathtt{T})$.

   Then $\Gamma \vdash \mathtt{S} \equiv \mathtt{T} :: \mathcal{S}(\mathtt{T})$ by Rule Q-ELIM, and $\Gamma \vdash \mathcal{S}(\mathtt{T}) <: *$ by SK-FORGET, so $\Gamma \vdash \mathtt{S} \equiv \mathtt{T} :: *$ by Rule Q-SUB.

   Case: $\mathtt{K} = \mathcal{S}(\mathtt{U})$, so $\mathcal{S}(\mathtt{T}::\mathtt{K}) = \mathcal{S}(\mathtt{T})$.

   By Rule Q-ELIM we have $\Gamma \vdash \mathtt{S} \equiv \mathtt{T} :: \mathcal{S}(\mathtt{S})$. By Proposition 10.3.5, inversion of K-SING, and SK-FORGET we have $\Gamma \vdash \mathcal{S}(\mathtt{S}) <: *$, so $\Gamma \vdash \mathtt{S} \equiv \mathtt{T} :: *$ by Q-SUB.

   Case: $\mathtt{K} = \Pi \mathtt{X}::\mathtt{K}_1 . \mathtt{K}_2$, so $\mathcal{S}(\mathtt{T}::\mathtt{K}) = \Pi \mathtt{X}::\mathtt{K}_1 . \mathcal{S}(\mathtt{T}\,\mathtt{X} :: \mathtt{K}_2)$.

   By Proposition 10.3.5 and inversion we have $\Gamma, \mathtt{X}::\mathtt{K}_1 \vdash \diamond$, so by Proposition 10.3.3 and K-APP, $\Gamma, \mathtt{X}::\mathtt{K}_1 \vdash \mathtt{S}\,\mathtt{X} :: \mathcal{S}(\mathtt{T}\,\mathtt{X} :: \mathtt{K}_2)$. By the same reasoning we have $\Gamma, \mathtt{X}::\mathtt{K}_1 \vdash \mathtt{T}\,\mathtt{X} :: \mathtt{K}_2$. By the inductive hypothesis, $\Gamma, \mathtt{X}::\mathtt{K}_1 \vdash \mathtt{S}\,\mathtt{X} \equiv \mathtt{T}\,\mathtt{X} :: \mathtt{K}_2$. Therefore, by Rule Q-FN-EXT we have $\Gamma \vdash \mathtt{S} \equiv \mathtt{T} :: \Pi \mathtt{X}::\mathtt{K}_1 . \mathtt{K}_2$.

   Case: $\mathtt{K} = \Sigma \mathtt{X}::\mathtt{K}_1 . \mathtt{K}_2$, so $\mathcal{S}(\mathtt{T}::\mathtt{K}) = \mathcal{S}(\pi_1\,\mathtt{T} :: \mathtt{K}_1) \times \mathcal{S}(\pi_2\,\mathtt{T} :: [\mathtt{X} \mapsto \pi_1\,\mathtt{T}]\mathtt{K}_2)$

   By K-FST and K-SND we have $\Gamma \vdash \pi_1\,\mathtt{S} :: \mathcal{S}(\pi_1\,\mathtt{T} :: \mathtt{K}_1)$ and $\Gamma \vdash \pi_2\,\mathtt{S} :: \mathcal{S}(\pi_2\,\mathtt{T} :: [\mathtt{X} \mapsto \pi_1\,\mathtt{T}]\mathtt{K}_2)$. Again by K-FST and K-SND we have $\Gamma \vdash \pi_1\,\mathtt{T} :: \mathtt{K}_1$ and $\Gamma \vdash \pi_2\,\mathtt{T} :: [\mathtt{X} \mapsto \pi_1\,\mathtt{T}]\mathtt{K}_2$, so by the inductive hypothesis we have we have $\Gamma \vdash \pi_1\,\mathtt{S} \equiv \pi_1\,\mathtt{T} :: \mathtt{K}_1$ and and $\Gamma \vdash \pi_1\,\mathtt{S} \equiv \pi_2\,\mathtt{T} :: [\mathtt{X} \mapsto \pi_1\,\mathtt{T}]\mathtt{K}_2)$. By Rule Q-PAIR-EXT, therefore, we have $\Gamma \vdash \mathtt{S} \equiv \mathtt{T} :: \Sigma \mathtt{X}::\mathtt{K}_1 . \mathtt{K}_2$ as desired.

□

10.3.8 EXERCISE [★★]: Why is the assumption $\Gamma \vdash \texttt{T} :: \texttt{K}$ required in Part 2 of this last proposition? □

At this point, it is not too hard to show that the BETA rules are admissible, as there is a natural proof involving labeled singletons.

10.3.9 EXERCISE [★★, RECOMMENDED]: Prove that Q-BETA-FST, Q-BETA-SND, and Q-APPABS are admissible. □

Given the BETA rules and extensionality, the usual eta rules follow.

10.3.10 EXERCISE [★★]: Prove that rules Q-ETA-FN and Q-ETA-PAIR are admissible. □

Aspinall, 1995 took a slightly different approach to formalizing a language with singletons. His language $\lambda_{\leq\{\}}$ included a very restricted form of extensionality, and so labeled singletons were taken as primitives rather than being defined.[7] The properties of Proposition 10.3.7 are then taken as axioms describing the behavior of these primitive singletons. In this formulation, the last part of Proposition 10.3.7 is necessary to have principal kinds; otherwise

$$\begin{aligned} \texttt{*} \;&:>\; \mathcal{S}(\texttt{Int}::\texttt{*}) \\ &:>\; \mathcal{S}(\texttt{Int}::\mathcal{S}(\texttt{Int}::\texttt{*})) \\ &:>\; \mathcal{S}(\texttt{Int}::\mathcal{S}(\texttt{Int}::\mathcal{S}(\texttt{Int}::\texttt{*}))) \\ &:>\; \cdots \end{aligned}$$

would be an infinite sequence of strictly more-precise kinds for the type `Int`.

An interesting consequence of making labeled singletons explicit is that Aspinall was able to *define* the equivalence judgment $\Gamma \vdash \texttt{S} \equiv \texttt{T} :: \texttt{K}$ as syntactic sugar for the judgment $\Gamma \vdash \texttt{S} :: \mathcal{S}(\texttt{T} :: \texttt{K})$.

A disadvantage of primitive labeled singletons rather than relying on extensionality is that most-precise classifiers can be large. For example, in an Aspinall-style system the most-precise kind of $\lambda\texttt{X}::\texttt{*}.\lambda\texttt{Y}::\texttt{*}.\texttt{X}$ would be

$$\begin{aligned} &\mathcal{S}((\lambda\texttt{X}::\texttt{*}.\lambda\texttt{Y}::\texttt{*}.\texttt{X}) :: \\ &\qquad \Pi\texttt{X}::\texttt{*}.\mathcal{S}((\lambda\texttt{Y}::\texttt{*}.\texttt{X}) :: (\Pi\texttt{Y}::\texttt{*}.\texttt{S}(\texttt{X} :: \texttt{*})))) \end{aligned}$$

rather than

$$\Pi\texttt{X}::\texttt{*}.\ \Pi\texttt{Y}::\texttt{*}.\ \mathcal{S}(\texttt{X})$$

7. More precisely, Aspinall studied term equivalence in a language with singleton types, but the ideas would work just as well for type equivalence with singleton kinds.

as in $\lambda^{\mathcal{S}}$. Of course conversely, $\mathcal{S}$`(Z :: *⇒*⇒*)` is simpler than Π`Y`$_1$`::*.`Π`Y`$_2$`::*.Z Y`$_1$ `Y`$_2$. A unlabelled higher-kind singleton $\mathcal{S}$`(Z)` for the type operator `Z` would be even more concise, but the metatheory of and algorithms for such generalized singletons are not well-studied.

Phase-Splitting

Adding second-class modules to a language noticeably increases the number of sorts of entities in the language; in addition to terms, types, and kinds we now have modules and interfaces. Interestingly, however, modules and interfaces can actually be decomposed into uses of terms, types, and kinds. Some implementations of Standard ML (Petersen, Cheng, Harper, and Stone, 2000; Shao, 1997, 1998) even implement modules using this technique.

The translation is possible if the module system has a phase distinction, as discussed in Chapter 9: types in modules must depend only on other types. $\lambda^{()}$ appears to violate this requirement, as the type `!W` may involve arbitrary module values, which turn can contain arbitrary term values. Similarly, a functor application `M`$_1$ `M`$_2$ can yield a module containing types, and the result syntactically appears to depend on all of `M`$_2$, which can contain arbitrary program terms.

As observed by Harper, Mitchell, and Moggi, 1990b, however, the dependency of types on the terms in modules is illusory. References to modules in types involve only the type components of that module; all else is superfluous. Similarly, in a functor application `M`$_1$ `M`$_2$ the types returned to depend only on types defined in `M`$_1$ and types defined in `M`$_2$; there is no way that the terms in these modules can affect the resulting types.

It is therefore possible to split *every* module into a type part and a term part; the former can be represented as a type (perhaps of a higher kind), and the latter as a term (potentially polymorphic).

A module containing many types and many values would split into a collection of only types (not to be confused with the type of a tuple of terms) and a collection of only values. In parallel, a functor application can be split into an application of types (the type part of the functor applied to the type part of the argument) and an application of a polymorphic term (the term part of the functor applied to both the type part of the argument and the term part of the argument). For example, the functor

```
module Diag = functor(S : sig
                            type t
                            val x : t
                          end) →
                struct
```

```
    type u = S.t * S.t
    val  y = (S.x, S.x)
end
```

which we encoded in the translucent sum calculus as

```
λm:(Σm':(|*|). (|!m'|)).
                (| (|!m.1 × !m.1|), (|{!m.2,!m.2}|) |)
```

could have its type portion be essentially

```
λX::*. X × X,
```

since `Diag` takes a type in its argument and returns the corresponding pair type in its result. Similarly, the term part could be

```
λX::*. λx:X.{x,x}.
```

representing that the functor takes a type and a term of that type in the argument, and that it returns a pair value.

Interfaces can be split correspondingly into a kind classifying the type portion of the module, and a type classifying the term portion of the module. So, the specification

```
diag : (Πm:(Σm':<*>. <!m'>). (Σm'':<*=!m.1 × !m.1>. <!m''>))
```

could be split into two specifications for the type part and term part of `diag` respectively:

$$\texttt{diag}_{type}\ \texttt{::}\ (\Pi \texttt{X::*}.\ \mathcal{S}(\texttt{X} \times \texttt{X}))$$

and

$$\texttt{diag}_{term}\ \texttt{:}\ \forall \texttt{X::*}.\ \texttt{X} \rightarrow \texttt{X} \times \texttt{X}.$$

Note that equations in $\lambda^{(|)}$ transparent signatures can simply become singleton kinds after phase-splitting.

The one construct in $\lambda^{(|)}$ which has no direct equivalent in $\lambda^{\mathcal{S}}$ is the generative sealing operation `M :> I`. The $\lambda^{\mathcal{S}}$ language has no abstraction mechanism, and there is no easy way to directly add a generative construct to its equational theory of types.

However, since sealing has no run-time effects it is possible to take a well-formed $\lambda^{(|)}$ term and erase all occurrence of sealing, yielding a behaviorally-equivalent term. Thus implementations of $\lambda^{(|)}$ based on phase-splitting first do type checking in $\lambda^{(|)}$ to ensure that abstractions are being respected, and thereafter erase the abstractions and perform the phase-splitting.

Natural Kind $\boxed{\Gamma \vdash T \uparrow K}$

$$\Gamma \vdash X \uparrow \Gamma(X)$$

$$\frac{\Gamma \vdash T_1 \uparrow \Pi X::K_1.K_2}{\Gamma \vdash T_1\ T_2 \uparrow [X \mapsto T_2]K_2}$$

$$\frac{\Gamma \vdash T_1 \uparrow \Sigma X::K_1.K_2}{\Gamma \vdash \pi_1\ T_1 \uparrow K_1}$$

$$\frac{\Gamma \vdash T_1 \uparrow \Sigma X::K_1.K_2}{\Gamma \vdash \pi_2\ T_2 \uparrow [X \mapsto \pi_1\ T_1]K_2}$$

$$\Gamma \vdash \forall X::K.T_2 \uparrow *$$

Weak Head Reduction $\boxed{\Gamma \vdash T \rightsquigarrow T'}$

$$\Gamma \vdash X \rightsquigarrow T \qquad \text{if } \Gamma \vdash X \uparrow \mathcal{S}(T)$$

$$\Gamma \vdash (\lambda X::K_{11}.T_{12})\ T_2 \rightsquigarrow [X \mapsto T_2]T_{12}$$

$$\Gamma \vdash \pi_1\ \{T_1, T_2\} \rightsquigarrow T_1$$

$$\Gamma \vdash \pi_2\ \{T_1, T_2\} \rightsquigarrow T_2$$

$$\frac{\Gamma \vdash T_1 \rightsquigarrow T'_1}{\Gamma \vdash T_1\ T_2 \rightsquigarrow T'_1\ T_2}$$

$$\frac{\Gamma \vdash T_1 \rightsquigarrow T'_1}{\Gamma \vdash \pi_1\ T_1 \rightsquigarrow \pi_1\ T'_1}$$

$$\frac{\Gamma \vdash T_1 \rightsquigarrow T'_1}{\Gamma \vdash \pi_2\ T_1 \rightsquigarrow \pi_2\ T'_1}$$

Head Normalization $\boxed{\Gamma \vdash S \Downarrow T}$

$$\frac{\Gamma \vdash S \rightsquigarrow S' \qquad \Gamma \vdash S' \Downarrow T}{\Gamma \vdash S \Downarrow T}$$

$$\frac{\Gamma \vdash S \not\rightsquigarrow}{\Gamma \vdash S \Downarrow S}$$

Type Equivalence $\boxed{\Gamma \vdash S \leftrightarrow T :: K}$

$$\frac{\Gamma \vdash S \Downarrow S' \qquad \Gamma \vdash T \Downarrow T' \qquad \Gamma \vdash S' \leftrightarrow T' \uparrow *}{\Gamma \vdash S \Leftrightarrow T :: *}$$

$$\frac{\Gamma \vdash S \Leftrightarrow T :: \mathcal{S}(T) \qquad X \notin dom(\Gamma) \qquad \Gamma, X::K_1 \vdash S\ X \Leftrightarrow T\ X :: K_2}{\Gamma \vdash S \Leftrightarrow T :: \Pi X::K_1.K_2}$$

$$\frac{\Gamma \vdash \pi_1\ S \Leftrightarrow \pi_1\ T :: K_1 \qquad \Gamma \vdash \pi_2\ S \Leftrightarrow \pi_2\ T :: [X \mapsto \pi_1\ S]K_2}{\Gamma \vdash S \Leftrightarrow T :: \Sigma X::K_1.K_2}$$

Structural Equivalence $\boxed{\Gamma \vdash S \leftrightarrow T \uparrow K}$

$$\Gamma \vdash \mathtt{int} \leftrightarrow \mathtt{int} \uparrow *$$

$$\Gamma \vdash X \leftrightarrow X \uparrow \Gamma(X)$$

$$\frac{\Gamma \vdash S_1 \leftrightarrow T_1 \uparrow * \qquad \Gamma \vdash S_2 \Leftrightarrow T_2 \uparrow *}{\Gamma \vdash S_1 \to S_2 \leftrightarrow T_1 \to T_2 \uparrow *}$$

$$\frac{X \notin dom(\Gamma) \qquad \Gamma, X::K \vdash T_1 \Leftrightarrow T_2 \uparrow *}{\Gamma \vdash \forall X::K.T_1 \leftrightarrow \forall X::K.T_2 \uparrow *}$$

$$\frac{\Gamma \vdash S_1 \leftrightarrow T_1 \uparrow \Pi X::K_1.K_2 \qquad \Gamma \vdash S_2 \Leftrightarrow T_2 :: K_1}{\Gamma \vdash S_1\ S_2 \leftrightarrow T_1\ T_2 \uparrow [X \mapsto S_2]K_2}$$

$$\frac{\Gamma \vdash S_1 \leftrightarrow T_1 \uparrow \Sigma X::K_1.K_2}{\Gamma \vdash \pi_1\ S_1 \leftrightarrow \pi_1\ T_1 \uparrow K_1}$$

$$\frac{\Gamma \vdash S_1 \leftrightarrow T_1 \uparrow \Sigma X::K_1.K_2}{\Gamma \vdash \pi_2\ S_1 \leftrightarrow \pi_2\ T_1 \uparrow [X \mapsto \pi_1\ S_1]K_2}$$

Figure 10-16: Algorithmic Equivalence with Definitions

Algorithmic Type Equivalence

Figure 10-16 shows a modified algorithmic version of equivalence of well-kinded types for $\lambda^{\mathcal{S}}$. The general framework is very similar to that for the `let` language, but a number of the details are different.

First of all, there is a judgment for computing "natural kinds" in analogy with the natural interfaces for module values in Figure 10-12. These are the most-precise kind available without using singleton introduction rules. The type has a definition if the natural kind is a singleton.

The algorithmic type equivalence relation becomes more elaborate, largely because of extensionality and the fact that singletons are a form of unit (see Chapter 2). Following Coquand, 1991b, the algorithmic equivalence relation is defined by induction on the classifying kind: type operators are compared by applying both sides to a fresh variable (to determine if they are pointwise equivalent), while pairs of types are compared componentwise. Types of kind `*` are head-normalized and compared structurally very much as before. Finally, types with singleton kinds are easy to compare because of the precondition that the two types actually have the kind at which they are being compared; any two types of kind $\mathcal{S}$`(T)` are equivalent to `T` and hence equivalent to each other.

The structural equivalence judgment is very similar to the preceeding algorithms. The main difference is that it simultaneously determines a natural kind. A kind is needed as in the Chapter 2 algorithms because when comparing the arguments of two irreducible applications, we need to choose the kind at which to compare these arguments.[8]

10.3.11 EXERCISE [★★,RECOMMENDED]: Show that

```
Y::(S(Int)⇒*)⇒* ⊢ Y(λX::*.X) ⇔ Y(λX::*.Int) :: *
```

is provable, but

```
Y::(*⇒*)⇒* ⊢ Y(λX::*.X) ⇔ Y(λX::*.Int) :: *
```

is not. □

The equivalence algorithm is correct and terminating.

8. The arguments are most likely to be found equal if they are compared at the greatest possible kind; by contravariance this kind can be found by looking at the domain of the principal kind for the type operators being applied, and it can be be shown that this is always equal to the domain of the natural kind.

10.3.12 FACT: If $\Gamma \vdash \mathtt{T} :: \mathtt{K}$ and $\Gamma \vdash \mathtt{T}' :: \mathtt{K}$ then $\Gamma \mapsto \mathtt{T} \Leftrightarrow \mathtt{T}' :: \mathtt{K}$ if and only if $\Gamma \vdash \mathtt{T} \equiv \mathtt{T}' :: \mathtt{K}$. Furthermore, the judgment $\Gamma \mapsto \mathtt{T} \Leftrightarrow \mathtt{T}' :: \mathtt{K}$ is always decidable (i.e, proof search will terminate whether or not $\mathtt{T}$ and $\mathtt{T}'$ are equivalent). □

The correctness of this equivalence algorithm is not trivial. The approach suggested in Chapter 2 does not directly apply because there is no a priori reason to believe that algorithmic equivalence is transitive, which means that the natural logical equivalence relation cannot directly be shown to be transitive. The problem is in the "asymmetry" of the rules caused by substitutions into kinds. The structural equivalence judgment $\Gamma \mapsto \mathtt{S} \leftrightarrow \mathtt{T} \uparrow \mathtt{K}$ computes the natural kind $\mathtt{K}$ of $\mathtt{S}$ as it goes along, but might have just as well computed the natural kind of $\mathtt{T}$ instead. Although we expect the two natural kinds to be *declaratively* equivalent, this does not guarantee that the algorithm is unaffected — kinds in the classifier end up in the context, which can affect how later weak head normalizations proceed, which could a priori result in a different answer or affect termination.

Stone and Harper, 2000a showed, however, that variants of Kripke logical relations can be used to prove the correctness of related algorithms, from which the correctness of the above algorithm can be derived.

10.4 Notes

Primitive definitions are permitted in most λ-calculus-based system implementations. The Automath system (de Bruijn, 1980; van Daalen, 1980), for example, relied vitally on definitions, as do more modern systems such as Coq (Barras, Boutin, Cornes, Courant, Filliatre, Gimenez, Herbelin, Huet, Munoz, Murthy, Parent, Paulin-Mohring, Saibi, and Werner, 1997).

Most directly related to the system $F^{\omega}_{\mathtt{let}}$ is that of Severi and Poll, 1994, who proved confluence and strong normalization of a pure type system with primitive definitions.

For references on the theory of module systems, see Chapter 9; the presentation of $\lambda^{(\!|}$ is most similar to that of Lillibridge, 1997.

Aspinall, 1995 suggested that if types were viewed as program specification, then singleton types would allow very specfic specifications (e.g., requiring that a particular function not only map natural numbers to natural numbers, but that it compute factorials). He presented a system of labeled singleton types with beta-equivalence and a limited form of extensionality for lambda abstractions. Stone and Harper, 2000a proved decidability of a $\lambda^{\mathcal{S}}$-style system very similar to, while Courant, 2002 was able to give a simpler reduction relation (along the lines of that for $F^{\omega}_{\mathtt{let}}$) by dropping exten-

sionality altogether.

Appendices

A *Solutions to Selected Exercises*

1.2.6 SOLUTION: The definition does not behave as expected, because `if` is a destructor, whose arguments—according to the call-by-value semantics of ML-the-calculus—are evaluated *before* R-TRUE or R-FALSE is allowed to fire. As a result, the semantics of the expression $\texttt{if}\ t_0\ \texttt{then}\ t_1\ \texttt{else}\ t_2$ is to evaluate *both* t_1 and t_2 before choosing one of them. Since these expressions may have side effects (for instance, they may fail to terminate, or update a reference), this semantics is undesirable. The desired evaluation order can be obtained by placing t_1 and t_2 within closures, which delays their evaluation, then invoking the closure returned by the conditional, forcing its body to be evaluated. In other words, the expression $\texttt{if}\ t_0\ \texttt{then}\ t_1\ \texttt{else}\ t_2$ should now be viewed as syntactic sugar for $\texttt{if}\ t_0\ (\lambda z.t_1)\ (\lambda z.t_2)\ \hat{0}$. The choice of the constant $\hat{0}$ is arbitrary, since it is discarded; any value would do.

1.2.24 SOLUTION: Within Damas and Milner's type system, we have:

$$\dfrac{\text{DM-LET}\ \dfrac{\text{DM-VAR}\ \dfrac{}{z_1 : X \vdash z_1 : X} \qquad \dfrac{}{z_1 : X; z_2 : X \vdash z_2 : X}\ \text{DM-VAR}}{z_1 : X \vdash \texttt{let}\ z_2 = z_1\ \texttt{in}\ z_2 : X}}{\text{DM-ABS}\ \dfrac{}{\varnothing \vdash \lambda z_1.\texttt{let}\ z_2 = z_1\ \texttt{in}\ z_2 : X \to X}}$$

Please note that, because X occurs free within the environment $z_1 : X$, it is impossible to apply DM-GEN to the judgement $z_1 : X \vdash z_1 : X$ in a nontrivial way. For this reason, z_2 cannot receive the type scheme $\forall X.X$, and the whole expression cannot receive type $X \to Y$, where X and Y are distinct.

1.2.25 SOLUTION: It is straightforward to prove that the identity function has type $\mathsf{int} \to \mathsf{int}$:

$$\dfrac{\dfrac{}{\Gamma_0; z : \mathsf{int} \vdash z : \mathsf{int}}\ \text{DM-VAR}}{\Gamma_0 \vdash \lambda z.z : \mathsf{int} \to \mathsf{int}}\ \text{DM-ABS}$$

In fact, nothing in this type derivation depends on the choice of int as the type of z. Thus, we may just as well use a type variable X instead. Furthermore, after forming the arrow type $X \to X$, we may employ DM-GEN to quantify universally over X, since X no longer appears in the environment.

$$\dfrac{\dfrac{\dfrac{}{\Gamma_0; z : X \vdash z : X}\,\text{DM-VAR}}{\Gamma_0 \vdash \lambda z.z : X \to X}\,\text{DM-ABS} \qquad X \notin \mathit{ftv}(\Gamma_0)}{\Gamma_0 \vdash \lambda z.z : \forall X.X \to X}\,\text{DM-GEN}$$

It is worth noting that, although the type derivation employs an arbitrary type variable X, the final typing judgement has no free type variables. It is thus independent of the choice of X. In the following, we refer to the above type derivation as Δ_0.

Next, we prove that the successor function has type $\mathsf{int} \to \mathsf{int}$ under the initial environment Γ_0. We write Γ_1 for $\Gamma_0; z : \mathsf{int}$, and make uses of DM-VAR implicit.

$$\dfrac{\dfrac{\dfrac{\Gamma_1 \vdash \hat{+} : \mathsf{int} \to \mathsf{int} \to \mathsf{int} \qquad \Gamma_1 \vdash z : \mathsf{int}}{\Gamma_1 \vdash \hat{+}\ z : \mathsf{int} \to \mathsf{int}}\,\text{DM-APP} \qquad \Gamma_1 \vdash \hat{1} : \mathsf{int}}{\Gamma_1 \vdash z\ \hat{+}\ \hat{1} : \mathsf{int}}\,\text{DM-APP}}{\Gamma_0 \vdash \lambda z.z\ \hat{+}\ \hat{1} : \mathsf{int} \to \mathsf{int}}\,\text{DM-ABS}$$

In the following, we refer to the above type derivation as Δ_1. We may now build a derivation for the third typing judgement. We write Γ_2 for $\Gamma_0; f : \mathsf{int} \to \mathsf{int}$.

$$\dfrac{\Delta_1 \qquad \dfrac{\Gamma_2 \vdash f : \mathsf{int} \to \mathsf{int} \qquad \Gamma_2 \vdash \hat{2} : \mathsf{int}}{\Gamma_2 \vdash f\ \hat{2} : \mathsf{int}}\,\text{DM-APP}}{\Gamma_0 \vdash \mathtt{let}\ f = \lambda z.z\ \hat{+}\ \hat{1}\ \mathtt{in}\ f\ \hat{2} : \mathsf{int}}\,\text{DM-LET}$$

To derive the fourth typing judgement, we re-use Δ_0, which proves that the identity function has polymorphic type $\forall X.X \to X$. We write Γ_3 for $\Gamma_0; f : \forall X.X \to X$. By DM-VAR and DM-INST, we have both $\Gamma_3 \vdash f : (\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int})$ and $\Gamma_3 \vdash f : \mathsf{int} \to \mathsf{int}$. Thus, we may build the following derivation:

$$\dfrac{\Delta_0 \qquad \dfrac{\dfrac{\Gamma_3 \vdash f : (\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int}) \qquad \Gamma_3 \vdash f : \mathsf{int} \to \mathsf{int}}{\Gamma_3 \vdash f\ f : \mathsf{int} \to \mathsf{int}}\,\text{DM-APP} \qquad \Gamma_3 \vdash \hat{2} : \mathsf{int}}{\Gamma_3 \vdash f\ f\ \hat{2} : \mathsf{int}}\,\text{DM-APP}}{\Gamma_0 \vdash \mathtt{let}\ f = \lambda z.z\ \mathtt{in}\ f\ f\ \hat{2} : \mathsf{int}}\,\text{DM-LET}$$

The first and third judgements are valid in the simply-typed λ-calculus, because they use neither DM-GEN nor DM-INST, and use DM-LET only to introduce the *monomorphic* binding $\mathtt{f} : \mathsf{int} \to \mathsf{int}$ into the environment. The second judgement, of course, is not: because it involves a nontrivial type scheme, it is not even a well-formed judgement in the simply-typed λ-calculus. The fourth judgement is well-formed, but *not* derivable, in the simply-typed λ-calculus. This is because $\mathtt{f}$ is used at two incompatible types, namely $(\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int})$ and $\mathsf{int} \to \mathsf{int}$, inside the expression $\mathtt{f}\ \mathtt{f}\ \hat{2}$. Both of these types are *instances* of $\forall \mathtt{X}.\mathtt{X} \to \mathtt{X}$, the type scheme assigned to $\mathtt{f}$ in the environment Γ_3.

By inspection of the rules, a derivation of $\Gamma_0 \vdash \hat{1} : \mathtt{T}$ must begin with an instance of DM-VAR, of the form $\Gamma_0 \vdash \hat{1} : \mathsf{int}$. It may be followed by an arbitrary number of instances of the sequence (DM-GEN; DM-INST), turning int into a type scheme of the form $\forall \bar{\mathtt{X}}.\mathsf{int}$, then back to int. Thus, $\mathtt{T}$ must be int. Because int is not an arrow type, there follows that the application $\hat{1}\ \hat{2}$ cannot be well-typed under Γ_0. In fact, because this expression is stuck, it cannot be well-typed in a sound type system.

The expression $\lambda\mathtt{f}.(\mathtt{f}\ \mathtt{f})$ is ill-typed in the simply-typed λ-calculus, because no type $\mathtt{T}$ may coincide with a type of the form $\mathtt{T} \to \mathtt{T}'$: indeed, $\mathtt{T}$ would be a subterm of itself. In DM, this expression is ill-typed as well, but the proof of this fact is slightly more complex. One must point out that, because $\mathtt{f}$ is λ-bound, it must be assigned a type $\mathtt{T}$ (as opposed to a type scheme) in the environment. Furthermore, one must note that DM-GEN is not applicable (except in a trivial way) to the judgement $\Gamma_0; \mathtt{f} : \mathtt{T} \vdash \mathtt{f} : \mathtt{T}$, because all of the type variables in the type $\mathtt{T}$ appear free in the environment $\Gamma_0; \mathtt{f} : \mathtt{T}$. Once these points are made, the proof is the same as in the simply-typed λ-calculus.

It is important to note that the above argument crucially relies on the fact that $\mathtt{f}$ is λ-bound and must be assigned a *type*, as opposed to a type scheme. Indeed, we have proved earlier in this exercise that the self-application $\mathtt{f}\ \mathtt{f}$ is well-typed when $\mathtt{f}$ is `let`-bound and is assigned the type scheme $\forall \mathtt{X}.\mathtt{X} \to \mathtt{X}$. For the same reason, $\lambda\mathtt{f}.(\mathtt{f}\ \mathtt{f})$ is well-typed in an implicitly-typed variant of System F. It also relies on the fact that types are *finite*: indeed, $\lambda\mathtt{f}.(\mathtt{f}\ \mathtt{f})$ is well-typed in an extension of the simply-typed λ-calculus with recursive types, where the equation $\mathtt{T} = \mathtt{T} \to \mathtt{T}'$ has a solution.

Later, we will develop a type inference algorithm for ML-the-type-system and prove that it is correct and complete. Then, to prove that a term is ill-typed, it will be sufficient to simulate a run of the algorithm and to check that it reports a failure.

1.2.26 SOLUTION: It is clear that the effect of DM-GEN may be obtained by a series of successive applications of DM-GEN′. Conversely, consider an instance of

DM-GEN′, whose premises are $\Gamma \vdash t : S$ **(1)** and $X \notin ftv(\Gamma)$ **(2)**. Let us write $S = \forall \bar{X}.T$, where $\bar{X} \# ftv(\Gamma)$ **(3)**. Applying DM-INST to (1) and to the identity substitution yields $\Gamma \vdash t : T$ **(4)**. Applying DM-GEN to (4), (2) and (3) yields $\Gamma \vdash t : \forall X\bar{X}.T$, that is, $\Gamma \vdash t : \forall X.S$. Thus, the effect of DM-GEN′ may be obtained by DM-INST and DM-GEN.

It is clear that DM-INST is a particular case of DM-INST′ where $\bar{Y}$ is empty. Conversely, consider an instance of DM-INST′, whose premises are $\Gamma \vdash t : \forall \bar{X}.T$ **(1)** and $\bar{Y} \# ftv(\forall \bar{X}.T)$ **(2)**. Let ρ be a renaming that exchanges $\bar{Y}$ with $\bar{Z}$, where $\bar{Z} \# ftv(\forall \bar{Y}.[\vec{X} \mapsto \vec{T}]T)$ **(3)** and $\bar{Z} \# ftv(\Gamma)$ **(4)**. Applying DM-INST to (1) yields $\Gamma \vdash t : [\vec{X} \mapsto \rho\vec{T}]T$ **(5)**. Applying DM-GEN to (5) and (4) yields $\Gamma \vdash t : \forall \bar{Z}.[\vec{X} \mapsto \rho\vec{T}]T$, that is, $\Gamma \vdash t : \forall \rho\bar{Y}.[\vec{X} \mapsto \rho\vec{T}]T$ **(6)**. Now, by (2) and (3), we have $[\vec{X} \mapsto \rho\vec{T}]T = \rho([\vec{X} \mapsto \vec{T}]T)$, so (6) may be written $\Gamma \vdash t : \forall \rho\bar{Y}.\rho([\vec{X} \mapsto \vec{T}]T)$, that is, $\Gamma \vdash t : \rho(\forall \bar{Y}.[\vec{X} \mapsto \vec{T}]T)$ **(7)**. By (3), this is exactly $\Gamma \vdash t : \forall \bar{Y}.[\vec{X} \mapsto \vec{T}]T$. Thus, the effect of DM-INST′ may be obtained by DM-INST and DM-GEN.

1.4.7 SOLUTION: Let us recall that a program t is well-typed if and only if a judgement of the form $C, \Gamma \vdash t : \sigma$, where C is satisfiable, holds. Let us show that it is in fact possible, without loss of generality, to require σ to be a monotype.

Assume $C, \Gamma \vdash t : \sigma$ **(1)** is derivable within HM(X). Let us write $\sigma = \forall \bar{X}[D].T$, where $\bar{X} \# ftv(C)$ **(2)**. Applying Lemma 1.4.1 to (1) yields $C \Vdash \exists \bar{X}.D$ **(3)**. By HMX-INST, (1) implies $C \wedge D, \Gamma \vdash t : T$ **(4)**. By (3), we have $C \equiv C \wedge \exists \bar{X}.D \equiv \exists \bar{X}.(C \wedge D)$. Because C is satisfiable, this implies that $C \wedge D$ is satisfiable as well. Thus, the judgement (4), which involves the monotype T, witnesses that t is well-typed.

We have shown that a program t is well-typed if and only if a judgement of the form $C, \Gamma \vdash t : T$, where C is satisfiable, holds. Thus, by Theorems 1.4.5 and 1.4.6, well-typedness is the same for both rule sets.

1.4.8 SOLUTION: By Theorem 1.4.5, every rule in Figure 1-8 is admissible in HM(X). Of course, so is HMX-GEN. So, every judgement that is derivable via the rules of Figure 1-8 and HMX-GEN is a valid HM(X) judgement.

Conversely, assume $C, \Gamma \vdash t : \sigma$ **(1)** holds in HM(X). We must show that it is derivable via the rules of Figure 1-8 and HMX-GEN. Let us write $\sigma = \forall \bar{X}[D].T$, where $\bar{X} \# ftv(C, \Gamma)$ **(2)**. By HMX-INST and (1), the judgement $C \wedge D, \Gamma \vdash t : T$ **(3)** holds in HM(X). This judgement involves a monotype, so, by By Theorem 1.4.6, it is derivable via the rules of Figure 1-8. Furthermore, from (3) and (2), HMX-GEN allows deriving $C \wedge \exists\sigma, \Gamma \vdash t : \sigma$ **(4)**. Applying Lemma 1.4.1 to (1) yields $C \Vdash \exists\sigma$, so the judgement (4) may be written $C, \Gamma \vdash t : \sigma$. We have shown that (1) is derivable via the rules of Figure 1-8 and HMX-GEN. In fact, it is possible to apply HMX-GEN only once, at the end of the derivation.

1.5.1 SOLUTION: Within the type system PCB(X), we have

$$\dfrac{\dfrac{\dfrac{}{z_1 \preceq Z \vdash z_1 : Z}\ \textsc{Var} \qquad \dfrac{}{z_2 \preceq Y \vdash z_2 : Y}\ \textsc{Var}}{\mathsf{let}\ z_2 : \forall Z[z_1 \preceq Z].Z\ \mathsf{in}\ z_2 \preceq Y \vdash \mathtt{let}\ z_2 = z_1\ \mathtt{in}\ z_2 : Y}\ \textsc{Let}}{\mathsf{let}\ z_1 : X; z_2 : \forall Z[z_1 \preceq Z].Z\ \mathsf{in}\ z_2 \preceq Y \vdash \lambda z_1.\mathtt{let}\ z_2 = z_1\ \mathtt{in}\ z_2 : X \to Y}\ \textsc{Abs}$$

The type variable Z, which occurs free in the left-hand instance of VAR, is generalized. However, z_2 does *not* receive the type scheme $\forall Z.Z$, which, as suggested earlier, is unsound; instead, it receives the *constrained* type scheme $\forall Z[z_1 \preceq Z].Z$. The latter is more restrictive than the former: indeed, the former claims that z_2 has every type, while the latter only claims that every valid type for z_1 is also a valid type for z_2. Let us now examine the constraint $\mathsf{let}\ z_1 : X; z_2 : \forall Z[z_1 \preceq Z].Z\ \mathsf{in}\ z_2 \preceq Y$, which appears at the root of the derivation. By C-INID and C-IN*, it is equivalent to $\mathsf{let}\ z_1 : X\ \mathsf{in}\ \exists Z.(z_1 \preceq Z \wedge Z \leq Y)$ and to $\exists Z.(X \leq Z \wedge Z \leq Y)$, which by C-EXTRANS is equivalent to $X \leq Y$. Thus, the judgement at the root of the above derivation may be written $X \leq Y \vdash \lambda z_1.\mathtt{let}\ z_2 = z_1\ \mathtt{in}\ z_2 : X \to Y$. In other words, the expression $\mathtt{let}\ z_2 = z_1\ \mathtt{in}\ z_2$ has type $X \to Y$ only under the assumption that X is a subtype of Y, which is sound. Even though LET allows unrestricted generalization of type variables, it remains sound, because the type scheme that it produces typically has *free program identifiers*, such as $\forall Z[z_1 \preceq Z].Z$ above.

?? SOLUTION: Let $X \notin \mathit{ftv}(\Gamma)$ **(1)**. Assume that there exist a satisfiable constraint C and a type T such that $C, \Gamma \vdash t : T$ **(2)** holds. Thanks to (1), we find that, up to a renaming of C and T, we may further assume $X \notin \mathit{ftv}(C, T)$ **(3)**. Then, applying Lemma 1.4.2 to (2), we obtain $C \wedge T = X, \Gamma \vdash t : T$, which by HMX-SUB yields $C \wedge T = X, \Gamma \vdash t : X$ **(4)**. Furthermore, by (3) and C-NAMEEQ, we have $\exists X.(C \wedge T = X) \equiv C$. Because C is satisfiable, this implies that $C \wedge T = X$ is satisfiable as well. As a result, we have found a satisfiable constraint C' such that $C', \Gamma \vdash t : X$ holds.

Now, assume Γ is closed and X is arbitrary. Then, (1) holds, so the previous paragraph proves that, if t is well-typed within Γ, then there exists a satisfiable constraint C' such that $C', \Gamma \vdash t : X$ holds. By the completeness property, we must then have $C' \Vdash [\![\Gamma \vdash t : X]\!]$. Since C' is satisfiable, this implies that $[\![\Gamma \vdash t : X]\!]$ is satisfiable as well. Conversely, if $[\![\Gamma \vdash t : X]\!]$ is satisfiable, then, by the soundness property, t is well-typed within Γ.

1.6.1 SOLUTION: Let $C \vdash t : T$ **(1)** hold, where C is satisfiable **(2)**. We note that our goal (which is to prove that there exists a satisfiable constraint C' such that $C' \vdash t : X$ holds) is preserved by a renaming of X, so we may assume, without loss of generality, $X \notin \mathit{ftv}(C, T)$ **(3)**. Applying SUB to (1) yields $C \wedge T \leq X \vdash$

$\mathtt{t} : \mathtt{X}$ **(4)**. Now, by (3), we have $\exists \mathtt{X}.(\mathtt{T} \leq \mathtt{X}) \equiv \mathsf{true}$, which by (3) again and by C-ExAnd implies $C \equiv \exists \mathtt{X}.(C \wedge \mathtt{T} \leq \mathtt{X})$. Given (2), this proves that $C \wedge \mathtt{T} \leq \mathtt{X}$ is satisfiable. Thus, (4) establishes the goal.

Let us now examine what it means for a closed term $\mathtt{t}$ to be well-typed **(5)**. By definition, (5) is equivalent to the existence of a satisfiable constraint C and a type $\mathtt{T}$ such that $C \vdash \mathtt{t} : \mathtt{T}$ holds **(6)**. By the above, (6) is equivalent to the existence of a satisfiable constraint C such that $C \vdash \mathtt{t} : \mathtt{X}$ holds **(7)**. By the argument that precedes the statement of this exercise, (7) is equivalent to the satisfiability of $[\![\mathtt{t} : \mathtt{X}]\!]$ **(8)**. Last, because no type variable other than $\mathtt{X}$ may appear free in $[\![\mathtt{t} : \mathtt{X}]\!]$, the constraint $\exists \mathtt{X}.[\![\mathtt{t} : \mathtt{X}]\!]$ must be equivalent to either true or false, and (8) is equivalent to the assertion $\exists \mathtt{X}.[\![\mathtt{t} : \mathtt{X}]\!] \equiv \mathsf{true}$.

1.7.11 SOLUTION: Let $\mathcal{E} = \mathtt{let}\ \mathtt{z} = \mathcal{E}_1\ \mathtt{in}\ \mathtt{t}_1$ and $\mathcal{E}_1[\mathtt{t}]/\mu \sqsubseteq \mathcal{E}_1[\mathtt{t}']/\mu'$ **(1)**. Then,

$$
\begin{array}{lll}
 & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![\mathcal{E}[\mathtt{t}]/\mu : \mathtt{T}/M]\!] & \\
= & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ ((\mathsf{let}\ \mathtt{z} : \forall \mathtt{X}[[\![\mathcal{E}_1[\mathtt{t}] : \mathtt{X}]\!]].\mathtt{X}\ \mathsf{in}\ [\![\mathtt{t}_1 : \mathtt{T}]\!]) \wedge [\![\mu : M]\!]) & \textbf{(2)} \\
\equiv & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M; \mathtt{z} : \forall \mathtt{X}[[\![\mathcal{E}_1[\mathtt{t}]/\mu : \mathtt{X}/M]\!]].\mathtt{X}\ \mathsf{in}\ [\![\mathtt{t}_1 : \mathtt{T}]\!] & \textbf{(3)} \\
\equiv & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M; \mathtt{z} : \forall \mathtt{X}[\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![\mathcal{E}_1[\mathtt{t}]/\mu : \mathtt{X}/M]\!]].\mathtt{X}\ \mathsf{in}\ [\![\mathtt{t}_1 : \mathtt{T}]\!] & \textbf{(4)} \\
\Vdash & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M; \mathtt{z} : \forall \mathtt{X}\bar{\mathtt{Y}}[\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M'\ \mathsf{in}\ [\![\mathcal{E}_1[\mathtt{t}']/\mu' : \mathtt{X}/M']\!]].\mathtt{X}\ \mathsf{in}\ [\![\mathtt{t}_1 : \mathtt{T}]\!] & \textbf{(5)}
\end{array}
$$

where (2) is by definition of constraint generation, where $\mathtt{X} \notin \mathit{ftv}(\mathtt{T}, M)$ **(6)**; (3) is by (6), C-LetAnd, and by definition of constraint generation; (4) is by (6) and C-LetDup; (5) follows from (1) and C-LetEx, for some $\bar{\mathtt{Y}}$ and M' such that $\bar{\mathtt{Y}} \# \mathit{ftv}(\mathtt{X}, M)$ **(7)** and $\mathit{ftv}(M') \subseteq \bar{\mathtt{Y}} \cup \mathit{ftv}(M)$ **(8)** and $\mathit{dom}(M') = \mathit{dom}(\mu')$ and M' extends M. Note that (6), (7) and (8) imply $\mathtt{X} \notin \mathit{ftv}(M')$ **(9)**.

At this point, the type variables $\bar{\mathtt{Y}}$, which determine the types of the newly allocated store cells, are universally quantified in the type scheme assigned to $\mathtt{z}$, which is undesirable. We are stuck, because we cannot in general apply C-LetAll to hoist $\exists \bar{\mathtt{Y}}$ out of the let constraint. Let us now assume that, by some external means, we are guaranteed $\bar{\mathtt{Y}} = \varnothing$ **(10)**. Then, we may proceed as follows:

$$
\begin{array}{lll}
\equiv & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M'; \mathtt{z} : \forall \mathtt{X}[\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M'\ \mathsf{in}\ [\![\mathcal{E}_1[\mathtt{t}']/\mu' : \mathtt{X}/M']\!]].\mathtt{X}\ \mathsf{in}\ [\![\mathtt{t}_1 : \mathtt{T}]\!] & \textbf{(11)} \\
\equiv & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M'\ \mathsf{in}\ [\![\mathcal{E}[\mathtt{t}']/\mu' : \mathtt{T}/M']\!] & \textbf{(12)}
\end{array}
$$

where (11) follows from the fact the the memory locations that appear free in $[\![\mathtt{t}_1 : \mathtt{T}]\!]$ are members of $\mathit{dom}(\mu)$, thus are not members of $\mathit{dom}(M') \setminus \mathit{dom}(M)$; (12) is obtained by performing the steps that lead to (4) in reverse.

The requirement that $\bar{\mathtt{Y}}$ be empty, that is, $\mathit{ftv}(M) = \mathit{ftv}(M')$, is classic (Tofte, 1988). How is it enforced? Assume that the left-hand side of every $\mathtt{let}$ construct is required to be a non-expansive expression. By assumptions (ii) and (iii), this invariant is preserved by reduction. So, $\mathcal{E}_1[\mathtt{t}]$ must be non-expansive, which, by assumption (i), guarantees that the reduction step does not allocate new memory cells. Then, μ' is μ, so M' is M.

1.9.2 SOLUTION: We must first ensure that R-ADD respects $\sqsubseteq$ (Definition 1.7.6). Since the rule is pure, it is sufficient to establish that $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\hat{k}_1 \mathbin{\hat{+}} \hat{k}_2 : \mathtt{T}]\!]$ entails $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\widehat{k_1 + k_2} : \mathtt{T}]\!]$. In fact, we have

$$
\begin{array}{lll}
 & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\hat{k}_1 \mathbin{\hat{+}} \hat{k}_2 : \mathtt{T}]\!] & \\
\equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ ([\![\hat{k}_1 : \mathsf{int}]\!] \wedge [\![\hat{k}_2 : \mathsf{int}]\!] \wedge \mathsf{int} \leq \mathtt{T}) & \textbf{(1)} \\
\equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ (\mathsf{int} \leq \mathsf{int} \wedge \mathsf{int} \leq \mathsf{int} \wedge \mathsf{int} \leq \mathtt{T}) & \textbf{(2)} \\
\equiv & \mathsf{int} \leq \mathtt{T} & \textbf{(3)} \\
\equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\widehat{k_1 + k_2} : \mathtt{T}]\!] & \textbf{(4)}
\end{array}
$$

where (1) and (2) are by Exercise 1.9.1; (3) is by C-IN* and by reflexivity of subtyping; (4) is by Exercise 1.9.1 again.

Second, we must check that if the configuration $\mathtt{c}\ \mathtt{v}_1\ \ldots\ \mathtt{v}_k/\mu$ (where $k \geq 0$) is well-typed, then either it is reducible, or $\mathtt{c}\ \mathtt{v}_1\ \ldots\ \mathtt{v}_k$ is a value.

We begin by checking that every value that is well-typed with type int is of the form $\hat{k}$. Indeed, suppose that $\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![\mathtt{v} : \mathsf{int}]\!]$ is satisfiable. Then, $\mathtt{v}$ cannot be a program variable, for a well-typed value must be closed. $\mathtt{v}$ cannot be a memory location m, for otherwise $\mathsf{ref}\ M(m) \leq \mathsf{int}$ would be satisfiable—but the type constructors ref and int are incompatible. $\mathtt{v}$ cannot be $\hat{+}$ or $\hat{+}\ \mathtt{v}'$, for otherwise $\mathsf{int} \to \mathsf{int} \to \mathsf{int} \leq \mathsf{int}$ or $\mathsf{int} \to \mathsf{int} \leq \mathsf{int}$ would be satisfiable—but the type constructors $\to$ and int are incompatible. Similarly, $\mathtt{v}$ cannot be a λ-abstraction. Thus, $\mathtt{v}$ must be of the form $\hat{k}$, for it is the only case left.

Next, we note that, according to the constraint generation rules, if the configuration $\mathtt{c}\ \mathtt{v}_1\ \ldots\ \mathtt{v}_k/\mu$ is well-typed, then a constraint of the form $\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ (\mathtt{c} \preceq \mathtt{X}_1 \to \ldots \to \mathtt{X}_k \to \mathtt{T} \wedge [\![\mathtt{v}_1 : \mathtt{X}_1]\!] \wedge \ldots \wedge [\![\mathtt{v}_k : \mathtt{X}_k]\!])$ is satisfiable. We now reason by cases on $\mathtt{c}$.

∘ *Case* $\mathtt{c}$ is $\hat{k}$. Then, $\Gamma_0(\mathtt{c})$ is int. Because the type constructors int and $\to$ are incompatible with each other, this implies $k = 0$. Since $\hat{k}$ is a constructor, the expression is a value.

∘ *Case* $\mathtt{c}$ is $\hat{+}$. We may assume $k \geq 2$, because otherwise the expression is a value. Then, $\Gamma_0(\mathtt{c})$ is $\mathsf{int} \to \mathsf{int} \to \mathsf{int}$, so, by C-ARROW, the above constraint entails $\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ (\mathtt{X}_1 \leq \mathsf{int} \wedge \mathtt{X}_2 \leq \mathsf{int} \wedge [\![\mathtt{v}_1 : \mathtt{X}_1]\!] \wedge [\![\mathtt{v}_2 : \mathtt{X}_2]\!])$, which, by Lemma 1.6.4, entails $\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ ([\![\mathtt{v}_1 : \mathsf{int}]\!] \wedge [\![\mathtt{v}_2 : \mathsf{int}]\!])$. Thus, $\mathtt{v}_1$ and $\mathtt{v}_2$ are well-typed with type int. By the remark above, they must be integer literals $\hat{k}_1$ and $\hat{k}_2$. As a result, the configuration is reducible by R-ADD.

1.9.6 SOLUTION: We must first ensure that R-REF, R-DEREF and R-ASSIGN respect $\sqsubseteq$ (Definition 1.7.6).

∘ *Case* R-REF. The reduction is $\mathtt{ref}\ \mathtt{v}/\varnothing \longrightarrow m/(m \mapsto \mathtt{v})$, where $m \notin \mathit{fpi}(\mathtt{v})$ **(1)**. Let $\mathtt{T}$ be an arbitrary type. According to Definition 1.7.6, the goal

is to show that there exist a set of type variables $\bar{Y}$ and a store type M' such that $\bar{Y} \# \mathit{ftv}(T)$ and $\mathit{ftv}(M') \subseteq \bar{Y}$ and $\mathit{dom}(M') = \{m\}$ and let Γ_0 in $[\![\mathtt{ref}\ v : T]\!]$ entails $\exists \bar{Y}.\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M'\ \mathsf{in}\ [\![m/(m \mapsto v) : T/M']\!]$. Now, we have

$$\begin{array}{lll} & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathtt{ref}\ v : T]\!] & \\ \equiv & \exists Y.\mathsf{let}\ \Gamma_0\ \mathsf{in}\ (\mathsf{ref}\ Y \leq T \wedge [\![v : Y]\!]) & \textbf{(2)} \\ \equiv & \exists Y.\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M'\ \mathsf{in}\ (m \preceq T \wedge [\![v : M'(m)]\!]) & \textbf{(3)} \\ \equiv & \exists Y.\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M'\ \mathsf{in}\ [\![m/(m \mapsto v) : T/M']\!] & \textbf{(4)} \end{array}$$

where (2) is by Exercise 1.9.1 and by C-InEx; (3) assumes M' is defined as $m \mapsto Y$, and follows from (1), C-InId and C-In*; and (4) is by definition of constraint generation.

Subcase R-Deref. The reduction is $!m/(m \mapsto v) \longrightarrow v/(m \mapsto v)$. Let T be an arbitrary type and let M be a store type of domain $\{m\}$. We have

$$\begin{array}{lll} & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![!m/(m \mapsto v) : T/M]\!] & \\ \equiv & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ \exists Y.(\mathsf{ref}\ M(m) \leq \mathsf{ref}\ Y \wedge Y \leq T \wedge [\![v : M(m)]\!]) & \textbf{(1)} \\ \equiv & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ \exists Y.(M(m) = Y \wedge Y \leq T \wedge [\![v : M(m)]\!]) & \textbf{(2)} \\ \equiv & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ (M(m) \leq T \wedge [\![v : M(m)]\!]) & \textbf{(3)} \\ \Vdash & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ ([\![v : T]\!] \wedge [\![v : M(m)]\!]) & \textbf{(4)} \\ \equiv & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![v/(m \mapsto v) : T/M]\!] & \textbf{(5)} \end{array}$$

where (1) is by Exercise 1.9.1 and by C-InId; (2) follows from C-ExTrans and from the fact that ref is an invariant type constructor; (3) is by C-NameEq; (4) is by Lemma 1.6.4 and C-Dup; and (5) is again by definition of constraint generation.

∘ *Case* R-Assign. The reduction is $m := v/(m \mapsto v_0) \longrightarrow v/(m \mapsto v)$. Let T be an arbitrary type and let M be a store type of domain $\{m\}$. We have

$$\begin{array}{lll} & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![m := v/(m \mapsto v_0) : T/M]\!] & \\ \Vdash & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![m := v : T]\!] & \textbf{(1)} \\ \equiv & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ \exists Z.(\mathsf{ref}\ M(m) \leq \mathsf{ref}\ Z \wedge [\![v : Z]\!] \wedge Z \leq T) & \textbf{(2)} \\ \equiv & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ \exists Z.(M(m) = Z \wedge Z \leq T \wedge [\![v : Z]\!]) & \textbf{(3)} \\ \equiv & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ (M(m) \leq T \wedge [\![v : M(m)]\!]) & \textbf{(4)} \\ \Vdash & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![v/(m \mapsto v) : T/M]\!] & \textbf{(5)} \end{array}$$

where (1) is by definition of constraint generation; (2) is by Exercise 1.9.1 and C-InId; (3) follows from the fact that ref is an invariant type constructor; (4) is by C-NameEq; and (5) is obtained as in the previous case.

Second, we must check that if the configuration $c\ v_1\ \ldots\ v_k/\mu$ (where $k \geq 0$) is well-typed, then either it is reducible, or $c\ v_1\ \ldots\ v_k$ is a value. We only give a sketch of this proof; see the solution to Exercise 1.9.2 for details of a similar proof.

We begin by checking that every value that is well-typed with a type of the form ref T is a memory location. This assertion relies on the fact that the type constructor ref is isolated.

Next, we note that, according to the constraint generation rules, if the configuration $c\ v_1\ \ldots\ v_k/\mu$ is well-typed, then a constraint of the form let Γ_0; ref M in $(c \preceq X_1 \to \ldots \to X_k \to T \wedge [\![v_1 : X_1]\!] \wedge \ldots \wedge [\![v_k : X_k]\!])$ is satisfiable. We now reason by cases on c.

∘ *Case* c is ref. If $k = 0$, then the expression is a value; otherwise, it is reducible by R-REF.

∘ *Case* c is !. We may assume $k \geq 1$, because otherwise the expression is a value. Then, by definition of $\Gamma_0(!)$, the above constraint entails let Γ_0; ref M in $\exists Y.(\text{ref } Y \to Y \leq X_1 \to \ldots \to X_k \to T \wedge [\![v_1 : X_1]\!])$, which, by C-ARROW, Lemma 1.6.4, and C-INEX, entails $\exists Y.$let Γ_0; ref M in $[\![v_1 : \text{ref } Y]\!]$. Thus, v_1 is well-typed with a type of the form ref Y. By the remark above, v_1 must be a memory location m. Furthermore, because every well-typed configuration is closed, m must be a member of $dom(\mu)$. As a result, the configuration $\text{ref}\ v_1\ \ldots\ v_k/\mu$ is reducible by R-DEREF.

∘ *Case* c is :=. We may assume $k \geq 2$, because otherwise the expression is a value. As above, we check that v_1 must be a memory location and a member of $dom(\mu)$. Thus, the configuration is reducible by R-ASSIGN.

1.9.7 SOLUTION: We must first ensure that R-FIX respects $\sqsubseteq$ (Definition 1.7.6). Since the rule is pure, it is sufficient to establish that let Γ_0 in $[\![\text{fix } v_1\ v_2 : T]\!]$ entails let Γ_0 in $[\![v_1\ (\text{fix } v_1)\ v_2 : T]\!]$. We have

$$
\begin{array}{lll}
 & \text{let } \Gamma_0 \text{ in } [\![\text{fix } v_1\ v_2 : T]\!] & \\
\equiv & \text{let } \Gamma_0 \text{ in } \exists XY.([\![v_1 : (X \to Y) \to (X \to Y)]\!] \wedge [\![v_2 : X]\!] \wedge Y \leq T) & \textbf{(1)} \\
\Vdash & \text{let } \Gamma_0 \text{ in } \exists XY.([\![v_1\ (\text{fix } v_1)\ v_2 : Y]\!] \wedge Y \leq T) & \textbf{(2)} \\
\equiv & \text{let } \Gamma_0 \text{ in } [\![v_1\ (\text{fix } v_1)\ v_2 : T]\!] & \textbf{(3)}
\end{array}
$$

where (1) is by Exercise 1.9.1; (2) follows from the fact that, under the context let Γ_0 in $\exists XY.[]$, the constraint $[\![\text{fix} : ((X \to Y) \to (X \to Y)) \to X \to Y]\!]$ is equivalent to true, so it may be added to the conjunction, and from the fact that $[\![t_1 : T \to T']\!] \wedge [\![t_2 : T]\!]$ entails $[\![t_1\ t_2 : T']\!]$; and (3) follows from Lemma 1.6.5 and C-EX*.

Second, we must check that if the configuration $\text{fix } v_1\ \ldots\ v_k/\mu$ (where $k \geq 0$) is well-typed, then either it is reducible, or $\text{fix } v_1\ \ldots\ v_k$ is a value. This is immediate, for it is a value when $k < 2$, and it is reducible by R-FIX when $k \geq 2$.

We now recall that the construct $\text{letrec f} = \lambda z.t_1 \text{ in } t_2$ provided by ML-the-programming-language may be viewed as syntactic sugar for

$\texttt{let f = fix } (\lambda \texttt{f}.\lambda \texttt{z}.\texttt{t}_1) \texttt{ in } \texttt{t}_2$, and set forth to discover the constraint generation rule that arises out of such a definition. We have

$$\begin{array}{lll} & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\texttt{fix } (\lambda \texttt{f}.\lambda \texttt{z}.\texttt{t}_1) : \texttt{T}]\!] & \\ \equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \texttt{XY}.(\texttt{X} \to \texttt{Y} \leq \texttt{T} \wedge [\![\lambda \texttt{f}.\lambda \texttt{z}.\texttt{t}_1 : (\texttt{X} \to \texttt{Y}) \to (\texttt{X} \to \texttt{Y})]\!]) & \textbf{(1)} \\ \equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \texttt{XY}.(\texttt{X} \to \texttt{Y} \leq \texttt{T} \wedge \mathsf{let}\ \texttt{f} : \texttt{X} \to \texttt{Y}; \texttt{z} : \texttt{X}\ \mathsf{in}\ [\![\texttt{t}_1 : \texttt{Y}]\!]) & \textbf{(2)} \end{array}$$

where (1) is by Exercise 1.9.1 and (2) follows from Lemma 1.6.6. This allows us to write

$$\begin{array}{lll} & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\texttt{let f = fix } (\lambda \texttt{f}.\lambda \texttt{z}.\texttt{t}_1) \texttt{ in } \texttt{t}_2 : \texttt{T}]\!] & \\ \equiv & \mathsf{let}\ \Gamma_0; \texttt{f} : \forall \texttt{Z}[[\![\texttt{fix } (\lambda \texttt{f}.\lambda \texttt{z}.\texttt{t}_1) : \texttt{Z}]\!].\texttt{Z}\ \mathsf{in}\ [\![\texttt{t}_2 : \texttt{T}]\!] & \textbf{(3)} \\ \equiv & \mathsf{let}\ \Gamma_0; \texttt{f} : \forall \texttt{Z}[\exists \texttt{XY}.(\texttt{X} \to \texttt{Y} \leq \texttt{Z} \wedge \mathsf{let}\ \texttt{f} : \texttt{X} \to \texttt{Y}; \texttt{z} : \texttt{X}\ \mathsf{in}\ [\![\texttt{t}_1 : \texttt{Y}]\!])].\texttt{Z} & \\ & \mathsf{in}\ [\![\texttt{t}_2 : \texttt{T}]\!] & \textbf{(4)} \\ \equiv & \mathsf{let}\ \Gamma_0; \texttt{f} : \forall \texttt{XY}[\mathsf{let}\ \texttt{f} : \texttt{X} \to \texttt{Y}; \texttt{z} : \texttt{X}\ \mathsf{in}\ [\![\texttt{t}_1 : \texttt{Y}]\!]].\texttt{X} \to \texttt{Y}\ \mathsf{in}\ [\![\texttt{t}_2 : \texttt{T}]\!] & \textbf{(5)} \end{array}$$

where (3) is by definition of constraint generation; (4) follows from C-LETDUP and from the previous series of equivalences; (5) is by C-LETEX, C-EXTRANS and Lemma 1.3.31.

1.9.23 SOLUTION: We have

$$\begin{array}{lll} & [\![\texttt{match } \texttt{t}_1 \texttt{ with } \texttt{z} \texttt{ .} \texttt{t}_2 : \texttt{T}]\!] & \\ \equiv & \mathsf{let}\ \forall \texttt{XX}'[[\![\texttt{t}_1 : \texttt{X}]\!] \wedge \mathsf{let}\ \texttt{z} : \texttt{X}'\ \mathsf{in}\ [\![\texttt{X} : \texttt{z}]\!]].(\texttt{z} : \texttt{X}')\ \mathsf{in}\ [\![\texttt{t}_2 : \texttt{T}]\!] & \textbf{(1)} \\ \equiv & \mathsf{let}\ \texttt{z} : \forall \texttt{X}'[\exists \texttt{X}.([\![\texttt{t}_1 : \texttt{X}]\!] \wedge \texttt{X} \leq \texttt{X}')].\texttt{X}'\ \mathsf{in}\ [\![\texttt{t}_2 : \texttt{T}]\!] & \textbf{(2)} \\ \equiv & \mathsf{let}\ \texttt{z} : \forall \texttt{X}'[[\![\texttt{t}_1 : \texttt{X}']\!]].\texttt{X}'\ \mathsf{in}\ [\![\texttt{t}_2 : \texttt{T}]\!] & \textbf{(3)} \\ \equiv & [\![\texttt{let z} = \texttt{t}_1 \texttt{ in } \texttt{t}_2 : \texttt{T}]\!] & \textbf{(4)} \end{array}$$

where (1) is by definition of constraint generation for `match`; (2) is by definition of constraint generation for patterns, by C-INID, C-IN*, and C-LETEX; (3) is by Lemma 1.6.5; (4) is by definition of constraint generation for `let`.

1.9.28 SOLUTION: The type scheme $\forall \bar{\texttt{X}}.\texttt{T} \to \texttt{T}$ may be written $\forall \bar{\texttt{X}}.[\texttt{X} \mapsto \texttt{T}](\texttt{X} \to \texttt{X})$. Furthermore, $\bar{\texttt{X}} \# \forall \texttt{X}.\texttt{X} \to \texttt{X}$ holds. Thus, $\forall \bar{\texttt{X}}.\texttt{T} \to \texttt{T}$ is an instance of $\forall \texttt{X}.\texttt{X} \to \texttt{X}$ in the sense of DM-INST′. Since DM-INST′ is an admissible rule for the type system DM, and since it is clear that the identity function $\lambda \texttt{z}.\texttt{z}$ has type $\forall \texttt{X}.\texttt{X} \to \texttt{X}$, it must also have type $\forall \bar{\texttt{X}}.\texttt{T} \to \texttt{T}$. (A more direct proof of this fact would not be difficult.) So, the destructor $(\cdot : \exists \bar{\texttt{X}}.\texttt{T})$ has not only identity semantics, but also an identity type. This shows that our definitions are sound.

Let us now check requirement (i) of Definition 1.7.7. Since R-ANNOTATION is pure, it suffices to show that $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![(\texttt{v} : \exists \bar{\texttt{X}}.\texttt{T}) : \texttt{T}']\!]$ entails $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\texttt{v} :$

$\mathtt{T}'\rrbracket$. Now, we have

$$\begin{array}{lll} & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![(\mathtt{v} : \exists\bar{\mathtt{X}}.\mathtt{T}) : \mathtt{T}']\!] & \\ \equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists\mathtt{X}\bar{\mathtt{X}}.(\mathtt{T} \to \mathtt{T} \leq \mathtt{X} \to \mathtt{T}' \wedge [\![\mathtt{v} : \mathtt{X}]\!]) & (1) \\ \equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists\mathtt{X}\bar{\mathtt{X}}.(\mathtt{X} \leq \mathtt{T} \leq \mathtt{T}' \wedge [\![\mathtt{v} : \mathtt{X}]\!]) & (2) \\ \Vdash & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathtt{v} : \mathtt{T}']\!] & (3) \end{array}$$

where (1) is by definition of constraint generation and by definition of $\Gamma_0((\cdot : \exists\bar{\mathtt{X}}.\mathtt{T}))$; (2) is by C-ARROW; and (3) follows from Lemma 1.6.4 and C-EX*.

1.10.8 SOLUTION: We have

$$\begin{array}{lll} & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists\mathtt{Z}.[\![(\lambda\mathtt{z}.\mathtt{z}\ \hat{+}\ \hat{1} : \forall\mathtt{X}.\mathtt{X} \to \mathtt{X}) : \mathtt{Z}]\!] & \\ \equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists\mathtt{Z}.(\forall\mathtt{X}.[\![\lambda\mathtt{z}.\mathtt{z}\ \hat{+}\ \hat{1} : \mathtt{X} \to \mathtt{X}]\!] \wedge \exists\mathtt{X}.(\mathtt{X} \to \mathtt{X} \leq \mathtt{Z})) & \textbf{(1)} \\ \equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \forall\mathtt{X}.\mathsf{let}\ \mathtt{z} : \mathtt{X}\ \mathsf{in}\ [\![\mathtt{z}\ \hat{+}\ \hat{1} : \mathtt{X}]\!] & \textbf{(2)} \\ \equiv & \forall\mathtt{X}.(\mathsf{int} \to \mathsf{int} \to \mathsf{int} \leq \mathtt{X} \to \mathsf{int} \to \mathtt{X}) & \textbf{(3)} \\ \equiv & \forall\mathtt{X}.(\mathtt{X} = \mathsf{int}) & \textbf{(4)} \\ \equiv & \mathsf{false} & \textbf{(5)} \end{array}$$

where (1) is by definition of constraint generation for universal type annotations; (2) is obtained by restricting the scope of $\exists\mathtt{Z}$ to the second conjunct, then dropping the latter altogether, since it is equivalent to true, and by Lemma 1.6.6; (3) is obtained by definition of constraint generation, by definition of $\Gamma_0(\hat{+})$ and of $\Gamma_0(\hat{1})$, and by a few simple equivalence laws; (4) follows from C-ARROW and antisymmetry of subtyping; (5) follows from the fact that int and (say) $\mathsf{int} \to \mathsf{int}$ have distinct interpretations, since the type constructors int and $\to$ are incompatible. On the other hand, we have

$$\begin{array}{lll} & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists\mathtt{Z}.[\![(\lambda\mathtt{z}.\mathtt{z} : \forall\mathtt{X}.\mathtt{X} \to \mathtt{X}) : \mathtt{Z}]\!] & \\ \equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \forall\mathtt{X}.\mathsf{let}\ \mathtt{z} : \mathtt{X}\ \mathsf{in}\ [\![\mathtt{z} : \mathtt{X}]\!] & \textbf{(1)} \\ \equiv & \forall\mathtt{X}.(\mathtt{X} \leq \mathtt{X}) & \textbf{(2)} \\ \equiv & \mathsf{true} & \textbf{(3)} \end{array}$$

where (1) is obtained as above; (2) by definition of constraint generation, C-INID and C-IN*; (3) is by reflexivity of subtyping.

1.10.9 SOLUTION: Under the naïve constraint generation rule for universal type variable introduction, the constraint $[\![\forall\mathtt{X}.(\lambda\mathtt{z}.\mathtt{z} : \mathtt{X} \to \mathtt{X}) : \mathtt{Z}]\!]$ is equivalent to $\forall\mathtt{X}.([\![\lambda\mathtt{z}.\mathtt{z} : \mathtt{X} \to \mathtt{X}]\!] \wedge \mathtt{X} \to \mathtt{X} \leq \mathtt{Z})$. Since the first conjunct is a tautology, this is in turn equivalent to $\forall\mathtt{X}.(\mathtt{X} \to \mathtt{X} \leq \mathtt{Z})$. In a nondegenerate free term model where subtyping is interpreted as equality, this constraint is unsatisfiable. In a non-structural subtyping model equipped with a least type $\bot$ and a greatest type $\top$, it is equivalent to $\bot \to \top \leq \mathtt{Z}$. This is a pretty restrictive

constraint: since no value has type $\bot$, a function whose type is (a supertype of) $\bot \to \top$ cannot ever be invoked at runtime. This situation is clearly unsatisfactory. Checking that $\forall X.[\![\lambda z.z : X \to X]\!]$ holds was indeed part of our intent, but constraining Z to be a supertype of $X \to X$ for *every* X was not.

1.10.10 SOLUTION: Let $\bar{X} \supseteq \mathit{ftv}(T)$ **(1)** and $\bar{X} \# \mathit{ftv}(t)$ **(2)**. We may assume, *w.l.o.g.*, $\bar{X} \# \mathit{ftv}(T')$ **(3)**. By (1), (2), (3), and by definition of constraint generation for local universal type annotations, $[\![(t : \forall \bar{X}.T) : T']\!]$ is well-defined and is $\forall \bar{X}.[\![t : T]\!] \wedge \exists \bar{X}.(T \le T')$ **(4)**. By (3) and by definition of constraint generation for introduction of universal type variables and for general type annotations, $[\![\forall \bar{X}.(t : T) : T']\!]$ is $\forall \bar{X}.\exists Z.([\![t : T]\!] \wedge T \le Z) \wedge \exists \bar{X}.([\![t : T]\!] \wedge T \le T')$, where Z is fresh, which we may immediately simplify to $\forall \bar{X}.[\![t : T]\!] \wedge \exists \bar{X}.([\![t : T]\!] \wedge T \le T')$ **(5)**. Using C-EXAND and Lemma 1.10.1, it is straightforward to check that (4) and (5) are equivalent.

1.10.12 SOLUTION: We have

$$\begin{array}{lll} & \exists Z.[\![\lambda z.\forall X.(z : X) : Z]\!] & \\ \Vdash & \exists Z_1 Z_2.\mathsf{let}\ z : Z_1\ \mathsf{in}\ [\![\forall X.(z : X) : Z_2]\!] & \textbf{(1)} \\ \Vdash & \exists Z_1.\forall X.(Z_1 \le X) & \textbf{(2)} \end{array}$$

where (1) is by definition of constraint generation for λ-abstractions, dropping the constraint that relates Z, Z_1, and Z_2; (2) is by definition of constraint generation for universal type variable introduction, this time dropping information about Z_2. Now, in a nondegenerate equality model, the constraint (2) is equivalent to false. In fact, for (2) to be satisfiable, the interpretation of subtyping must admit a least element $\bot$. We now see that $[\![\lambda z.\forall X.(z : X) : Z]\!]$ is a very restrictive constraint. Indeed, it requires z to have every type at once. Because z is λ-bound—hence monomorphic—it must in fact have type $\bot$. On the other hand, we have

$$\begin{array}{lll} & \exists Z.[\![\forall X.\lambda z.(z : X) : Z]\!] & \\ \equiv & \forall X.\exists Z.[\![\lambda z.(z : X) : Z]\!] & \textbf{(1)} \\ \equiv & \forall X.\exists Z Z_1 Z_2.(Z_1 \le X \wedge X \le Z_2 \wedge Z_1 \to Z_2 \le Z) & \textbf{(2)} \\ \equiv & \mathsf{true} & \textbf{(3)} \end{array}$$

where (1) is by definition of constraint generation for universal type variable introduction, dropping the second conjunct, which is entailed by the first; (2) is by Lemma 1.6.6, by definition of constraint generation for general type annotations, and by a few simple equivalence laws; (3) follows from C-NAMEEQ and the witness substitution $[Z_1 \mapsto X, Z_2 \mapsto X, Z \mapsto (X \to X)]$.

1.10.13 SOLUTION: We have

$$
\begin{array}{lll}
 & [\![\texttt{letrec f} : \texttt{S} = \lambda\texttt{z}.\texttt{t}_1 \texttt{ in } \texttt{t}_2 : \texttt{T}]\!] & \\
\equiv & \mathsf{let}\ \texttt{f} : \forall\texttt{X}[[\![\texttt{fix f} : \texttt{S}.\lambda\texttt{z}.\texttt{t}_1 : \texttt{X}]\!].\texttt{X}\ \mathsf{in}\ [\![\texttt{t}_2 : \texttt{T}]\!] & \textbf{(1)} \\
\equiv & \mathsf{let}\ \texttt{f} : \forall\texttt{X}[\mathsf{let}\ \texttt{f} : \texttt{S}\ \mathsf{in}\ [\![\lambda\texttt{z}.\texttt{t}_1 : \texttt{S}]\!] \wedge \texttt{S} \preceq \texttt{X}].\texttt{X}\ \mathsf{in}\ [\![\texttt{t}_2 : \texttt{T}]\!] & \textbf{(2)} \\
\equiv & \mathsf{let}\ \texttt{f} : \texttt{S}\ \mathsf{in}\ [\![\lambda\texttt{z}.\texttt{t}_1 : \texttt{S}]\!] \wedge \mathsf{let}\ \texttt{f} : \forall\texttt{X}[\texttt{S} \preceq \texttt{X}].\texttt{X}\ \mathsf{in}\ [\![\texttt{t}_2 : \texttt{T}]\!] & \textbf{(3)} \\
\equiv & \mathsf{let}\ \texttt{f} : \texttt{S}\ \mathsf{in}\ ([\![\lambda\texttt{z}.\texttt{t}_1 : \texttt{S}]\!] \wedge [\![\texttt{t}_2 : \texttt{T}]\!]) & \textbf{(4)}
\end{array}
$$

where (1) is by definition of the `letrec` syntactic sugar and by the definition of constraint generation for `let` constructs; we have $\texttt{X} \notin \mathit{ftv}(\texttt{S}, \texttt{t}_1)$; (2) is by definition of constraint generation for `fix`; (3) is by C-LETAND; (4) follows from the equivalence between the type schemes $\forall\texttt{X}[\texttt{S} \preceq \texttt{X}].\texttt{X}$ and `S`—which itself is a direct consequence of C-EXTRANS—and from C-INAND.

1.11.5 SOLUTION: We let the reader check that `X` must have kind $\star.\mathit{Type}$ and `Y` must have kind $\star.\mathit{Row}(\{\ell\})$. The type with all superscripts made explicit is

$$\texttt{X} \rightarrow^{\mathit{Type}} \texttt{"}\,(\ell^{\star,\mathit{Row}(\varnothing)} : \mathsf{int}^{\mathit{Type}}\ ;\ (\texttt{Y} \rightarrow^{\mathit{Row}(\{\ell\})} \partial^{\star,\mathit{Row}(\{\ell\})}\texttt{X})).$$

In this case, because the type constructor " occurs on the right-hand side of the toplevel arrow, it is possible to guess that the type must have kind $\star.\mathit{Type}$. There are cases where it is not possible to guess the kind of a type, because it may have several kinds; consider, for instance, $\partial\mathsf{int}$.

1.11.28 SOLUTION: For the sake of generality, we perform the proof in the presence of subtyping, that is, we do not assume that subtyping is interpreted as equality. We formulate some hypotheses about the interpretation of subtyping: the type constructors $(\ell : \cdot\ ;\ \cdot)$, ∂, and " must be covariant; the type constructors $\rightarrow$ and " must be isolated.

We begin with a preliminary fact: *if the domain of* `V` *is* $\{\ell_1, \ldots, \ell_n\}$, *where* $\ell_1 < \ldots < \ell_n$, *then the constraint* $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\{\texttt{V};\texttt{v}\} : \texttt{T}]\!]$ *is equivalent to* $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists\texttt{Z}_1 \ldots \texttt{Z}_n\texttt{Z}.(\bigwedge_{\texttt{i}=1}^{\texttt{n}} [\![\texttt{V}(\ell_\texttt{i}) : \texttt{Z}_\texttt{i}]\!] \wedge [\![\texttt{v} : \texttt{Z}]\!] \wedge \texttt{"}\,(\ell_1 : \texttt{Z}_1; \ldots; \ell_\texttt{n} : \texttt{Z}_\texttt{n}; \partial\texttt{Z}) \leq \texttt{T})$. We let the reader check this fact using the constraint generation rules, the definition of Γ_0 and rule C-INID, and the above covariance hypotheses. We note that, by C-ROW-LL, the above constraint is invariant under a permutation of the labels $\ell_1, \ldots, \ell_n$, so the above fact still holds when the hypothesis $\ell_1 < \ldots < \ell_n$ is removed.

We now prove that rules R-UPDATE, R-ACCESS-1, and R-ACCESS-2 enjoy subject reduction (Definition 1.7.6). Because the store is not involved, the goal is to establish that $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\texttt{t} : \texttt{T}]\!]$ entails $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\texttt{t}' : \texttt{T}]\!]$, where `t` is the redex and $\texttt{t}'$ is the reduct.

∘ *Case* R-UPDATE. We have:

$$\begin{array}{ll} & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\{\mathtt{V};\mathtt{v}\}\ \mathsf{with}\ \ell = \mathtt{v}'\} : \mathtt{T}]\!] \\ \equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \mathtt{X}\mathtt{X}'\mathtt{Y}.([\![\{\mathtt{V};\mathtt{v}\} : \tilde{}\ (\ell : \mathtt{X}\ ;\ \mathtt{Y})]\!] \wedge [\![\mathtt{v}' : \mathtt{X}']\!] \wedge \tilde{}\ (\ell : \mathtt{X}'\ ;\ \mathtt{Y}) \leq \mathtt{T}) \quad \textbf{(1)} \\ \equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \mathtt{X}\mathtt{X}'\mathtt{Y}\mathtt{Z}_1 \ldots \mathtt{Z}_n\mathtt{Z}.(\textstyle\bigwedge_{i=1}^{n} [\![\mathtt{V}(\ell_i) : \mathtt{Z}_i]\!] \wedge [\![\mathtt{v} : \mathtt{Z}]\!] \quad \textbf{(2)} \\ & \qquad\qquad \wedge \tilde{}\ (\ell_1 : \mathtt{Z}_1; \ldots; \ell_n : \mathtt{Z}_n; \partial\mathtt{Z}) \leq \tilde{}\ (\ell : \mathtt{X}\ ;\ \mathtt{Y}) \\ & \qquad\qquad \wedge [\![\mathtt{v}' : \mathtt{X}']\!] \wedge \tilde{}\ (\ell : \mathtt{X}'\ ;\ \mathtt{Y}) \leq \mathtt{T}) \end{array}$$

where (1) is by Exercise 1.9.1, and (2) follows from the preliminary fact and from C-EXAND, provided $\{\ell_1, \ldots, \ell_n\}$ is the domain of $\mathtt{V}$. We now distinguish two subcases:

Subcase $\ell \in dom(\mathtt{V})$. We may assume, *w.l.o.g.*, that ℓ is ℓ_1. Then, by our covariance hypotheses, the subconstraint in the second line of (2) entails $(\ell_2 : \mathtt{Z}_2; \ldots; \ell_n : \mathtt{Z}_n; \partial\mathtt{Z}) \leq \mathtt{Y}$, which in turn entails $\tilde{}\ (\ell_1 : \mathtt{X}'; \ell_2 : \mathtt{Z}_2; \ldots; \ell_n : \mathtt{Z}_n; \partial\mathtt{Z}) \leq \tilde{}\ (\ell : \mathtt{X}'\ ;\ \mathtt{Y})$. By transitivity of subtyping, the subconstraint in the second and third lines of (2) entails $\tilde{}\ (\ell_1 : \mathtt{X}'; \ell_2 : \mathtt{Z}_2; \ldots; \ell_n : \mathtt{Z}_n; \partial\mathtt{Z}) \leq \mathtt{T}$. By this remark and by C-EX*, (2) entails

$$\begin{array}{ll} & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \mathtt{X}'\mathtt{Z}_2 \ldots \mathtt{Z}_n\mathtt{Z}.([\![\mathtt{v}' : \mathtt{X}']\!] \wedge \textstyle\bigwedge_{i=2}^{n} [\![\mathtt{V}(\ell_i) : \mathtt{Z}_i]\!] \wedge [\![\mathtt{v} : \mathtt{Z}]\!] \quad \textbf{(3)} \\ & \qquad\qquad \wedge \tilde{}\ (\ell_1 : \mathtt{X}'; \ell_2 : \mathtt{Z}_2; \ldots; \ell_n : \mathtt{Z}_n; \partial\mathtt{Z}) \leq \mathtt{T}) \end{array}$$

which by our preliminary fact is precisely $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\{\mathtt{V}[\ell \mapsto \mathtt{v}'];\mathtt{v}\} : \mathtt{T}]\!]$.

Subcase $\ell \notin dom(\mathtt{V})$. By C-ROW-DL and C-ROW-LL, the term $(\ell_1 : \mathtt{Z}_1; \ldots; \ell_n : \mathtt{Z}_n; \partial\mathtt{Z})$ may be replaced with $(\ell : \mathtt{Z}; \ell_1 : \mathtt{Z}_1; \ldots; \ell_n : \mathtt{Z}_n; \partial\mathtt{Z})$. Thus, reasoning as in the previous subcase, we find that (2) entails

$$\begin{array}{ll} & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \mathtt{X}'\mathtt{Z}_1 \ldots \mathtt{Z}_n\mathtt{Z}.([\![\mathtt{v}' : \mathtt{X}']\!] \wedge \textstyle\bigwedge_{i=1}^{n} [\![\mathtt{V}(\ell_i) : \mathtt{Z}_i]\!] \wedge [\![\mathtt{v} : \mathtt{Z}]\!] \quad \textbf{(4)} \\ & \qquad\qquad \wedge \tilde{}\ (\ell_1 : \mathtt{X}'; \ell_1 : \mathtt{Z}_1; \ldots; \ell_n : \mathtt{Z}_n; \partial\mathtt{Z}) \leq \mathtt{T}) \end{array}$$

which by our preliminary fact is precisely $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\{\mathtt{V}[\ell \mapsto \mathtt{v}'];\mathtt{v}\} : \mathtt{T}]\!]$.

∘ *Cases* R-ACCESS-1, R-ACCESS-2. We have:

$$\begin{array}{ll} & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\{\mathtt{V};\mathtt{v}\}.\{\ell\} : \mathtt{T}]\!] \\ \equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \mathtt{X}\mathtt{Y}.([\![\{\mathtt{V};\mathtt{v}\} : \tilde{}\ (\ell : \mathtt{X}\ ;\ \mathtt{Y})]\!] \wedge \mathtt{X} \leq \mathtt{T}) \quad \textbf{(1)} \\ \equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \mathtt{X}\mathtt{Y}\mathtt{Z}_1 \ldots \mathtt{Z}_n\mathtt{Z}.(\textstyle\bigwedge_{i=1}^{n} [\![\mathtt{V}(\ell_i) : \mathtt{Z}_i]\!] \wedge [\![\mathtt{v} : \mathtt{Z}]\!] \quad \textbf{(2)} \\ & \qquad\qquad \wedge \tilde{}\ (\ell_1 : \mathtt{Z}_1; \ldots; \ell_n : \mathtt{Z}_n; \partial\mathtt{Z}) \leq \tilde{}\ (\ell : \mathtt{X}\ ;\ \mathtt{Y}) \\ & \qquad\qquad \wedge \mathtt{X} \leq \mathtt{T}) \end{array}$$

where (1) is by Exercise 1.9.1, and (2) follows from the preliminary fact and from C-EXAND, provided $\{\ell_1, \ldots, \ell_n\}$ is the domain of $\mathtt{V}$. We now distinguish two subcases:

Subcase $\ell \in dom(\mathtt{V})$, *i.e.*, (R-ACCESS-1). We may assume, *w.l.o.g.*, that ℓ is ℓ_1. Then, by our covariance hypotheses, the subconstraint in the second line

of (2) entails $\mathtt{Z}_1 \leq \mathtt{X}$. By transitivity of subtyping, by Lemma 1.6.4, and by C-EX*, we find that (2) entails $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathtt{V}(\ell) : \mathtt{T}]\!]$.

Subcase $\ell \notin dom(\mathtt{V})$, *i.e.*, (R-ACCESS-2). By C-ROW-DL and C-ROW-LL, the term $(\ell_1 : \mathtt{Z}_1; \ldots; \ell_n : \mathtt{Z}_n; \partial\mathtt{Z})$ may be replaced with $(\ell : \mathtt{Z}; \ell_1 : \mathtt{Z}_1; \ldots; \ell_n : \mathtt{Z}_n; \partial\mathtt{Z})$. Thus, reasoning as in the previous subcase, we find that (2) entails $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathtt{v} : \mathtt{T}]\!]$.

Before attacking the proof of the progress property, let us briefly check that every value $\mathtt{v}$ that is well-typed with type $''\ \mathtt{T}$ must be a record value, that is, must be of the form $\{\mathtt{V}; \mathtt{w}\}$. Indeed, assume that $\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![\mathtt{v} : ''\ \mathtt{T}]\!]$ is satisfiable. Then, $\mathtt{v}$ cannot be a program variable, for a well-typed value must be closed. Furthermore, $\mathtt{v}$ cannot be a memory location m, because $\mathsf{ref}\ M(m) \leq ''\ \mathtt{T}$ is unsatisfiable: indeed, the type constructors ref and $''$ are incompatible (recall that $''$ is isolated). Similarly, $\mathtt{v}$ cannot be a partially applied constant or a λ-abstraction, because $\mathtt{T}' \to \mathtt{T}'' \leq ''\ \mathtt{T}$ is unsatisfiable. Thus, $\mathtt{v}$ must be a fully applied constructor. Since the only constructors in the language are the record constructors $\{\}_L$, $\mathtt{v}$ must be a record value. (If there were other constructors in the language, they could be ruled out as well, provided their return types are incompatible with $''$.)

We must now prove that if the configuration $\mathtt{c}\ \mathtt{v}_1\ \ldots\ \mathtt{v}_k/\mu$ is is well-typed, then either it is reducible, or $\mathtt{c}\ \mathtt{v}_1\ \ldots\ \mathtt{v}_k$ is a value. By the well-typedness hypothesis, a constraint of the form $\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![\mathtt{c}\ \mathtt{v}_1\ \ldots\ \mathtt{v}_k : \mathtt{T}]\!]$ is satisfiable.

∘ *Case* $\mathtt{c}$ is $\{\}_L$. If k is less than or equal to $n + 1$, where n is the cardinal of L, then $\mathtt{c}\ \mathtt{v}_1\ \ldots\ \mathtt{v}_k$ is a value. Otherwise, unfolding the above constraint, we find that it cannot be satisfiable, because $''$ and $\to$ are incompatible; this yields a contradiction.

∘ *Case* $\mathtt{c}$ is $\{\cdot\ \mathsf{with}\ \ell = \cdot\}$. Analogous to the next case.

∘ *Case* $\mathtt{c}$ is $\cdot.\{\ell\}$. If $k = 0$, then $\mathtt{c}\ \mathtt{v}_1\ \ldots\ \mathtt{v}_k$ is a value. Assume $k \geq 1$. Then, the constraint $\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![\mathtt{c}\ \mathtt{v}_1 : \mathtt{T}]\!]$ is satisfiable. By Exercise 1.9.1, this implies that $\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![\mathtt{v}_1 : ''\ (\ell : \mathtt{X}\ ;\ \mathtt{Y})]\!]$ is satisfiable. Thus, $\mathtt{v}_1$ must be a record value, and the configuration is reducible by R-ACCESS-1 or R-ACCESS-2.

1.11.30 SOLUTION: Because establishing progress for this extension is straightforward (see Exercise 1.11.28), we concentrate on subject reduction. Our covariance hypotheses are as in Exercise 1.11.28.

∘ *Case* R-APPLY-1. We have:

$$
\begin{array}{ll}
& \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathsf{apply}\ \{V;v\}\{V';v'\} : T]\!] \\
\equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists XY.([\![\{V;v\} : \Cup (X \to Y)]\!] \wedge [\![\{V';v'\} : \Cup X]\!] \wedge \Cup Y \leq T) \qquad \textbf{(1)} \\
\equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists XYZ_1 \ldots Z_n Z Z'_1 \ldots Z'_n Z'.(\qquad \textbf{(2)} \\
& \qquad \textstyle\bigwedge_{i=1}^{n} [\![V(\ell_i) : Z_i]\!] \wedge [\![v : Z]\!] \wedge \\
& \qquad \Cup (\ell_1 : Z_1; \ldots; \ell_n : Z_n; \partial Z) \leq \Cup (X \to Y) \wedge \\
& \qquad \textstyle\bigwedge_{i=1}^{n} [\![V'(\ell_i) : Z'_i]\!] \wedge [\![v' : Z']\!] \wedge \\
& \qquad \Cup (\ell_1 : Z'_1; \ldots; \ell_n : Z'_n; \partial Z') \leq \Cup X \wedge \\
& \qquad \Cup Y \leq T)
\end{array}
$$

where (1) is by Exercise 1.9.1, and (2) follows from the preliminary fact in Exercise 1.11.28 and from C-EXAND, provided $\{\ell_1, \ldots, \ell_n\}$ is the common domain of V and V'. Now, consider the constraint $\Cup (\ell_1 : Z_1; \ldots; \ell_n : Z_n; \partial Z) \leq \Cup (X \to Y)$ in the third line of (2). Any ground assignment that satisfies it must map each of $Z_1, \ldots, Z_n, Z$ to a subtype of an arrow type. Together with Lemma 1.6.4, this allows *expanding* each of $Z_1, \ldots, Z_n, Z$ into an arrow type. In other words, (2) is equivalent to

$$
\begin{array}{l}
\mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists XYZ^1_1 Z^2_1 \ldots Z^1_n Z^2_n Z^1 Z^2 Z'_1 \ldots Z'_n Z'.(\qquad \textbf{(3)} \\
\qquad \textstyle\bigwedge_{i=1}^{n} [\![V(\ell_i) : Z^1_i \to Z^2_i]\!] \wedge [\![v : Z^1 \to Z^2]\!] \wedge \\
\qquad \Cup (\ell_1 : Z^1_1 \to Z^2_1; \ldots; \ell_n : Z^1_n \to Z^2_n; \partial Z^1 \to Z^2) \leq \Cup (X \to Y) \wedge \\
\qquad \textstyle\bigwedge_{i=1}^{n} [\![V'(\ell_i) : Z'_i]\!] \wedge [\![v' : Z']\!] \wedge \\
\qquad \Cup (\ell_1 : Z'_1; \ldots; \ell_n : Z'_n; \partial Z') \leq \Cup X \wedge \\
\qquad \Cup Y \leq T)
\end{array}
$$

Let us now consider the third line in (3). By C-ROW-GL, its left-hand side may be written $\Cup ((\ell_1 : Z^1_1; \ldots; \ell_n : Z^1_n; \partial Z^1) \to (\ell_1 : Z^2_1; \ldots; \ell_n : Z^2_n; \partial Z^2))$. By covariance of $\Cup$ and by contra- and covariance of $\to$, the third line thus entails $X \leq (\ell_1 : Z^1_1; \ldots; \ell_n : Z^1_n; \partial Z^1)$ and $(\ell_1 : Z^2_1; \ldots; \ell_n : Z^2_n; \partial Z^2) \leq Y$. Combining this with the last two lines in (3) and again using our covariance hypotheses, we find that (3) entails

$$
\begin{array}{l}
\mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists Z^1_1 Z^2_1 \ldots Z^1_n Z^2_n Z^1 Z^2 Z'_1 \ldots Z'_n Z'.(\qquad \textbf{(4)} \\
\qquad \textstyle\bigwedge_{i=1}^{n} [\![V(\ell_i) : Z^1_i \to Z^2_i]\!] \wedge [\![v : Z^1 \to Z^2]\!] \wedge \\
\qquad \textstyle\bigwedge_{i=1}^{n} Z'_i \leq Z^1_i \wedge Z' \leq Z^1 \wedge \\
\qquad \textstyle\bigwedge_{i=1}^{n} [\![V'(\ell_i) : Z'_i]\!] \wedge [\![v' : Z']\!] \wedge \\
\qquad \Cup (\ell_1 : Z^2_1; \ldots; \ell_n : Z^2_n; \partial Z^2) \leq T)
\end{array}
$$

By Lemma 1.6.5, (4) is equivalent to

$$
\begin{array}{l}
\mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists Z^1_1 Z^2_1 \ldots Z^1_n Z^2_n Z^1 Z^2.(\qquad \textbf{(5)} \\
\qquad \textstyle\bigwedge_{i=1}^{n} [\![V(\ell_i) : Z^1_i \to Z^2_i]\!] \wedge [\![v : Z^1 \to Z^2]\!] \wedge \\
\qquad \textstyle\bigwedge_{i=1}^{n} [\![V'(\ell_i) : Z^1_i]\!] \wedge [\![v' : Z^1]\!] \wedge \\
\qquad \Cup (\ell_1 : Z^2_1; \ldots; \ell_n : Z^2_n; \partial Z^2) \leq T)
\end{array}
$$

which, by definition of constraint generation for applications, is equivalent to

$$\begin{array}{l}\mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \mathtt{Z}_1^2 \ldots \mathtt{Z}_n^2\, \mathtt{Z}^2.(\\ \qquad \bigwedge_{i=1}^{n} [\![\mathtt{V}(\ell_i)\ \mathtt{V}'(\ell_i) : \mathtt{Z}_i^2]\!] \wedge [\![\mathtt{v}\ \mathtt{v}' : \mathtt{Z}^2]\!] \wedge \\ \qquad \text{''}\,(\ell_1 : \mathtt{Z}_1^2; \ldots; \ell_n : \mathtt{Z}_n^2; \partial \mathtt{Z}^2) \leq \mathtt{T})\end{array} \tag{6}$$

To conclude, by the preliminary fact of Exercise 1.11.28, (6) is exactly $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\{\mathtt{V}\ \mathtt{V}'; \mathtt{v}\ \mathtt{v}'\} : \mathtt{T}]\!]$.

◦ *Cases* R-APPLY-2, R-APPLY-3. In both rules, part of the term is preserved, and may be ignored in the subject reduction proof. As a result, in order to establish subject reduction for both rules, it suffices to establish subject reduction for the (imaginary) rule $\{\mathtt{V}; \mathtt{v}\} \longrightarrow \{\mathtt{V}[\ell \mapsto \mathtt{v}]; \mathtt{v}\}$ where $\ell \notin dom(\mathtt{V})$. This is a straightforward consequence of the preliminary fact of Exercise 1.11.28 and of C-ROW-DL; the details are left to the reader.

1.11.34 SOLUTION: To make extension strict, it suffices to restrict its binding in the initial environment Γ_0, as follows:

$$\langle \cdot\ \mathsf{with}\ \ell = \cdot \rangle : \forall \mathtt{XY}.\text{''}\,(\ell : \mathsf{abs}\ ;\ \mathtt{Y}) \to \mathtt{X} \to \text{''}\,(\ell : \mathsf{pre}\ \mathtt{X}\ ;\ \mathtt{Y}).$$

The new binding, which is less general than the former, requires the field ℓ to be absent in the input record. The operational semantics need not be modified, since strict extension coincides with free extension when it is defined.

Defining the operational semantics of (free) restriction is left to the reader. Its binding in the initial environment should be:

$$\cdot \setminus \langle \ell \rangle : \forall \mathtt{XY}.\text{''}\,(\ell : \mathtt{X}\ ;\ \mathtt{Y}) \to \text{''}\,(\ell : \mathsf{abs}\ ;\ \mathtt{Y})$$

In principle, there is no need to guess this binding: it may be discovered through the encoding of finite records in terms of full records (Exercise 1.11.33). Strict restriction, which requires the field to be present in the input record, may be assigned the following type scheme:

$$\cdot \setminus \langle \ell \rangle : \forall \mathtt{XY}.\text{''}\,(\ell : \mathsf{pre}\ \mathtt{X}\ ;\ \mathtt{Y}) \to \text{''}\,(\ell : \mathsf{abs}\ ;\ \mathtt{Y})$$

1.11.35 SOLUTION: The informal sentence "supplying a record with more fields in a context where a record with fewer fields is expected" may be understood as "providing an argument of type $\text{''}\,(\ell : \mathsf{pre}\ \mathtt{T}\ ;\ \mathtt{T}')$ to a function whose domain type is $\text{''}\,(\ell : \mathsf{abs}\ ;\ \mathtt{T}')$," or, more generally, as "writing a program whose well-typedness requires some constraint of the form $\text{''}\,(\ell{:}\mathsf{pre}\ \mathtt{T}\ ;\ \mathtt{T}') \leq \text{''}\,(\ell{:}\mathsf{abs}\ ;\ \mathtt{T}')$ to be satisfiable." Now, in a nonstructural subtyping order where $\mathsf{pre} \leqslant \mathsf{abs}$ holds, such a constraint is equivalent to true. On the opposite, if subtyping is

interpreted as equality, then such a constraint is equivalent to false. In other words, it is the law $\mathsf{pre}\ \mathtt{T} \leq \mathsf{abs} \equiv \mathsf{true}$ that gives rise to width subtyping.

It is worth drawing a comparison with the way width subtyping is defined in type systems that do not have rows. In such type systems, a record type is of the form $\{\ell_1 : \mathtt{T}_1; \ldots; \ell_n : \mathtt{T}_n\}$. Let us forget about the types $\mathtt{T}_1, \ldots, \mathtt{T}_n$, because they describe the contents of fields, not their presence, and are thus orthogonal to the issue at hand. Then, a record type is a set $\{\ell_1, \ldots, \ell_n\}$, and width subtyping is obtained by letting subtyping coincide with (the reverse of) set containment. In a type system that exploits rows, on the other hand, a record type is a total mapping from row labels to either pre or abs. (Because we are ignoring $\mathtt{T}_1, \ldots, \mathtt{T}_n$, let us temporarily imagine that pre is a nullary type constructor.) The above record type is then written $\{\ell_1 : \mathsf{pre}; \ldots; \ell_n : \mathsf{pre}; \partial\mathsf{abs}\}$. In other words, a set is now encoded as its characteristic function. Width subtyping is obtained by letting $\mathsf{pre} \leqslant \mathsf{abs}$ and by lifting this ordering, pointwise, to rows (which corresponds to our convention that rows are covariant).

1.11.36 SOLUTION: Because the subject reduction proof is analogous to that found in Exercise 1.11.28, we concentrate on progress. The proof is analogous to that of Exercise 1.11.28. However, access may now fail: thus, we must check not only that the argument to $\cdot.\langle\ell\rangle$ is a record value, but also that it contains a field labelled ℓ.

As in Exercise 1.11.28, we assume that the type constructors $\to$ and $''$ are isolated. Furthermore, we assume that every constraint of the form $\mathsf{abs} \leq \mathsf{pre}\ \mathtt{T}$ is equivalent to false. Thus, our proof is valid with respect to both of the interpretations of subtyping given in Remark 1.11.11. This assumption may be summed up as follows: in order for progress to hold, it does not matter whether depth subtyping is allowed or not, *i.e.*, whether subsumption does or does not allow an existing record field to *disappear*. What matters is that subsumption must not allow a nonexistent field to *appear* out of thin air.

The proof that every value $\mathtt{v}$ that is well-typed with type $''\ \mathtt{T}$ must be a record value is carried out as in Exercise 1.11.28. Let us further prove that, if the record value $\langle\mathtt{V}\rangle$ is well-typed with type $''\ (\ell : \mathsf{pre}\ \mathtt{T}\ ;\ \mathtt{T}')$, then ℓ must be in the domain of $\mathtt{V}$. Indeed, assume that $\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![\langle\mathtt{V}\rangle : ''\ (\ell : \mathsf{pre}\ \mathtt{T}\ ;\ \mathtt{T}')]\!]$ **(1)** is satisfiable. Because (1) is equivalent to

$$\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ \exists \mathtt{Z}_1 \ldots \mathtt{Z}_n.(\textstyle\bigwedge_{i=1}^{n} [\![\mathtt{V}(\ell_i) : \mathtt{Z}_i]\!] \wedge\ '' (\ell_1 : \mathtt{Z}_1; \ldots; \ell_n : \mathtt{Z}_n; \partial\mathsf{abs}) \leq\ '' (\ell : \mathsf{pre}\ \mathtt{T}\ ;\ \mathtt{T}')),$$

where $\{\ell_1, \ldots, \ell_n\}$ is the domain of $\mathtt{V}$, there follows that the subconstraint $''\ (\ell_1 : \mathtt{Z}_1; \ldots; \ell_n : \mathtt{Z}_n; \partial\mathsf{abs}) \leq\ ''\ (\ell : \mathsf{pre}\ \mathtt{T}\ ;\ \mathtt{T}')$ **(2)** must be satisfiable as well. Now, assume, by way of contradiction, that ℓ is not in the domain of $\mathtt{V}$. Then,

by C-ROW-DL and C-ROW-LL, the left-hand member of (2) may be written $''(\ell : \mathsf{abs}; \ell_1 : \mathtt{Z}_1; \ldots; \ell_n : \mathtt{Z}_n; \partial\mathsf{abs})$. Thus, by our covariance hypotheses, (2) entails $\mathsf{abs} \leq \mathsf{pre}\ \mathtt{T}$, that is, false. We have reached a contradiction.

The remainder of the proof is left to the reader.

1.4.3 SOLUTION: Our hypotheses are $\mathsf{C}, \Gamma \vdash \mathtt{t} : \forall \bar{\mathtt{X}}[\mathsf{D}].\mathtt{T}$ **(1)** and $\mathsf{C} \Vdash [\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}]\mathsf{D}$ **(2)**. We may also assume, *w.l.o.g.*, $\bar{\mathtt{X}} \# \mathit{ftv}(\mathsf{C}, \Gamma, \vec{\mathtt{T}})$ **(3)**. By HMX-INST and (1), we have $\mathsf{C} \wedge \mathsf{D}, \Gamma \vdash \mathtt{t} : \mathtt{T}$, which by Lemma 1.4.2 yields $\mathsf{C} \wedge \mathsf{D} \wedge \vec{\mathtt{X}} = \vec{\mathtt{T}}, \Gamma \vdash \mathtt{t} : \mathtt{T}$ **(4)**. Now, we claim that $\vec{\mathtt{X}} = \vec{\mathtt{T}} \Vdash \mathtt{T} \leq [\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}]\mathtt{T}$ **(5)** holds; the proof appears in the next paragraph. Applying HMX-SUB to (4) and to (5), we obtain $\mathsf{C} \wedge \mathsf{D} \wedge \vec{\mathtt{X}} = \vec{\mathtt{T}}, \Gamma \vdash \mathtt{t} : [\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}]\mathtt{T}$ **(6)**. By C-EQ and by (2), we have $\mathsf{C} \wedge \vec{\mathtt{X}} = \vec{\mathtt{T}} \Vdash \mathsf{D}$, so (6) may be written $\mathsf{C} \wedge \vec{\mathtt{X}} = \vec{\mathtt{T}}, \Gamma \vdash \mathtt{t} : [\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}]\mathtt{T}$ **(7)**. Last, (3) implies $\bar{\mathtt{X}} \# \mathit{ftv}(\Gamma, [\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}]\mathtt{T})$ **(8)**. Applying rule HMX-EXISTS to (7) and (8), we get $\exists \bar{\mathtt{X}}.(\mathsf{C} \wedge \vec{\mathtt{X}} = \vec{\mathtt{T}}), \Gamma \vdash \mathtt{t} : [\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}]\mathtt{T}$ **(9)**. By C-NAMEEQ and by (3), $\exists \bar{\mathtt{X}}.(\mathsf{C} \wedge \vec{\mathtt{X}} = \vec{\mathtt{T}})$ is equivalent to C, hence (9) is the goal $\mathsf{C}, \Gamma \vdash \mathtt{t} : [\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}]\mathtt{T}$.

There now remains to establish (5). One possible proof method is to unfold the definition of $\Vdash$ and reason by structural induction on $\mathtt{T}$. Here is another, axiomatic approach. Let $\mathtt{Z}$ be fresh for $\mathtt{T}$, $\vec{\mathtt{X}}$, and $\vec{\mathtt{T}}$. By reflexivity of subtyping and by C-EXTRANS, we have $\mathsf{true} \equiv \mathtt{T} \leq \mathtt{T} \equiv \exists \mathtt{Z}.(\mathtt{T} \leq \mathtt{Z} \wedge \mathtt{Z} \leq \mathtt{T})$, which by congruence of $\equiv$ and by C-EXAND implies $\vec{\mathtt{X}} = \vec{\mathtt{T}} \equiv \exists \mathtt{Z}.(\mathtt{T} \leq \mathtt{Z} \wedge \vec{\mathtt{X}} = \vec{\mathtt{T}} \wedge \mathtt{Z} \leq \mathtt{T})$ **(10)**. Furthermore, by C-EQ, we have $(\vec{\mathtt{X}} = \vec{\mathtt{T}} \wedge \mathtt{Z} \leq \mathtt{T}) \equiv (\vec{\mathtt{X}} = \vec{\mathtt{T}} \wedge \mathtt{Z} \leq [\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}]\mathtt{T}) \Vdash (\mathtt{Z} \leq [\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}]\mathtt{T})$ **(11)**. Combining (10) and (11) yields $\vec{\mathtt{X}} = \vec{\mathtt{T}} \Vdash \exists \mathtt{Z}.(\mathtt{T} \leq \mathtt{Z} \wedge \mathtt{Z} \leq [\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}]\mathtt{T})$, which by C-EXTRANS may be read $\vec{\mathtt{X}} = \vec{\mathtt{T}} \Vdash \mathtt{T} \leq [\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}]\mathtt{T}$.

1.4.4 SOLUTION: The simplest possible derivation of $\mathsf{true}, \varnothing \vdash \lambda \mathtt{z}.\mathtt{z} : \mathsf{int} \to \mathsf{int}$ is syntax-directed. It closely resembles the Damas-Milner derivation given in Exercise 1.2.25.

$$\dfrac{\dfrac{}{\mathsf{true}, \mathtt{z} : \mathsf{int} \vdash \mathtt{z} : \mathsf{int}}\ \text{HMX-VAR}}{\mathsf{true}, \varnothing \vdash \lambda \mathtt{z}.\mathtt{z} : \mathsf{int} \to \mathsf{int}}\ \text{HMX-ABS}$$

As in Exercise 1.2.25, we may use a type variable $\mathtt{X}$ instead of the type int, then employ HMX-GEN to quantify universally over $\mathtt{X}$.

$$\dfrac{\dfrac{\dfrac{}{\mathsf{true}, \mathtt{z} : \mathtt{X} \vdash \mathtt{z} : \mathtt{X}}\ \text{HMX-VAR}}{\mathsf{true}, \varnothing \vdash \lambda \mathtt{z}.\mathtt{z} : \mathtt{X} \to \mathtt{X}}\ \text{HMX-ABS} \qquad \mathtt{X} \# \mathit{ftv}(\mathsf{true}, \varnothing)}{\mathsf{true}, \varnothing \vdash \lambda \mathtt{z}.\mathtt{z} : \forall \mathtt{X}[\mathsf{true}].\mathtt{X} \to \mathtt{X}}\ \text{HMX-GEN}$$

The validity of this instance of HMX-GEN relies on the equivalence $\mathsf{true} \wedge \mathsf{true} \equiv \mathsf{true}$ and on the fact that judgements are identified up to equivalence of their constraint assumptions.

If we now wish to instantiate X with int, we may use HMX-INST′ as follows:

$$\frac{\mathsf{true}, \varnothing \vdash \lambda z.z : \forall X[\mathsf{true}].X \to X \qquad \mathsf{true} \Vdash [X \mapsto \mathsf{int}]\mathsf{true}}{\mathsf{true}, \varnothing \vdash \lambda z.z : \mathsf{int} \to \mathsf{int}}\ \text{HMX-INST}'$$

This is not, strictly speaking, an HM(X) derivation, since HMX-INST′ is not part of the rules of Figure 1-7. However, since the proof of Lemma 1.4.2 and the solution of Exercise 1.4.3 are constructive, it is possible to exhibit the HM(X) derivation that underlies it. We find:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{X = \mathsf{int}, z : X \vdash z : X}\ \text{HMX-VAR}}{X = \mathsf{int}, \varnothing \vdash \lambda z.z : X \to X}\ \text{HMX-ABS}}{X = \mathsf{int}, \varnothing \vdash \lambda z.z : \forall X.X \to X}\ \text{HMX-GEN}}{X = \mathsf{int}, \varnothing \vdash \lambda z.z : X \to X}\ \text{HMX-INST}}{X = \mathsf{int} \Vdash X \to X \le \mathsf{int} \to \mathsf{int}}\ \text{HMX-SUB}}{\exists X.(X = \mathsf{int}), \varnothing \vdash \lambda z.z : \mathsf{int} \to \mathsf{int}}\ \text{HMX-EXISTS}$$

Since $\exists X.(X = \mathsf{int})$ is equivalent to true, the conclusion is indeed the desired judgement.

?? SOLUTION: We have:

$$\begin{aligned}
&[\![\mathtt{let\ f} = \lambda z.z\ \mathtt{in\ f\ f} : \mathtt{T}]\!] \\
&= \mathsf{let}\ \mathtt{f} : \forall X[[\![\lambda z.z : X]\!]].X\ \mathsf{in}\ [\![\mathtt{f\ f} : \mathtt{T}]\!] && \textbf{(1)} \\
&= \mathsf{let}\ \mathtt{f} : \forall X[\exists Z_1 Z_2.(\mathsf{let}\ \mathtt{z} : Z_1\ \mathsf{in}\ [\![\mathtt{z} : Z_2]\!] \wedge Z_1 \to Z_2 \le X)].X \\
&\quad \mathsf{in}\ \exists Y.([\![\mathtt{f} : Y \to \mathtt{T}]\!] \wedge [\![\mathtt{f} : Y]\!]) && \textbf{(2)} \\
&= \mathsf{let}\ \mathtt{f} : \forall X[\exists Z_1 Z_2.(\mathsf{let}\ \mathtt{z} : Z_1\ \mathsf{in}\ \mathtt{z} \preceq Z_2 \wedge Z_1 \to Z_2 \le X)].X \\
&\quad \mathsf{in}\ \exists Y.(\mathtt{f} \preceq Y \to \mathtt{T} \wedge \mathtt{f} \preceq Y) && \textbf{(3)} \\
&\equiv \mathsf{let}\ \mathtt{f} : \forall X[\exists Z_1 Z_2.(Z_1 \le Z_2 \wedge Z_1 \to Z_2 \le X)].X \\
&\quad \mathsf{in}\ \exists Y.(\mathtt{f} \preceq Y \to \mathtt{T} \wedge \mathtt{f} \preceq Y) && \textbf{(4)} \\
&\equiv \mathsf{let}\ \mathtt{f} : \forall X[\exists Z.(Z \to Z \le X)].X\ \mathsf{in}\ \exists Y.(\mathtt{f} \preceq Y \to \mathtt{T} \wedge \mathtt{f} \preceq Y) && \textbf{(5)} \\
&\equiv \mathsf{let}\ \mathtt{f} : \forall Z.Z \to Z\ \mathsf{in}\ \exists Y.(\mathtt{f} \preceq Y \to \mathtt{T} \wedge \mathtt{f} \preceq Y) && \textbf{(6)} \\
&\equiv \exists Y.(\exists Z.(Z \to Z \le Y \to \mathtt{T}) \wedge \exists Z.(Z \to Z \le Y)) && \textbf{(7)} \\
&\equiv \exists YZ.(Y \le \mathtt{T} \wedge Z \to Z \le Y) && \textbf{(8)} \\
&\equiv \exists Z.(Z \to Z \le \mathtt{T}) && \textbf{(9)}
\end{aligned}$$

where (1), (2), and (3) follow from the definition of constraint generation; (4) is by C-INID, C-IN*, and Definition 1.3.3; (5) may be obtained by exploiting the fact that $\to$ is contravariant is in its domain, or the fact that it is covariant in its codomain; (6) follows from the equivalence of the type schemes

$\forall X[\exists Z.(Z \to Z \leq X)].X$ and $\forall Z.Z \to Z$, which itself is a direct consequence of Definition 1.3.29 and of C-EXTRANS; (7) is by C-INID, C-IN*, and Definition 1.3.3; (8) is obtained by using the contra- and covariance of the arrow, as well as C-EXTRANS, to simplify the first conjunct, then applying C-EXAND to lift the second $\exists Z$ quantifier to the top level; (9) follows by C-EXTRANS. Please note that we have simplified f's type scheme as much as possible *before* eliminating the let f construct: otherwise, this simplification work would have been duplicated at each of the two instantiation sites.

1.7.16 SOLUTION: Let t be the unique ground type such $t = t \to t$. Let ϕ be the ground assignment that maps every type variable to t. Then, provided the type T contains no type constructors other than $\to$, we must have $\phi(T) = t$; that is, all types are mapped to a single point in the model. Now, let t be a closed source program. The constraint $\exists Z.[\![t : Z]\!]$ has no free program identifiers, involves no type constructors other than $\to$ and no predicates other than $\leq$. Thus, it must be satisfied by ϕ. (This proof is left to the reader.) Thus, t is well-typed.

1.9.1 SOLUTION: We have

$$
\begin{array}{lll}
 & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![c\ t_1\ \ldots\ t_n : T']\!] & \\
\equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists Z_1 \ldots Z_n.(\bigwedge_{i=1}^{n}[\![t_i : Z_i]\!] \wedge c \preceq Z_1 \to \ldots \to Z_n \to T') & \textbf{(1)} \\
\equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists Z_1 \ldots Z_n \bar{X}.(\bigwedge_{i=1}^{n}[\![t_i : Z_i]\!] & \textbf{(2)} \\
 & \qquad \wedge T_1 \to \ldots \to T_n \to T \leq Z_1 \to \ldots \to Z_n \to T') & \\
\equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \bar{X}.(\bigwedge_{i=1}^{n}[\![t_i : T_i]\!] \wedge T \leq T') & \textbf{(3)}
\end{array}
$$

where (1) is by definition of constraint generation; (2) is by C-INID; (3) is by C-ARROW, C-EXAND, and by Lemma 1.6.5.

2.2.1 SOLUTION: The if direction ($s' = t'$ implies $s \Leftrightarrow^* t$) follows directly from symmetry and transitivity. For the only-if direction, suppose $s \Leftrightarrow^* t$. We claim that s and t have a common reduct (that is, there exists u such that $s \Rightarrow^* u$ and $t \Rightarrow^* u$):

Proof: The proof is by induction on $s \Leftrightarrow^* t$.

Base step:

Suppose $s \Leftrightarrow^* t$ holds because $s \Rightarrow t$. Then let u be t.

Induction step: (Symmetry)

Suppose $s \Leftrightarrow^* t$ holds because $t \Leftrightarrow^* s$. By induction, t and s have a common reduct u.

Induction step: (Transitivity)

Suppose $s \Leftrightarrow^* t$ holds because $s \Leftrightarrow^* u'$ and $u' \Leftrightarrow^* t$. By induction, s and u' have a common reduct s'', and u' and t have a common reduct t''. Thus $u' \Rightarrow^* s''$ and $u' \Rightarrow^* t''$, so by confluence there exists u such that $s'' \Rightarrow^* u$ and $t'' \Rightarrow^* u$. Therefore u is a common reduct of s and t. □

We have shown that s and t have a common reduct u. Observe that $s \Rightarrow^* s'$ and $s \Rightarrow^* u$. By confluence, there exists r such that $s' \Rightarrow^* r$ and $u \Rightarrow^* r$. But s' is a normal form, so $r = s'$. Hence $u \Rightarrow^* s'$. Similarly $u \Rightarrow^* t'$. Then, by confluence, s' and t' must have a common reduct, but again they are normal forms so they must be equal.

2.2.2 SOLUTION: Let T and T' be any two distinct types. Then let t be:

$$\lambda x{:}T.\,(\lambda y{:}T'.\,y)\,x$$

By QR-ABS and QR-BETA, t reduces to $\lambda x{:}T.\,x$, and by QR-ETA, t reduces to $\lambda y{:}T'.\,y$. These two terms are distinct normal forms, so they have no common reduct.

2.3.1 SOLUTION:

$$\begin{array}{lcl} s & = & \lambda x{:}1.\,x \\ t & = & \lambda x{:}1.\,\star \end{array}$$

2.4.4 SOLUTION:

2.6.3 SOLUTION: The logical equivalence $x{:}b \vdash x$ **is** $x : b$ holds but $\vdash x$ **is** $x : b$ does not.

2.6.4 SOLUTION: The logical equivalence $\vdash \lambda x{:}b.\,x$ **is** $\lambda x{:}b.\,k : b \to b$ holds, but $y{:}b \vdash \lambda x{:}b.\,x$ **is** $\lambda x{:}b.\,k : b \to b$ does not.

Proof: We begin by showing the former logical equivalence holds. Suppose $\vdash s$ **is** $t : b$. We wish to show that:

$$\vdash (\lambda x{:}b.\,x)\ s \textbf{ is } (\lambda x{:}b.\,k)\ t : b$$

It is sufficient to show that:

$$\vdash (\lambda x{:}b.\,x)\ s \Leftrightarrow (\lambda x{:}b.\,k)\ t : b$$

Since $\vdash s$ **is** $t : b$, we have that $\vdash s \Leftrightarrow t : b$. By inversion $s \Downarrow s'$, $t \Downarrow t'$, and $\vdash s' \leftrightarrow t' : b$. Since the context is empty, and there exists only one

constant, it is easy to verify that $s' = t' = k$. Therefore $((\lambda x{:}b.x)\ s) \Downarrow k$ and $((\lambda x{:}b.k)\ t) \Downarrow k$. The desired conclusion follows.

Now we show that the latter logical equivalence does not hold. Let $s = t = y$. Then certainly $y{:}b \vdash s$ **is** $t : b$. However, $((\lambda x{:}b.x)\ s) \Downarrow y$ and $((\lambda x{:}b.k)\ t) \Downarrow k$, and y and k are not path equivalent. Therefore $(\lambda x{:}b.x)\ s$ and $(\lambda x{:}b.k)\ t$ are not algorithmically equivalent and hence cannot be logically equivalent at b. □

2.9.2 SOLUTION: The proof is by induction on the structure of T. The case $T = 1$ is trivial, and the case $T = b$ follows from algorithmic transitivity (Lemma 2.5.4).

Suppose $T = T_1 \to T_2$. Then $\Gamma \vdash s$ **is** $t : T_1 \to T_2$ and $\Gamma \vdash t$ **is** $u : T_1 \to T_2$. We wish to show that $\Gamma \vdash s$ **is** $u : T_1 \to T_2$. Suppose $\Gamma' \supseteq \Gamma$ and $\Gamma' \vdash s'$ **is** $u' : T_1$. Then we wish to show that $\Gamma' \vdash s\ s'$ **is** $u\ u' : T_2$.

By logical symmetry (Lemma 2.9.1), $\Gamma' \vdash u'$ **is** $s' : T_1$, and then by induction, $\Gamma' \vdash u'$ **is** $u' : T_1$. By the definition of logical equivalence, we may deduce $\Gamma' \vdash s\ s'$ **is** $t\ u' : T_2$ and also $\Gamma' \vdash t\ u'$ **is** $u\ u' : T_2$. By induction, $\Gamma' \vdash s\ s'$ **is** $u\ u' : T_2$.

2.9.9 SOLUTION:

Case T-CONST:
$$t = k$$
$$T = b$$

By the Main Lemma, $\Gamma' \vdash k$ **is** $k : b$. Therefore $\Gamma' \vdash \gamma(k)$ **is** $\delta(k) : b$, since k contains no free variables.

Case Q-REFL:

Immediate by the first clause of the induction hypothesis.

Case Q-SYMM:

Immediate from the induction hypothesis and logical symmetry.

Case Q-TRANS:

By logical symmetry, $\Gamma' \vdash \delta$ **is** $\gamma : \Gamma$, so by logical transitivity, $\Gamma' \vdash \delta$ **is** $\delta : \Gamma$. Therefore, by induction (using γ and δ), $\Gamma' \vdash \gamma(s)$ **is** $\delta(t) : T$, and also by induction (using δ and δ), $\Gamma' \vdash \delta(t)$ **is** $\delta(u) : T$. By logical transitivity, $\Gamma' \vdash \gamma(s)$ **is** $\delta(u) : T$.

Case Q-ABS:
$$s = \lambda x{:}T_1.s_2$$
$$t = \lambda x{:}T_1.t_2$$
$$T = T_1 \to T_2$$

We wish to show that $\Gamma' \vdash \gamma(\lambda x{:}T_1.s_2)$ **is** $\delta(\lambda x{:}T_1.t_2) : T_1 \to T_2$. Suppose $\Gamma'' \supseteq \Gamma'$ and $\Gamma'' \vdash s'$ **is** $t' : T_1$. We wish to show that $\Gamma'' \vdash (\lambda x{:}T_1.\gamma(s_2))\ s'$ **is** $(\lambda x{:}T_1.\delta(t_2))\ t' : T_2$. By logical weak head closure, it is sufficient to show that $\Gamma'' \vdash [x \mapsto s']\gamma(s_2)$ **is** $[x \mapsto t']\delta(t_2) : T_2$.

By logical monotonicity, $\Gamma'' \vdash \gamma$ **is** δ : Γ. Thus, $\Gamma'' \vdash \gamma[x \mapsto s']$ **is** $\delta[x \mapsto t']$: $(\Gamma, x:T_1)$. Therefore, by induction, $\Gamma'' \vdash \gamma[x \mapsto s'](s_2)$ **is** $\delta[x \mapsto t'](t_2)$: T_2, which is equivalent to the desired conclusion.

Case Q-APP:
$$s = s_1\ s_2$$
$$t = t_1\ t_2$$
$$T = T_{12}$$

By induction, $\Gamma' \vdash \gamma(s_1)$ **is** $\delta(t_1)$: $T_1 \to T_2$ and $\Gamma' \vdash \gamma(s_2)$ **is** $\delta(t_2)$: T_1. By the definition of the logical relation, since $\Gamma' \supseteq \Gamma$, we may conclude $\Gamma' \vdash \gamma(s_1)\gamma(s_2)$ **is** $\delta(t_1)\delta(t_2)$: T_2. That is, $\Gamma' \vdash \gamma(s_1\ s_2)$ **is** $\delta(t_1\ t_2)$: T_2.

Case Q-EXT:
$$s = s$$
$$t = t$$
$$T = T_1 \to T_2$$

We wish to show that $\Gamma' \vdash \gamma(s)$ **is** $\delta(t)$: $T_1 \to T_2$. Suppose $\Gamma'' \supseteq \Gamma'$ and $\Gamma'' \vdash s'$ **is** t' : T_1. We wish to show that $\Gamma'' \vdash \gamma(s)\ s'$ **is** $\delta(t)\ t'$: T_2.

By logical monotonicity, $\Gamma'' \vdash \gamma$ **is** δ : Γ. Thus, $\Gamma'' \vdash \gamma[x \mapsto s']$ **is** $\delta[x \mapsto t']$: $(\Gamma, x:T_1)$. Therefore, by induction, $\Gamma'' \vdash \gamma[x \mapsto s'](s\ x)$ **is** $\delta[x \mapsto t'](t\ x)$: T_2. That is, $\Gamma'' \vdash \gamma(s)\ s'$ **is** $\delta(t)\ t'$: T_2, as desired.

2.9.11 SOLUTION: By soundness, $\Gamma \vdash s_1 \equiv t_1 : T_1 \to T_2$ and $\Gamma \vdash s_2 \equiv t_2 : T_1$. By Q-APP, $\Gamma \vdash s_1\ s_2 \equiv t_1\ t_2 : T_2$. By completeness, $\Gamma \vdash s_1\ s_2 \Leftrightarrow t_1\ t_2 : T_2$.

2.9.12 SOLUTION: The key observation is that the left- and right-hand sides of the algorithm do not interact, except insofar as a failure to match in path equivalence allows the algorithm to quit early. Therefore we can devise a termination metric that takes each side into account independently.

Therefore, define the metric $M(\Gamma \vdash s \Leftrightarrow t : T)$ to be the size of the derivation (if it exists) of $\Gamma \vdash s \Leftrightarrow s : T$ plus the size of the derivation (if it exists) of $\Gamma \vdash t \Leftrightarrow t : T$. Define the metric $M(\Gamma \vdash p \leftrightarrow q : T)$ similarly. It is straightforward to show that the metric decreases in each recursive call of the algorithm. It is also straightforward to show that all normalizations terminate, since the normalization derivations being sought already exist within the derivations measured by the metric.

Thus, it remains to show only that the metric is actually defined, that is, that there exist derivations of $\Gamma \vdash s \Leftrightarrow s : T$ and $\Gamma \vdash t \Leftrightarrow t : T$. This follows by the completeness of the algorithm from our assumptions that $\Gamma \vdash s : T$ and $\Gamma \vdash t : T$.

2.9.13 SOLUTION:

3.3.2 HINT: First prove

$$\langle S_1, t_1\rangle \longrightarrow \langle S_2, t_2\rangle \Rightarrow (\forall S)(\langle S@S_2, t_2\rangle\downarrow \Rightarrow \langle S@S_1, t_1\rangle\downarrow)$$

by considering the different cases for $\longrightarrow$. Deduce the 'if' part of (3.6) from this. For the 'only if' part, show that

$$\{(S,t) \mid (\exists S_1, S_2, v)\ S = S_1@S_2 \ \&\ \langle S_2, t\rangle \longrightarrow^* \langle Id, v\rangle \ \&\ \langle S_1, v\rangle \downarrow\}$$

is closed under the axiom and rules in Figure 3-2 inductively defining the termination relation.

3.5.3 SOLUTION: Since $(-)^{st}$ is inflationary we have $r \subseteq r^{st}$; and since r only relates values, this implies $r \subseteq r^{stv}$. Then since $(-)^{st}$ is monotone, we have $r^{st} \subseteq r^{stvst}$.

Conversely, since $(r')^v \subseteq r'$ for any r', we have $r^{stv} \subseteq r^{st}$; and then since $(-)^{st}$ is monotone and idempotent, $r^{stvst} \subseteq r^{stst} = r^{st}$.

3.5.8 HINT: The proof of (3.21) is just like the proof of (3.19), using the following property of the termination relation:

$$(\langle S, v\,\mathtt{.1}\rangle \downarrow \Leftrightarrow \langle S', v'\,\mathtt{.1}\rangle \downarrow) \text{ iff } (\langle S \circ (\mathtt{x.x.1}), v\rangle \downarrow \Leftrightarrow \langle S' \circ (\mathtt{x.x.1}), v'\rangle \downarrow).$$

Similarly, the proof of (3.22) follows from:

$$(\langle S, v\ \mathtt{T}\rangle \downarrow \Leftrightarrow \langle S', v'\ \mathtt{T}'\rangle \downarrow) \text{ iff } (\langle S \circ (\mathtt{x.x\ T}), v\rangle \downarrow \Leftrightarrow \langle S' \circ (\mathtt{x.x\ T}'), v'\rangle \downarrow).$$

3.5.11 SOLUTION: It suffices to show

$$(\forall n = 0, 1, \ldots)\ (\mathtt{F}_n, \mathtt{F}'_n) \in \mathtt{fun}(r_1, r_2) \tag{A.1}$$

where $\mathtt{F}_n$ and $\mathtt{F}'_n$ are the unwindings associated with $\mathtt{F}$ and $\mathtt{F}'$ respectively, as in Theorem 3.3.4. For if (A.1) holds, then using the fact that $(-)^{st}$ is inflationary

$$(\mathtt{F}_n, \mathtt{F}'_n) \in \mathtt{fun}(r_1, r_2) \subseteq \mathtt{fun}(r_1, r_2)^{st}$$

for each n; so by the **Admissibility** property in Lemma 3.5.5 we have $(\mathtt{F}, \mathtt{F}') \in \mathtt{fun}(r_1, r_2)^{st}$. Thus $(\mathtt{F}, \mathtt{F}') \in \mathtt{fun}(r_1, r_2)^{stv} = \mathtt{fun}(r_1, r_2)$ by Lemma 3.5.7, since $(r_2)^{st} = r_2$. (A.1) is proved by induction on n:

Base case $n = 0$**:** By definition of $\mathtt{F}_0$, $\langle S, \mathtt{F}_0\ v_1\rangle \downarrow$ does not hold for any $S \in \mathit{Stack}(\mathtt{T}_2)$ and $v_1 \in \mathit{Val}(\mathtt{T}_1)$; similarly for $\mathtt{F}'_0$. Hence for all $(v_1, v'_1) \in (r_1)^v$, $(\mathtt{F}_0\ v_1, \mathtt{F}'_0\ v'_1) \in s^t$ for any $s \in \mathit{SRel}(\mathtt{T}_2, \mathtt{T}'_2)$ and hence in particular for $s = (r_2)^s$. So $(\mathtt{F}_0\ v_1, \mathtt{F}'_0\ v'_1) \in (r_2)^{st} = r_2$ for all $(v_1, v'_1) \in (r_1)^v$. Therefore $(\mathtt{F}_0, \mathtt{F}'_0) \in \mathtt{fun}(r_1, r_2)$.

Induction step: Suppose $(\mathtt{F}_n, \mathtt{F}'_n) \in \mathtt{fun}(r_1, r_2)$. Then for any $(v_1, v'_1) \in (r_1)^v$, from (3.24) we have

$$([\mathtt{f} \mapsto \mathtt{F}_n][\mathtt{x} \mapsto v_1]t, [\mathtt{f} \mapsto \mathtt{F}'_n][\mathtt{x} \mapsto v'_1]t') \in r_2.$$

By definition of $\mathtt{F}_{n+1}$ and Corollary 3.4.6 we have $\emptyset \vdash \mathtt{F}_{n+1}\mathtt{v}_1 =_{\mathrm{ctx}} [\mathtt{f} \mapsto \mathtt{F}_n][\mathtt{x} \mapsto \mathtt{v}_1]\mathtt{t}$; and similarly, $\emptyset \vdash \mathtt{F}'_{n+1}\mathtt{v}'_1 =_{\mathrm{ctx}} [\mathtt{f} \mapsto \mathtt{F}'_n][\mathtt{x} \mapsto \mathtt{v}'_1]\mathtt{t}'$. So since $\mathtt{r}_2$ is closed, we can apply the **Equality-respecting** property in Lemma 3.5.5 to conclude that $(\mathtt{F}_{n+1}\mathtt{v}_1, \mathtt{F}'_{n+1}\mathtt{v}'_1) \in \mathtt{r}_2$. Since this holds for any $(\mathtt{v}_1, \mathtt{v}'_1) \in (\mathtt{r}_1)^v$, we have $(\mathtt{F}_{n+1}, \mathtt{F}'_{n+1}) \in \mathtt{fun}(\mathtt{r}_1, \mathtt{r}_2)$.

3.6.2 SOLUTION: One possible solution is $S_c = Id \circ (\mathtt{x}.\mathtt{v}_c\ \mathtt{x})$ where

```
val v_c = fun f(x:Gnd) = if x=c then c else f x
```

3.6.8 SOLUTION: Since `N` has no closed values, neither does $\{\exists \mathtt{X}, \mathtt{N}\}$. On the other hand

```
val v = ΛY.fun f(x:∀X.N→Y) = (f x):Y
```

is a closed value of type $\forall \mathtt{Y}.(\forall \mathtt{X}.\mathtt{N} \to \mathtt{Y}) \to \mathtt{Y}$. If `i` and `j` were to exist with the stated properties we could use them to construct from `v` a closed value of type $\{\exists \mathtt{X}, \mathtt{N}\}$, which is impossible. (For `i (j v)` and `v` are ciu-equivalent (Theorem 3.4.5); so since $\mathtt{v}\downarrow$, we also have $\mathtt{i}\,(\mathtt{j}\ \mathtt{v})\downarrow$. Hence by Exercise 3.3.2, $\langle Id, \mathtt{j}\ \mathtt{v}\rangle \longrightarrow^* \langle Id, \mathtt{v}'\rangle$ for some $\mathtt{v}'$, which is a closed value of type $\{\exists \mathtt{X}, \mathtt{N}\}$, by Exercise 3.3.3.)

III.1 SOLUTION: We can introduce a type family for rectangular matrices thus:

```
Matrix     :: Nat → Nat → *
idmatrix   : Πn:Nat. Matrix n n
multmatrix : Πl:Nat. Πm:Nat. Πn:Nat.
                     Matrix l m → Matrix m n → Matrix l n
```

Suppose we have a dependent type for ranges of integers: `{n...m}` denotes the type of integers between `n` and `m` inclusive, either `n` or `m` may be omitted. A possible typing for dates is given by:

```
Year  =  {2003..} :: *
Month =  {1..12}  :: *
Day   :: Month → *
```

where

```
Day(n)  =  {1..31}  if n ∈ {1,3,5,7,8,10,12}
Day(n)  =  {1..30}  if n ∈ {4,6,9,11}
Day(2)  =  {1..29}
```

A date is then given by an element of the Σ-type (see page 258):

```
Date :: Σy:Year. Σm:Month. Day(m)
```

Of course, we could gain more accuracy by making the type of days also depend on the year.

III.2 SOLUTION: A type representing the constructive axiom of choice for a predicate `P` is:

```
(Πa:A.Σb:B. P(a,b)) → (Σf:A→B. Πx:A. P(x, f x))
```

It can be shown in Martin-Löf's type theory that this type is inhabited (Martin-Löf, 1984).

III.3 SOLUTION: `Σa:A.Σb:B.Id(f a, g b)`

III.4 SOLUTION: Here are some terms representing β-reduction and its closure on lambda terms:

```
Eval     :: ΠA:Ty. Tm A → Tm A → *
evalAppAbs : ΠA:Ty. ΠB:Ty.
               Πt1:(Tm A → Tm B).
               Πt2:(Tm A) → Eval (app (lam t1) t2) (t1 t2)
evalLam  :  ΠA:Ty. ΠB:Ty.
             Πft1,ft1':(Tm A → Tm B).
              (Πx: Tm A. Eval (ft1 x) (ft1' x))
                 → Eval (lam ft1) (lam ft2)
evalApp1 :  ΠA:Ty. ΠB:Ty.
             Πt1,t1':(Tm (arrow A B)).
              Πt2: Tm B. Eval t1 t1' → Eval (app t1 t2) (app t1' t2)
evalApp2 :  ΠA:Ty. ΠB:Ty.
             Πt1: (Tm (arrow A B)).
              Πt2,t2': Tm B. Eval t2 t2' → Eval (app t1 t2) (app t1 t2')
```

4.5.2 SOLUTION: To simplify the translation we assume that there is no syntactic distinction between type and term variables in F^ω and that a single colon is used for both typing and kinding.

The types of the Calculus of Constructions are divided into two categories. The *kind types* are those of the form

$$\Pi x_1{:}T_1.\ldots.\Pi x_n{:}T_n.\mathtt{Prop}$$

The *proper types* are those of the form

$$\Pi x_1{:}T_1.\ldots.\Pi x_n{:}T_n.\mathtt{Prf}(t)$$

A mapping $(-)^K$ from kind types to F^ω kinds is defined by

$$(\Pi x{:}T_1.T_2)^K = (T_1)^K \to (T_2)^K, \text{ if } T_1 \text{ is a kind type}$$

$(\Pi x:T_1.T_2)^K = (T_2)^K$, if T_1 is a proper type

A mapping $(-)^T$ from proper types to F^ω-types is defined by

$(\Pi x:T_1.T_2)^T = \forall x:(T_1)^K.(T_2)^T$, if T_1 is a kind type

$(\Pi x:T_1.T_2)^T = (T_1)^T \to (T_2)^T$, if T_1 is a proper type

Let us define the mapping $(-)^*$ on all types as the union of the two mappings $(-)^K$ and $(-)^T$. Contexts are mapped to F^ω-contexts by applying $(-)^*$ to the bindings. Notice that while a context in the Calculus of Constructions consists entirely of type bindings the translation will involve both type and kind bindings.

The translation is now extended to terms by

$(x)^* = x$
$(\lambda x:T.t)^* = \lambda x:(T)^*.(t)^*$
$(t_1\ t_2)^* = (t_1)^*(t_2)^*$
$(\mathtt{all}\ x:T.t)^* = \forall x:(T)^K.(t)^*$ if T is a kind type
$(\mathtt{all}\ x:T.t)^* = (T)^K \to (t)^*$ if T is a proper type

It is now clear that $\Gamma \vdash t:T$ implies $(\Gamma)^* \vdash (t)^* : (T)^*$.

4.5.4 SOLUTION: We give the solution in the syntax of the implementation.

```
eqSucc = λx:Prf(nat).λy:Prf(nat).λh:Prf(eq nat x y).
       h(λz:Prf(nat).eq nat (succ x) (succ z)) (eqRefl nat (succ x))
      : Πx:Prf(nat).Πy:Prf(nat).Prf(eq nat x y) →
                                      Prf(eq nat (succ x) (succ y));

addAssoc = λx:Prf(nat).λy:Prf(nat).λz:Prf(nat).
          eq nat (add x (add y z)) (add (add x y) z);

proofOfAddAssoc = λx:Prf(nat).λy:Prf(nat).λz:Prf(nat).
      natInd (λx1:Prf(nat).addAssoc x1 y z)
           (eqRefl nat (add y z))
           (λx1:Prf(nat).λp:Prf(addAssoc x1 y z).
                eqSucc (add x1 (add y z)) (add (add x1 y) z) p)
           x
 : Πx:Prf(nat).Πy:Prf(nat).Πz:Prf(nat).Prf(addAssoc x y z);
```

4.6.1 SOLUTION: Let $i : \mathit{Syntax}(\lambda LF) \to \mathit{Syntax}(\lambda P)$ be the obvious mapping between the syntaxes, which collapses each λLF λ-construct to the single λP λ-operator, etc. (Except that type and term variables have disoint images). Then we would hope to show:

1. $\Gamma \vdash_{\lambda LF} \mathtt{t} : \mathtt{T} \iff \mathtt{i}(\Gamma) \vdash_{\lambda P} \mathtt{i}(\mathtt{t}) : \mathtt{i}(\mathtt{T})$
2. $\Gamma \vdash_{\lambda LF} \mathtt{T} :: \mathtt{K} \iff \mathtt{i}(\Gamma) \vdash_{\lambda P} \mathtt{i}(\mathtt{T}) : \mathtt{i}(\mathtt{K})$
3. $\Gamma \vdash_{\lambda LF} \mathtt{K} \iff \mathtt{i}(\Gamma) \vdash_{\lambda P} \mathtt{i}(\mathtt{K}) : \Box$

There are two difficulties in establishing this equivalence. First, the presentation of λLF includes Q-ETA, but η equalities are not included in the definition of PTS we gave. If Q-ETA is removed from λLF, the left to right direction is straightforward. The right to left direction raises the second difficulty: we must show that the untyped conversion relation of PTS can be simulated by the declarative equality in λLF. This requires showing the Church-Rosser property for the PTS conversion.

6.1.4 SOLUTION: The proof of each lemma proceeds by induction on the typing derivation. Almost all cases follow directly from the induction hypothesis. The base cases are straightforward as well, but some slight amount of work is involved. For instance, in the base case for weakening we are given the judgement $\Gamma_1, \mathtt{x}:\mathtt{T}, \Gamma_2 \vdash \mathtt{x} : \mathtt{T}$. and must prove that for arbitrary Γ_3, $\Gamma_1, \mathtt{x}:\mathtt{T}, \Gamma_2, \Gamma_3 \vdash \mathtt{x} : \mathtt{T}$. The latter judgement follows directly from the variable rule as the rule schema allows the context $\Gamma_1, \mathtt{x}:\mathtt{T}, \Gamma_2, \Gamma_3$. Notice, however, that if we were not careful in the definition of the variable rule and had omitted Γ_2 from the context in the rule schema, we would be unable to prove this weakening lemma. Hence, while simple, the rules for the variables and constants play an integral role in defining the structural properties of a type system.

6.2.1 SOLUTION: Since the variable may only appear on the extreme right-hand side of the context, we will be unable to prove the exchange lemma. In the literature, you will see this formulation of the variable rule all the time because authors often treat contexts as finite partial maps. In other words, contexts that differ only in the order in which we write down their elements are treated equally and are never distinguished from one another. In this chapter, we choose not to take this perspective so that we may study the complete set of structure rules directly.

6.3.2 SOLUTION: The type of linear trees with elements of type `T` follows.

```
type T tree = rec a.lin (unit + lin (T * a * a))
```

It will be convenient to define some constructors for trees of type `T` as well.

```
fun nilT (nil:unit) : T tree = roll (lin inl nil)

fun nodeT (arg : lin (T * T tree * T tree)) : TL = roll (lin inr arg)
```

In order to do a constant-space tree traversal, we must reuse the tree cells to create our own list (stack) that remembers what to do next after we have completed processing the current subtree. In ML, we might define such a list using the datatype:

```
datatype (T1,T2) TL =
  done
| right of T2 * T1 tree * TL
| left of T2 * T2 tree * TL
```

The first constructor indicates there is nothing left to do. The second constructor indicates we have finished processing a left subtree, but we still need to process the right subtree (the object with type `T1 tree`) and glue the processed tree element (with type `T2`) together with the left and right subtrees. We also need to recursively process the rest of the list. The last constructor indicates we have just finished processing a right subtree and we need to assembly the tree element, left and right subtrees and recursively process the rest of the list.

In our linear lambda calculus, the ML type definition given above and its associated constructors will be defined as follows. We will use $\texttt{in}_0$, $\texttt{in}_1$,... $\texttt{in}_{n-1}$ to inject into a `n`-ary sum when `n` is greater than two. For brevity, we will not bother to index these types and constructors with parameters `T1` and `T2`.

```
type TL =
  mu a. lin (unit + lin (T2 * T1 tree * TL) + lin (T2 * T2 tree * TL))

fun done (nil:unit) : TL = roll (lin in0 nil)

fun left (arg : lin (T2 * T2 tree * TL)) : TL = roll (lin in1 arg)

fun right (arg : lin (T2 * T1 tree * TL)) : TL = roll (lin in2 arg)
```

The algorithm is factored into a top-level function `treeMap` and two helpers. The first processes a subtree we have not seen yet. The second determines what to do next by looking at the `TL` stack.

```
type FT = T1 → T2

fun treeMap(f:FT,t:T1 tree) : T2 tree =
  procTree (f,t,empty())

and procTree(f:FT,t:T1 tree,tl:TL) : T2 tree =
  case unroll t (
```

```
      inl nil ⇒ procTL (f,empty(),tl)
    | inr tree ⇒
       split tree as elem,t1,t2 in
       procTree (f,t1,right lin <f elem,t2,tl>)

and procTL(f:FT,t:T2 tree,tl:TL): T2 tree =
  case unroll tl (
    in0 nil  ⇒ t
  | in1 arg ⇒
      split arg as elem,t2,tl in
      procTree (f,t2,left lin <elem,t,tl>)
  | in2 arg ⇒
      split arg as elem,t1,tl in
      procTL (f,nodeT lin <elem,t1,t2>,tl)
```

6.3.5 SOLUTION: If an unrestricted array can contain a linear object, the linear object might never be used. Interestingly, due to our swapping operational semantics for arrays, even though an unrestricted array (containing linear objects) could be used many times, the linear objects themselves could never be used more than once.

6.4.1 SOLUTION: Since `ord` is the least qualifier in our ordered type system and `un` is the greatest qualifier in our ordered type system, the rules dealing with polymorphic qualifiers should be following.

$$\texttt{ord} \sqsubseteq \texttt{p} \qquad \text{(Q-ORDP)}$$

$$\texttt{p} \sqsubseteq \texttt{un} \qquad \text{(Q-PUN)}$$

6.4.2 SOLUTION: Consider the following expression. If we generalized the syntax to allow nested sub expressions but made no change to the typing rules, it would type check despite the fact that booleans are confused with integers.

```
let x = ord <true,true> in
let y = ord <ord <3,2>,x> in
split y as z1,z2 in
split z2 as b1,b2 in
if b1 then ...  (* using an int as if it was a bool *)
```

6.4.5 SOLUTION: There are undoubtedly several solutions to this exercise. The simplest solution is to add a new `junk` type that can serve as a placeholder for the used function pointer. The only thing that can be done with an object of type `junk` is to pop it off the stack. The idea is that rather than having the compiler implicitly insert instructions to shift the ordered argument down the stack at the point of a function call, the function will be left on the stack,

but given the unusable type `junk`. When the function body has used the argument, the junk item will appear at the top of the stack. The programmer will have to explicitly pop it off the stack before using the objects in the function closure, which will appear deeper on the stack.

The typing rule for the specialized ordered abstraction with ordered argument as well as the typing rule for the command $\texttt{pop}\ t_1\texttt{;}\ t_2$, which pops its argument (t_1) off the top of the stack and continues execution with t_2, appear below. It is up to you to define their operational rules.

$$\frac{q(\Gamma) \qquad \Gamma, \texttt{f:ord junk}, \texttt{x:ord}\ P_1 \vdash t_2 : T_2}{\Gamma \vdash \texttt{ord}\ \lambda_f\ \texttt{x:(ord}\ P_1\texttt{)}.t_2 : \texttt{ord (ord}\ P_1\texttt{)} \rightarrow T_2} \qquad \text{(T-ABS)}$$

$$\frac{\Gamma_1 \vdash t_1 : \texttt{ord junk} \qquad \Gamma_2 \vdash t_2 : T}{\Gamma_1 \circ \Gamma_2 \vdash \texttt{pop}\ t_1\texttt{;}\ t_2 : T} \qquad \text{(T-POP)}$$

7.2.1 SOLUTION: The assertion that x has type singleton for value v can be written simply as $x = v$. The singleton type for the value v can be written as

$$\{x \mid x = v\}$$

and correspondingly the assertion can also be written as $x : \{x \mid x = v\}$

7.2.2 SOLUTION:

$$\{x \mid x : \texttt{ptr}\,\{\{y \mid y = 0\}; \texttt{int}\} \vee x : \texttt{ptr}\,\{\{y \mid y = 1\}; \texttt{int}; \texttt{int}\}\}$$

7.2.3 SOLUTION: We assume that the same `listinv` formula constructor is used to specify that the contents of the memory is well-typed. The function specification is then:

$$\begin{array}{lcl} \texttt{Pre} & = & r_1 : \texttt{ptr}\,\{\texttt{int}; \texttt{int}\} \wedge r_2 : \texttt{list int} \wedge \texttt{listinv}\ r_M \\ \texttt{Post} & = & r_R : \texttt{list int} \wedge \texttt{listinv}\ r_M \end{array}$$

7.2.4 SOLUTION: The challenge here is to express the sequence property. We can do that either by adding a new type constructor or simply by using a universal quantifier.

$$\begin{array}{lcl} \texttt{Pre} & = & \texttt{listinv}\ r_M \wedge \forall i.(0 \leq i \wedge i < r_2) \Rightarrow (r_1 + 4 * i) : \texttt{ptr}\,\{\texttt{list int}\} \\ \texttt{Post} & = & \texttt{listinv}\ r_M \end{array}$$

7.2.5 SOLUTION: We show the solution for the more complicated case when the array elements are structures. We must add the `array S` type constructor, where S is a structure type. We also add the `(sizeof S N)` formula to state that the size of the structure S is N bytes.

In order to handle the `sizeof` formula constructor we add the following two rules:

$$\frac{\texttt{sizeof}\ W\ 4 \qquad \texttt{sizeof}\ S\ N}{\texttt{sizeof}\ (W;S)\ (N+4)}$$

By indexing into an array we can obtain pointers to elements, provided that the index is in the bounds of the array. For the purpose of bounds checking we must fetch the length of the array from memory and hence we must add a requirement that the memory contents is well-typed.

$$\frac{A : \texttt{array}\ S \qquad (\texttt{sizeof}\ S\ N) \qquad 0 \leq I \qquad I < (\texttt{sel}\ M\ A) \qquad \texttt{listinv}\ M}{(A + 4 + I * N) : \texttt{ptr}\ \{S\}}$$

7.3.5 SOLUTION:

We assume that for each function staring at address `L` in the agent we have a precondition $Pre_{\texttt{L}}$ and a postcondition $Post_{\texttt{L}}$. These can be specified by the agent producer using annotations. For example, in JVML they are specified as types in a special table in the `.class` file that contains the agent.

Assume also that the set of registers is $\texttt{r}_1, \ldots, \texttt{r}_n$ and that the callee-save registers are $\texttt{r}_1, \ldots, \texttt{r}_{CS}$. Unlike in the original symbolic evaluator we must identify for each `return` instruction to which function it belongs, and thus what postcondition to use. This can be done by carrying an additional parameter in the symbolic evaluator to specify the postcondition to use for the `return` instructions. Instead, we are going to assume that return instructions are annotated with the starting address of the function to which they belong. We also assume that each start of a function contains an invariant corresponding to the precondition.

Now we can extend the symbolic evaluator as follows:

$$SE(i, \sigma) = \begin{cases} \ldots & \\ (\sigma\ Post_{\texttt{L}}) & \text{if } \Pi_i = \texttt{return}_{\texttt{L}} \\ (\sigma\ Pre_{\texttt{L}}) \wedge & \text{if } \Pi_i = \texttt{call L} \\ \forall x_{CS+1} \ldots . x_n . (\sigma'\ Post_{\texttt{L}}) \Rightarrow SE(i+1, \sigma') & \end{cases}$$

where $\sigma' = \sigma[\texttt{r}_{CS+1} = x_{CS+1}, \ldots, \texttt{r}_n = x_n]$. Thus a function call first asserts that the precondition holds, then modifies the symbolic state so that the non callee-save registers are modified to have arbitrary values. The state σ' models the state after the call. In this state the postcondition is assumed to hold and the symbolic evaluation continues.

7.3.6 SOLUTION: We extend the symbolic evaluator as follows:

$$SE(i, \sigma) = \begin{cases} \ldots & \\ \texttt{false} & \text{if } \Pi_i = \texttt{UNREACHABLE} \end{cases}$$

Notice that indeed we stop the evaluation at that point, but we require that the agent producer proves that this context is never reachable. The agent producer can actually produce a proof of `false` if this program point follows a function call to a function that never returns and whose postcondition is `false`, as is the case with the `myexit` function in the problem statement. It is also possible to prove `false` at a program point following a conditional branch that can be proved to be always taken.

7.3.7 SOLUTION: We shall consider that each label in the program also acts as a nullary constructor in the logic, denoting the program counter where it is placed. We extend the symbolic evaluator to read the annotation that follows an indirect jump and to require a proof that the address being jumped to is equal to one of the declared destinations. Otherwise, the indirect jump is handled as a conditional branch.

$$SE(i,\sigma) = \begin{cases} \ldots & \\ ((\sigma\ e) = \texttt{L1} \vee (\sigma\ e) = \texttt{L2}) \wedge & \text{if } \Pi_i = \texttt{jump at } e \\ ((\sigma\ e) = \texttt{L1} \Rightarrow SE(\texttt{L1},\sigma)) \wedge & \text{and } \Pi_{i+1} = \texttt{JUMPDEST(L1,L2)} \\ ((\sigma\ e) = \texttt{L2} \Rightarrow SE(\texttt{L2},\sigma)) & \end{cases}$$

7.4.6 SOLUTION: Let ρ_1 be a state such that $\models_{\mathcal{M}} \rho_1\ \mathit{Pre}$. We assume that $\mathit{Dom}(\mathcal{M}) \subseteq \mathcal{A}\mathtt{ddr}$ and $\models_{\mathcal{M}} \mathit{VC}$. By convention the first instruction in the program (at program counter 1) is an invariant `INV` *Pre*. Let $\sigma_1 = \{\mathtt{r}_1 = x_1, \ldots, \mathtt{r}_n = x_n\}$ and $\tau_1 = \{x_1 = \rho_1\ \mathtt{r}_1, \ldots, x_n = \rho_1\ \mathtt{r}_n\}$. This means that $\rho_1 = \tau_1 \circ \sigma_1$. We also know that $SE(1,\sigma_1) = \sigma_1\ \mathit{Pre}$ and therefore we know that $\models_{\mathcal{M}} \tau_1\ SE(1,\sigma_1)$. This allows us to establish that the induction hypothesis holds for the first instruction: $IH(1,\rho_1,\sigma_1,\tau_1)$.

We can prove by induction on the number of transition steps, that for any (i,ρ) reachable from the initial state $(1,\rho_1)$, there exist σ and τ such that $IH(i,\rho,\sigma,\tau)$. Furthermore, either i points to a `return` instruction or else we can make further progress. The base case follows from the argument above and the inductive step is proved using Theorem 7.4.4.

7.5.1 SOLUTION:

```
all (λa : ι.
    (imp (hastype a (ptr (seq1 int)))
         (addr a)))
```

7.5.2 SOLUTION:

The proof of the predicate $\forall a.a : \texttt{ptr}\,\{\texttt{int}\} \Rightarrow \texttt{addr}\ a$ is shown below:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{a : \texttt{ptr}\,\{\texttt{int}\}}\,u}{\texttt{addr}\ a}\,\text{PTRADDR}}{a : \texttt{ptr}\,\{\texttt{int}\} \Rightarrow \texttt{addr}\ a}\,\text{IMPI}^u}{\forall a.a : \texttt{ptr}\,\{\texttt{int}\} \Rightarrow \texttt{addr}\ a}\,\text{ALLI}^a$$

The LF representation of this proof is shown below. Notice how the parameter a and the hypothesis u are properly scoped by using higher-order representation.

```
alli (λa : ι.(imp (hastype a (ptr (seq1 int)))
                  (addr a)))
     (impi (hastype a (ptr (seq1 int)))
           (addr a)
           (λu : pf (hastype a (ptr (seq1 int)))
                (ptraddr a (seq1 int) u)))
```

In the above representation we used LF constants declared in Figure 7-11 along with the following declaration for `ptraddr`:

```
ptraddr : ΠA : ι.ΠS : s.pf (hastype A (ptr S)) → pf (addr A)
```

9.2.1 SOLUTION: As of this writing, the question of how far nominal module systems can be pushed is still quite open. A step in this direction was recently taken by Odersky et al. (2003).

9.5.1 SOLUTION: Define M_1 to be the module

```
module M1 = mod {
  type X = Int
  val x = 0
  val f = succ
}
```

and define M_2 to be the module

```
module M2 = mod {
  type Y = Bool
  val x = true
  val f = not
}.
```

Define `M` to be the expression `if flip() then` M_1 `else` M_2, where the function `flip:unit→bool` alternates between `true` and `false` on each call.

Note that `M` implements the interface `I` given in the question. Now consider the term `t = M.f(M.x)`. This is well-typed, because `M.f : M.X→M.X` and `M.x : M.X`. But evaluation of `t` goes wrong by either applying `succ` to a value of type `Bool`, or `not` to a value of type `Int`!

9.5.2 SOLUTION: In a call-by-name setting, variables are no longer determinate. So, if we have an indeterminate module expression, we cannot "determinize" it by binding it to a variable, and there is no way to use its type components!

9.5.4 SOLUTION: For example, we might hash the same value in the two different hash tables, producing two hash codes that, because they have the same type, could be compared and (surprisingly) found to be different. Conversely, we could get unlucky and hash two *different* values to the same hash code.

9.5.9 SOLUTION: Since variables are determinate, `m` has the (principal) interface `I'` given by

```
interface I' = int {
  type X = m.X
  val x : X
  val f : X→X
}
```

which may be taken as the type of `n`.

9.5.10 SOLUTION: The interface

```
interface J = int {
  type X : *→*
  type Y : *
}
```

is a super-interface of the interface `I` that avoids `m`. For each type `A` the interface

```
interface K_A = int {
  type X : *→*
  type Y = X(A)
}
```

is also a super-interface of the interface `I` that avoids `m`. But the interface `J` is incomparable with every interface K_A, and the interfaces K_A and K_B are incomparable whenever `A` and `B` are inequivalent types.

For an example in $F_{\leq}$, see Ghelli and Pierce (1998).

9.7.3 SOLUTION: The interface of "value modules" is trivial:

```
interface VI = int {
  type Y
}
```

The parameterized version:

```
interface D =
  λk:K. λv:VI.
    int {
      type Dict
      val new : Dict
      val add : Dict → k.X → v.Y → Dict
      val member : Dict → k.X → Bool
      val lookup : Dict → k.X → v.Y
    }
```

An instance would be written `D` k_1 v_1, assuming that v_1 is some implementation of `VI`.

The fibered version:

```
interface D = int {
  module k : K
  module v : VI
  type Dict
  val new : Dict
  val add : Dict → k.X → v.Y → Dict
  val member : Dict → k.X → Bool
  val lookup : Dict → k.X → v.Y
}
```

9.8.1 SOLUTION: A functor interface is an interface *describing* module-level functions; a family of interfaces *is* a function from modules to interfaces. The body of a functor interface is a family of interfaces (indexed by the functor parameter). (In a sense, in the case of first-order module systems, there is no real need for functor interfaces *per se*, because there are no variable functors. We could just say that a family of modules has a family of interfaces, each module instance determining a corresponding interface instance. But for higher-order or separate compilation purposes we need a notation meaning "`F` is a functor implementing interface `I`".)

9.8.2 SOLUTION: Yes, but we need to make sure that the `dictFun` functor includes its parameter as a submodule of its result:

```
module dictFun =
  λ(k:ordered)
```

```
    mod {
      module key = k
      ... (as before)
    }

interface DictFun =
  Π(k:Ordered)
     int {
       module key = k
       type Dict : *→*
       val new : ∀V. Dict V
       val add : ∀V. Dict V → key.X → V → Dict V
       val member : ∀V. Dict V → key.X → Bool
       val lookup : ∀V. Dict V → key.X → V
     }
```

Or, more concisely,

```
DictFun = Π(m:K): (DF where k = m)
```

9.8.4 SOLUTION: The `compose8` functor will require 9 type parameters; `compose16` will require 17. Note that, in this series of examples, the part of each functor that is doing useful work is the same size as as its predecessor, while the amount of "nuisance parameterization" increases exponentially.

9.10.1 SOLUTION: A module in our sense corresponds to a "`.c`" file, which contains procedure and function definitions, type definitions, and declarations of global variables. Procedures, functions, and variables may be made private by declaring them `static`; otherwise they are presumed to be exported. An interface in our sense corresponds to a "`.h`" file, which contains procedure and function headers, type definitions, and declarations of global variables. The compiled versions of modules correspond to "`.o`" files, which are linked (e.g., using the `ld` command in Unix) into complete executable programs.

9.10.2 SOLUTION: A rigorous comparison of Java's modularity features with the ones described in this chapter is actually quite difficult. Here are some observations, however.

A class in Java is a medium-scale program structuring device, and is often a unit of abstraction, maintaining interesting invariants among its fields and allowing access to the fields only through its own methods. In these ways, a class is like a module. However, Java classes do not have type components. Conversely, class instantiation (in the sense of saying `new` to a class to get an object) is something that we don't do with modules. Also, classes in Java

are not units of compilation—it is not generally possible to compile a class separately from other classes that it refers to (e.g., because mutually recursive references are allowed).

An object in Java is also something like a module, providing a collection of named components; like classes, however, objects do not contain type components[1] —just methods (functions) and fields (reference cells holding pointers to objects).

Both Java interfaces and abstract classes (all of whose methods are virtual) are something like interfaces in the sense of this chapter, since they describe the components of an object without providing implementations.

Interfaces and abstract classes can be used to achieve separate compilation in Java, but in a somewhat different style from the separate compilation discussed in this chapter. One defines an interface `I`, then defines one class that implements `I` and, separately, another that expects to be given an object implementing `I`. These two classes can be compiled separately from each other.

Java's *packages* are also useful in structuring and decomposing the namespaces of large software systems, but they do not have many of the characteristics of modules in our sense: packages are not units of separate compilation, and there is no notion of an "interface of a package." This suggests that packages could perhaps be turned into something more like real modules by equipping them with interfaces. This extension has been explored by Bauer et al. (1999).

9.10.3 SOLUTION: Neither! In SML the so-called principal signature of a module is an internal data structure that cannot alway be written as an ML signature. In O'Caml the avoidance problem precludes existence of principal signatures.

10.0.1 SOLUTION: Assume that the subtyping relation contains `Bool <: Int`. Then we have

```
x : Int ⊢ (λy:Typeof(x). y+1)(3) : Int
```

since the principal type of `x` is `Int`. But subsumption yields `true : Int`, so by the obvious substitution we get

```
⊢ (λy:Typeof(true). y+1)(3) : Int
```

i.e., that

```
⊢ (λt:Bool. y+1)(3) : Int
```

1. ...except in experimental extensions with *virtual types* (Thorup, 1997; Torgersen, 1998; Igarashi and Pierce, 1999, etc.).

but this last term is ill-typed.

A similar example could be constructed for any distinct subtypes.

10.1.1 SOLUTION: Omitting the side condition allows not only

$$\vdash \texttt{(let X=Int in 3) : Int}$$

but also

$$\vdash \texttt{(let X=Int in 3) : X}$$

where the latter is referring to `X` outside of the scope of its definition. Thus, without this constraint, code such as

```
(let X=Int in λy:X.y+1) (let X=Int × Int in {5,4})
```

would typecheck because the function could be given type `X→Int` and the argument could be given type `X`. Clearly, however, this application must be rejected.

10.1.2 SOLUTION: If primitive definitions were added to the simply-typed lambda calculus, we would have to allow type variables to appear in types (requiring an extension of the syntax of types) and to allow type definitions to appear in the context. In contrast to $F^{\omega}_{\texttt{let}}$, though, every type variable would have a definition (and be of kind `*`, since the language lacks type operators). These definitions would induce a context-sensitive type equivalence relation based purely on definition expansion; this would be used by adding the F^{ω} rule T-EQ to the language.

Given the lack of type operators, a further extension might be to allow parameterized type definitions with fixed arity, i.e., to allow

$$\texttt{X(}Y_1\texttt{,}Y_2\texttt{) = }Y_1\texttt{→}Y_2$$

to appear in the context and then to allow fully-applied uses of `X` such as `X(Int,Bool→Bool)` to appear in types. This leads to the possibility of ill-formed types supplying the wrong number of arguments (e.g., `X` by itself or `X(Int)`) and so might require a type well-formedness judgment.

10.1.3 SOLUTION: This is an open question, but see (Cardelli, 1997) for some thoughts on the topic.

10.1.24 SOLUTION: We simultaneously prove that if $\Gamma \vdash \texttt{S :: K}$ and $\Gamma \vdash \texttt{T :: K}$ are both head-normal then $\Gamma \vdash \texttt{S} \equiv \texttt{T :: K}$ if and only if $\Gamma \mapsto \texttt{S} \leftrightarrow \texttt{T}$.

The "if" direction follows by induction on the execution of the equivalence algorithm. Assume that $\Gamma \vdash \texttt{S :: K}$ and $\Gamma \vdash \texttt{T :: K}$.

Case: $\Gamma \mapsto S \Leftrightarrow T$ because $\Gamma \mapsto S \Downarrow S'$, $\Gamma \mapsto T \Downarrow T'$, and $\Gamma \mapsto S' \longleftrightarrow T'$.

then by Proposition 10.1.8 $\Gamma \vdash S \equiv S' :: K$ and $\Gamma \vdash T \equiv T' :: K$, and further by Proposition 10.1.7 we have $\Gamma \vdash S' :: K$ and $\Gamma \vdash T' :: K$. Thus by the inductive hypothesis we have $\Gamma \vdash S' \equiv T' :: K$ and so by transitivity and symmetry we have $\Gamma \vdash S \equiv T :: K$.

Case: $\Gamma \mapsto S \leftrightarrow T$ because $S = T = X$.

Then $K = \Gamma(X)$ and the desired result follows using Q-REFL.

Case: $\Gamma \mapsto S \leftrightarrow T$ because $S = S_1 \to S_2$, $T = T_1 \to T_2$, $\Gamma \mapsto S_1 \Leftrightarrow T_1$, and $\Gamma \mapsto S_2 \Leftrightarrow T_2$.

Then $K = *$, and $\Gamma \vdash S_1 :: *$ and $\Gamma \vdash T_1 :: *$, so inductively we have we have $\Gamma \vdash S_1 \equiv T_1 :: *$. Similarly $\Gamma \vdash S_2 \equiv T_2 :: *$, so by Rule Q-ARROW we have $\Gamma \vdash S_1 \to S_2 \equiv T_1 \to T_2 :: *$ as required. The remaining cases for the "if" direction are similar.

To prove the "only if" direction, we first observe that an easy inductive proof shows that the algorithmic and structural type equivalence relations are symmetric and transitive. Further, if $\Gamma \vdash T :: K$ then we can show that $\Gamma \mapsto T \Leftrightarrow T$ (and $\Gamma \mapsto T \leftrightarrow T$ if T is weak head-normal) by induction first on the number of steps required to normalize T using a leftmost-outermost reduction sequence, and secondly on the size of T.

10.1.25 SOLUTION: As one example, the equivalence

$$X :: (*\Rightarrow *\Rightarrow *) = (\lambda Y :: (*\Rightarrow *) . Y\ \mathtt{Int}) \vdash X(\lambda Z :: * . Z) \equiv X(\lambda Z :: * . \mathtt{Int})$$

is provable, because both sides are provably equivalent to `Int`. However, given the same definition for X we have $X(\lambda Z :: * . \mathtt{Int}) \not\equiv X(\lambda Z :: * . \mathtt{Bool})$.

Pfenning and Schürmann, 1998 give a syntactic criterion for detecting a collection of *injective* type operators that yield equivalent results only when given equivalent arguments, and hence can be treated specially by an implementation.

10.2.1 SOLUTION: An equivalent most-precise signature would be

$$\Pi m : (\Sigma m' : (\!|*|\!) . (\!|!m'|\!)) . (\Sigma m'' : (\!|*=!m.1 \times !m.1|\!) . (\!|!m.1 \times !m.1|\!)),$$

which is a subsignature of infinitely many signatures, including

$$\Pi m : (\Sigma m' : (\!|*|\!) . (\!|!m'|\!)) . (\Sigma m'' : (\!|*|\!) . (\!|!m''|\!))$$

and

```
Πm: (Σm′:⦇*=Int⦈. ⦇Int⦈) .
          (Σm″:⦇*=Int × Int⦈. ⦇!m″⦈) .
```

10.2.2 SOLUTION: If `Int <: Top` then we should expect ⦇`Int`⦈ <: ⦇`Top`⦈ in analogy with depth subtyping for records. In contrast, the interfaces ⦇`*=Int`⦈ and ⦇`*=Top`⦈ must be unrelated, because a module containing the type `Int` is not a module containing the type `Top` nor vice versa. To see what goes wrong, assume that ⦇`*=Int`⦈ <: ⦇`*=Top`⦈ and let `M` be the module ⦇`Int`⦈. Then `M` : ⦇`*=Int`⦈, which by subsumption would further yield `M` : ⦇`*=Top`⦈. At this point, we could then show that `!M` ≡ `Int` and `!M` ≡ `Top` and hence that `Int` ≡ `Top`.

10.2.3 SOLUTION: The module defined by

```
let m = ⦇ ⦇Int::*⦈, ⦇3::Int⦈ ⦈ :> Σm:⦇*⦈.⦇!m⦈
in
   ⦇ ⦇ λX::*.(!m.1)::*⇒*⦈, !m.2 ⦈
```

satisfies the signature

```
Σm′:⦇*⇒*⦈. ⦇(!m′)(Int)⦈
```

and more generally the signature

```
Σm′:⦇*⇒*⦈. ⦇(!m′)(T)⦈
```

for any type `T`, but it satisfies no signature that is a subsignature of all of these.

10.2.4 SOLUTION: The only difficulty, in comparision with computing principal types for a language with subtyping is computing the types of variables. Due to the SELF rules, the principal signature of a module variable `m` is not the same as the interface associated with `m` in the typing context.

Therefore the algorithm is divided into two parts: there is an

A.1 PROPOSITION: 1. If Γ ⊢▸ `P : I` ⇒ `J` then Γ ⊢▸ `P : J` ⇒ `J`.

2. Γ ⊢▸ `P : J`$_1$ × `J`$_2$ if and only if Γ ⊢▸ `P.1 : J`$_1$ and Γ ⊢▸ `P.2 : J`$_2$.

3. If Γ ⊢ `P : I` then there exists a unique `J` such that Γ ⊢▸ `P : I` ⇒ `J` and Γ ⊢▸ `P : J` and Γ ⊢▸ `J <: I`.

4. If Γ ⊢ `M : I` then there exists `J` (determined by only Γ and `M`) such that Γ ⊢▸ `M : J`, Γ ⊢▸ `P : J` and Γ ⊢▸ `J <: I`.

□

Path Signatures $\boxed{\Gamma \vdash P : I \Rightarrow J}$

$$\Gamma \vdash P : (\!|T|\!) \Rightarrow (\!|T|\!)$$

$$\Gamma \vdash P : (\!|K|\!) \Rightarrow (\!|K=!P|\!)$$

$$\Gamma \vdash P : (\!|K=T|\!) \Rightarrow (\!|K=T|\!)$$

$$\Gamma \vdash P : \Pi m:I_1.I_2 \Rightarrow \Pi m:I_1.I_2$$

$$\Gamma \vdash P.1 : I_1 \Rightarrow J_1$$

$$\frac{\Gamma \vdash P.2 : [m \mapsto P.1]I_2 \Rightarrow J_2}{\Gamma \vdash P : \Sigma m:I_1.I_2 \Rightarrow J_1 \times J_2}$$

Principal Signatures $\boxed{\Gamma \vdash P : J}$

$$\frac{\Gamma \vdash m : \Gamma(m) \Rightarrow J}{\Gamma \vdash m : J}$$

$$\frac{\Gamma \vdash P_1 : J_1 \times J_2}{\Gamma \vdash P_1.1 : J_1}$$

$$\frac{\Gamma \vdash P_1 : J_1 \times J_2}{\Gamma \vdash P_1.2 : J_2}$$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash (\!|t|\!) : (\!|T|\!)}$$

$$\frac{\Gamma \vdash T :: K}{\Gamma \vdash (\!|T|\!) : (\!|K=T|\!)}$$

$$\frac{\Gamma \vdash P_1 : J_1 \qquad \Gamma \vdash P_2 : J_2}{\Gamma \vdash (\!|P_1, P_2|\!) : J_1 \times J_2}$$

$$\frac{\Gamma, m:J_1 \vdash M_2 : J_2}{\Gamma \vdash \lambda m:J_1.M_2 : \Pi m:J_1.J_2}$$

$$\frac{\Gamma \vdash P_1 : \Pi m:J_{11}.J_{12} \qquad \Gamma \vdash P_2 : J_{11}}{\Gamma \vdash P_1\, P_2 : [m \mapsto P_2]J_{12}}$$

$$\Gamma \vdash (P :> J) : J$$

Figure A-1: Principal Signature Algorithms

10.3.2 SOLUTION: We divide the proof into 5 steps:

1.
$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\cdot \vdash \diamond}{\vdash \mathtt{Int} :: *}}{\vdash \mathcal{S}(\mathtt{Int})}}{\mathtt{Y}::\mathcal{S}(\mathtt{Int}) \vdash \diamond} \qquad \dfrac{\vdots}{\mathtt{Y}::\mathcal{S}(\mathtt{Int}) \vdash \mathtt{Int} :: *}}{\mathtt{Y}::\mathcal{S}(\mathtt{Int}) \vdash \mathtt{Y} :: \mathcal{S}(\mathtt{Int})} \qquad \dfrac{\mathtt{Y}::\mathcal{S}(\mathtt{Int}) \vdash \mathtt{Y} :: \mathcal{S}(\mathtt{Int})}{\mathtt{Y}::\mathcal{S}(\mathtt{Int}) \vdash \mathcal{S}(\mathtt{Int}) <: *}}{\mathtt{Y}::\mathcal{S}(\mathtt{Int}) \vdash \mathtt{Y} :: *}}{\mathtt{Y}::\mathcal{S}(\mathtt{Int}) \vdash (\lambda \mathtt{X}::*.\mathtt{X})\,\mathtt{Y} \equiv \mathtt{Y} :: *}$$

2.
$$\dfrac{\dfrac{\dfrac{\vdots}{\mathtt{Y}::\mathcal{S}(\mathtt{Int}) \vdash \diamond} \qquad \dfrac{\vdots}{\mathtt{Y}::\mathcal{S}(\mathtt{Int}) \vdash \mathtt{Int} :: *}}{\mathtt{Y}::\mathcal{S}(\mathtt{Int}) \vdash \mathtt{Y} \equiv \mathtt{Int} :: \mathtt{S}(\mathtt{Int})} \qquad \dfrac{\mathtt{Y}::\mathcal{S}(\mathtt{Int}) \vdash \mathtt{Y} \equiv \mathtt{Int} :: \mathtt{S}(\mathtt{Int})}{\mathtt{Y}::\mathcal{S}(\mathtt{Int}) \vdash \mathcal{S}(\mathtt{Int}) <: *}}{\mathtt{Y}::\mathcal{S}(\mathtt{Int}) \vdash \mathtt{Y} \equiv \mathtt{Int} :: *}$$

3.

$$\dfrac{\dfrac{\vdots}{\texttt{Y::}\mathcal{S}\texttt{(Int), X::*} \vdash \texttt{X :: *}} \quad \dfrac{\vdots}{\texttt{Y::}\mathcal{S}\texttt{(Int)} \vdash \texttt{Y :: *}}}{\dfrac{\texttt{Y::}\mathcal{S}\texttt{(Int)} \vdash \texttt{(}\lambda\texttt{X::*.Int) Y} \equiv \texttt{Int :: *}}{\texttt{Y::}\mathcal{S}\texttt{(Int)} \vdash \texttt{Int} \equiv \texttt{(}\lambda\texttt{X::*.Int) Y} \equiv\texttt{:: *}}}$$

4.
$$\dfrac{\dfrac{\vdots}{\texttt{Y::}\mathcal{S}\texttt{(Int)} \vdash \texttt{(}\lambda\texttt{X::*.X) Y} \equiv \texttt{Y :: *}} \quad \dfrac{\vdots}{\texttt{Y::}\mathcal{S}\texttt{(Int)} \vdash \texttt{Y} \equiv \texttt{Int :: *}}}{\texttt{Y::}\mathcal{S}\texttt{(Int)} \vdash \texttt{(}\lambda\texttt{X::*.X) Y} \equiv \texttt{Int :: *}}$$

5.
$$\dfrac{\dfrac{\textit{Part 4} \qquad \dfrac{\vdots}{\texttt{Y::}\mathcal{S}\texttt{(Int)} \vdash \texttt{Int} \equiv \texttt{(}\lambda\texttt{X::*.X) Y :: *}}}{\texttt{Y::}\mathcal{S}\texttt{(Int)} \vdash \texttt{(}\lambda\texttt{X::*.X) Y} \equiv \texttt{(}\lambda\texttt{X::*.X) Y :: *}}\ \text{Q-TRANS}}{\vdash \texttt{(}\lambda\texttt{X::*.X)} \equiv \texttt{(}\lambda\texttt{X::*.X) :: S(Int)}\Rightarrow\texttt{*}}\ \text{Q-EXT}$$

10.3.8 SOLUTION: By the definitions in Figure 10-14 it is possible for $\mathcal{S}\,\texttt{(T::K)}$ to be a well-formed kind even if $\texttt{T}$ does not satisfy kind $\texttt{K}$; for example, take $\texttt{T} = \texttt{Int}$ and $\texttt{K} = \mathcal{S}\,\texttt{(Int}\rightarrow\texttt{Int)}$. Then we have $\Gamma \vdash \texttt{Int :: }\mathcal{S}\,\texttt{(Int::}\mathcal{S}\,\texttt{(Int}\rightarrow\texttt{Int))}$ but not $\Gamma \vdash \texttt{Int} \equiv \texttt{Int :: }\mathcal{S}\,\texttt{(Int}\rightarrow\texttt{Int)}$.

10.3.9 SOLUTION: Using the properties of Fact 10.3.7, we can show the admissibility of Q-BETA-FST.

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \texttt{T}_1\texttt{ :: K}_1}{\Gamma \vdash \texttt{T}_1\texttt{ :: }\mathcal{S}\,\texttt{(T}_1\texttt{ :: K}_1\texttt{)}} \quad \Gamma \vdash \texttt{T}_2\texttt{ :: }\mathcal{S}\,\texttt{(T}_1\texttt{ :: K}_2\texttt{)}}{\Gamma \vdash \texttt{\{T}_1\texttt{, T}_2\texttt{\}}\texttt{ :: }\mathcal{S}\,\texttt{(T}_1\texttt{ :: K}_1\texttt{)} \times \texttt{K}_2}}{\Gamma \vdash \pi_1\,\texttt{\{T}_1\texttt{, T}_2\texttt{\}}\texttt{ :: }\mathcal{S}\,\texttt{(T}_1\texttt{ :: K}_1\texttt{)}}}{\Gamma \vdash \pi_1\,\texttt{\{T}_1\texttt{, T}_2\texttt{\}} \equiv \texttt{T}_1\texttt{ :: K}_1}$$

The proof for Q-BETA-SND is exactly analogous, and a similar idea works for Q-BETA:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, \texttt{X::K}_{11} \vdash \texttt{T}_{12}\texttt{ :: K}_{12}}{\Gamma, \texttt{X::K}_{11} \vdash \texttt{T}_{12}\texttt{ :: }\mathcal{S}\,\texttt{(T}_{12}\texttt{ :: K}_{12}\texttt{)}} \quad \Gamma \vdash \texttt{T}_2\texttt{ :: K}_{12}}{\Gamma \vdash \texttt{(}\lambda\texttt{X::K}_{11}\texttt{.T}_{12}\texttt{) :: (}\Pi\texttt{X::K}_{11}\texttt{.}\mathcal{S}\,\texttt{(T}_{12}\texttt{ :: K}_{12}\texttt{))}}}{\Gamma \vdash \texttt{(}\lambda\texttt{X::K}_{11}\texttt{.T}_{12}\texttt{) T}_2\texttt{ :: }\mathcal{S}\,\texttt{([X}\mapsto\texttt{T}_2\texttt{]T}_{12}\texttt{ :: [X}\mapsto\texttt{T}_2\texttt{]K}_{12}\texttt{)}}}{\Gamma \vdash \texttt{(}\lambda\texttt{X::K}_{11}\texttt{.T}_{12}\texttt{) T}_2 \equiv \texttt{[X}\mapsto\texttt{T}_2\texttt{]T}_{12}\texttt{ :: [X}\mapsto\texttt{T}_2\texttt{]K}_{12}}$$

10.3.11 SOLUTION: Let $\Gamma_1 \stackrel{\text{def}}{=} \texttt{Y} :: (\mathcal{S}\,(\texttt{Int}) \Rightarrow *) \Rightarrow *$. Then

$$\texttt{Y} :: (\mathcal{S}\,(\texttt{Int}) \Rightarrow *) \Rightarrow * \;\vdash\!\blacktriangleright\; \texttt{Y}\,(\lambda \texttt{X} :: * . \texttt{X}) \Leftrightarrow \texttt{Y}\,(\lambda \texttt{X} :: * . \texttt{Int}) :: *$$

because

- $\Gamma_1 \;\vdash\!\blacktriangleright\; \texttt{Y}\,(\lambda \texttt{X} :: * . \texttt{X}) \Downarrow \texttt{Y}\,(\lambda \texttt{X} :: * . \texttt{X})$
- $\Gamma_1 \;\vdash\!\blacktriangleright\; \texttt{Y}\,(\lambda \texttt{X} :: * . \texttt{Int}) \Downarrow \texttt{Y}\,(\lambda \texttt{X} :: * . \texttt{Int})$
- $\Gamma_1 \;\vdash\!\blacktriangleright\; \texttt{Y}\,(\lambda \texttt{X} :: * . \texttt{X}) \leftrightarrow \texttt{Y}\,(\lambda \texttt{X} :: * . \texttt{X}) \uparrow *$, because
  - $\Gamma_1 \;\vdash\!\blacktriangleright\; \texttt{Y} \leftrightarrow \texttt{Y} \uparrow (\mathcal{S}\,(\texttt{Int}) \Rightarrow *) \Rightarrow *$, and
  - $\Gamma_1 \;\vdash\!\blacktriangleright\; \lambda \texttt{X} :: * . \texttt{X} \Leftrightarrow \lambda \texttt{X} :: * . \texttt{Int} : \mathcal{S}\,(\texttt{Int}) \Rightarrow *$, because
    - $\Gamma_1, \texttt{Z} :: \texttt{S}\,(\texttt{Int}) \;\vdash\!\blacktriangleright\; (\lambda \texttt{X} :: * . \texttt{X})\,\texttt{Z} \Leftrightarrow (\lambda \texttt{X} :: * . \texttt{Int})\,\texttt{Z} :: *$, because
      - $\Gamma_1, \texttt{Z} :: \texttt{S}\,(\texttt{Int}) \;\vdash\!\blacktriangleright\; (\lambda \texttt{X} :: * . \texttt{X})\,\texttt{Z} \Downarrow \texttt{Int}$
      - $\Gamma_1, \texttt{Z} :: \texttt{S}\,(\texttt{Int}) \;\vdash\!\blacktriangleright\; (\lambda \texttt{X} :: * . \texttt{Int})\,\texttt{Z} \Downarrow \texttt{Int}$
      - $\Gamma_1, \texttt{Z} :: \texttt{S}\,(\texttt{Int}) \;\vdash\!\blacktriangleright\; \texttt{Int} \leftrightarrow \texttt{Int} \uparrow *$.

References

Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991. Summary in *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*, 1990.

Rolf Adams, Walter Tichy, and Annette Weinert. The cost of selective recompilation and environment processing. *ACM Transactions on Software Engineering and Methodology*, 3(1):3–28, January 1994.

Amal Ahmed and David Walker. The logical approach to stack typing. In *ACM Workshop on Types in Compilation*, New Orleans, Lousiana, January 2003.

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

Alexander Aiken, Manuel Fähndrich, and Raph Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Programming Language Design and Implementation*, volume 30(6), pages 174–185, 18–21 June 1995. URL `http://www.cs.berkeley.edu/~aiken/publications/papers/pldi95.ps`.

Alexander Aiken and Edward L. Wimmers. Solving systems of set constraints (extended abstract). In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 329–340, Santa Cruz, California, 22–25 June 1992. IEEE Computer Society Press.

Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *ACM Symposium on Functional Programming Languages and Computer Architecture (FPCA)*, pages 31–41, 1993.

Alexander S. Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, pages 163–173, January 1994.

Thorsten Altenkirch. *Constructions, Inductive Types and Strong Normalization*. PhD thesis, LFCS, University of Edinburgh, 1993. URL `http://www.lfcs.informatics.ed.ac.uk/reports/93/ECS-LFCS-93-279/index.h%tml`.

Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. In POPL POPL (a), pages 104–118.

Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993. Summary in *ACM Symposium on Principles of Programming Languages (POPL), Orlando, Florida*, pp. 104–118; also DEC/Compaq Systems Research Center Research Report number 62, August 1990.

Torben Amtoft, Flemming Nielson, and Hanne Riis Nielson. *Type and Effect Systems: Behaviours for Concurrency*. IC Press, 1999.

Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *ACM Symposium on Principles of Programming Languages (POPL), Boston, Massachusetts*, pages 243–253, January 2000.

David Aspinall. Subtyping with Singleton Types. In *Proceedings of Computer Science Logic (CSL '94)*, 1995. In LNCS 933.

David Aspinall and Martin Hofmann. Another type system for in-place update. In *European Symposium on Programming*, number 2305 in Lecture Notes in Computer Science, pages 36–52, Grenoble, France, April 2002. Springer.

Lennart Augustsson. Cayenne — a language with dependent types. In *International Conference on Functional Programming (ICFP), Baltimore, Maryland, USA*, pages 239–250, 1998.

Henry G. Baker. Lively linear Lisp – 'Look Ma, no garbage!'. *ACM Sigplan Notices*, 27 (8):89–98, August 1992.

Henk P. Barendregt. *The Lambda Calculus*. North Holland, revised edition, 1984.

Henk P. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991.

Henk P. Barendregt. Lambda calculi with types. In Abramsky, Gabbay, and Maibaum, editors, *Handbook of Logic in Computer Science*, volume II. Oxford University Press, 1992.

Erik Barendsen and Sjaak Smetsers. Conventional and uniqueness typing in graph rewrite systems. In Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science*, number 761 in LNCS, pages 41–51, Bombay, India, 1993. Springer-Verlag.

Bruno Barras, Samuel Boutin, Cristina Cornes, Judicael Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saibi, and Benjamin Werner. The Coq proof assistant reference manual : Version 6.1. Technical Report RT-0203, Inria (Institut National de Recherche en Informatique et en Automatique), France, 1997.

Lujo Bauer, Andrew W. Appel, and Edward W. Felten. Mechanisms for secure modular programming in java. Technical Report TR-603-99, 1999. URL `citeseer.nj.nec.com/bauer99mechanisms.html`.

Mike Beaven and Ryan Stansifer. Explaining type errors in polymorphic languages. *ACM Letters on Programming Languages and Systems*, 2(4):17–30, March 1993.

Stephan Bellantoni and Stephan Cook. A new recursion-theoretic characterization of polytime functions. *Computational Complexity*, 2(2):97–110, 1992.

Stephan Bellantoni, K.-H. Niggl, and H. Schwichtenberg. Higher type recursion, ramification and polynomial time. *Annals of Pure and Applied Logic*, 104:17–30, 2000.

Stefano Berardi. Towards a mathematical analysis of the Coquand-Huet calculus of constructions and the other systems in Barendregt's cube. Technical report, Department of Computer Science, CMU, and Dipartimento Matematica, Universita di Torino, 1988.

K. Bernstein and E. W. Stark. Debugging type errors. `http://bsd7.starkhome.cs.sunysb.edu/~stark/REPORTS/debugtype.ps.gz`, November 1995.

Bernard Berthomieu. Tagged types. a theory of order sorted types for tagged expressions. Research Report 93083, LAAS, 7, avenue du Colonel Roche, 31077 Toulouse, France, March 1993.

Bernard Berthomieu and Camille le Moniès de Sagazan. A calculus of tagged types, with applications to process languages. In *Workshop on Types for Program Analysis*, pages 1–15, May 1995.

Edoardo Biagioni, Nicholas Haines, Robert Harper, Peter Lee, Brian G. Milnes, and Eliot B. Moss. Signatures for a protocol stack: A systems application of Standard ML. In *LISP and Functional Programming*, Orlando, FL, June 1994.

G. M. Bierman, A. M. Pitts, and C. V. Russo. Operational properties of Lily, a polymorphic linear lambda calculus with recursion. In *Fourth International Workshop on Higher Order Operational Techniques in Semantics, Montréal*, volume 41 of *Electronic Notes in Theoretical Computer Science*. Elsevier, September 2000. URL `http://www.elsevier.nl/locate/entcs/volume41.html`.

L. Birkedal and R. W. Harper. Constructing interpretations of recursive types in an operational setting. *Information and Computation*, 155:3–63, 1999.

Lars Birkedal and Mads Tofte. A constraint-based region inference algorithm. *Theoretical Computer Science*, 258:299–392, 2001. URL `http://www.it-c.dk/people/birkedal/papers/conria.ps.gz`.

Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Principles of Programming Languages*, pages 171–183, New York, NY, USA, 21–24 January 1996. ACM Press. ISBN 0-89791-769-3. URL `http://www.it-c.dk/people/birkedal/papers/reginm.ps.gz`.

Matthias Blume. *The SML/NJ Compilation and Library Manager*, May 2002. URL `http://www.smlnj.org/doc/CM/index.html`.

Matthias Blume and Andrew W. Appel. Hierarchical modularity. *ACM Transactions on Programming Languages and Systems*, 21(4):813–847, 1999. URL `citeseer.nj.nec.com/blume98hierarchical.html`.

Daniel Bonniot. Type-checking multi-methods in ML (a modular approach). In *Workshop on Foundations of Object-Oriented Languages (FOOL), informal proceedings*, January 2002.

Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. In Roger Hindley, editor, *Proc. 3d Int'l Conf. on Typed Lambda Calculi and Applications (TLCA), Nancy, France, April 2–4, 1997*, volume 1210 of *Lecture Notes in Computer Science (LNCS)*, pages 63–81. Springer-Verlag, April 1997. Full version in Fundamenta Informaticae, Vol. 33, pp. 309–338, 1998.

Kim B. Bruce. Typing in object-oriented languages: Achieving expressibility and safety, 1995. Available through `http://www.cs.williams.edu/simkim`.

Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, the Hopkins Objects Group (Jonathan Eifrig, Scott Smith, Valery Trifonov), Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1996.

Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. In *Theoretical Aspects of Computer Software (TACS), Sendai, Japan*, September 1997. An earlier version was presented as an invited lecture at the Third International Workshop on Foundations of Object Oriented Languages (FOOL 3), July 1996.

T. H. Brus, M. C. J. D. van Eekelen, M. O. van Leer, and M. J. Plasmeijer. Clean: A language for functional graph rewriting. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 364–384. Springer-Verlag, Berlin, DE, 1987. ISBN 3-540-18317-5. Lecture Notes in Computer Science 274; Proceedings of Conference held at Portland, OR.

Michele Bugliesi and Santiago M. Pericás-Geertsen. Type inference for variant object types. *Information and Computation*, 177(1):2–27, August 2002.

Rod Burstall and Butler Lampson. A kernel language for abstract data types and modules. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 1–50. Springer-Verlag, 1984.

Rod Burstall, David MacQueen, and Donald Sannella. HOPE: An experimental applicative language. In *Proceedings of the 1980 LISP Conference*, pages 136–143, Stanford, California, 1980. Stanford University.

Cristiano Calcagno. Stratified operational semantics for safety and correctness of region calculus. In POPL POPL (b), pages 155–165. ISBN 1-58113-336-7. URL `ftp://ftp.disi.unige.it/person/CalcagnoC/regions.ps`.

Cristiano Calcagno, Simon Helsen, and Peter Thiemann. Syntactic type soundness results for the region calculus. *Information & Computation*, 173(2):199–221, 2002. URL `http://www.swen.uwaterloo.ca/~shelsen/calcagno-helsen-thiemann-iandc-20%01.pdf`.

Luca Cardelli. A polymorphic λ-calculus with Type:Type. Research report 10, DEC/Compaq Systems Research Center, May 1986.

Luca Cardelli. Phase distinctions in type theory. unpublished manuscript, 1988a.

Luca Cardelli. Typechecking dependent types and subtypes. In M. Boscarol, L. Carlucci Aiello, and G. Levi, editors, *Foundations of Logic and Functional Programming, Workshop Proceedings, Trento, Italy, (Dec. 1986)*, volume 306 of *Lecture Notes in Computer Science*, pages 45–57. Springer-Verlag, 1988b.

Luca Cardelli. Program fragments, linking, and modularization. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 266–277, Paris, France, January 1997. ACM Press.

Luca Cardelli, Jim Donahue, Mick Jordan, Bill Kalso, and Greg Nelson. The modula-3 type system. In *SixteenthACM Symposium on Principles of Programming Languages (POPL)*, pages 202–212, Austin, TX, January 1989.

Luca Cardelli and Xavier Leroy. Abstract types and the dot notation. In *Proceedings of the IFIP TC2 Working Conference on Programming Concepts and Methods*. North Holland, 1990a. Also appeared as DEC/Compaq SRC technical report 56.

Luca Cardelli and Xavier Leroy. Abstract types and the dot notation. Technical Report 56, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, March 1990b.

Luca Cardelli and Giuseppe Longo. A semantic basis for Quest. *Journal of Functional Programming*, 1(4):417–458, October 1991. Summary in ACM Conference on Lisp and Functional Programming, June 1990. Also available as DEC/Compaq SRC Research Report 55, Feb. 1990.

Luca Cardelli and John Mitchell. Operations on records. *Mathematical Structures in Computer Science*, 1:3–48, 1991. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, MIT Press, 1994; available as DEC/Compaq Systems Research Center Research Report #48, August, 1989, and in the proceedings of MFPS '89, Springer LNCS volume 442.

J. Cartmell. Generalised algebraic theories and contextual categories. *Annals of Pure and Applied Logic*, 32:209–243, 1986.

R. Cartwright and M. Fagan. Soft typing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 278–292, June 1991. Also available as SIGPLAN Notices 26(6) June 1991.

Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. *Theoretical Computer Science*, 232(1–2):133–163, February 2000.

Iliano Cervesato and Frank Pfenning. A linear logical framework. *Information and Computation*, 179(1):19–75, November 2002.

Sagar Chaki, Sriram K. Rajamani, and Jakob Rehof. Types as models: Model checking message-passing programs. In *Principles of Programming Languages*, pages 45–57, New York, NY, USA, 16–18 January 2002. ACM Press. ISBN 1-58113-450-9.

Jawahar Chirimar, Carl A. Gunter, and Jon G. Riecke. Reference counting as a computational interpretation of linear logic. *Journal of Functional Programming*, 6(2): 195–244, March 1996.

Olaf Chitil. Compositional explanation of types and algorithmic debugging of type errors. In *International Conference on Functional Programming (ICFP)*, pages 193–204, September 2001.

Venkatesh Choppella. *Unification Source-tracking with Application to Diagnosis of Type Inference*. PhD thesis, Indiana University, August 2002.

Alonzo Church. The calculi of lambda-conversion. *The Annals of Mathematical Studies*, 6, 1941.

Alonzo Church. The weak theory of implication. *Kontroliertes Denken: Untersuchungen zum Logikkalk ul und zur Logik der Einzelwissenschaften*, pages 22–37, 1951.

Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ML. In *ACM Symposium on Lisp and Functional Programming (LFP)*, pages 13–27, August 1986.

Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for Java. *ACM SIGPLAN Notices*, 35(5):95–107, May 2000. ISSN 0362-1340.

Hubert Comon. Constraints in term algebras (short survey). In *Conference on Algebraic Methodology and Software Technology (AMAST)*, Workshops in Computing. Springer-Verlag, 1993.

Hubert Comon and Pierre Lescanne. Equational problems and disunification. *Journal of Symbolic Computation*, 7:371–425, 1989.

R. L. et al Constable. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, NJ, 1986. URL `http://www.nuprl.org/book/doc.html`.

Catarina Coquand. The AGDA proof system homepage, 1998. At `http://www.cs.chalmers.se/~catarina/agda/`.

Thierry Coquand. An analysis of Girard's paradox. In *Proceedings, Symposium on Logic in Computer Science* IEE (1986), pages 227–236.

Thierry Coquand. An algorithm for testing conversion in type theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 255–279. Cambridge University Press, 1991a.

Thierry Coquand. An Algorithm for Testing Conversion in Type Theory. In Gérard Huet and G. Plotkin, editors, *Logical frameworks*, pages 255–277. Cambridge University Press, 1991b.

Thierry Coquand. Pattern matching with dependent types. In *Proceedings of the Workshop on Types for Proofs and Programs, Baastad*. Informal proceedings available by ftp from `ftp://ftp.cs.chalmers.se/pub/cs-reports/baastad.92/proc.ps.Z`, 1992.

Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

Judicaël Courant. Strong normalization with singleton types. In *Proceeedings of the Second Workshop on Intersection Types and Related Systems (ITRS '02)*, volume 70 of *ENTCS*, 2002.

Bruno Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25(2):95–169, March 1983.

Erik Crank and Matthias Felleisen. Parameter-passing and the lambda calculus. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 233–244, January 1991.

Karl Crary. Toward a foundational typed assembly language. In *ACM Symposium on Principles of Programming Languages (POPL), New Orleans, Louisiana*, pages 198–212, January 2003.

Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? In *SIGPLAN '99 Conference on Programming Language Design and Implementation (PLDI)*, pages 50–63, Atlanta, GA, 1999a. ACM SIGPLAN.

Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 262–275, January 1999b.

Karl Crary, Stephanie Weirich, and J. Gregory Morrisett. Intensional polymorphism in type-erasure semantics. In *International Conference on Functional Programming (ICFP), Baltimore, Maryland, USA*, pages 301–312, 1998.

Pavel Curtis. *Constrained Quantification in Polymorphic Type Analysis*. PhD thesis, Cornell University, February 1990.

Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *ACM Symposium on Principles of Programming Languages (POPL), Albuquerque, New Mexico*, pages 207–212, 1982.

Olivier Danvy. Functional unparsing. *Journal of Functional Programming*, 8(6), 1998.

Nicolas G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606. Academic Press, 1980.

Rob DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 59–69, Snowbird, Utah, June 2001.

Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah*, pages 56–69, June 2001.

Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 59–69, June 2001.

James Donahue and Alan Demers. Data types are values. *ACM Transactions on Programming Languages and Systems*, 7(3):426–445, July 1985.

Kosta Došen and Peter Schroeder-Heister, editors. *Substructural Logics*, chapter A historical introduction to substructural logics, pages 1–30. Oxford University Press, 1993.

Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Higher order unification via explicit substitutions. Research Report 2709, INRIA, November 1995.

Gilles Dowek, Thérèse Hardin, Claude Kirchner, and Frank Pfenning. Unification via explicit substitutions: the case of higher-order patterns. Research Report 3591, INRIA, December 1998.

Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. In *POPL 2003: Proceedings of the 30th ACM SIGPLAN-SIGACT Sumposium on Principles of Programming Languages*, pages 236–249, New Orleans, January 2003.

Dominic Duggan and Frederick Bent. Explaining type inference. *Science of Computer Programming*, 27(1), June 1996.

Dirk Dussart, Fritz Henglein, and Christian Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In Alan Mycroft, editor, *Static Analysis Symposium (SAS)*, volume 983 of *Lecture Notes in Computer Science*, pages 118–135. Springer-Verlag, September 1995a.

Dirk Dussart, Fritz Henglein, and Christian Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In Alan Mycroft, editor, *Static Analysis Symposium*, volume 983 of *Lecture Notes in Computer Science*, pages 118–135, Heidelberg, Germany, 25–27 September 1995b. Springer-Verlag. ISBN 3-540-60360-3.

Thomas Erhard. A categorical semantics of constructions. In *Symposium on Logic in Computer Science*, pages 264–273, July 1988.

Manuel Fähndrich and Rob DeLine. Adoption and focus: Practical linear types for imperative programming. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 13–24, Berlin, June 2002.

Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Programming Language Design and Implementation*, volume 35(5) of *SIGPLAN Notices*, pages 253–263, New York, NY, USA, 18–21 June 2000. ACM Press. ISBN 1-58113-199-2.

M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103:235–271, 1992.

Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. Catching bugs in the web of program invariants. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1996.

Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 236–248, 1998. URL `citeseer.nj.nec.com/flatt98unit.html`.

Matthew Fluet and Riccardo Pucella. Phantom types and subtyping. In *IFIP International Conference on Theoretical Computer Science (TCS)*, pages 448–460, August 2002.

Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Principles of Programming Languages*, January 1996.

Alexandre Frey. Satisfying subtype inequalities in polynomial space. In Pascal Van Hentenryck, editor, *International Symposium on Static Analysis (SAS)*, number 1302 in Lecture Notes in Computer Science, pages 265–277. Springer-Verlag, September 1997.

You-Chin Fuh and Prateek Mishra. Type inference with subtypes. In H. Ganzinger, editor, *European Symp. on Programming (ESOP)*, volume 300 of *Lecture Notes in Computer Science*, pages 94–114. Springer-Verlag, 1988.

Jun P. Furuse and Jacques Garrigue. A label-selective lambda-calculus with optional arguments and its compilation method. RIMS Preprint 1041, Kyoto University, October 1995.

Manuel Fähndrich. BANE*: A Library for Scalable Constraint-Based Program Analysis*. PhD thesis, University of California at Berkeley, 1999.

Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable context-sensitive flow analysis using instantiation constraints. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, British Columbia, Canada*, June 2000.

Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13(3–5):341–363, July 2002.

Jacques Garrigue. Programming with polymorphic variants. In *Workshop on ML*, September 1998.

Jacques Garrigue. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering*, November 2000.

Jacques Garrigue. Simple type inference for structural polymorphism. In *Workshop on Foundations of Object-Oriented Languages (FOOL), informal proceedings*, January 2002.

Jacques Garrigue. Relaxing the value restriction. Draft., August 2003.

Jacques Garrigue and Didier Rémy. Extending ML with semi-explicit higher-order polymorphism. *Information and Computation*, 155(1):134–169, 1999.

Jaques Garrigue and Hassan Aït-Kaci. The typed polymorphic label-selective lambda-calculus. In *ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, pages 35–47, 1994.

Benedict R. Gaster. *Records, variants and qualified types*. PhD thesis, University of Nottingham, July 1998.

Benedict R. Gaster and Mark P. Jones. A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, Department of Computer Science, University of Nottingham, November 1996.

David Gay and Alexander Aiken. Language support for regions. In *Programming Language Design and Implementation*, volume 36(5) of *SIGPLAN Notices*, pages 70–80, New York, NY, USA, 20–22 June 2001. ACM Press. ISBN 1-58113-414-2. URL `http://www.cs.berkeley.edu/~dgay/papers/pldi01.ps`.

Giorgio Ghelli and Benjamin Pierce. Bounded existentials and minimal typing. *Theoretical Computer Science*, 193:75–96, 1998. Circulated in manuscript form in 1992.

David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *LISP and Functional Programming*, pages 28–38, New York, NY, USA, 4–6 August 1986. ACM Press.

J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmetique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972a. Thèse de doctorat d'état.

Jean-Yves Girard. *Interprétation Fonctionnelle et Élimination des Coupures dans l'Arithmétique d'Ordre Supérieure*. PhD thesis, Université Paris VII, 1972b.

Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

Jean-Yves Girard. Light linear logic. *Information and Computation*, 143, 1998.

Neal Glew. Type dispatch for named hierarchical types. In *International Conference on Functional Programming (ICFP), Paris, France*, pages 172–182, 1999.

GNU. GNU C library, version 2.2.5, 2001. URL `http://www.gnu.org/manual/glibc-2.2.5/html_mono/libc.html`.

Healfdene Goguen. *A Typed Operational Semantics for Type Theory*. PhD thesis, LFCS, University of Edinburgh, 1994. URL `http://www.lfcs.informatics.ed.ac.uk/reports/94/ECS-LFCS-94-304/index.h%tml`. Report ESC-LFCS-94-304.

A. D. Gordon. *Functional Programming and Input/Output*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.

A. D. Gordon. Bisimilarity as a theory of functional programming. In *Eleventh Conference on the Mathematical Foundations of Programming Semantics, New Orleans, 1995*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995.

A. D. Gordon. Operational equivalences for untyped and polymorphic object calculi. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute, pages 9–54. Cambridge University Press, 1998.

Andrew D. Gordon and Alan Jeffrey. Authenticity by typing for security protocols. In *Proc. 14th IEEE Computer Security Foundations Workshop (CSFW 2001), Cape Breton*, pages 145–159, 2001a.

Andrew D. Gordon and Alan Jeffrey. Typing correspondence assertions for communiation protocols. *Electronic Notes in Theoretical Computer Science*, 45:22 pages, 2001b. `http://www.elsevier.nl/locate/entcs/volume45.html`.

Andrew D. Gordon and Alan Jeffrey. Types and effects for asymmetric cryptographic protocols. In *Proc. 15th IEEE Computer Security Foundations Workshop (CSFW 2002), Cape Breton*, pages 77–91, 2002.

Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979a.

Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*. Springer-Verlag LNCS 78, 1979b.

Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 282–293, June 2002a.

Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation (PLDI'02)*, pages 282–293. ACM Press, 2002b.

Jörgen Gustavsson and Josef Svenningsson. Constraint abstractions. In Olivier Danvy and Andrzej Filinski, editors, *Programs as Data Objects*, volume 2053 of *LNCS*, pages 63–83, Heidelberg, Germany, 21–23 May 2001a. Springer-Verlag. ISBN 3-540-42068-1.

Jörgen Gustavsson and Josef Svenningsson. Constraint abstractions. In *Symposium on Programs as Data Objects*, volume 2053 of *Lecture Notes in Computer Science*. Springer-Verlag, May 2001b.

Jr. Guy L. Steele. *Common Lisp the Language*. Digital Press, 1990.

Christian Haack and J. B. Wells. Type error slicing in implicitly typed, higher-order languages. In *European Symp. on Programming (ESOP)*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*. ACM Press, June 2002. Berlin, Germany.

Thomas Hallgren and Aarne Ranta. An extensible proof text editor (abstract). In *Logic for Programming and Automated Reasoning (LPAR'2000)*, pages 70–84. Springer-Verlag LNCS/LNAI. 1955, 2000.

Nadeem Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. A syntactic approach to foundational proof-carrying code. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 89–100, July 2002.

David R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software—Practice and Experience*, 20(1):5–12, 1990.

R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *JACM*, 40(1): 143–184, 1993a. URL `http://www.cs.cmu.edu/~fp/elf-papers/jacm93.dvi.gz`.

Robert Harper. On equivalence and canonical forms in the LF type theory. (Submitted for publication.), August 2002.

Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993b. Summary in LICS'87.

Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, pages 123–137, January 1994.

Robert Harper and John C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, April 1993.

Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *SeventeenthACM Symposium on Principles of Programming Languages (POPL)*, San Francisco, CA, January 1990a.

Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*, pages 341–354, January 1990b.

Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *ACM Symposium on Principles of Programming Languages (POPL), Orlando, Florida*, pages 131–142, January 1991. Extended version available as Carnegie Mellon Technical Report CMU-CS-90-157.

Robert Harper and Robert Pollack. Type checking with universes. *Theoretical Computer Science*, 89:107–136, 1991.

Robert Harper and Chris Stone. A type-theoretic interpretation of Standard ML. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction: Essays in Honor of Robin Milner*. MIT Press, 2000a.

Robert Harper and Christopher Stone. A type-theoretic interpretation of Standard ML. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000b.

Nevin Heintze. Set based analysis of ML programs. Technical Report CMU-CS-93-193, Carnegie Mellon University, School of Computer Science, July 1993.

Simon Helsen and Peter Thiemann. Syntactic type soundness for the region calculus. In Alan Jeffrey, editor, *ACM Workshop on Higher Order Operational Techniques in Semantics*, volume 41(3) of *Electronic Notes in Theoretical Computer Science*, pages 1–20. Elsevier, September 2000. URL `http://www.elsevier.nl/locate/entcs/volume41.html`.

Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, 1993.

Fritz Henglein, Henning Makholm, and Henning Niss. A direct approach to control-flow sensitive region-based memory management. In *Proceedings of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 175–186, Firenze, Italy, September 2001. ACM Press. URL `http://www.diku.dk/~hniss/publications/ppdp2001-abstract.html`.

Fritz Henglein and Christian Mossin. Polymorphic binding-time analysis. In Donald Sannella, editor, *5th European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 287–301, Heidelberg, Germany, 11–13 April 1994. Springer-Verlag. ISBN 3-540-57880-3.

Tom Hirschowitz and Xavier Leroy. Mixin modules in a call-by-value setting. In *European Symposium on Programming*, pages 6–20, 2002. URL `citeseer.nj.nec.com/article/hirschowitz02mixin.html`.

C. A. R. Hoare. Proof of correctness of data representation. *Acta Informatica*, 1:271–281, 1972.

Martin Hofmann. A mixed modal/linear lambda calculus with applications to bellantoni-cook safe recursion. In *Conference for Computer Science Logic*, Aarhus, Denmark, August 1997a.

Martin Hofmann. Syntax and semantics of dependent types. In P. Dybjer and A. Pitts, editors, *Semantics of Logics of Computation*, chapter 3. Cambridge University Press, 1997b. URL `http://www.mathematik.th-darmstadt.de/~mh/cupart.dvi.gz`.

Martin Hofmann. Linear types and non-size-increasing polynomial time computation. In *Logic in computer science*, pages 464–473, Los Alamitos, CA, June 1999.

Martin Hofmann. Safe recursion with higher types and BCK-algebra. *Annals of Pure and Applied Logic*, 2000.

F. Honsell, I. A. Mason, S. F. Smith, and C. L. Talcott. A variable typed logic of effects. *Information and Computation*, 119(1):55–90, 1995.

William A. Howard. Hereditarily majorizable functionals of finite type. In Anne Sjerp Troelstra, editor, *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, volume 344 of *Lecture Notes in Mathematics*, pages 454–461. Springer-Verlag, Berlin, 1973. Appendix.

William A. Howard. The formulas-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, New York, 1980. Reprint of 1969 article.

D. J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, 1996.

Paul Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzman, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. Report on the programming language Haskell, version 1.2. *SIGPLAN Notices*, 27(5), May 1992.

Gérard Huet. *Résolution d'equations dans les langages d'ordre 1,2, ...,ω*. Thèse de Doctorat d'Etat, Université de Paris 7 (France), 1976.

Proceedings, Symposium on Logic in Computer Science, Cambridge, Massachusetts, 16–18 June 1986. IEEE Computer Society.

Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. In POPL POPL (b), pages 128–141. ISBN 1-58113-336-7.

Atsushi Igarashi and Benjamin C. Pierce. Foundations for virtual types. In *European Conference on Object-Oriented Programming (ECOOP)*, 1999. Also in informal proceedings of the Sixth International Workshop on Foundations of Object-Oriented Languages (FOOL). Full version to appear in *Information and Computation*.

Bart Jacobs. *Categorical Logic and Type Theory*. Studies in Logic and the Foundations of Mathematics 141. North Holland, Elsevier, 1999.

Lalita A. Jategaonkar. ML with extended pattern matching and subtypes. Master's thesis, MIT, August 1989.

Lalita A. Jategaonkar and John C. Mitchell. ML with extended pattern matching and subtypes (preliminary version). In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 198–211, Snowbird, Utah, July 1988.

Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report*. Springer-Verlag, second edition, 1975.

Thomas Jensen. Inference of polymorphic and conditional strictness properties. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 209–221. ACM Press, January 1998. `http://www.irisa.fr/lande/jensen/papers/popl98.ps`.

Trevor Jim. What are principal typings and what are they good for? Technical Report MIT/LCS TM-532, Massachusetts Institute of Technology, August 1995.

Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 275–288. USENIX Association, 2002.

Trevor Jim and Jens Palsberg. Type inference in systems of recursive types with subtyping. Manuscript, 1999.

P. Johann. A generalization of short-cut fusion and its correctness proof. *Higher-Order and Symbolic Computation*, 15(4):273–300, 2002.

Gregory F. Johnson and Janet A. Walz. A maximum-flow approach to anomaly isolation in unification-based incremental type inference. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 44–57, January 1986.

Mark P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, Cambridge, England, 1994.

Mark P. Jones. Using parameterized signatures to express modular structure. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, St. Petersburg, Florida, 21–24, 1996. ACM Press. URL `citeseer.nj.nec.com/jones96using.html`.

Mark P. Jones. Typing Haskell in Haskell. In *Haskell Workshop*, October 1999.

Mark P. Jones and Simon Peyton Jones. Lightweight extensible records for Haskell. In *Haskell Workshop*, October 1999.

Jean-Pierre Jouannaud and Claude Kirchner. Solving equations in abstract algebras: a rule-based survey of unification. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 8, pages 257–321. MIT Press, 1991.

Pierre Jouvelot and David K. Gifford. Reasoning about continuations with control effects. In *Programming Language Design and Implementation*, volume 24(7), pages 218–226, 21–23 June 1989.

Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In POPL POPL (a), pages 303–310.

A. Jung and A. Stoughton. Studying the fully abstract model of PCF within its continuous function model. In M. Bezem and J.M. Groote, editors, *Proc. Typed Lambda Calculi and Applications (TLCA)*, volume 664 of *Lecture Notes in Computer Science*, pages 230–244. Springer-Verlag, 1993.

L.S. van Benthem Jutting, James McKinna, and Robert Pollack. Checking algorithms for Pure Type Systems. In Henk Barendregt and Tobias Nipkow, editors, *Proceedings of the International Workshop on Types for Proofs and Programs*, pages 19–61, Nijmegen, The Netherlands, May 1994. Springer-Verlag LNCS 806.

Assaf J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, April 1993a.

Assaf J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. The undecidability of the semi-unification problem. *Information and Computation*, 102(1):83–101, January 1993b. Summary in *STOC 1990*.

Assaf J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. An analysis of ML typability. *Journal of the ACM*, 41(2):368–398, March 1994.

Claude Kirchner and F. Klay. Syntactic theories and unification. In *Proceedings 5th IEEE Symposium on Logic in Computer Science, Philadelphia (Pa., USA)*, pages 270–277, June 1990.

Naoki Kobayashi. Quasi-linear types. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming languages*, pages 29–42, San Antonio, January 1999.

Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient recursive subtyping. *Mathematical Structures in Computer Science*, 5(1):113–125, 1995.

Viktor Kuncak and Martin Rinard. Structural subtyping of non-recursive types is decidable. In *IEEE Symposium on Logic in Computer Science (LICS)*, June 2003.

Yves Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988.

Joachim Lambek. The mathematics of sentence structure. *American Mathematical Monthly*, 65:154–170, 1958.

B. Lampson and R. Burstall. Pebble, a kernel language for modules and abstract data types. *Information and Computation*, 76:278–346, February/March 1988.

S. B. Lassen. *Relational Reasoning about Functions and Nondeterminism*. PhD thesis, Department of Computer Science, University of Aarhus, 1998.

Jean-Louis Lassez, Michael J. Maher, and Kim G. Marriott. Unification revisited. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, chapter 15, pages 587–625. Morgan Kaufmann, 1988.

Konstantin Läufer and Martin Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1411–1430, September 1994. Summary in *Phoenix Seminar and Workshop on Declarative Programming*, Nov. 1991.

Fabrice Le Fessant and Luc Maranget. Optimizing pattern matching. In *International Conference on Functional Programming (ICFP), Firenze, Italy*. ACM Press, 2001.

Oukseh Lee and Kwangkeun Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4): 707–723, 1998.

Daniel Leivant. Stratified functional programs and computational complexity. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming languages*, pages 325–333, jan 1993.

Xavier Leroy. Polymorphic typing of an algorithmic language. Research Report 1778, INRIA, October 1992.

Xavier Leroy. Manifest types, modules, and separate compilation. In *Proceedings of the Twenty-first Annual ACM Symposium on Principles of Programming Languages, Portland*. ACM, January 1994a.

Xavier Leroy. Manifest types, modules and separate compilation. In *ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, pages 109–122, January 1994b.

Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Conference Record of POPL '95: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 142–153, San Francisco, CA, January 1995a.

Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Proceedings of the Twenty-Second ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, pages 142–153, San Francisco, California, January 1995b.

Xavier Leroy. The Objective Caml system: Documentation and user's guide. Available at `http://pauillac.inria.fr/ocaml/htmlman/`., 1996a.

Xavier Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5):667–698, 1996b.

Xavier Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5):667–698, September 1996c.

Xavier Leroy. The Objective Caml system: Documentation and user's manual, 2000. With Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. Available from `http://caml.inria.fr`.

Xavier Leroy and François Pessaux. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems*, 22(2):340–377, March 2000. Summary in *ACM Symposium on Principles of Programming Languages (POPL), San Antonio, Texas*, 1999.

Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, December 1996.

Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1997.

Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, January 1997. ISBN 0-201-63452-X. URL `http://www.aw.com/cp/javaseries.html`.

Barbara Liskov. A history of CLU. *ACM SIGPLAN Notices*, 28(3):133–147, 1993.

Ralph Loader. Finitary PCF is not decidable. *Theoretical Computer Science*, 266(1-2): 341–364, September 2001.

John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Principles of Programming Languages*, pages 47–57, New York, NY, USA, January 1988. ACM Press.

Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Number 11 in International Series of Monographs on Computer Science. Oxford University Press, 1994.

Zhaohui Luo and Robert Pollack. The LEGO proof development system: A user's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, May 1992.

David MacQueen. Modules for Standard ML. In *1984ACM Conference on LISP and Functional Programming*, pages 198–207, 1984.

David MacQueen. Using dependent types to express modular structure. In *ThirteenthACM Symposium on Principles of Programming Languages (POPL)*, 1986.

David B. MacQueen and Mads Tofte. A semantics for higher-order functors. In Donald T. Sannella, editor, *Programming Languages and Systems — ESOP '94*, volume 788 of *Lecture Notes in Computer Science*, pages 409–423. Springer-Verlag, 1994.

Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In *Types for Proofs and Programs*, volume 806, pages 213–237. Springer-Verlag LNCS 806, 1994.

Harry G. Mairson, Paris C. Kanellakis, and John C. Mitchell. Unification and ml type reconstruction. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 444–478. MIT Press, 1991.

Henning Makholm. Region-based memory management in Prolog. Master's thesis, Department of Computer Science, University of Copenhagen (DIKU), March 2000. URL `ftp://ftp.diku.dk/diku/semantics/papers/D-421.ps.gz`. DIKU Technical Report 00/09.

Henning Makholm and Kostis Sagonas. On enabling the WAM with region support. In Peter J. Stuckey, editor, *Logic Programming*, volume 2401 of *Lecture Notes in Computer Science*, pages 163–178, Heidelberg, Germany, 29 July–1 August 2002. Springer-Verlag. ISBN 3-540-43930-7.

Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

I. A. Mason, S. F. Smith, and C. L. Talcott. From operational semantics to domain theory. *Information and Computation*, 128(1):26–47, 1996.

I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1:287–327, 1991.

Bruce J. McAdam. On the Unification of Substitutions in Type Inference. In Kevin Hammond, Anthony J. T. Davie, and Chris Clack, editors, *Implementation of Functional Languages (IFL'98)*, volume 1595 of *Lecture Notes in Computer Science*, pages 139–154. Springer-Verlag, September 1998.

David McAllester. On the complexity analysis of static analyses. *Journal of the ACM*, 49(4):512–537, July 2002.

David McAllester. A logical algorithm for ML type inference. In *Rewriting Techniques and Applications (RTA)*, volume 2706 of *Lecture Notes in Computer Science*, pages 436–451. Springer-Verlag, June 2003.

Conor McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, LFCS, University of Edinburgh, 2000. URL `http://www.lfcs.informatics.ed.ac.uk/reports/00/ECS-LFCS-00-419/index.h%tml`.

Conor McBride and James McKinna. The view from the left. Submitted, 2002.

James McKinna and Robert Pollack. Pure Type Sytems formalized. In M. Bezem and J. F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 289–305. Springer-Verlag LNCS 664, March 1993.

David Melski and Thomas Reps. Interconvertibility of a class of set constraints and context-free language reachability. *Theoretical Computer Science*, 248(1–2), November 2000.

R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997a.

Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997b.

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997c.

Yasuhiko Minamide. A functional representation of data structures with a hole. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, pages 75–84, January 1998.

Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, pages 271–283, January 1996.

J. C. Mitchell. On the equivalence of data representations. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 305–330. Academic Press, 1991a.

J. C. Mitchell and G. D. Plotkin. Abstract types have existential types. *ACM Transactions on Programming Languages and Systems*, 10:470–502, 1988a.

John C. Mitchell. Coercion and type inference (summary). In *ACM Symposium on Principles of Programming Languages (POPL), Salt Lake City, Utah*, pages 175–185, January 1984.

John C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–286, July 1991b.

John C. Mitchell. *Foundations for Programming Languages.* MIT Press, Cambridge, Massachussetts, 1996a.

John C. Mitchell. *Foundations for Programming Languages.* MIT Press, 1996b.

John C. Mitchell and Gordon Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988b.

John C. Mitchell and Gordon D. Plotkin. Abstract types have existential types. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502, 1988c. Summary in *ACM Symposium on Principles of Programming Languages (POPL), New Orleans, Louisiana*, 1985.

Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991. Presented at LICS '89.

Shaw-Kwei Moh. The deduction theorems and two new logical systems. *Methodos*, 2: 56–75, 1950.

Christine Mohring. Algorithm development in the calculus of constructions. In *Proceedings, Symposium on Logic in Computer Science* IEE (1986), pages 84–91.

Stefan Monnier, Bratin Saha, and Zhong Shao. Principled scavenging. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah*, pages 81–91, June 2001.

Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. *Journal of Functional Programming*, 12(1):43–88, January 2002.

Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System-F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21 (3):527–568, May 1999.

Christian Mossin. *Flow Analysis of Typed Higher-Order Programs*. PhD thesis, DIKU, University of Copenhagen, Copenhagen, Denmark, 1997. URL `http://www.diku.dk/research/published/97-1.ps.gz`. Technical Report DIKU-TR-97/1.

Martin Müller. A constraint-based recast of ML-polymorphism. In *International Workshop on Unification*, June 1994. Technical Report 94-R-243, CRIN, Nancy, France. `http://www.ps.uni-sb.de/Papers/abstracts/UNIF94.ps`.

Martin Müller. Notes on HM(X). `http://www.ps.uni-sb.de/~mmueller/papers/HMX.ps.gz`, August 1998.

Martin Müller, Joachim Niehren, and Ralf Treinen. The first-order theory of ordering constraints over feature trees. *Discrete Mathematics and Theoretical Computer Science*, 4(2):193–234, 2001.

Martin Müller and Susumu Nishimura. Type inference for first-class messages with feature constraints. In Jieh Hsiang and Atsushi Ohori, editors, *Asian Computer Science Conference (ASIAN 98)*, volume 1538 of *LNCS*, pages 169–187, Manila, The Philippines, December 1998. Springer-Verlag.

Alan Mycroft. Polymorphic type schemes and recursive definitions. In M. Paul and B. Robinet, editors, *Proceedings of the International Symposium on Programming*, volume 167 of *LNCS*, pages 217–228, Toulouse, France, April 1984. Springer.

George C. Necula. Proof-carrying code. In *ACM Symposium on Principles of Programming Languages (POPL), Paris, France*, pages 106–119, 15–17 January 1997.

George C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, September 1998. Also available as CMU-CS-98-154.

George C. Necula. Translation validation for an optimizing compiler. In *ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PDLI)*, pages 83–94, Vancouver, BC, Canada, 18–21 June 2000. ACM SIGPLAN.

George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996, Seattle, WA*, pages 229–243, Berkeley, CA, USA, October 1996. USENIX press.

George C. Necula and Peter Lee. Efficient representation and validation of logical proofs. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 93–104. IEEE Computer Society Press, 1998.

Joachim Niehren, Martin Müller, and Andreas Podelski. Inclusion constraints over non-empty sets of trees. In Max Dauchet, editor, *International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, volume 1214 of *Lecture Notes in Computer Science*, pages 217–231. Springer-Verlag, April 1997.

Joachim Niehren and Tim Priesnitz. Non-structural subtype entailment in automata theory. *Information and Computation*, 2003. To appear.

Flemming Nielson and Hanne Riis Nielson. From CML to its process algebra. *Theoretical Computer Science*, 155:179–219, 1996.

Flemming Nielson, Hanne Riis Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer, 1999.

Flemming Nielson, Hanne Riis Nielson, and Helmut Seidl. A succinct solver for ALFP. *Nordic Journal of Computing*, 9(4):335–372, 2002. `http://www.informatik.uni-trier.de/~seidl/papers/succinct.pdf`.

Hanne Riis Nielson and Flemming Nielson. Higher-order concurrent programs with finite communication topology. In *Principles of Programming Languages*, pages 84–97, New York, NY, USA, 1994. ACM Press. ISBN 0-89791-636-0.

Susumu Nishimura. Static typing for dynamic messages. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pages 266–278, New York, 1998. ACM Press.

Henning Niss. *Regions are Imperative: Unscoped Regions and Control-Flow Sensitive Memory Management*. PhD thesis, Department of Computer Science, University of Copenhagen (DIKU), 2002.

E.G.J.M.H. N ocker and J.E.W. Smetsers. Partially strict non-recursive data types. *Journal of Functional Programming*, 3(2):191–215, 1993.

E.G.J.M.H. N ocker, J.E.W. Smetsers, M.C.J.D. van Eekelen, and M.J. Plasmeijer. Concurrent CLEAN. In Leeuwen and Rem, editors, *Parallel Architectures and Languages Europe*, number 505 in LNCS, pages 202–219. Springer-Verlag, 1991.

Martin Odersky. Observers for linear types. In B. Krieg-Brückner, editor, *ESOP '92: 4th European Symposium on Programming, Rennes, France, Proceedings*, pages 390–407. Springer-Verlag, February 1992. Lecture Notes in Computer Science 582.

Martin Odersky, Vincent Cremet, Christine Rockl, and Matthias Zenger. A nominal theory of objects with dependent types. In *Workshop on Foundations of Object-Oriented Languages (FOOL), informal proceedings*, 2003.

Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, pages 54–67, St. Petersburg, Florida, January 21–24, 1996. ACM Press.

Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999. Summary in *Workshop on Foundations of Object-Oriented Languages (FOOL), informal proceedings*, 1997.

Peter O'Hearn and David Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, November 1995.

Atsushi Ohori and Peter Buneman. Type inference in a database programming language. In *1988 ACM Conference on Lisp and Functional Programming*, pages 174–183, Snowbird, Utah, July 1988. Revised manuscript, September, 1988.

Atsushi Ohori and Peter Buneman. Static type inference for parametric classes. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 445–456, October 1989. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, MIT Press, 1994.

I. E. Orlov. The calculus of compatibility of propositions (in russian). *Matematicheskii Sbornik*, 35:263–286, 1928.

Jens Palsberg. Efficient inference of object types. *Information and Computation*, 123(2): 198–209, 1995.

Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. In *Principles of Programming Languages*, pages 367–378, New York, NY, USA, 1995. ACM Press. ISBN 0-89791-692-1.

Jens Palsberg and Michael Schwartzbach. Type substitution for object-oriented programming. In N. Meyrowitz, editor, *Proc. Conf. Object-Oriented Programming: Systems, Languages, and Applications and European Conf. on Object-Oriented Programming*, pages 151–160, Ottawa, Canada, October 1990. ACM Press.

Jens Palsberg and Michael Schwartzbach. *Object-oriented Type Systems*. John Wiley & Sons, 1994.

Jens Palsberg, Mitchell Wand, and Patrick M. O'Keefe. Type inference with nonstructural subtyping. *Formal Aspects of Computing*, 9:49–67, 1997.

Christine Paulin-Mohring. Extracting F_ω's programs from proofs in the calculus of constructions. In *ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*, pages 89–104, January 1989.

Leaf Petersen, Perry Cheng, Robert Harper, and Chris Stone. Implementing the TILT internal language. Technical Report CMU–CS–00–180, Carnegie Mellon University, Department of Computer Science, 2000.

Leaf Petersen, Robert Harper, Karl Crary, and Frank Pfenning. A type theory for memory allocation and data layout. In *POPL 2003: The 30th ACM SIPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 172–184, New Orleans, LA, January 2003a.

Leaf Petersen, Robert Harper, Karl Crary, and Frank Pfenning. A type theory for memory allocation and data layout. In *ACM Symposium on Principles of Programming Languages (POPL), New Orleans, Louisiana*, pages 172–184, January 2003b.

Simon Peyton Jones. Special issue: Haskell 98 language and libraries. *Journal of Functional Programming*, 13, January 2003.

Simon Peyton Jones and Mark Shields. Lexically-scoped type variables. Submitted to ICFP'03., March 2003.

Frank Pfenning and Carsten Schürmann. Algorithms for equality and unification in the presence of notational definitions. In T. Altenkirch, W. Naraschewski, and B. Reus, editors, *Types for Proofs and Programs*, number LNCS 1657. Springer-Verlag, 1998.

Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Logic in Computer Science*, 1993. Full version in *Mathematical Structures in Computer Science* , Vol. 6, No. 5, 1996.

Benjamin C. Pierce and David N. Turner. Object-oriented programming without recursive types. In *ACM Symposium on Principles of Programming Languages (POPL), Charleston, South Carolina*, January 1993.

A. M. Pitts. Relational properties of domains. *Information and Computation*, 127:66–90, 1996.

A. M. Pitts. Existential types: Logical relations and operational equivalence. In K. G. Larsen, S. Skyum, and G. Winskel, editors, *Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 1998, Proceedings*, volume 1443 of *Lecture Notes in Computer Science*, pages 309–326. Springer-Verlag, Berlin, 1998.

A. M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10:321–359, 2000.

A. M. Pitts. Operational semantics and program equivalence. In G. Barthe, P. Dybjer, and J. Saraiva, editors, *Applied Semantics, Advanced Lectures*, volume 2395 of *Lecture Notes in Computer Science, Tutorial*, pages 378–412. Springer-Verlag, 2002a. International Summer School, APPSEM 2000, Caminha, Portugal, September 9–15, 2000.

A. M. Pitts and I. D. B. Stark. Operational reasoning for functions with local state. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute, pages 227–273. Cambridge University Press, 1998.

Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 2002b. To appear.

Andrew M. Pitts and Ian D. B. Stark. Observable properties of higher order functions that dynamically create local names, or: What's new? In *Mathematical Foundations of Computer Science, Proc. 18th Int. Symp., Gdańsk, 1993*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141. Springer-Verlag, Berlin, 1993.

G. D. Plotkin. Lambda-definability and logical relations. Memorandum SAI-RM-4, School of Artificial Intelligence, University of Edinburgh, 1973.

G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

G. D. Plotkin and M. Abadi. A logic for parametric polymorphism. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculus and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 361–375. Springer-Verlag, Berlin, 1993.

Gordon D. Plotkin. Lambda-definability in the full type hierarchy. In Jonathan P. Seldin and J. Roger Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 363–373. Academic Press, London, 1980.

Jeff Polakow and Frank Pfenning. Natural deduction for intuitionistic non-commutative linear logic. In *International Conference on Typed Lambda Calculi*, number 1581 in LNCS, pages 295–309. Springer-Verlag, April 1999.

Jeff Polakow and Frank Pfenning. Properties of terms in continuation-passing style in an ordered logical framework. In *Workshop on Logical Frameworks and Meta-Languages*, Santa Barbara, June 2000.

Erik Poll. Expansion Postponement for Normalising Pure Type Systems. *Journal of Functional Programming*, 8(1):89–96, 1998.

Robert Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1994.

POPL. *Principles of Programming Languages*, New York, NY, USA, January 1991a. ACM Press.

POPL. *Principles of Programming Languages*, New York, NY, USA, 17–19 January 2001b. ACM Press. ISBN 1-58113-336-7.

François Pottier. A versatile constraint-based type inference system. *Nordic Journal of Computing*, 7(4):312–347, November 2000.

François Pottier. A semi-syntactic soundness proof for HM(X). Research Report 4150, INRIA, March 2001a.

François Pottier. Simplifying subtyping constraints: a theory. *Information and Computation*, 170(2):153–183, November 2001b.

François Pottier. A constraint-based presentation and generalization of rows. In *Eighteenth Annual IEEE Symposium on Logic In Computer Science (LICS'03)*, Ottawa, Canada, June 2003. URL `http://pauillac.inria.fr/~fpottier/publis/fpottier-lics03.ps.gz`.

François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, January 2003.

François Pottier, Christian Skalka, and Scott Smith. A systematic approach to static access control. In *European Symp. on Programming (ESOP)*, volume 2028 of *Lecture Notes in Computer Science*, pages 30–45. Springer-Verlag, April 2001.

Vaughan Pratt and Jerzy Tiuryn. Satisfiability of inequalities in a poset. *Fundamenta Informaticæ*, 28(1–2):165–182, 1996.

Sriram K. Rajamani and Jakob Rehof. A behavioral module system for the pi-calculus. In *SAS 01: Static Analysis*, LNCS 2126, pages 375–394. Springer-Verlag, 2001.

Sriram K. Rajamani and Jakob Rehof. Conformance checking for models of asynchronous message passing software. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 166–179, Heidelberg, Germany, July 27–31 2002. Springer-Verlag. ISBN 3-540-43997-8.

Jakob Rehof. Minimal typings in atomic subtyping. In *ACM Symposium on Principles of Programming Languages (POPL), Paris, France*, pages 278–291, January 1997.

Jakob Rehof and Manuel Fähndrich. Type-based flow analysis: From polymorphic subtyping to CFL reachability. In POPL POPL (b), pages 54–66. ISBN 1-58113-336-7.

Jakob Rehof and Manuel Fähndrich. Type-based flow analysis: From polymorphic subtyping to CFL-reachability. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 54–66, January 2001.

Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit: Component composition for systems software. In *Proc. of the 4th Operating Systems Design and Implementation (OSDI)*, pages 347–360, October 2000. URL `citeseer.nj.nec.com/reid00knit.html`.

Didier Rémy. Extending ML type system with a sorted equational theory. Research Report 1766, Institut National de Recherche en Informatique et Automatisme, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France, 1992a.

Didier Rémy. Projective ML. In *ACM Symposium on Lisp and Functional Programming (LFP)*, pages 66–75, 1992b.

Didier Rémy. Syntactic theories and the algebra of record terms. Research Report 1869, Institut National de Recherche en Informatique et Automatisme, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France, 1993a.

Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993b.

Didier Rémy. Programming objects with ML-ART: An extension to ML with abstract and record types. In Masami Hagiya and John C. Mitchell, editors, *International Symposium on Theoretical Aspects of Computer Software (TACS)*, pages 321–346, Sendai, Japan, April 1994. Springer-Verlag.

Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Object Systems*, 4(1):27–50, 1998. Summary in *ACM Symposium on Principles of Programming Languages (POPL), Paris, France*, 1997.

Greg Restall. *An introduction to substructural logics*. Routledge, January 2000.

Greg Restall. Handbook of the history and philosophy of logic. To appear., 2001.

J. C. Reynolds. Towards a theory of type structure. In *Paris Colloquium on Programming*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, Berlin, 1974a.

J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523. North-Holland, Amsterdam, 1983a.

John C. Reynolds. Automatic computation of data set definitions. In A. J. H. Morrell, editor, *Information Processing 68*, volume 1, pages 456–461. North Holland, 1969a.

John C. Reynolds. Automatic computation of data set definitions. In A. J. H. Morrell, editor, *Information Processing 68*, volume 1, pages 456–461, Edinburgh, Scotland, 1969b. North Holland.

John C. Reynolds. Towards a theory of type structure. In *Colloq. sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer-Verlag, 1974b.

John C. Reynolds. Syntactic control of interference. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 39–46, Tucson, 1978.

John C. Reynolds. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523, Amsterdam, 1983b. Elsevier Science Publishers B. V. (North-Holland).

John C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, June 1988. Reprinted in O'Hearn and Tennent, *ALGOL-like Languages*, vol. 1, pages 173–233, Birkhäuser, 1997.

John C. Reynolds. Syntactic control of interference, part 2. In *International Colloquium on Automata, Languages and Programming*, July 1989.

J. Alan Robinson. Computational logic: The unification computation. *Machine Intelligence*, 6:63–72, 1971.

Douglas T. Ross. The AED free storage package. *Communications of the ACM*, 10(8): 481–492, 1967.

Claudio V. Russo. *Types for Modules*. PhD thesis, Edinburgh University, Edinburgh, Scotland, 1998. LFCS Thesis ECS–LFCS–98–389.

Claudio V. Russo. Non-dependent types for standard ML modules. In *Principles and Practice of Declarative Programming*, pages 80–97, 1999. URL `citeseer.nj.nec.com/russo99nondependent.html`.

Claudio V. Russo. Recursive structures for standard ml. In *Proc. Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01)*, pages 50–61, Florence, Italy, September 2001.

Didier Rémy. Type checking records and variants in a natural extension of ML. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 77–88, 1989.

Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

Jacob T. Schwartz. Optimization of very high level languages (parts I and II). *Computer Languages*, 1(2 & 3):161–194, 197–218, 1975.

R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. Adaptive pattern matching. *SIAM Journal on Computing*, 24(6):1207–1234, December 1995.

Jonathan Seldin. Curry's anticipation of the types used in programming languages. Presented at Ann. Meet. Can. Soc. Hist. and Phil. Math, Toronto. Available from Seldin's homepage, 2002.

Miley Semmelroth and Amr Sabry. Monadic encapsulation in ML. In *International Conference on Functional Programming*, pages 8–17, 1999. URL `citeseer.nj.nec.com/semmelroth99monadic.html`.

Peter Sestoft. Replacing function parameters by global variables. Technical Report 88-7-2, DIKU, University of Copenhagen, October 1988. SE88.

Peter Sestoft. Replacing function parameters by global variables. In *Proc. Functional Programming Languages and Computer Architecture (FPCA), London, England*, pages 39–53. ACM Press, September 1989.

Peter Sestoft. Moscow ml, 2003. URL `http://www.dina.dk/~sestoft/mosml.html`.

Paula Severi and Erik Poll. Pure type systems with definitions. In *Proceedings of Logical Foundations of Computer Science (LFCS)*, pages 316–328. Springer-Verlag, 1994. LNCS volume 813.

Zhong Shao. An overview of the FLINT/ML compiler. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation (TIC'97)*, Amsterdam, The Netherlands, June 1997.

Zhong Shao. Typed cross-module compilation. In *Proc. 1998 ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 141–152, Baltimore, Maryland, September 1998.

Zhong Shao. Transparent modules with fully syntactic signatures. In *International Conference on Functional Programming*, pages 220–232, Paris, France, September 1999.

Zhong Shao, Christopher League, and Stefan Monnier. Implementing typed intermediate languages. In *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming*, pages 313–323, Baltimore, MD, September 1998. ACM SIGPLAN.

Mark Shields and Erik Meijer. Type-indexed rows. In *ACM Symposium on Principles of Programming Languages (POPL), London, England*, pages 261–275, January 2001.

Mark B. Shields and Simon Peyton Jones. First class modules for Haskell. In *Workshop on Foundations of Object-Oriented Languages (FOOL), informal proceedings*, pages 28–40, January 2002.

Olin Shivers. Control flow analysis in Scheme. In *Programming Language Design and Implementation*, volume 23(7), pages 164–174, 22–24 June 1988.

Olin Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, Carnegie Mellon University, May 1991.

Vincent Simonet. Type inference with structural subtyping: a faithful formalization of an efficient constraint solver. In *Asian Symposium on Programming Languages and Systems*, November 2003.

Christian Skalka and François Pottier. Syntactic type soundness for HM(X). In *Workshop on Types in Programming (TIP'02)*, volume 75 of *Electronic Notes in Theoretical Computer Science*, July 2002.

Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *European Symp. on Programming (ESOP)*, pages 366–381, Berlin, March 2000a.

Frederick Smith, David Walker, and Greg Morrisett. Alias types. In Gert Smolka, editor, *Ninth European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–381. Springer-Verlag, April 2000b.

Geoffrey S. Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23(2–3):197–226, December 1994.

Jan Smith, Bengt Nordström, and Kent Petersson. *Programming in Martin-Löf's Type Theory. An Introduction*. Oxford University Press, 1990.

R. Statman. Logical relations and the typed lambda calculus. *Information and Control*, 65:85–97, 1985a.

Richard Statman. Logical relations and the typed λ-calculus. *Information and Control*, 65(2–3):85–97, May–June 1985b.

Christopher A. Stone. *Singleton Kinds and Singleton Types*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, August 2000.

Christopher A. Stone and Robert Harper. Deciding type equivalence in a language with singleton kinds. In *ACM Symposium on Principles of Programming Languages (POPL), Boston, Massachusetts*, Boston, January 2000a.

Christopher A. Stone and Robert Harper. Deciding type equivalence in a language with singleton kinds. In *Twenty SeventhACM Symposium on Principles of Programming Languages (POPL)*, pages 214–227, Boston, January 2000b.

Thomas Streicher. *Semantics of Type Theory*. Springer-Verlag, 1991.

Zhendong Su and Alexander Aiken. Entailment with conditional equality constraints. In *European Symp. on Programming (ESOP)*, volume 2028 of *Lecture Notes in Computer Science*, pages 170–189, April 2001.

Zhendong Su, Alexander Aiken, Joachim Niehren, Tim Priesnitz, and Ralf Treinen. The first-order theory of subtyping constraints. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 203–216, January 2002.

Martin Sulzmann. *A general framework for Hindley/Milner type systems with constraints*. PhD thesis, Yale University, Department of Computer Science, May 2000.

Martin Sulzmann, Martin Müller, and Christoph Zenger. Hindley/Milner style type systems in constraint form. Research Report ACRC–99–009, University of South Australia, School of Computer and Information Science, July 1999.

William W. Tait. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic*, 32(2):198–212, June 1967.

C. Talcott. Reasoning about functions with effects. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute, pages 347–390. Cambridge University Press, 1998.

Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming (JFP)*, 2(2), 1992.

Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 111:245–296, 1994.

David Tarditi, Greg Morrisett, Perry Cheng, Christopher Stone, Robert Harper, and Peter Lee. TIL : A type-directed optimizing compiler for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Philadephia, Pennsylvania*, pages 181–192, May 21–24 1996.

Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, April 1975.

Robert Endre Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26(4):690–715, October 1979.

J. Terlouw. Een nadere bewijstheoretische analyse van GSTTs. Manuscript, University of Nijmegen, Netherlands, 1989.

Kresten Krab Thorup. Genericity in Java with virtual types. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 444–471, Jyväskylä, Finland, June 1997. Springer-Verlag.

Luc Maranget Thérèse Hardin and Bruno Pagano. Functional runtimes within the lambda-sigma calculus. *Journal of Functional Programming*, 8(2), March 1998.

Jerzy Tiuryn. Subtype inequalities. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 308–317, June 1992.

Jerzy Tiuryn and Mitchell Wand. Type reconstruction with recursive types and atomic subtyping. In *Proceedings of TAPSOFT '93*, volume 668 of *Lecture Notes in Computer Science*, pages 686–701. Springer-Verlag, April 1993.

Mads Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, University of Edinburgh, 1988.

Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):724–767, 1998. URL `http://www.itu.dk/research/mlkit/kit_general/toplas98.ps.gz`.

Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. Region-based memory management in perspective. To appear, 2003.

Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, and Peter Sestoft. Programming with regions in the ML Kit (for version 4). Technical report, IT University of Copenhagen, October 2001.

Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, Peter Sestoft, and Peter Bertelsen. Programming with regions in the ML Kit (for version 3). Technical Report DIKU-TR-98/25, Department of Computer Science, University of Copenhagen (DIKU), 1998. URL `http://www.it-c.dk/research/mlkit/kit3/manual.ps.gz`.

Mads Tofte and Jean-Pierre Talpin. Implementing the call-by-value lambda-calculus using a stack of regions. In *ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, January 1994.

Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1 February 1997.

Mads Torgersen. Virtual types are statically safe. In *Proceedings of the 5th Workshop on Foundations of Object-Oriented Languages (FOOL)*, San Diego, CA, January 1998.

Valery Trifonov and Scott Smith. Subtyping constrained types. In *Static Analysis Symposium (SAS)*, volume 1145 of *Lecture Notes in Computer Science*, pages 349–365. Springer-Verlag, September 1996.

David N. Turner. *The Polymorphic Pi-calulus: Theory and Implementation*. PhD thesis, University of Edinburgh, 1995.

David N. Turner and Philip Wadler. Operational interpretations of linear logic. *Theoretical Computer Science*, 227:231–248, 1999. Special issue on linear logic.

David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *ACM International Conference on Functional Programming and Computer Architecture*, San Diego, CA, June 1995.

Diederik T. van Daalen. *The Language Theory of Automath*. PhD thesis, Technische Hogeschool Eindhoven, 1980.

Robbert van Renesse, Kenneth P. Birman, Mark Hayden, Alexey Vaysburd, and David Karr. Building adaptive systems using ensemble. *Software: Practice and Experience*, 28(9):963–979, August 1998.

Per Velschow and Morten Voetmann Christensen. Region-based memory management in Java. Master's thesis, Department of Computer Science, University of Copenhagen (DIKU), 1998. URL `http://www.worldonline.dk/~voet/thesis.ps.gz`.

Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Progarmming Concepts and Methods*, Sea of Galilee, Israel, April 1990. North Holland. IFIP TC 2 Working Conference.

Philip Wadler. The marriage of effects and monads. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1), pages 63–74, 1999. URL `citeseer.nj.nec.com/article/wadler98marriage.html`. Journal version submitted to ACM Transactions on Computational Logic (2003).

R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *14th ACM Symposium on Operating Systems Principles*, pages 203–216. ACM, December 1993.

David Walker, Karl Crary, and Greg Morrisett. Typed memory management in a calculus of capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4):701–771, July 2000a.

David Walker, Karl Crary, and Greg Morrisett. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4):701–771, July 2000b. URL `http://www.cs.princeton.edu/~dpw/capabilities-toplas.pdf`.

David Walker and Greg Morrisett. Alias types for recursive data structures. *Lecture Notes in Computer Science*, 2071:177+, 2001.

David Walker and Kevin Watkins. On linear types and regions. In *International Conference on Functional Programming*, Florence, September 2001a. ACM Press.

David Walker and Kevin Watkins. On regions and linear types. In *6th International Conference on Functional Programming*, pages 181–192, New York, NY, USA, 3–5 September 2001b. ACM Press. ISBN 1-58113-415-0. URL `http://www.cs.princeton.edu/~dpw/papers/lr.pdf`.

Mitchell Wand. Finding the source of type errors. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 38–43, January 1986.

Mitchell Wand. Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, Ithaca, NY, June 1987.

Mitchell Wand. Corrigendum: Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, 1988.

Mitchell Wand. Type inference for record concatenation and multiple inheritance. In *Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 92–97, Pacific Grove, CA, June 1989.

Mitchell Wand. Type inference for objects with instance variables and inheritance. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 97–120. MIT Press, 1994.

Daniel C. Wang and Andrew W. Appel. Type-preserving garbage collectors. In *ACM Symposium on Principles of Programming Languages (POPL), London, England*, pages 166–178, January 2001.

Keith Wansbrough and Simon Peyton Jones. Once upon a polymorphic type. In *Twenty-sixth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 15–28, San Antonio, January 1999.

J. B. Wells. Typability and type checking in system F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1–3):111–156, 1999.

Benjamin Werner. *Une Th'eorie des Constructiones Inductives*. PhD thesis, L'Universite Paris, 1994.

Niklaus Wirth. *Systematic Programming: An Introduction*. Prentice Hall, 1973.

Niklaus Wirth. *Programming in Modula-2*. Texts and Monographs in Computer Science. Springer-Verlag, 1983.

A. K. Wright and R. Cartwright. A practical soft type system for scheme. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*. ACM, June 1994. Also available as LISP Pointers VII(3) July-September 1994.

A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994a.

Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–356, December 1995.

Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994b.

Hongwei Xi and Robert Harper. A dependently typed assembly language. In *International Conference on Functional Programming (ICFP), Firenze, Italy*, 2001.

Jan Zwanenburg. Pure type systems with subtyping. In J.-Y. Girard, editor, *Typed Lambda Calculus and Applications (TLCA)*, pages 381–396. Springer-Verlag, 1999. Lecture Notes in Computer Science, volume 1581.

Index