1 *The Essence of ML Type Inference*

By François Pottier and Didier Rémy

1.1 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) families 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 (Peyton Jones, 2003a) and Clean (Brus, van Eekelen, van Leer, and Plasmeijer, 1987), whose semantics is rather different—indeed, it is nonstrict and pure (Sabry, 1998)—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.

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

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 manipulate 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, 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 *Damas and Milner's type system*.

1.1.1 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,

$\mathtt{x},\mathtt{y} ::=$		*Identifiers:*
	$\mathtt{z}$	*Variable*
	m	*Memory location*
	$\mathtt{c}$	*Constant*
$\mathtt{t} ::=$		*Expressions:*
	$\mathtt{x}$	*Identifier*
	$\lambda\mathtt{z.t}$	*Function*
	$\mathtt{t}\ \mathtt{t}$	*Application*
	$\mathtt{let}\ \mathtt{z} = \mathtt{t}\ \mathtt{in}\ \mathtt{t}$	*Local definition*
$\mathtt{v},\mathtt{w} ::=$		*Values:*
	$\mathtt{z}$	*Variable*
	m	*Memory location*
	$\lambda\mathtt{z.t}$	*Function*
	$\mathtt{c}\ \mathtt{v}_1 \ldots \mathtt{v}_k$	*Data*
		$\mathtt{c} \in \mathcal{Q}^+ \wedge k \leq a(\mathtt{c})$
	$\mathtt{c}\ \mathtt{v}_1 \ldots \mathtt{v}_k$	*Partial application*
		$\mathtt{c} \in \mathcal{Q}^- \wedge k < a(\mathtt{c})$
$\mathcal{E} ::=$		*Evaluation Contexts:*
	$[]$	*Empty context*
	$\mathcal{E}\ \mathtt{t}$	*Left side of an application*
	$\mathtt{v}\ \mathcal{E}$	*Right side of an application*
	$\mathtt{let}\ \mathtt{z} = \mathcal{E}\ \mathtt{in}\ \mathtt{t}$	*Local definition*

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

$\lambda\_.\mathtt{t}$ stands for $\lambda\mathtt{z.t}$, provided $\mathtt{z}$ is fresh for $\mathtt{t}$. *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 $\mathtt{c}$ has a nonnegative integer *arity* $a(\mathtt{c})$. 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.

1.1.1 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 $\mathtt{t}_1 \hat{+} \mathtt{t}_2$ stands for the double application $\hat{+}\ \mathtt{t}_1\ \mathtt{t}_2$ of the destructor $\hat{+}$ to the expressions $\mathtt{t}_1$ and $\mathtt{t}_2$. □

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* $\lambda\mathtt{z.t}$ represents the function of one parameter named $\mathtt{z}$ whose result is the expression $\mathtt{t}$, or, in other words, the function that maps $\mathtt{z}$ to $\mathtt{t}$. Note that the variable $\mathtt{z}$ is bound within the term $\mathtt{t}$, so (for instance) $\lambda\mathtt{z}_1.\mathtt{z}_1$ and $\lambda\mathtt{z}_2.\mathtt{z}_2$ are the same object. The *application* $\mathtt{t}_1\ \mathtt{t}_2$ represents the result of calling the function $\mathtt{t}_1$ with actual parameter $\mathtt{t}_2$, or, in other words, the result of applying $\mathtt{t}_1$ to $\mathtt{t}_2$. Application is left-associative, that is, $\mathtt{t}_1\ \mathtt{t}_2\ \mathtt{t}_3$ stands for $(\mathtt{t}_1\ \mathtt{t}_2)\ \mathtt{t}_3$. The construct $\mathtt{let}\ \mathtt{z} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2$ represents the result of evaluating $\mathtt{t}_2$ after binding the variable $\mathtt{z}$ to $\mathtt{t}_1$. Note that the variable $\mathtt{z}$ is bound within $\mathtt{t}_2$, but not within $\mathtt{t}_1$, so for instance $\mathtt{let}\ \mathtt{z}_1 = \mathtt{z}_1\ \mathtt{in}\ \mathtt{z}_1$ and $\mathtt{let}\ \mathtt{z}_2 = \mathtt{z}_1\ \mathtt{in}\ \mathtt{z}_2$ are the same object. The construct $\mathtt{let}\ \mathtt{z} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2$ has the same meaning as $(\lambda\mathtt{z}.\mathtt{t}_2)\ \mathtt{t}_1$, 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 $\mathtt{let}$ 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 $\mathtt{c}\ \mathtt{v}_1\ \ldots\ \mathtt{v}_k$, where k does not exceed $\mathtt{c}$'s arity if $\mathtt{c}$ is a constructor, and k is smaller than $\mathtt{c}$'s arity if $\mathtt{c}$ 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 $\mathtt{v}$ and $\mathtt{w}$ for values.

1.1.2 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 $\hat{+}\ \mathtt{v}$. 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. □

1.1.3 EXAMPLE [PAIRS]: Let $(\cdot,\cdot)$ be a binary constructor. If $\mathtt{t}_1$ are $\mathtt{t}_2$ are expressions, then the double application $(\cdot,\cdot)\ \mathtt{t}_1\ \mathtt{t}_2$ may be called the *pair* of $\mathtt{t}_1$ and $\mathtt{t}_2$, and may be written $(\mathtt{t}_1, \mathtt{t}_2)$. By the definition above, $(\mathtt{t}_1, \mathtt{t}_2)$ is a value if and only if $\mathtt{t}_1$ and $\mathtt{t}_2$ 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 values, 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 $\mu[m \mapsto \mathtt{v}]$ for the store that maps m to $\mathtt{v}$ 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 $\mathtt{t}/\mu$ is a pair of an expression $\mathtt{t}$ and a store μ. The memory locations in the domain of μ are considered bound within $\mathtt{t}/\mu$, 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 $\mathtt{t}/\mu$ such that $\mathtt{t}$ has no free program variables and every memory location that appears within $\mathtt{t}$ or within the range of μ is in the domain of μ. If $\mathtt{t}$ is a closed source program, its evaluation begins within an empty store—that is, with the configuration $\mathtt{t}/\varnothing$. 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. 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 $\lambda\mathtt{z}.[]$ 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 $\mathtt{t}\ \mathcal{E}$ is an evaluation context only if $\mathtt{t}$ is a value means that, to evaluate an application $\mathtt{t}_1\ \mathtt{t}_2$, one should fully evaluate $\mathtt{t}_1$ before attempting to evaluate $\mathtt{t}_2$. 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 $\mathcal{E} ::= \ldots \mid \mathtt{t}\ \mathcal{E} \mid \mathcal{E}\ \mathtt{v} \mid \ldots$ would enforce a right-to-left evaluation order, while defining $\mathcal{E} ::= \ldots \mid \mathtt{t}\ \mathcal{E} \mid \mathcal{E}\ \mathtt{t} \mid \ldots$ would leave the evaluation order unspecified, effectively allowing reduction to alternate between both subexpressions, and making evaluation nondeterministic. The fact that $\mathtt{let\ z} = \mathtt{v\ in}\ \mathcal{E}$ 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 $\mathcal{E}[\mathtt{t}]$ for the expression obtained by replacing the hole in $\mathcal{E}$ with the expression $\mathtt{t}$.

Figure 1-2 defines first a relation $\longrightarrow$ between configurations, then a relation $\longrightarrow\!\!\!\!\!\!\rightarrow$ between *closed* configurations. If $\mathtt{t}/\mu \longrightarrow \mathtt{t}'/\mu'$ or $\mathtt{t}/\mu \longrightarrow\!\!\!\!\!\!\rightarrow \mathtt{t}'/\mu'$ holds, then we say that the configuration $\mathtt{t}/\mu$ *reduces* to the configuration

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

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

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

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

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

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

$\mathtt{t}'/\mu'$; the ambiguity involved in this definition is benign. If $\mathtt{t}/\mu \longrightarrow \mathtt{t}'/\mu$ holds for every store μ, then we write $\mathtt{t} \longrightarrow \mathtt{t}'$ 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 $(\lambda \mathtt{z.t})\ \mathtt{v}$ reduces to the function body, namely $\mathtt{t}$, where every occurrence of the formal argument $\mathtt{z}$ has been replaced with the actual argument $\mathtt{v}$. The λ construct, which prevented the function body $\mathtt{t}$ 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; Hardin, Maranget, and Pagano, 1998). 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, 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 $\mathtt{c}\ \mathtt{v}_1\ \ldots\ \mathtt{v}_n$, 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 $\mathit{dom}(\mu' \setminus \mu)$. The notation $A \# B$ stands for $A \cap B = \varnothing$.

Rule R-CONTEXT completes the definition of the operational semantics by defining $\longrightarrow\!\!\!\!\!\!\triangleright$, 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 $\longrightarrow\!\!\!\!\!\!\triangleright$ relates closed configurations only, we do not need to require that the memory locations in $\mathit{dom}(\mu' \setminus \mu)$ be fresh for $\mathcal{E}$: indeed, every memory location that appears within $\mathcal{E}$ must be a member of $\mathit{dom}(\mu)$.

1.1.4 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 $\hat{+}\ \hat{k}_1\ \hat{k}_2$, 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. □

1.1.5 EXAMPLE [PAIRS, CONTINUED]: In addition to the pair constructor defined in Example 1.1.3, 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. □

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

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

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

Let us use the syntactic sugar `if` t_0 `then` t_1 `else` t_2 for the triple application of `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 `if` t_0 `then` t_1 `else` t_2 that corrects the problem. □

1.1.7 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).
□

1.1.8 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. □

1.1.9 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:

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

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

$$m := \mathsf{v}/(m \mapsto \mathsf{v}_0) \xrightarrow{\delta} \mathsf{v}/(m \mapsto \mathsf{v}) \qquad \text{(R-ASSIGN)}$$

According to R-REF, evaluating $\texttt{ref}\ \mathsf{v}$ 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 $m := \mathsf{v}$ 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 $m := \mathsf{v}$ is v itself; in ML-the-programming-language, it is usually a nullary constructor $()$, pronounced *unit*. □

1.1.10 EXAMPLE [RECURSION]: Let $\texttt{fix}$ be a binary destructor, whose operational semantics is:

$$\texttt{fix}\ \mathsf{v}_1\ \mathsf{v}_2 \xrightarrow{\delta} \mathsf{v}_1\ (\texttt{fix}\ \mathsf{v}_1)\ \mathsf{v}_2 \qquad \text{(R-FIX)}$$

$\texttt{fix}$ is a fixpoint combinator: it effectively allows recursive definitions of functions. Indeed, the construct $\texttt{letrec}\ \mathsf{f} = \lambda \mathsf{z}.\mathsf{t}_1\ \texttt{in}\ \mathsf{t}_2$ provided by ML-the-programming-language may be viewed as syntactic sugar for $\texttt{let}\ \mathsf{f} = \texttt{fix}\ (\lambda \mathsf{f}.\lambda \mathsf{z}.\mathsf{t}_1)\ \texttt{in}\ \mathsf{t}_2$. □

1.1.11 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. The proof is left as an exercise to the reader. The converse, however, does not hold: if t/μ is irreducible with respect to $\longrightarrow$, 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 $\mathsf{t}/\varnothing$ reduces to a stuck configuration.

1.1.12 EXAMPLE: Runtime errors typically arise when destructors are applied to arguments of an unexpected nature. For instance, the expressions $\hat{+}\ \hat{1}\ m$ and

$\pi_1\ \hat{2}$ and $!\hat{3}$ are stuck, regardless of the current store. The program `let z =` $\hat{+}\ \hat{+}$ `in z 1` is not stuck, because $\hat{+}\ \hat{+}$ is a value. However, its reduct through R-LET is $\hat{+}\ \hat{+}\ 1$, which is stuck, so this program goes wrong. The primary purpose of type systems is to prevent such situations from arising. □

1.1.13 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, 2002). □

1.1.2 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.

We must first define *types*. In DM, types are terms built out of *type constructors* and *type variables*. Furthermore, they are *first-order* terms: that is, in the grammar of types, none of the productions *binds* a type variable. This situation is identical to that of the simply-typed λ-calculus.

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 $\mathsf{T} \rightarrow \mathsf{T}'$ out of arbitrary types T and T'; that is, we need a binary type constructor $\rightarrow$. 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 1.1.1, then a nullary type constructor int is needed; if they include pair construction and destruction, as in Examples 1.1.3 and 1.1.5, 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 $\mathsf{T} \rightarrow \mathsf{T}'$, it is understood that the type constructor $\rightarrow$ has arity 2, that T is its *first* parameter, known as its *domain*, and that 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 1.2.9) and in the definition of rows (§1.8).

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 §1.8. 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.

1.1.14 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 $\{domain \mapsto \star, codomain \mapsto \star\} \Rightarrow \star$. $\square$

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 §1.8, 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.

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

The set of types, ranged over by 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:

1.1.16 DEFINITION: A *type* of kind κ is either a member of $\mathcal{V}_\kappa$, or a term of the form $\mathsf{F}\{d_1 \mapsto \mathsf{T}_1, \ldots, d_n \mapsto \mathsf{T}_n\}$, where F has signature $\{d_1 \mapsto \kappa_1, \ldots, d_n \mapsto \kappa_n\} \Rightarrow \kappa$ and $\mathsf{T}_1, \ldots, \mathsf{T}_n$ are types of kind $\kappa_1, \ldots, \kappa_n$, respectively. $\square$

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

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.

1.1.17 DEFINITION: A *type substitution* θ 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. $\square$

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

As pointed out earlier, types are first-order terms. As a result, every type variable that appears within a type T appears *free* within T. Things become

$$\frac{\Gamma(\mathtt{x}) = \mathtt{S}}{\Gamma \vdash \mathtt{x} : \mathtt{S}} \quad (\text{DM-VAR}) \qquad \frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{S} \qquad \Gamma; \mathtt{z} : \mathtt{S} \vdash \mathtt{t}_2 : \mathtt{T}}{\Gamma \vdash \mathtt{let}\ \mathtt{z} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}} \quad (\text{DM-LET})$$

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

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T} \to \mathtt{T}' \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}'} \quad (\text{DM-APP}) \qquad \frac{\Gamma \vdash \mathtt{t} : \forall \bar{\mathtt{X}}.\mathtt{T}}{\Gamma \vdash \mathtt{t} : [\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}]\mathtt{T}} \quad (\text{DM-INST})$$

Figure 1-3: Typing rules for DM

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.

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

One may view the type $\mathtt{T}$ as the trivial type scheme $\forall \varnothing.\mathtt{T}$, where no type variables are universally quantified, so types of kind $\star$ may be viewed as a subset of type schemes. The type scheme $\forall \bar{\mathtt{X}}.\mathtt{T}$ 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 1.2.2).

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

1.1.19 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 $\Gamma(\mathtt{x}) = \mathtt{S}$ if and only if Γ is of the form $\Gamma_1; \mathtt{x} : \mathtt{S}; \Gamma_2$, where Γ_2 contains no assumption about $\mathtt{x}$. The set of *defined program identifiers* of an environment Γ, written $\mathit{dpi}(\Gamma)$, is defined by $\mathit{dpi}(\varnothing) = \varnothing$ and $\mathit{dpi}(\Gamma; \mathtt{x} : \mathtt{S}) = \mathit{dpi}(\Gamma) \cup \{\mathtt{x}\}$. □

To complete the definition of Damas and Milner's type system, there remains 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 (§1.5). Rule DM-ABS specifies how to typecheck a λ-abstraction $\lambda \mathtt{z.t}$. 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 $\mathtt{T} \rightarrow \mathtt{T}'$ 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 1.1.20 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.

1.1.20 EXAMPLE: It is unsound to allow generalizing type variables that appear free in the environment. For instance, consider the typing judgement $\mathtt{z} : \mathtt{X} \vdash \mathtt{z} : \mathtt{X}$ **(1)**, which, according to DM-VAR, is valid. Applying an unrestricted version of DM-GEN to it, we obtain $\mathtt{z} : \mathtt{X} \vdash \mathtt{z} : \forall \mathtt{X}.\mathtt{X}$ **(2)**, whence, by DM-

INST, $\mathtt{z} : \mathsf{X} \vdash \mathtt{z} : \mathsf{Y}$ **(3)**. By DM-ABS and DM-GEN, we then have $\varnothing \vdash \lambda\mathtt{z}.\mathtt{z} : \forall\mathsf{XY}.\mathsf{X} \rightarrow \mathsf{Y}$. In other words, the identity function has unrelated argument and result types! Then, the expression $(\lambda\mathtt{z}.\mathtt{z})\ \hat{0}\ \hat{0}$, which reduces to the stuck expression $\hat{0}\ \hat{0}$, has type scheme $\forall\mathsf{Z}.\mathsf{Z}$. 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 $\mathtt{z}$ 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 $\mathtt{z} : \mathsf{X} \vdash \mathtt{z} : \forall\mathsf{Y}.\mathsf{Y}$—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 $\mathsf{T} \rightarrow \mathsf{T}'$. If instead the rule were to introduce the assumption $\mathtt{z} : \mathsf{S}$ into the environment, then its conclusion would have to form $\mathsf{S} \rightarrow \mathsf{T}'$, 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.

1.1.21 EXERCISE [RECOMMENDED, ★]: Build a type derivation for the expression $\lambda\mathtt{z}_1.\mathtt{let}\ \mathtt{z}_2 = \mathtt{z}_1\ \mathtt{in}\ \mathtt{z}_2$. □

1.1.22 EXERCISE [RECOMMENDED, ★]: Let int be a nullary type constructor of signature $\star$. Let Γ_0 consist of the bindings $\hat{+} : \mathsf{int} \rightarrow \mathsf{int} \rightarrow \mathsf{int}$ 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 $\mathtt{let}\ \mathtt{z} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2$ is syntactic sugar for $(\lambda\mathtt{z}.\mathtt{t}_2)\ \mathtt{t}_1$?

$$\begin{gathered}
\Gamma_0 \vdash \lambda\mathtt{z}.\mathtt{z} : \mathsf{int} \rightarrow \mathsf{int} \\
\Gamma_0 \vdash \lambda\mathtt{z}.\mathtt{z} : \forall\mathsf{X}.\mathsf{X} \rightarrow \mathsf{X} \\
\Gamma_0 \vdash \mathtt{let}\ \mathtt{f} = \lambda\mathtt{z}.\mathtt{z}\hat{+}\hat{1}\ \mathtt{in}\ \mathtt{f}\ \hat{2} : \mathsf{int} \\
\Gamma_0 \vdash \mathtt{let}\ \mathtt{f} = \lambda\mathtt{z}.\mathtt{z}\ \mathtt{in}\ \mathtt{f}\ \mathtt{f}\ \hat{2} : \mathsf{int}
\end{gathered}$$

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

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 $\mathtt{t}$, tells whether $\mathtt{t}$ is well-typed with respect to Γ, and if so, produces a *principal* type scheme $\mathtt{S}$. A principal type scheme is such that (i) it is valid, that is, $\Gamma \vdash \mathtt{t} : \mathtt{S}$ holds, and (ii) it is most general, that is, every judgement of the form $\Gamma \vdash \mathtt{t} : \mathtt{S}'$ follows from $\Gamma \vdash \mathtt{t} : \mathtt{S}$ 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 $\mathtt{t}$ may be summarized in the form of a single (principal) type scheme, which is very convenient.

1.1.3 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 §1.2. In §1.3, 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 §1.4, we show that type inference may be viewed as a combination of constraint generation and constraint solving, as promised earlier. Then, in §1.5, 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 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 name HM(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 §1.6, 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 §1.7, we explain how to extend ML-the-calculus with a number of features, including products, sums, references, recursion, algebraic data types, and recursive types. Last, in §1.8, we extend the type language with *rows* and use them to assign polymorphic type schemes to operations on records and variants.

1.2 Constraints

In this section, we define the syntax and logical meaning of constraints. Both are partly unspecified. Indeed, the set of *type constructors* (Definition 1.1.14) 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.

1.2.1 Syntax

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

1.2.1 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 $\kappa \otimes \kappa \Rightarrow \cdot$. □

1.2.2 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. □

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 1.2.3*
	let $x : \sigma$ in C	*Definition 1.2.3*
	$\exists\sigma$	*Definition 1.2.3*
	def Γ in C	*Definition 1.2.4*
	let Γ in C	*Definition 1.2.4*
	$\exists\Gamma$	*Definition 1.2.4*

Figure 1-4: Syntax of type schemes and constraints

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 def $x : \sigma$ in $C \wedge D$ is (def $x : \sigma$ 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 def $x : \sigma$ 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. 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 (2001). In order to be able to explain these last two forms, we must first introduce a number of derived constraint forms:

1.2.3 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 let $x : \sigma$ in C for $\exists\sigma \wedge$ def $x : \sigma$ 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 1.1.18). 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 $\mathsf{Y} \rightarrow \mathsf{Z}$ is *not* an instance of $\forall \mathsf{X}.\mathsf{X} \rightarrow \mathsf{X}$ in Damas and Milner's sense, because Y and Z are distinct type variables. In our presentation, on the other hand, $\forall \mathsf{X}.\mathsf{X} \rightarrow \mathsf{X} \preceq \mathsf{Y} \rightarrow \mathsf{Z}$ is not an assertion; rather, it is a constraint, which by definition is $\exists \mathsf{X}.(\mathsf{true} \wedge \mathsf{X} \rightarrow \mathsf{X} \leq \mathsf{Y} \rightarrow \mathsf{Z})$. We later prove that it is equivalent to $\exists \mathsf{X}.(\mathsf{Y} \leq \mathsf{X} \wedge \mathsf{X} \leq \mathsf{Z})$ and to $\mathsf{Y} \leq \mathsf{Z}$, and, if subtyping is interpreted as equality, to $\mathsf{Y} = \mathsf{Z}$. That is, $\sigma \preceq \mathsf{T}'$ represents a condition on (the ground types denoted by) the type variables in $ftv(\sigma, \mathsf{T}')$ 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 rule C-ExTrans on page 29). 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 $\forall \bar{\mathsf{X}}[C].\mathsf{T}$. The constraint C, whose free type variables may or may not be members of $\bar{\mathsf{X}}$, is meant to restrict the set of instances of the type scheme $\forall \bar{\mathsf{X}}[C].\mathsf{T}$. This is evident in the instantiation constraint $\forall \bar{\mathsf{X}}[C].\mathsf{T} \preceq \mathsf{T}'$, which by Definition 1.2.3 stands for $\exists \bar{\mathsf{X}}.(C \wedge \mathsf{T} \leq \mathsf{T}')$: the values that $\bar{\mathsf{X}}$ 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 $\mathsf{def}\ \mathsf{x} : \sigma\ \mathsf{in}\ C$ binds the name x to the type scheme σ within the constraint C. If C contains a subconstraint of the form $\mathsf{x} \preceq \mathsf{T}$, where this occurrence of x is free in C, then this subconstraint acquires the meaning $\sigma \preceq \mathsf{T}$. Thus, the constraint $\mathsf{x} \preceq \mathsf{T}$ is indeed an instantiation constraint, where the type scheme that is being instantiated is referred to by name. The constraint $\mathsf{def}\ \mathsf{x} : \sigma\ \mathsf{in}\ C$ may be viewed as an *explicit substitution* of the type scheme σ for the name x within C. Later (§1.4), 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 1.4, 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 1-9 on page 42.) 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, 2001). 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 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 T, whether T is an instance of σ (that is, whether the constraint $\sigma \preceq \mathsf{T}$ is satisfied) depends on the meaning assigned to the type variables in $\mathit{ftv}(\sigma, \mathsf{T})$. Similarly, given a type scheme, whether *some* type is an instance of σ (that is, whether the constraint $\exists \mathsf{Z}.\sigma \preceq \mathsf{Z}$, where 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 $\exists \mathsf{Z}.\sigma \preceq \mathsf{Z}$. In fact, we later prove that it is equivalent to $\exists\sigma$, as defined above. We also use the constraint form $\mathsf{let}\ \mathsf{x} : \sigma\ \mathsf{in}\ C$, which requires σ to have an instance and at the same time associates it with the name 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 §1.2; we always use let instead.

1.2.4 DEFINITION: Environments Γ remain as in Definition 1.1.19, except DM type schemes 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 $\mathit{fpi}(\Gamma; \mathsf{x} : \sigma) = \mathit{fpi}(\Gamma) \cup \mathit{fpi}(\sigma)$. We write $\mathit{dfpi}(\Gamma)$ for $\mathit{dpi}(\Gamma) \cup \mathit{fpi}(\Gamma)$. We define $\mathsf{def}\ \varnothing\ \mathsf{in}\ C$ as C and $\mathsf{def}\ \Gamma; \mathsf{x} : \sigma\ \mathsf{in}\ C$ as $\mathsf{def}\ \Gamma\ \mathsf{in}\ \mathsf{def}\ \mathsf{x} : \sigma\ \mathsf{in}\ C$. Similarly, we define $\mathsf{let}\ \varnothing\ \mathsf{in}\ C$ as C and $\mathsf{let}\ \Gamma; \mathsf{x} : \sigma\ \mathsf{in}\ C$ as $\mathsf{let}\ \Gamma\ \mathsf{in}\ \mathsf{let}\ \mathsf{x} : \sigma\ \mathsf{in}\ C$. We define $\exists\varnothing$ as true and $\exists(\Gamma; \mathsf{x} : \sigma)$ as $\exists\Gamma \wedge \mathsf{def}\ \Gamma\ \mathsf{in}\ \exists\sigma$. □

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:

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

The application of a constraint context $\mathcal{C}$ to a constraint C, written $\mathcal{C}[C]$, 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 $\mathcal{C}[C]$. We write $dtv(\mathcal{C})$ and $dpi(\mathcal{C})$ for the sets of type variables and program identifiers, respectively, that $\mathcal{C}$ may thus capture. We write $\mathsf{let}\ x : \forall\bar{X}[\mathcal{C}].T\ \mathsf{in}\ C$ for $\exists\bar{X}.\mathcal{C} \wedge \mathsf{def}\ x : \forall\bar{X}[\mathcal{C}].T\ \mathsf{in}\ C$. (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 $\mathcal{X} ::= [] \mid \exists\bar{X}.\mathcal{X}$.

1.2.2 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*:

1.2.5 DEFINITION: For every kind κ, let $\mathcal{M}_\kappa$ be a nonempty set, whose elements are called 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 system HM(X), described in §1.3, is *parameterized* by X.

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

1.2.6 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*. □

1.2.7 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. Note that free finite tree models coincide with syntactic models, as defined in the previous example. □

Rows (§1.8) are interpreted in a tree model, albeit not a free one. The following examples suggest different ways of interpreting the subtyping predicate.

1.2.8 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*. □

1.2.9 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; introducing, in addition to the type constructor $\rightarrow$, two type constructors $\perp$ and $\top$ of signature $\star$; and letting $\perp \leqslant \rightarrow \leqslant \top$. This gives rise to a model where $\perp$ 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, $\perp$ and $\perp \rightarrow \top$.

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 $\{content \mapsto \star\} \Rightarrow \circ$, and let $\mathsf{pre} \leqslant \mathsf{abs}$. This gives rise to a model where $\mathsf{pre}\ \mathrm{t} \leq \mathsf{abs}$ holds for every $\mathrm{t} \in \mathcal{M}_\star$. Again, comparable ground types may have different shapes: consider, for instance, $\mathsf{pre}\ \top$ and abs. §1.8 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 $\mathrm{t}_1 \leq \mathrm{t}_2$ holds, then $dom(\mathrm{t}_1)$ and $dom(\mathrm{t}_2)$ 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, 1991; 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. □

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,

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

$$\frac{P(\phi(T_1),\ldots,\phi(T_n))}{\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ P\ T_1\ \ldots\ T_n} \qquad \text{(CM-Predicate)}$$

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

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

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

Figure 1-5: Meaning of constraints

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.

1.2.10 Definition: A *ground assignment* ϕ is a total, kind-preserving mapping from $\mathcal{V}$ into $\mathcal{M}$. Ground assignments are extended to types by $\phi(F\,T_1\,\ldots\,T_n) = F(\phi(T_1),\ldots,\phi(T_n))$. Then, for every type T of kind κ, $\phi(T)$ 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. Which rule applies 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 $\mathsf{def}\ \Gamma\ \mathsf{in\ true}$ is a tautology, that is, holds under every ground assignment. No rule matches constraints of the form $\mathsf{def}\ \Gamma\ \mathsf{in\ false}$, 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 1.2.5, 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 $\vec{\mathsf{X}}$ to denote arbitrary ground types $\vec{t}$ within C, independently of their image through ϕ. We implicitly require $\vec{\mathsf{X}}$ and $\vec{t}$ to have matching kinds, so that $\phi[\vec{\mathsf{X}} \mapsto \vec{t}]$ remains a kind-preserving ground assignment. The side condition $\bar{\mathsf{X}} \mathrel{\#} \mathit{ftv}(\Gamma)$—which may always be satisfied by suitable α-conversion of the constraint $\exists\bar{\mathsf{X}}.\mathsf{C}$—prevents free occurrences of the type variables $\bar{\mathsf{X}}$ within Γ from being unduly affected. CM-INSTANCE concerns constraints of the form $\mathsf{def}\ \Gamma\ \mathsf{in}\ \mathsf{x} \preceq \mathsf{T}'$. The constraint $\mathsf{x} \preceq \mathsf{T}'$ is turned into $\sigma \preceq \mathsf{T}'$, where, according to the second premise, σ is $\Gamma(\mathsf{x})$. Recall that constraints of such a form were introduced in Definition 1.2.3. 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 $\mathsf{def}\ \mathsf{x} : \sigma\ \mathsf{in}\ \mathsf{C}$ and $[\mathsf{x} \mapsto \sigma]\mathsf{C}$ 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. 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 $\exists \mathsf{Z}.\mathsf{x} \preceq \mathsf{Z}$. Because the identifier x is free, CM-INSTANCE is not applicable, so the constraint is not satisfiable. Furthermore, placing it within the context $\mathsf{let}\ \mathsf{x} : \forall \mathsf{X}.\mathsf{X}\ \mathsf{in}\ []$ makes it satisfied by every ground assignment, so it is not false. In a standard first-order logic, the assertions "C is satisfiable" and "C is false" are complementary. Here, however, they may not be so; they are so when $\mathit{fpi}(\mathsf{C}) = \varnothing$ holds.

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.

1.2.11 DEFINITION: We write $\mathsf{C}_1 \Vdash \mathsf{C}_2$, and say that C_1 *entails* C_2, if and only if, for every ground assignment ϕ and for every environment Γ, $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ \mathsf{C}_1$ implies $\phi \models \mathsf{def}\ \Gamma\ \mathsf{in}\ \mathsf{C}_2$. We write $\mathsf{C}_1 \equiv \mathsf{C}_2$, and say that C_1 and C_2 are *equivalent*, if and only if $\mathsf{C}_1 \Vdash \mathsf{C}_2$ and $\mathsf{C}_2 \Vdash \mathsf{C}_1$ 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 $\mathsf{C} \equiv \mathsf{false}$ holds. It is straightforward to check that entailment is reflexive and transitive and that $\equiv$ is indeed an equivalence relation.

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 F be a type constructor of signature $\kappa_1 \otimes \ldots \otimes \kappa_n \Rightarrow \kappa$. Let $i \in \{1, \ldots, n\}$. F is *covariant* (resp. *contravariant*, *invariant*) with respect to its i^{th} parameter if and only if, for all types $T_1, \ldots, T_n$ and T'_i of appropriate kinds, the constraint $F\,T_1 \ldots T_i \ldots T_n \leq F\,T_1 \ldots T'_i \ldots T_n$ is equivalent to $T_i \leq T'_i$ (resp. $T'_i \leq T_i$, $T_i = T'_i$). 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:

$$T_1 \rightarrow T_2 \leq T'_1 \rightarrow T'_2 \equiv T'_1 \leq T_1 \wedge T_2 \leq T'_2 \qquad \text{(C-ARROW)}$$

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 $F_1\,\vec{T}_1 \leq F_2\,\vec{T}_2$ and $F_2\,\vec{T}_2 \leq F_1\,\vec{T}_1$ are false. 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.

1.2.3 Reasoning with constraints

In this section, we give a number of equivalence laws that are often useful and help understand the meaning of constraints. To begin, we note that entailment is preserved by arbitrary constraint contexts, as stated by the following theorem. As a result, constraint equivalence is a congruence. Throughout this chapter, these facts are often used implicitly.

1.2.12 THEOREM [CONGRUENCE]: $\mathsf{C}_1 \Vdash \mathsf{C}_2$ implies $\mathcal{C}[\mathsf{C}_1] \Vdash \mathcal{C}[\mathsf{C}_2]$. □

Next, we define what it means for a constraint to determine a set of type variables. In short, C *determines* $\bar{\mathsf{Y}}$ if and only if, given a ground assignment for $ftv(\mathsf{C}) \setminus \bar{\mathsf{Y}}$ and given that C holds, it is possible to reconstruct, in a unique way, a ground assignement for $\bar{\mathsf{Y}}$. Determinacy appears in the equivalence law C-LETALL on page 29 and is exploited by the constraint solver in §1.6.

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

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.

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

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 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 1.2.3, $\mathsf{T}' \preceq \mathsf{T}$ and $\mathsf{T}' \leq \mathsf{T}$ are equivalent. As a result, specializing C-EXTRANS to the case where σ is a monotype, we find that $\mathsf{T}' \leq \mathsf{T}$ is equivalent to $\exists \mathsf{Z}.(\mathsf{T}' \leq \mathsf{Z} \wedge \mathsf{Z} \leq \mathsf{T})$, for fresh Z, a standard equivalence law. When oriented from left to right, it becomes

$$
\begin{array}{rcll r}
C_1 \wedge C_2 & \equiv & C_2 \wedge C_1 & & (\text{C-AND}) \\
(C_1 \wedge C_2) \wedge C_3 & \equiv & C_1 \wedge (C_2 \wedge C_3) & & (\text{C-ANDAND}) \\
C_1 \wedge C_2 & \equiv & C_1 & \text{if } C_1 \Vdash C_2 & (\text{C-DUP}) \\
\exists \bar{X}.\exists \bar{Y}.C & \equiv & \exists \bar{X}\bar{Y}.C & & (\text{C-EXEX}) \\
\exists \bar{X}.C & \equiv & C & \text{if } \bar{X} \# \mathit{ftv}(C) & (\text{C-EX*}) \\
(\exists \bar{X}.C_1) \wedge C_2 & \equiv & \exists \bar{X}.(C_1 \wedge C_2) & \text{if } \bar{X} \# \mathit{ftv}(C_2) & (\text{C-EXAND}) \\
\exists Z.(\sigma \preceq Z \wedge Z \leq T) & \equiv & \sigma \preceq T & \text{if } Z \notin \mathit{ftv}(\sigma, T) & (\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] & & (\text{C-INID}) \\
 & & \multicolumn{2}{l}{\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 & \text{if } \mathit{dpi}(\Gamma) \# \mathit{fpi}(C) & (\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) & & (\text{C-INAND}) \\
\text{let } \Gamma \text{ in } (C_1 \wedge C_2) & \equiv & (\text{let } \Gamma \text{ in } C_1) \wedge C_2 & \text{if } \mathit{dpi}(\Gamma) \# \mathit{fpi}(C_2) & (\text{C-INAND*}) \\
\text{let } \Gamma \text{ in } \exists \bar{X}.C & \equiv & \exists \bar{X}.\text{let } \Gamma \text{ in } C & \text{if } \bar{X} \# \mathit{ftv}(\Gamma) & (\text{C-INEX}) \\
\text{let } \Gamma_1; \Gamma_2 \text{ in } C & \equiv & \text{let } \Gamma_2; \Gamma_1 \text{ in } C & & (\text{C-LETLET}) \\
 & & \multicolumn{2}{l}{\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 & \text{if } \bar{X} \# \mathit{ftv}(C_1) & (\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 & & (\text{C-LETDUP}) \\
 & & \multicolumn{2}{l}{\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 & \text{if } \bar{Y} \# \mathit{ftv}(T) & (\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 & & (\text{C-LETALL}) \\
 & & \multicolumn{2}{l}{\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 & \text{if } X \notin \mathit{ftv}(T, C) & (\text{C-LETSUB}) \\
\vec{X} = \vec{T} \wedge [\vec{X} \mapsto \vec{T}]C & \equiv & \vec{X} = \vec{T} \wedge C & & (\text{C-EQ}) \\
\text{true} & \equiv & \exists \bar{X}.(\vec{X} = \vec{T}) & \text{if } \bar{X} \# \mathit{ftv}(T) & (\text{C-NAME}) \\
[\vec{X} \mapsto \vec{T}]C & \equiv & \exists \bar{X}.(\vec{X} = \vec{T} \wedge C) & \text{if } \bar{X} \# \mathit{ftv}(T) & (\text{C-NAMEEQ})
\end{array}
$$

Figure 1-6: Constraint equivalence laws

an interesting *simplification* law: in a chain of subtyping constraints, an intermediate variable such as Z may be suppressed, provided it is *local*, as witnessed by the existential quantifier $\exists\mathsf{Z}$. C-InId states that, within the scope of the binding $\mathsf{x} : \sigma$, every *free* occurrence of x may be safely replaced with σ. The restriction to free occurrences stems from the side-condition $\mathsf{x} \notin dpi(\mathcal{C})$. When the rule is oriented from left to right, its other side-conditions, which require the context $\mathsf{let}\ \mathsf{x} : \sigma\ \mathsf{in}\ \mathcal{C}$ 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 $\forall\bar{\mathsf{X}}[\mathsf{C}_1 \wedge \mathsf{C}_2].\mathsf{T}$, provided it does not involve any of the universally quantified type variables $\bar{\mathsf{X}}$. When oriented from left to right, the rule yields an important simplification law: indeed, taking an instance of $\forall\bar{\mathsf{X}}[\mathsf{C}_2].\mathsf{T}$ is less expensive than taking an instance of $\forall\bar{\mathsf{X}}[\mathsf{C}_1 \wedge \mathsf{C}_2].\mathsf{T}$, 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 $\bar{\mathsf{Y}}$ 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 $\bar{\mathsf{Y}}$ is existentially quantified. 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.

1.2.15 Example: C-LetAll would be invalid without the condition that $\exists\bar{\mathsf{X}}.\mathsf{C}_1$ determines $\bar{\mathsf{Y}}$. Consider, for instance, the constraint $\mathsf{let}\ \mathsf{x} : \forall\mathsf{Y}.\mathsf{Y} \rightarrow \mathsf{Y}\ \mathsf{in}\ (\mathsf{x} \preceq \mathsf{int} \rightarrow \mathsf{int} \wedge \mathsf{x} \preceq \mathsf{bool} \rightarrow \mathsf{bool})$ **(1)**, where int and bool are incompatible nullary type constructors. By C-InId and C-In*, it is equivalent to $\forall\mathsf{Y}.\mathsf{Y} \rightarrow \mathsf{Y} \leq \mathsf{int} \rightarrow \mathsf{int} \wedge \forall\mathsf{Y}.\mathsf{Y} \rightarrow \mathsf{Y} \leq \mathsf{bool} \rightarrow \mathsf{bool}$ which, by Definition 1.2.3, mean $\exists\mathsf{Y}.(\mathsf{Y} \rightarrow \mathsf{Y} \leq \mathsf{int} \rightarrow \mathsf{int}) \wedge \exists\mathsf{Y}.(\mathsf{Y} \rightarrow \mathsf{Y} \leq \mathsf{bool} \rightarrow \mathsf{bool})$, that is, true. Now, if C-LetAll was valid without its side-condition, then (1) would also be equivalent to $\exists\mathsf{Y}.\mathsf{let}\ \mathsf{x} : \mathsf{Y} \rightarrow \mathsf{Y}\ \mathsf{in}\ (\mathsf{x} \preceq \mathsf{int} \rightarrow \mathsf{int} \wedge \mathsf{x} \preceq \mathsf{bool} \rightarrow \mathsf{bool})$, which by C-InId and C-In* is $\exists\mathsf{Y}.(\mathsf{Y} \rightarrow \mathsf{Y} \leq \mathsf{int} \rightarrow \mathsf{int} \wedge \mathsf{Y} \rightarrow \mathsf{Y} \leq \mathsf{bool} \rightarrow \mathsf{bool})$.

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 $\forall \mathsf{Y}$ quantifier, every instance of σ receives its own $\exists \mathsf{Y}$ 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. It is possible to prove 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 $\mathsf{let}\ \mathsf{x} : \forall \mathsf{X}[\exists \mathsf{Y}.\mathsf{X} = \mathsf{Y} \rightarrow \mathsf{Y}].\mathsf{X}\ \mathsf{in}\ (\mathsf{x} \preceq \mathsf{int} \rightarrow \mathsf{int} \wedge \mathsf{x} \preceq \mathsf{bool} \rightarrow \mathsf{bool})$.

Naturally, in the above examples, the side-condition "true determines Y" does *not* hold: by Definition 1.2.13, 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 $\exists \mathsf{X}.\mathsf{Y} = \mathsf{int}$ determines Y; see Lemma 1.6.7. As a result, C-LETALL states that $\mathsf{let}\ \mathsf{x} : \forall \mathsf{XY}[\mathsf{Y} = \mathsf{int}].\mathsf{Y} \rightarrow \mathsf{X}\ \mathsf{in}\ C$ **(1)** is equivalent to $\exists \mathsf{Y}.\mathsf{let}\ \mathsf{x} : \forall \mathsf{X}[\mathsf{Y} = \mathsf{int}].\mathsf{Y} \rightarrow \mathsf{X}\ \mathsf{in}\ C$ **(2)**, provided $\mathsf{Y} \notin ftv(C)$. The intuition is simple: because Y is forced to assume the value int by the equation $\mathsf{Y} = \mathsf{int}$, it makes no difference whether Y is or isn't universally quantified. We remark that, by C-LETAND, (2) is equivalent to $\exists \mathsf{Y}.(\mathsf{Y} = \mathsf{int} \wedge \mathsf{let}\ \mathsf{x} : \forall \mathsf{X}.\mathsf{Y} \rightarrow \mathsf{X}\ \mathsf{in}\ C)$ **(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 $\mathsf{Y} = \mathsf{int}$ 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 $\vec{\mathsf{X}}$ for the types $\vec{\mathsf{T}}$. As always, $\vec{\mathsf{X}}$ stands for a vector of *distinct* type variables; $\vec{\mathsf{T}}$ 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 $\bar{\mathsf{X}}$ do not occur free within the terms 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.

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 (omitted!) 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.

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

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

1.2.4 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*. These results are exploited in §1.3.3.

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 Theorem 1.3.7.

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 §1.6 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.

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

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.

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

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

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

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

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

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.

1.2.22 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 1.2.17, 1.2.19, and 1.2.21, every satisfiable constraint admits a most general unifier.

1.3 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 22. 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).

1.3.1 Definition

Our presentation of HM(X) relies on the constraint language introduced in §1.2. 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 (§1.2). 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*. Note that this does not prevent them from containing let forms.

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

$$\frac{\Gamma(\mathtt{x}) = \sigma \qquad C \Vdash \exists\sigma}{C, \Gamma \vdash \mathtt{x} : \sigma} \quad (\textsc{hmx-Var})$$

$$\frac{C, (\Gamma; \mathtt{z} : \mathsf{T}) \vdash \mathtt{t} : \mathsf{T}'}{C, \Gamma \vdash \lambda \mathtt{z}.\mathtt{t} : \mathsf{T} \rightarrow \mathsf{T}'} \quad (\textsc{hmx-Abs})$$

$$\frac{C, \Gamma \vdash \mathtt{t}_1 : \mathsf{T} \rightarrow \mathsf{T}' \qquad C, \Gamma \vdash \mathtt{t}_2 : \mathsf{T}}{C, \Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathsf{T}'} \quad (\textsc{hmx-App})$$

$$\frac{C, \Gamma \vdash \mathtt{t}_1 : \sigma \qquad C, (\Gamma; \mathtt{z} : \sigma) \vdash \mathtt{t}_2 : \mathsf{T}}{C, \Gamma \vdash \mathtt{let}\ \mathtt{z} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathsf{T}} \quad (\textsc{hmx-Let})$$

$$\frac{C \wedge D, \Gamma \vdash \mathtt{t} : \mathsf{T} \qquad \bar{\mathsf{X}} \# \mathit{ftv}(C, \Gamma)}{C \wedge \exists \bar{\mathsf{X}}.D, \Gamma \vdash \mathtt{t} : \forall \bar{\mathsf{X}}[D].\mathsf{T}} \quad (\textsc{hmx-Gen})$$

$$\frac{C, \Gamma \vdash \mathtt{t} : \forall \bar{\mathsf{X}}[D].\mathsf{T}}{C \wedge D, \Gamma \vdash \mathtt{t} : \mathsf{T}} \quad (\textsc{hmx-Inst})$$

$$\frac{C, \Gamma \vdash \mathtt{t} : \mathsf{T} \qquad C \Vdash \mathsf{T} \leq \mathsf{T}'}{C, \Gamma \vdash \mathtt{t} : \mathsf{T}'} \quad (\textsc{hmx-Sub})$$

$$\frac{C, \Gamma \vdash \mathtt{t} : \sigma \qquad \bar{\mathsf{X}} \# \mathit{ftv}(\Gamma, \sigma)}{\exists \bar{\mathsf{X}}.C, \Gamma \vdash \mathtt{t} : \sigma} \quad (\textsc{hmx-Exists})$$

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

gram identifiers. 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 $C, \Gamma \vdash \mathtt{t} : \sigma$ and $D, \Gamma \vdash \mathtt{t} : \sigma$ 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. Note that a valid judgement may involve an unsatisfiable constraint. A program $\mathtt{t}$ is *well-typed* within the (closed) environment Γ if and only if a judgement of the form $C, \Gamma \vdash \mathtt{t} : \sigma$ 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, hence the two requirements are equivalent.

Let us now explain the rules. Like dm-Var, hmx-Var looks up the environment to determine the type scheme associated with the program identifier $\mathtt{x}$. 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 $\forall\varnothing[\mathsf{true}].\mathsf{T}$ (Definitions 1.1.18 and 1.2.2). As a result, types form a subset of type schemes, which implies that $\Gamma; \mathtt{z} : \mathsf{T}$ is a well-formed environment and

$\mathsf{C}, \Gamma \vdash \mathtt{t} : \mathsf{T}$ 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:

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

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 $\forall \bar{\mathsf{X}}[\mathsf{D}].\mathsf{T}$, where the type variables $\bar{\mathsf{X}}$ have become universally quantified, but are still subject to the constraint D. Note that the type variables that occur free in D may include not only $\bar{\mathsf{X}}$, 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 $\bar{\mathsf{X}}$. 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 $\mathsf{C} \wedge \exists \bar{\mathsf{X}}.\mathsf{D}$ 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 1.3.1. 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 $\vec{\mathsf{X}} \mapsto \vec{\mathsf{X}}$. 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 1.3.2 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 T'. Because both T and T' may have free type variables, whether $\mathsf{T} \leq \mathsf{T}'$ 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. 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, $C \equiv D$ implies $\exists\bar{X}.C \equiv \exists\bar{X}.D$, but the converse does not hold in general. For instance, there is in general no way of simplifying the judgement $X \leq Y \leq Z, \Gamma \vdash t : \sigma$, but if it is known that Y does not appear free in Γ or σ, then HMX-EXISTS allows deriving $\exists Y.(X \leq Y \leq Z), \Gamma \vdash t : \sigma$, which is the same judgement as $X \leq Z, \Gamma \vdash t : \sigma$. Thus, an interesting simplification has been enabled. Note that $X \leq Y \leq Z \equiv X \leq Z$ does *not* hold, while, according to C-EXTRANS, $\exists Y.(X \leq Y \leq Z) \equiv X \leq Z$ does.

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.

1.3.1 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. □

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

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

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. □

1.3.3 EXERCISE [★]: Give a derivation of $\mathsf{true}, \varnothing \vdash \lambda z.z : \mathsf{int} \rightarrow \mathsf{int}$. Give a derivation of $\mathsf{true}, \varnothing \vdash \lambda z.z : \forall X.X \rightarrow X$. Check that the former judgement also follows from the latter via HMX-INST′ (Exercise 1.3.2), and determine which derivation of $\mathsf{true}, \varnothing \vdash \lambda z.z : \mathsf{int} \rightarrow \mathsf{int}$ this path gives rise to. □

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

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

$$\frac{\mathsf{C}, \Gamma \vdash \mathtt{t}_1 : \mathsf{T} \to \mathsf{T}' \qquad \mathsf{C}, \Gamma \vdash \mathtt{t}_2 : \mathsf{T}}{\mathsf{C}, \Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathsf{T}'} \quad \text{(HMD-APP)}$$

$$\frac{\begin{array}{c}\mathsf{C} \wedge \mathsf{D}, \Gamma \vdash \mathtt{t}_1 : \mathsf{T}_1 \qquad \bar{\mathsf{X}} \# \mathit{ftv}(\mathsf{C}, \Gamma) \\ \mathsf{C} \wedge \exists\bar{\mathsf{X}}.\mathsf{D}, (\Gamma; \mathtt{z} : \forall\bar{\mathsf{X}}[\mathsf{D}].\mathsf{T}_1) \vdash \mathtt{t}_2 : \mathsf{T}_2\end{array}}{\mathsf{C} \wedge \exists\bar{\mathsf{X}}.\mathsf{D}, \Gamma \vdash \mathtt{let}\ \mathtt{z} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathsf{T}_2} \quad \text{(HMD-LETGEN)}$$

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

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

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

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 (1994), 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).

1.3.2 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 $\mathsf{C}, \Gamma \vdash \mathtt{t} : \mathsf{T}$, rather than $\mathsf{C}, \Gamma \vdash \mathtt{t} : \sigma$. The following theorem states that the two presentations are indeed equivalent.

1.3.4 THEOREM: $\mathsf{C}, \Gamma \vdash \mathtt{t} : \mathsf{T}$ is derivable via the rules of Figure 1-8 if and only if it is a valid HM(X) judgement. □

This theorem shows that the rule sets of Figures 1-7 and 1-8 derive the same monomorphic judgements, that is, the same judgements of the form $\mathsf{C}, \Gamma \vdash \mathtt{t} : \mathsf{T}$. The fact that judgements of the form $\mathsf{C}, \Gamma \vdash \mathtt{t} : \sigma$, where σ is a not a monotype, cannot be derived using the new rule set is a technical simplification, without deep significance.

1.3.5 EXERCISE [★★★, $\not\rightarrow$]: 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 $\Gamma \vdash \mathtt{t} : \mathsf{T}$; 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). □

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

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

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 1.2.22), whose properties we make essential use of.

Unfortunately, by lack of space, we cannot give the details of this translation, which are fairly involved. Let us merely say that, given a type scheme σ and an idempotent type substitution θ such that $ftv(\sigma) \subseteq dom(\theta)$ and $\exists\theta \Vdash \exists\sigma$ hold, the translation of σ under θ is a DM type scheme, written $[\![\sigma]\!]_\theta$. Its meaning is intended to be the same as that of the HM(X) type scheme $\theta(\sigma)$. The translation is extended to environments in such a way that $[\![\Gamma]\!]_\theta$ is defined when $ftv(\Gamma) \subseteq dom(\theta)$ holds. We are now ready to state the main theorem.

1.3.7 THEOREM: Let $C, \Gamma \vdash \mathtt{t} : \sigma$ 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. □

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.

Together, Theorems 1.3.6 and 1.3.7 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 1.3.6 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.

1.4 Constraint generation

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

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 $C, \Gamma \vdash \mathtt{t} : \mathsf{T}$ holds, then, by the completeness property, $C \Vdash [\![\Gamma \vdash \mathtt{t} : \mathsf{T}]\!]$ holds, so $[\![\Gamma \vdash \mathtt{t} : \mathsf{T}]\!]$ is satisfiable. Conversely, by the soundness property, if $[\![\Gamma \vdash \mathtt{t} : \mathsf{T}]\!]$

is satisfiable, then we have a satisfiable constraint C such that $C, \Gamma \vdash \mathtt{t} : \mathtt{T}$ holds. In other words, $\mathtt{t}$ is well-typed with type $\mathtt{T}$ under environment Γ if and only if $[\![\Gamma \vdash \mathtt{t} : \mathtt{T}]\!]$ is satisfiable.

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

1.4.1 Exercise [Recommended, ★]: Let $\mathtt{X} \notin \mathit{ftv}(\Gamma)$. Show that, if there exist a satisfiable constraint C and a type $\mathtt{T}$ such that $C, \Gamma \vdash \mathtt{t} : \mathtt{T}$ holds, then there exists a satisfiable constraint C' such that $C', \Gamma \vdash \mathtt{t} : \mathtt{X}$ holds. Conclude that, given a closed environment Γ and an arbitrary type variable $\mathtt{X}$, the term $\mathtt{t}$ is well-typed within Γ if and only if $[\![\Gamma \vdash \mathtt{t} : \mathtt{X}]\!]$ is satisfiable. □

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 Γ and $\mathtt{t}$ 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 Γ and $\mathtt{t}$ 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 $[\![\Gamma \vdash \mathtt{t} : \mathtt{T}]\!]$ is least specific in the sense that all valid constraints entail it. More about principal types and principal typings may be found in (Jim, 1996; Wells, 2002).

How do we perform constraint generation? A standard approach (Sulzmann, Müller, and Zenger, 1999; Bonniot, 2002) is to define $[\![\Gamma \vdash \mathtt{t} : \mathtt{T}]\!]$ by induction on the structure of $\mathtt{t}$. At every let node, following HMD-LetGen, part of the current constraint, namely D, is turned into a type scheme, namely $\forall \bar{\mathtt{X}}[D].\mathtt{T}$, which is used to extend the environment. Then, at every occurrence of the program variable that was bound at this $\mathtt{let}$ node, following HMD-VarInst, this type scheme is retrieved from the environment, and a copy of D is added back to the current constraint. If such an approach is adopted, it

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

Figure 1-9: Constraint generation

is important to *simplify* the type scheme $\forall \bar{\mathsf{X}}[D].\mathsf{T}$ *before* it is stored in the environment, because it would be inefficient to copy an unsimplified constraint. In other words, in an efficient implementation of this standard approach, constraint generation and constraint simplification cannot be separated.

Type scheme introduction and elimination constraints, which we introduced in §1.2 but did not use in the specification of HM(X), are intended as a means of solving this problem. By extending our vocabulary, we are able to achieve the desired separation between constraint generation, on the one hand, and constraint solving and simplification, on the other hand, without compromising efficiency. Indeed, by exploiting these new constraint forms, we may define a constraint generation procedure whose time and space complexity is linear, because it no longer involves copying subconstraints back and forth between the environment and the constraint that is being generated. (It is then up to the constraint solver to perform simplification and copying, if and when necessary.) In fact, the environment is suppressed altogether: we define $[\![\mathtt{t} : \mathsf{T}]\!]$ by induction on the structure of $\mathtt{t}$—notice the absence of the parameter Γ. Then, the constraint $[\![\Gamma \vdash \mathtt{t} : \mathsf{T}]\!]$ discussed above becomes syntactic sugar for $\mathsf{let}\ \Gamma\ \mathsf{in}\ [\![\mathtt{t} : \mathsf{T}]\!]$. We now employ the full constraint language: the program identifiers that appear free in $\mathtt{t}$ may also appear free in $[\![\mathtt{t} : \mathsf{T}]\!]$, as part of instantiation constraints. They become bound when $[\![\mathtt{t} : \mathsf{T}]\!]$ is placed within the context $\mathsf{let}\ \Gamma\ \mathsf{in}\ []$. A similar approach to constraint generation appears in (Müller, 1994).

The defining equations for $[\![\mathtt{t} : \mathsf{T}]\!]$ appear in Figure 1-9. We refer to them as the *constraint generation rules*. The definition is quite terse, and certainly simpler than the declarative specification of HM(X) given in Figure 1-7; yet, we prove below that the two are equivalent.

Before explaining the definition, we state the requirements that bear on the type variables X_1, X_2, and 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, $\mathsf{X}_1 \neq \mathsf{X}_2$ 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 X_1, X_2, and 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 T. However, this simplification would be rendered invalid by the introduction of open type annotations within expressions. Note that we are able to state a *precise* (as opposed to informal) freshness requirement. This is made possible by the fact that $[\![t : T]\!]$ has no free type variables other than those of T, which in turn depends on our explicit use of existential quantification to limit the scope of auxiliary variables.

Let us now review the four equations. The first equation may be read: x *has type* T *if and only if* T *is an instance of the type scheme associated with* x. Note that we no longer consult the type scheme associated with x in the environment—indeed, there is no environment. Instead, we merely generate an instantiation constraint, where x appears free. (For this reason, every program identifier that occurs free within t typically also occurs free within $[\![t : T]\!]$.) This constraint acquires its full meaning when it is later placed within a context of the form $\text{let } x : \sigma \text{ in } []$. This equation roughly corresponds to HMD-VARINST. The second equation may be read: $\lambda z.t$ *has type* T *if and only if, for some* X_1 *and* X_2, *(i) under the assumption that* z *has type* X_1, t *has type* X_2, *and (ii)* T *is a supertype of* $X_1 \rightarrow X_2$. Here, the types associated with z and t must be fresh type variables, namely X_1 and X_2, because we cannot in general guess them. These type variables are *bound* so as to guarantee that the generated constraint is unique up to α-conversion. They are *existentially* bound because we intend the constraint solver to discover their value. Condition (i) is expressed by the subconstraint $\text{let } z : X_1 \text{ in } [\![t : X_2]\!]$. This makes sense as follows. $[\![t : X_2]\!]$ typically contains a number of instantiation constraints bearing on z, of the form $z \preceq T_i$. By wrapping it within the context $\text{let } z : X_1 \text{ in } []$, we effectively require every T_i to be a supertype of X_1. Note that z does not occur free in the constraint $\text{let } z : X_1 \text{ in } [\![t : X_2]\!]$, which is necessary for well-formedness of the definition, since it does not occur free in $\lambda z.t$. This equation roughly corresponds to HMD-EXISTS, HMD-ABS, and HMD-SUB. The third equation may be read: $t_1\ t_2$ *has type* T *if and only if, for some* X_2, t_1 *has type* $X_2 \rightarrow T$ *and* t_2 *has type* X_2. Here, the fresh type variable X_2 stands for the unknown type of t_2. This equation roughly corresponds to HMD-APP. The last equation, which roughly corresponds to HMD-LETGEN, may be read: $\texttt{let } z = t_1 \texttt{ in } t_2$ *has type* T *if and only if, under the assumption that* z *has every type* X *such that* $[\![t_1 : X]\!]$ *holds,* t_2 *has type* T. As in the case of λ-abstractions, the instantiation constraints bearing on $\mathtt{z}$ that appear within $[\![\mathtt{t}_2 : \mathsf{T}]\!]$ are given a meaning via a let prefix. The difference is that $\mathtt{z}$ may now be assigned a type scheme, as opposed to a monotype. An appropriate type scheme is built as follows. The constraint $[\![\mathtt{t}_1 : \mathsf{X}]\!]$ is the *least specific* constraint that must be imposed on the fresh type variable X so as to make it a valid type for $\mathtt{t}_1$. In other words, $\mathtt{t}_1$ has every type X such that $[\![\mathtt{t}_1 : \mathsf{X}]\!]$ holds, and none other. That is, the type scheme $\forall \mathsf{X}[[\![\mathtt{t}_1 : \mathsf{X}]\!]].\mathsf{X}$, abbreviated σ in the following, is a *principal* type scheme for $\mathtt{t}_1$. It is interesting to note that there is no question of *which* type variables to generalize. Indeed, by construction, no type variables other than X may appear free in $[\![\mathtt{t}_1 : \mathsf{X}]\!]$, so we cannot generalize *more* variables. On the other hand, it is valid to generalize X, since it does not appear free anywhere else. This interesting simplification is inspired by (Sulzmann, Müller, and Zenger, 1999), where a similar technique is used. Now, what happens when $[\![\mathtt{t}_2 : \mathsf{T}]\!]$ is placed inside the context $\mathsf{let}\ \mathtt{z} : \sigma\ \mathsf{in}\ []$? When placed inside this context, an instantiation constraint of the form $\mathtt{z} \preceq \mathsf{T}'$ acquires the meaning $\sigma \preceq \mathsf{T}'$, which by definition of σ and by Lemma 1.4.6 (see below) is equivalent to $[\![\mathtt{t}_1 : \mathsf{T}']\!]$. Thus, the constraint produced by the fourth equation simulates a textual expansion of the $\mathtt{let}$ 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.

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

1.4.3 EXERCISE [RECOMMENDED, ★, ↛]: Compute and simplify, as best as you can, the constraint $[\![\mathtt{let}\ \mathtt{f} = \lambda \mathtt{z}.\mathtt{z}\ \mathtt{in}\ \mathtt{f}\ \mathtt{f} : \mathsf{T}]\!]$. □

We now state several properties of constraint generation. We begin with soundness, whose statement was explained above.

1.4.4 THEOREM [SOUNDNESS]: $\mathsf{let}\ \Gamma\ \mathsf{in}\ [\![\mathtt{t} : \mathsf{T}]\!], \Gamma \vdash \mathtt{t} : \mathsf{T}$. □

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

1.4.5 LEMMA: $[\![\mathtt{t} : \mathsf{T}]\!] \wedge \mathsf{T} \leq \mathsf{T}' \Vdash [\![\mathtt{t} : \mathsf{T}']\!]$. □

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

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.

1.4.7 LEMMA: $[\![\lambda z.t : T_1 \to T_2]\!]$ is equivalent to $\mathsf{let}\ z : T_1\ \mathsf{in}\ [\![t : T_2]\!]$. □

We conclude with the completeness property. The theorem states that if, within HM(X), t has type T under assumptions C and Γ, then C must be at least as specific as $\mathsf{let}\ \Gamma\ \mathsf{in}\ [\![t : T]\!]$. 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 1.5.2.

1.4.8 THEOREM [COMPLETENESS]: Let $C \Vdash \exists\Gamma$. Assume $\mathit{fpi}(C,\Gamma) = \varnothing$. If $C, \Gamma \vdash t : T$ holds in HM(X), then C entails $\mathsf{let}\ \Gamma\ \mathsf{in}\ [\![t : T]\!]$. □

1.5 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 1.3.6 and 1.4.8 imply that type soundness also holds when well-typedness is defined with respect to the typing judgements of DM or HM(X). We establish type soundness by following Wright and Felleisen's so-called *syntactic approach* (1994). 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 is a version of this lemma, stated in terms of the constraint generation rules.

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

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.

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

1.5.3 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 $m \in dom(M)$ to $\mathsf{ref}\ M(m)$. Assuming $dom(\mu)$ and $dom(M)$ coincide, the constraint $[\![\mu : M]\!]$ is defined as the conjunction of the constraints $[\![\mu(m) : M(m)]\!]$, where m ranges over $dom(\mu)$. Under the same assumption, the constraint $[\![\mathtt{t}/\mu : \mathsf{T}/M]\!]$ is defined as $[\![\mathtt{t} : \mathsf{T}]\!] \wedge [\![\mu : M]\!]$. A configuration $\mathtt{t}/\mu$ is *well-typed* if and only if there exist a type T and a store type M such that $dom(\mu) = dom(M)$ and the constraint $\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![\mathtt{t}/\mu : \mathsf{T}/M]\!]$ is satisfiable. □

The type $\mathsf{ref}\ \mathsf{T}$ is the type of references (that is, memory locations) that store data of type T (TAPL Chapter 13). 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 $[\![\mathtt{t}/\mu : \mathsf{T}/M]\!]$ within the context $\mathsf{let}\ \mathsf{ref}\ M\ \mathsf{in}\ []$, we give meaning to free occurrences of memory locations within $[\![\mathtt{t}/\mu : \mathsf{T}/M]\!]$, 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 1.2.17), every well-typed configuration must be closed. The context $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ []$ gives meaning to occurrences of constants within $[\![\mathtt{t}/\mu : \mathsf{T}/M]\!]$.

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 T under a certain constraint, then its reduct $\mathtt{t}'$ has type T under the same constraint. In terms of generated constraints, this statement becomes: $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathtt{t} : \mathsf{T}]\!]$ entails $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathtt{t}' : \mathsf{T}]\!]$. 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 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:

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

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

The relation $\sqsubseteq$ is intended to express a connection between a configuration and its reduct. Thus, subject reduction may be stated as: $(\longrightarrow) \subseteq (\sqsubseteq)$, 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.

1.5.5 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:

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

This expression reduces to $\mathtt{true}\ \hat{+}\ \hat{1}$, so it must not be well-typed. Yet, if natural type schemes are assigned to $\mathtt{ref}$, $!$, and $:=$ (see Example 1.7.5), then it *is* well-typed with respect to the rules given so far, because $\mathtt{r}$ receives the polymorphic type scheme $\forall \mathsf{X}.\mathsf{ref}\ (\mathsf{X} \rightarrow \mathsf{X})$, which allows writing a function of type $\mathsf{int} \rightarrow \mathsf{int}$ into $\mathtt{r}$ and reading it back with type $\mathsf{bool} \rightarrow \mathsf{bool}$. The problem is that $\mathtt{let}$-polymorphism simulates a textual duplication of the $\mathtt{let}$-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).

1.5.6 DEFINITION: A program satisfies the *value restriction* if and only if all subexpressions of the form $\mathtt{let}\ \mathtt{z} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2$ are in fact of the form $\mathtt{let}\ \mathtt{z} = \mathtt{v}_1\ \mathtt{in}\ \mathtt{t}_2$. 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 $\mathtt{let}\ \mathtt{z} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2$ into $(\lambda\mathtt{z.t}_2)\ \mathtt{t}_1$ 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 $\mathtt{c}\ \mathtt{v}_1\ \ldots\ \mathtt{v}_k$, where $\mathtt{c}$ is a destructor of arity k. This is excessive, because many destructors have pure semantics; only a few, such as $\mathtt{ref}$, allocate new mutable storage. Furthermore, we use pure destructors to encode numerous language features (§1.7). 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 $\mathtt{c}\ \mathtt{t}_1\ \ldots\ \mathtt{t}_k$, where $\mathtt{c}$ 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 1.5.6) is summarized by the following result.

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

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

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

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

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

Let us now state the progress property.

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

We may now conclude:

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

Let us recall that this result holds only if the requirements of Definition 1.5.5 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 §1.7 and §1.8.

1.6 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.

1.6.1 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.

1.6.1 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 $=^n_\kappa\ \mathsf{T}_1 \ldots \mathsf{T}_n$ is written $\mathsf{T}_1 = \ldots = \mathsf{T}_n$, 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:

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

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

1.6.2 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 $\epsilon = \mathsf{T}_1 = \mathsf{T}_2$, 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 $\epsilon = \mathsf{T}_1$, together with a pending request to solve $\mathsf{T}_1 = \mathsf{T}_2$ 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 description, where state (standard conjunctions of multi-equations) and control (pending binary equations) are distinguished.

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

The specification of the unification algorithm consists of a set of constraint rewriting rules, given in Figure 1-10. 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.

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

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

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

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

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

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

$$T \quad \rightarrow \quad \text{true} \qquad \text{(S-Single)}$$

$$U \wedge \text{true} \quad \rightarrow \quad U \qquad \text{(S-True)}$$

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

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

Figure 1-10: Unification

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-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 1-10 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 $\exists \bar{\mathrm{X}}.\mathrm{U}$, where U is satisfiable. The latter two properties indicate that the algorithm is indeed a constraint solver.

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

1.6.5 LEMMA: $\mathrm{U}_1 \rightarrow \mathrm{U}_2$ implies $\mathrm{U}_1 \equiv \mathrm{U}_2$. □

1.6.6 LEMMA: Every normal form is either false or of the form $\mathcal{X}[\mathrm{U}]$, 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. □

1.6.2 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 U, as previously studied, and *stacks*, whose syntax is as follows:

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

In the second and fourth productions, C is an external constraint. In the last production, we require σ to be of the form $\forall \bar{\mathsf{X}}[\mathsf{U}].\mathsf{X}$, and we demand $\exists \sigma \equiv \mathsf{true}$. Every stack may be viewed as a one-hole constraint context (page 21): 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 $S; \mathsf{U}; \mathsf{C}$, where S is a stack, U is a unification constraint, and C is an external constraint. The state $S; \mathsf{U}; \mathsf{C}$ is to be understood as a representation of the constraint $S[\mathsf{U} \wedge \mathsf{C}]$, 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 $[]; \mathsf{true}; \mathsf{C}$, 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 $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ \mathsf{C}$, 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 $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ []; \mathsf{true}; \mathsf{C}$ 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 1-11. Rewriting is performed modulo α-conversion. S-UNIFY makes

$$
\begin{array}{rcll}
S; U; C & \rightarrow & S; U'; C & \text{(S-Unify)} \\
 & & \text{if } U \rightarrow U' & \\
S; \exists \bar{X}.U; C & \rightarrow & S[\exists \bar{X}.[]]; U; C & \text{(S-Ex-1)} \\
 & & \text{if } \bar{X} \# \mathit{ftv}(C) & \\
S[(\exists \bar{X}.S') \wedge D]; U; C & \rightarrow & S[\exists \bar{X}.(S' \wedge D)]; U; C & \text{(S-Ex-2)} \\
 & & \text{if } \bar{X} \# \mathit{ftv}(D) & \\
S[\mathsf{let}\ x : \forall \bar{X}[\exists \bar{Y}.S'].T\ \mathsf{in}\ D]; U; C & \rightarrow & S[\mathsf{let}\ x : \forall \bar{X}\bar{Y}[S].'T\ \mathsf{in}\ D]; U; C & \text{(S-Ex-3)} \\
 & & \text{if } \bar{Y} \# \mathit{ftv}(T) & \\
S[\mathsf{let}\ x : \sigma\ \mathsf{in}\ \exists \bar{X}.S']; U; C & \rightarrow & S[\exists \bar{X}.\mathsf{let}\ x : \sigma\ \mathsf{in}\ S']; U; C & \text{(S-Ex-4)} \\
 & & \text{if } \bar{X} \# \mathit{ftv}(\sigma) & \\
S; U; T_1 = T_2 & \rightarrow & S; U \wedge T_1 = T_2; \mathsf{true} & \text{(S-Solve-Eq)} \\
S; U; x \preceq T & \rightarrow & S; U; S(x) \preceq T & \text{(S-Solve-Id)} \\
S; U; C_1 \wedge C_2 & \rightarrow & S[[] \wedge C_2]; U; C_1 & \text{(S-Solve-And)} \\
S; U; \exists \bar{X}.C & \rightarrow & S[\exists \bar{X}.[]]; U; C & \text{(S-Solve-Ex)} \\
 & & \text{if } \bar{X} \# \mathit{ftv}(U) & \\
S; U; \mathsf{let}\ x : \forall \bar{X}[D].T\ \mathsf{in}\ C & \rightarrow & S[\mathsf{let}\ x : \forall \bar{X}[[]].T\ \mathsf{in}\ C]; U; D & \text{(S-Solve-Let)} \\
 & & \text{if } \bar{X} \# \mathit{ftv}(U) & \\
S[[] \wedge C]; U; \mathsf{true} & \rightarrow & S; U; C & \text{(S-Pop-And)} \\
S[\mathsf{let}\ x : \forall \bar{X}[[]].T\ \mathsf{in}\ C]; U; \mathsf{true} & \rightarrow & S[\mathsf{let}\ x : \forall \bar{X}X[[]].X\ \mathsf{in}\ C]; & \\
 & & U \wedge X = T; \mathsf{true} & \text{(S-Name-2)} \\
 & & \text{if } X \notin \mathit{ftv}(U, T) \wedge T \notin \mathcal{V} & \\
S[\mathsf{let}\ x : \forall \bar{X}Y[[]].X\ \mathsf{in}\ C]; Y = Z = \epsilon \wedge U; \mathsf{true} & \rightarrow & S[\mathsf{let}\ x : \forall \bar{X}Y[[]].\theta(X)\ \mathsf{in}\ C]; & \\
 & & Y \wedge Z = \theta(\epsilon) \wedge \theta(U); \mathsf{true} & \text{(S-Compress)} \\
 & & \text{if } Y \neq Z \wedge \theta = [Y \mapsto Z] & \\
S[\mathsf{let}\ x : \forall \bar{X}Y[[]].X\ \mathsf{in}\ C]; Y = \epsilon \wedge U; \mathsf{true} & \rightarrow & S[\mathsf{let}\ x : \forall \bar{X}[[]].X\ \mathsf{in}\ C]; \epsilon \wedge U; \mathsf{true} & \text{(S-UnName)} \\
 & & \text{if } Y \notin X \cup \mathit{ftv}(\epsilon, U) & \\
S[\mathsf{let}\ x : \forall \bar{X}\bar{Y}[[]].X\ \mathsf{in}\ C]; U; \mathsf{true} & \rightarrow & S[\exists \bar{Y}.\mathsf{let}\ x : \forall \bar{X}[[]].X\ \mathsf{in}\ C]; U; \mathsf{true} & \text{(S-LetAll)} \\
 & & \text{if } \bar{Y} \# \mathit{ftv}(C) \wedge \exists \bar{X}.U \text{ determines } \bar{Y} & \\
S[\mathsf{let}\ x : \forall \bar{X}[[]].X\ \mathsf{in}\ C]; U_1 \wedge U_2; \mathsf{true} & \rightarrow & S[\mathsf{let}\ x : \forall \bar{X}[U_2].X\ \mathsf{in}\ []]; U_1; C & \text{(S-Pop-Let)} \\
 & & \text{if } \bar{X} \# \mathit{ftv}(U_1) \wedge \exists \bar{X}.U_2 \equiv \mathsf{true} & \\
S[\mathsf{let}\ x : \sigma\ \mathsf{in}\ []]; U; \mathsf{true} & \rightarrow & S; U; \mathsf{true} & \text{(S-Pop-Env)}
\end{array}
$$

Figure 1-11: A constraint solver

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 $S; \mathsf{U}; \mathsf{C}$ is a normal form with respect to these five rules, then U must be either false or a conjunction of standard multi-equations, 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 $\mathsf{x} \preceq \mathsf{T}$ and replaces it with $\sigma \preceq \mathsf{T}$, where the type scheme $\sigma = S(\mathsf{x})$ is the type scheme carried by the nearest environment frame that defines x in the stack S. It is defined as follows:

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

If $\mathsf{x} \in dpi(S)$ does not hold, then $S(\mathsf{x})$ 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. Recall that, if σ is of the form $\forall \bar{\mathsf{X}}[\mathsf{U}].\mathsf{X}$, where $\bar{\mathsf{X}} \# ftv(\mathsf{T})$, then $\sigma \preceq \mathsf{T}$ stands for $\exists \bar{\mathsf{X}}.(\mathsf{U} \wedge \mathsf{X} = \mathsf{T})$. 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 $\mathsf{let}\ \mathsf{x} : \sigma\ \mathsf{in}\ []$ 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 $\forall\bar{\mathsf{X}}[\mathsf{U}].\mathsf{X}$, whereas the type scheme $\forall\bar{\mathsf{X}}[\mathsf{D}].\mathsf{T}$ 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 $S[\mathsf{let}\ \mathsf{x} : \forall\bar{\mathsf{X}}[[]].\mathsf{T}\ \mathsf{in}\ \mathsf{C}]; \mathsf{U}; \mathsf{true}$ into $S[\mathsf{let}\ \mathsf{x} : \forall\bar{\mathsf{X}}[\mathsf{U}].\mathsf{T}\ \mathsf{in}\ []]; \mathsf{true}; \mathsf{C}$, 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 1.6.7). 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 $\bar{\mathsf{X}} \# \mathit{ftv}(\mathsf{U}_1)$—and U_2 constrains *young* variables only—as expressed by the side-condition $\exists\bar{\mathsf{X}}.\mathsf{U}_2 \equiv \mathsf{true}$. Note that U_2 may still contain free occurrences of old type variables, so the type scheme $\forall\bar{\mathsf{X}}[\mathsf{U}_2].\mathsf{X}$ that appears on the right-hand side is not necessarily closed. It is not obvious why such a decomposition must exist; Lemma 1.6.10 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 $\mathsf{U}_1 \wedge \mathsf{U}_2$ 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* and corresponds to the difference between hmx-Gen′ and hmx-Gen discussed in §1.3. The unification constraint U_2, on the other hand, becomes part of the newly built type scheme $\forall\bar{\mathsf{X}}[\mathsf{U}_2].\mathsf{X}$. The property $\exists\bar{\mathsf{X}}.\mathsf{U}_2 \equiv \mathsf{true}$ guarantees that the newly created environment frame meets the requirements imposed on such frames. 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 $\forall\bar{\mathsf{X}}[\mathsf{U}_2].\mathsf{X}$, making it cheaper to instantiate.

To complete our description of the constraint solver, there remains to explain how to decide when $\exists\bar{\mathsf{X}}.\mathsf{U}$ determines $\bar{\mathsf{Y}}$, 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 $\bar{\mathsf{Y}}$ (Definition 1.2.13). In the first situation, the type variables $\bar{\mathsf{Y}}$ 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 $\bar{\mathsf{Y}}$ are determined by the value of X—they are subtrees of it at specific positions. For instance, $\mathsf{X} = \mathsf{Y}_1 \rightarrow \mathsf{Y}_2$ determines $\mathsf{Y}_1\mathsf{Y}_2$, while $\exists\mathsf{Y}_1.(\mathsf{X} = \mathsf{Y}_1 \rightarrow \mathsf{Y}_2)$ determines Y_2. In the second situation, the type variables $\bar{\mathsf{Y}}$ are equated with a term T, *all* of whose type variables are *free* in C. Again, the value of the type variables $\bar{\mathsf{Y}}$ is then determined by the values of the type variables $\mathit{ftv}(\mathsf{T})$. For instance, $\mathsf{X} = \mathsf{Y}_1 \rightarrow \mathsf{Y}_2$ determines X, while $\exists\mathsf{Y}_1.(\mathsf{X} = \mathsf{Y}_1 \rightarrow \mathsf{Y}_2)$ does not. In the second situation, no assumption is in fact made about the model. Note that $\mathsf{X} = \mathsf{Y}_1 \rightarrow \mathsf{Y}_2$ determines $\mathsf{Y}_1\mathsf{Y}_2$ and determines X, but does *not* simultaneously determine $\mathsf{X}\mathsf{Y}_1\mathsf{Y}_2$.

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

Thanks to Lemma 1.6.7, 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.

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.

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

1.6.8 LEMMA: The reduction system $\rightarrow$ is strongly normalizing. $\square$

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 $S[U \wedge C]$.

1.6.9 LEMMA: $S; U; C \rightarrow S'; U'; C'$ implies $S[U \wedge C] \equiv S'[U' \wedge C']$. $\square$

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

1.6.10 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. $\square$

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 1.2.16, 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 1.6.9 and 1.6.10 indeed prove that the algorithm is a constraint solver.

1.6.11 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). □

1.7 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. Then, we describe algebraic data type definitions. Last, the issues associated with recursive types are briefly discussed. Exceptions are not discussed; the reader is referred to (TAPL Chapter 14).

1.7.1 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 1.5.5 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.

1.7.1 Exercise [Recommended, ★]: Let Γ_0 contain the binding $\mathtt{c} : \forall\bar{X}.T_1 \rightarrow \ldots \rightarrow T_n \rightarrow T$. Prove that $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathtt{c}\ \mathtt{t}_1\ \ldots\ \mathtt{t}_n : T']\!]$ is equivalent to $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists\bar{X}.(\bigwedge_{i=1}^{n}[\![\mathtt{t}_i : T_i]\!] \wedge T \leq T')$. □

1.7.2 Exercise [Integers, Recommended, ★★]: Integer literals and integer addition have been introduced and given an operational semantics in Examples 1.1.1, 1.1.2 and 1.1.4. 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 $\hat{+} : \mathsf{int} \rightarrow \mathsf{int} \rightarrow \mathsf{int}$. Check that these definitions meet the requirements of Definition 1.5.5. □

1.7.3 Exercise [Pairs, ★★, ↛]: Pairs and pair projections have been introduced and given an operational semantics in Examples 1.1.3 and 1.1.5. 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:

$$\begin{array}{rl} (\cdot,\cdot) : & \forall XY.X \rightarrow Y \rightarrow X \times Y \\ \pi_1 : & \forall XY.X \times Y \rightarrow X \\ \pi_2 : & \forall XY.X \times Y \rightarrow Y \end{array}$$

Check that these definitions meet the requirements of Definition 1.5.5. □

1.7.4 Exercise [Sums, ★★, ↛]: Sums have been introduced and given an operational semantics in Example 1.1.7. 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:

$$\begin{array}{rl} \mathtt{inj}_1 : & \forall XY.X \rightarrow X + Y \\ \mathtt{inj}_2 : & \forall XY.Y \rightarrow X + Y \\ \mathtt{case} : & \forall XYZ.(X + Y) \rightarrow (X \rightarrow Z) \rightarrow (Y \rightarrow Z) \rightarrow Z \end{array}$$

Check that these definitions meet the requirements of Definition 1.5.5. □

1.7.5 Exercise [References, ★★★]: References have been introduced and given an operational semantics in Example 1.1.9. The type constructor ref has been introduced in Definition 1.5.3. Let us now extend the initial environment Γ_0 with the following bindings:

$$\begin{array}{rl} \mathtt{ref} : & \forall X.X \rightarrow \mathsf{ref}\ X \\ ! : & \forall X.\mathsf{ref}\ X \rightarrow X \\ := : & \forall X.\mathsf{ref}\ X \rightarrow X \rightarrow X \end{array}$$

Check that these definitions meet the requirements of Definition 1.5.5. □

1.7.6 EXERCISE [RECURSION, RECOMMENDED, ★★★, ↛]: The fixpoint combinator `fix` has been introduced and given an operational semantics in Example 1.1.10. Let us now extend the initial environment Γ_0 with the following binding:

$$\mathtt{fix} : \quad \forall \mathsf{XY}.((\mathsf{X} \to \mathsf{Y}) \to (\mathsf{X} \to \mathsf{Y})) \to \mathsf{X} \to \mathsf{Y}$$

Check that these definitions meet the requirements of Definition 1.5.5. Recall how the `letrec` syntactic sugar was defined in Example 1.1.10, and check that this gives rise to the following constraint generation rule:

$$\begin{array}{ll} & \text{let } \Gamma_0 \text{ in } [\![\mathtt{letrec}\ \mathtt{f} = \lambda \mathtt{z}.\mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathsf{T}]\!] \\ \equiv & \text{let } \Gamma_0 \text{ in let } \mathtt{f} : \forall \mathsf{XY}[\text{let } \mathtt{f} : \mathsf{X} \to \mathsf{Y}; \mathtt{z} : \mathsf{X} \text{ in } [\![\mathtt{t}_1 : \mathsf{Y}]\!]].\mathsf{X} \to \mathsf{Y} \text{ in } [\![\mathtt{t}_2 : \mathsf{T}]\!] \end{array}$$

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 $\mathtt{t}_1$ to be assigned the *monomorphic* type $\mathsf{X} \to \mathsf{Y}$. This type is generalized and turned into a type scheme before inspecting $\mathtt{t}_2$, however, so every occurrence of `f` within $\mathtt{t}_2$ may receive a different type, as usual with `let`-polymorphism. A more powerful way of typechecking recursive function definitions, proposed by (Mycroft, 1984) and known as *polymorphic recursion*, allows the types of occurrences of `f` within $\mathtt{t}_1$ to be instances of a valid type scheme for $\mathtt{t}_1$. However, type inference for this extension is equivalent to semi-unification (Henglein, 1993), which has been proved undecidable (Kfoury, Tiuryn, and Urzyczyn, 1993). Hence, type inference must rely on a semi-algorithm and is thus necessarily incomplete. □

1.7.2 Algebraic data types

Exercises 1.7.3 and 1.7.4 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 "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 $|\,\mathsf{T}\,|$ is a function of T. More precisely, up to a constant factor, we have $|\,\mathsf{int}\,| = |\,\mathsf{bool}\,| = 1$, $|\,\mathsf{T}_1 \times \mathsf{T}_2\,| = 1 + |\,\mathsf{T}_1\,| + |\,\mathsf{T}_2\,|$, and $|\,\mathsf{T}_1 + \mathsf{T}_2\,| = 1 + \max(|\,\mathsf{T}_1\,|, |\,\mathsf{T}_2\,|)$. The following example describes another facet of the same problem.

1.7.7 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:

$$\texttt{letrec length} = \lambda \texttt{l}.\texttt{case l}\ (\lambda\_.\hat{0})\ (\lambda \texttt{z}.\hat{1}\ \hat{+}\ \texttt{length}\ (\pi_2\ \texttt{z}))$$

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 $\mathsf{X} = \mathsf{Y} + (\mathsf{Z} \times \mathsf{X})$, 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 (§1.7.3). □

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

$$\mathsf{D}\,\vec{\mathsf{X}} \approx \sum_{i=1}^{k} \ell_i : \mathsf{T}_i \qquad \text{and} \qquad \mathsf{D}\,\vec{\mathsf{X}} \approx \prod_{i=1}^{k} \ell_i : \mathsf{T}_i.$$

In either case, k must be nonnegative. If D has signature $\vec{\kappa} \Rightarrow \star$, then the type variables $\vec{\mathsf{X}}$ must have kind $\vec{\kappa}$. Every T_i must have kind $\star$. We refer to $\bar{\mathsf{X}}$ as the *parameters* and to $\vec{\mathsf{T}}$ (the vector formed by $\mathsf{T}_1, \ldots, \mathsf{T}_k$) as the *components* of the definition. The parameters are bound within the components, and the definition must be closed, that is, $ftv(\vec{\mathsf{T}}) \subseteq \bar{\mathsf{X}}$ must hold. Last, for an algebraic data type definition to be valid, the behavior of the type constructor D with respect to subtyping must match its definition. This requirement is clarified below.

1.7.8 DEFINITION: Consider an algebraic data type definition whose parameters and components are respectively $\vec{\mathsf{X}}$ and $\vec{\mathsf{T}}$. Let $\vec{\mathsf{X}}'$ and $\vec{\mathsf{T}}'$ be their images under an arbitrary renaming. Then, $\mathsf{D}\,\vec{\mathsf{X}} \leq \mathsf{D}\,\vec{\mathsf{X}}' \Vdash \vec{\mathsf{T}} \leq \vec{\mathsf{T}}'$ 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 $\mathsf{D}\,\vec{\mathsf{X}}$ is declared to be isomorphic to (a sum or a product of) $\vec{\mathsf{T}}$, whenever two types built with 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 D *invariant* in all of its parameters. Indeed, in that case, $\mathsf{D}\,\vec{\mathsf{X}} \leq \mathsf{D}\,\vec{\mathsf{X}}'$ entails $\vec{\mathsf{X}} = \vec{\mathsf{X}}'$, which must entail $\vec{\mathsf{T}} = \vec{\mathsf{T}}'$ since $\vec{\mathsf{T}}'$ is precisely $[\vec{\mathsf{X}} \mapsto \vec{\mathsf{X}}']\vec{\mathsf{T}}$. 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 1.7.8 while assigning D a less restrictive variance. The following example illustrates such a case.

1.7.9 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:

$$\mathsf{list}\,\mathsf{X} \approx \Sigma\,(\mathtt{Nil} : \mathsf{unit}; \mathtt{Cons} : \mathsf{X} \times \mathsf{list}\,\mathsf{X})$$

Because data types form a subset of type constructors, it is valid to form the type $\mathsf{list}\,\mathsf{X}$ 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 $\mathsf{list}\,\mathsf{X}$ is *not* a recursive type: it is nothing but an application of the unary type constructor list to the type variable X. To check that the definition of list satisfies the requirement of Definition 1.7.8, we must ensure that

$$\mathsf{list}\,\mathsf{X} \leq \mathsf{list}\,\mathsf{X}' \Vdash \mathsf{unit} \leq \mathsf{unit} \wedge \mathsf{X} \times \mathsf{list}\,\mathsf{X} \leq \mathsf{X}' \times \mathsf{list}\,\mathsf{X}'$$

holds. This assertion is equivalent to $\mathsf{list}\,\mathsf{X} \leq \mathsf{list}\,\mathsf{X}' \Vdash \mathsf{X} \leq \mathsf{X}'$. To satisfy the requirement, it is sufficient to make list a *covariant* type constructor, that is, to define subtyping in the model so that $\mathsf{list}\,\mathsf{X} \leq \mathsf{list}\,\mathsf{X}' \equiv \mathsf{X} \leq \mathsf{X}'$ holds.

Let tree be a data type of signature $\star \Rightarrow \star$. Let $\mathtt{root}$ and $\mathtt{sons}$ be record labels. Then, the following is a definition of tree as a record type:

$$\mathsf{tree}\,\mathsf{X} \approx \Pi\,(\mathtt{root} : \mathsf{X}; \mathtt{sons} : \mathsf{list}\,(\mathsf{tree}\,\mathsf{X}))$$

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. □

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 1.1.7. A record type definition extends it with a record formation construct and several projections, as in Examples 1.1.3 and 1.1.5. 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 $\mathsf{D}\,\vec{\mathsf{X}} \approx \sum_{i=1}^{k} \ell_i : \mathsf{T}_i$. 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 $\mathtt{case}_{\mathsf{D}}$, is introduced. When

$\mathsf{k} > 0$, it is common to write $\mathtt{case}\ \mathtt{t}\ [\ell_i : \mathtt{t}_i]_{i=1}^{k}$ for the application $\mathtt{case}_\mathtt{D}\ \mathtt{t}$ $\mathtt{t}_1\ \ldots\ \mathtt{t}_n$. The operational semantics is extended with the following reduction rules, for $i \in \{1, \ldots, k\}$:

$$\mathtt{case}\ (\ell_i\ \mathsf{v})\ [\ell_j : \mathsf{v}_j]_{j=1}^{k} \xrightarrow{\delta} \mathsf{v}_i\ \mathsf{v} \qquad \text{(R-ALG-CASE)}$$

For each $i \in \{1, \ldots, k\}$, the initial environment is extended with the binding $\ell_i : \forall \bar{\mathsf{X}}.\mathsf{T}_i \to \mathsf{D}\,\vec{\mathsf{X}}$. It is further extended with the binding $\mathtt{case}_\mathtt{D} : \forall \bar{\mathsf{X}}\mathsf{Z}.\mathsf{D}\,\vec{\mathsf{X}} \to (\mathsf{T}_1 \to \mathsf{Z}) \to \ldots (\mathsf{T}_k \to \mathsf{Z}) \to \mathsf{Z}$.

Now, suppose the prologue contains the definition $\mathsf{D}\,\vec{\mathsf{X}} \approx \prod_{i=1}^{k} \ell_i : \mathsf{T}_i$. 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 $\mathtt{make}_\mathtt{D}$, 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 $\mathtt{make}_\mathtt{D}\ \mathtt{t}_1\ \ldots\ \mathtt{t}_k$. The operational semantics is extended with the following reduction rules, for $i \in \{1, \ldots, k\}$:

$$(\{\ell_j = \mathsf{v}_j\}_{j=1}^{k}).\ell_i \xrightarrow{\delta} \mathsf{v}_i \qquad \text{(R-ALG-PROJ)}$$

For each $i \in \{1, \ldots, k\}$, the initial environment is extended with the binding $\ell_i : \forall \bar{\mathsf{X}}.\mathsf{D}\,\vec{\mathsf{X}} \to \mathsf{T}_i$. It is further extended with the binding $\mathtt{make}_\mathtt{D} : \forall \bar{\mathsf{X}}.\mathsf{T}_1 \to \ldots \to \mathsf{T}_k \to \mathsf{D}\,\vec{\mathsf{X}}$.

1.7.10 EXAMPLE: The effect of defining list (Example 1.7.9) is to make $\mathtt{Nil}$ and $\mathtt{Cons}$ data constructors of arity 1 and to introduce a binary destructor $\mathtt{case}_{\mathsf{list}}$. The definition also extends the initial environment as follows:

$$\begin{array}{rl}
\mathtt{Nil} : & \forall \mathsf{X}.\mathsf{unit} \to \mathsf{list}\,\mathsf{X} \\
\mathtt{Cons} : & \forall \mathsf{X}.\mathsf{X} \times \mathsf{list}\,\mathsf{X} \to \mathsf{list}\,\mathsf{X} \\
\mathtt{case}_{\mathsf{list}} : & \forall \mathsf{XZ}.\mathsf{list}\,\mathsf{X} \to (\mathsf{unit} \to \mathsf{Z}) \to (\mathsf{X} \times \mathsf{list}\,\mathsf{X} \to \mathsf{Z}) \to \mathsf{Z}
\end{array}$$

Thus, the value $\mathtt{Cons}(\hat{0}, \mathtt{Nil}())$, an integer list of length 1, has type $\mathsf{list}\,\mathsf{int}$. A function that computes the length of a list may now be written as follows:

$$\mathtt{letrec}\ \mathtt{length} = \lambda \mathtt{l}.\mathtt{case}\ \mathtt{l}\ [\mathtt{Nil} : \lambda\_.\hat{0} \mid \mathtt{Cons} : \lambda \mathtt{z}.\hat{1}\ \hat{+}\ \mathtt{length}\ (\pi_2\ \mathtt{z})]$$

Recall that this notation is syntactic sugar for

$$\mathtt{letrec}\ \mathtt{length} = \lambda \mathtt{l}.\mathtt{case}_{\mathsf{list}}\ \mathtt{l}\ (\lambda\_.\hat{0})\ (\lambda \mathtt{z}.\hat{1}\ \hat{+}\ \mathtt{length}\ (\pi_2\ \mathtt{z}))$$

The difference with the code in Example 1.7.7 appears minimal: the $\mathtt{case}$ 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 $\mathtt{case}$ construct, given in Exercise 1.7.4. This is good for a couple of reasons. First, the former is more informative than the latter, because it contains the type T_i associated with the data constructor ℓ_i. Here, for instance, the generated constraint requires the type of `z` to be $\mathsf{X} \times \mathsf{list}\,\mathsf{X}$ 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 1.7.7, 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 $\mathsf{list}\,\mathsf{X}$ 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 $\forall \mathsf{X}.\mathsf{list}\,\mathsf{X} \to \mathsf{int}$, as expected. □

Example 1.7.10 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 $\forall \mathsf{X}_1 \ldots \mathsf{X}_k.\mathsf{X}_i \to \mathsf{X}_1 + \ldots + \mathsf{X}_k$, or, more concisely, $\forall \mathsf{X}_1 \ldots \mathsf{X}_k.\mathsf{X}_i \to \sum_{i=1}^{k} \mathsf{X}_i$. By instantiating each X_i with T_i and generalizing again, we find that a more specific type scheme is $\forall \bar{\mathsf{X}}.\mathsf{T}_i \to \sum_{i=1}^{k} \mathsf{T}_i$. Perhaps this could have been the type scheme assigned to ℓ_i? Instead, however, it is $\forall \bar{\mathsf{X}}.\mathsf{T}_i \to \mathsf{D}\,\vec{\mathsf{X}}$. 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 $\sum_{i=1}^{k} \mathsf{T}_i$—which forms the definition's right-hand side—into the parameterized abstract type $\mathsf{D}\,\vec{\mathsf{X}}$—which is the definition's left-hand side. Conversely, the type scheme assigned to $\mathtt{case}_{\mathsf{D}}$ *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.

1.7.11 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. □

1.7.12 EXERCISE [★★★, ↛]: Check that the above definitions meet the requirements of Definition 1.5.5. □

1.7.13 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 $\mathsf{D}\,\vec{\mathsf{X}} \approx \sum_{i=1}^{k} \ell_i : \vec{\mathsf{T}}_i$, where the arity of the data constructor ℓ_i is the length of the vector $\vec{\mathsf{T}}_i$, 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 1.5.5 are still met. □

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 §1.8.

1.7.3 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 $\mathsf{X} = \mathsf{X} \rightarrow \mathsf{X}$, are then satisfiable. Our theorems about type inference and type soundness, which are independent of the model, remain valid. The constraint solver described in §1.6 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.

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 $\mu\mathsf{X}.\mathsf{X} \rightarrow \mathsf{X}$ corresponds, in our constraint-based approach, to the type scheme $\forall\mathsf{X}[\mathsf{X} = \mathsf{X} \rightarrow \mathsf{X}].\mathsf{X}$.

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 $\forall \mathsf{XYZ}[\mathsf{Y} = \mathsf{list}\,\mathsf{Y} \wedge \mathsf{Z} = \mathsf{list}\,\mathsf{Z}].\mathsf{X} \rightarrow \mathsf{Y} \rightarrow \mathsf{Z}$. In the absence of recursive types, it is ill-typed, since the constraint $\mathsf{Y} = \mathsf{list}\,\mathsf{Y} \wedge \mathsf{Z} = \mathsf{list}\,\mathsf{Z}$ 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.

1.8 Rows

In §1.7, 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 T in the definition of the simple record type $\mathsf{D}\,\vec{\mathsf{X}}$, then the associated record access operation has type $\forall \vec{\mathsf{X}}.\mathsf{D}\,\vec{\mathsf{X}} \rightarrow \mathsf{T}$. If ℓ appears with type T' in the definition of another simple record type, say $\mathsf{D}'\,\vec{\mathsf{X}}'$, then the associated record access operation has type $\forall \vec{\mathsf{X}}'.\mathsf{D}'\,\vec{\mathsf{X}}' \rightarrow \mathsf{T}'$; and so on. The most precise type scheme that subsumes all of these incomparable type schemes is $\forall XY.X \to Y$. 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 $D\,\vec{X}$, 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 $T_1 \times T_2$ is the (undeclared) type of pairs whose first component has type T_1 and whose second component has type 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.

1.8.1 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 $''\ T_a\ T_b\ T_c$ represents all records where the field ℓ_a (resp. ℓ_b, ℓ_c) contains a value of type T_a (resp. T_b, T_c). 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:

$$\begin{aligned} \{\cdot\} &: \quad \forall \mathsf{X}.\mathsf{X} \rightarrow {}^{\prime\prime}\, \mathsf{X}\ \mathsf{X}\ \mathsf{X} \\ \{\cdot \text{ with } \ell_b = \cdot\} &: \quad \forall \mathsf{X}_a \mathsf{X}_b \mathsf{X}'_b \mathsf{X}_c. {}^{\prime\prime}\, \mathsf{X}_a\ \mathsf{X}_b\ \mathsf{X}_c \rightarrow \mathsf{X}'_b \rightarrow {}^{\prime\prime}\, \mathsf{X}_a\ \mathsf{X}'_b\ \mathsf{X}_c \\ \cdot.\{\ell_b\} &: \quad \forall \mathsf{X}_a \mathsf{X}_b \mathsf{X}_c. {}^{\prime\prime}\, \mathsf{X}_a\ \mathsf{X}_b\ \mathsf{X}_c \rightarrow \mathsf{X}_b \end{aligned}$$

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 $^{\prime\prime}$ 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 $\mathsf{option}\,\mathsf{X} \approx \mathsf{pre}\ \mathsf{X} + \mathsf{abs}$. We use pre for *present* and abs for *absent*: indeed, a field that is defined with value v is encoded as a field with value $\mathsf{pre}\ \mathsf{v}$, while an undefined field is encoded as a field with value abs. Thus, an arbitrary record whose fields, *if present*, have types T_a, T_b, and T_c, respectively, may be encoded as a full record of type $^{\prime\prime}\ (\mathsf{option}\ \mathsf{T}_a)\ (\mathsf{option}\ \mathsf{T}_b)\ (\mathsf{option}\ \mathsf{T}_c)$. 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 $\mathsf{pre}\,\mathsf{X} \approx \mathsf{pre}\ \mathsf{X}$ 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 $^{\prime\prime}\ \mathsf{abs}\ (\mathsf{pre}\ \mathsf{T}_b)\ (\mathsf{pre}\ \mathsf{T}_c)$ 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 $\forall \mathsf{X}.\mathsf{pre}\ \mathsf{X} \rightarrow \mathsf{X}$, 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) $\langle \cdot \text{ with } \ell_b = \cdot \rangle$ is defined as

$\lambda \mathsf{r}.\lambda \mathsf{z}.\{\mathsf{r} \text{ with } \ell_b = \mathsf{pre}\ \mathsf{z}\}$. *Access* at a particular field (say, ℓ_b) $\cdot.\langle \ell_b \rangle$ is defined as $\lambda \mathsf{z}.\mathsf{pre}^{-1}\mathsf{z}.\{\ell_b\}$. It is straightforward to check that these operations have the following principal type schemes:

$$\begin{array}{rl}
\langle\rangle : & ″\ \mathsf{abs}\ \mathsf{abs}\ \mathsf{abs} \\
\langle \cdot \text{ with } \ell_b = \cdot \rangle : & \forall \mathsf{X}_a \mathsf{X}_b \mathsf{X}'_b \mathsf{X}_c.″\ \mathsf{X}_a\ \mathsf{X}_b\ \mathsf{X}_c \rightarrow \mathsf{X}'_b \rightarrow ″\ \mathsf{X}_a\ (\mathsf{pre}\ \ \mathsf{X}'_b)\ \mathsf{X}_c \\
\cdot.\langle \ell_b \rangle : & \forall \mathsf{X}_a \mathsf{X}_b \mathsf{X}_c.″\ \mathsf{X}_a\ (\mathsf{pre}\ \ \mathsf{X}_b)\ \mathsf{X}_c \rightarrow \mathsf{X}_b
\end{array}$$

It is important to notice that the type schemes associated with extension and access at ℓ_b are polymorphic in X_a and 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 X_b, which means that it is insensitive to the presence or absence of the field ℓ_b in its argument. The subterm $\mathsf{pre}\ \mathsf{X}'_b$ in its result type reflects the fact that ℓ_b is defined in the extended record. Conversely, the subterm $\mathsf{pre}\ \ \mathsf{X}_b$ 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: $\mathsf{pre}\colon \star \Rightarrow \circ$, $\mathsf{abs} : \circ$, and $″ : \circ \otimes \circ \otimes \circ \Rightarrow \star$.

1.8.1 Exercise [Recommended, ★, ↛]: Check that the three type schemes given above are well-kinded. What is the kind of each type variable? □

1.8.2 Exercise [Recommended, ★★, ↛]: Our ″ 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 $\{\ell_a : \mathsf{T}_a\}$; a record that defines the fields ℓ_b and ℓ_c only would have a type of the form $\{\ell_b : \mathsf{T}_b; \ell_c : \mathsf{T}_c\}$; 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? □

1.8.2 Records with infinite carrier

The treatment of records described in §1.8.1 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, $\{\{\texttt{false}\}\ \texttt{with}\ \ell = 1\}$ 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 §1.8.1, only in the setting of an infinite number of fields.

If we were to continue as in §1.8.1, we would now introduce a type constructor $\ddot{}$, *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, $\partial\mathsf{T}$ may be a *complete or incomplete* row. Second, $(\ell : \mathsf{T}\ ;\ \mathsf{T}')$ is defined only if ℓ is not in the domain of the row T', so this construction is strict extension, not update. These aspects are made clear by a *kinding discipline*, to be introduced later on (§1.8.3).

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 ″ a *unary* type constructor, *whose parameter is a row*. Then, (say) $\text{″}\ (\ell : \mathsf{int}\ ;\ \partial\mathsf{bool})$ is a record type, and we intend it to be a valid type for the record $\{\{\mathtt{false}\}\ \mathsf{with}\ \ell = 1\}$. The basic operations on records may be assigned the following type schemes:

$$\begin{aligned}
\{\cdot\} &: \quad \forall \mathsf{X}.\mathsf{X} \rightarrow \text{″}\,(\partial\mathsf{X}) \\
\{\cdot\ \mathsf{with}\ \ell = \cdot\} &: \quad \forall \mathsf{X}\mathsf{X}'\mathsf{Y}.\text{″}\,(\ell : \mathsf{X}\ ;\ \mathsf{Y}) \rightarrow \mathsf{X}' \rightarrow \text{″}\,(\ell : \mathsf{X}'\ ;\ \mathsf{Y}) \\
\cdot.\{\ell\} &: \quad \forall \mathsf{X}\mathsf{Y}.\text{″}\,(\ell : \mathsf{X}\ ;\ \mathsf{Y}) \rightarrow \mathsf{X}
\end{aligned}$$

These type schemes are reminiscent of those given in §1.8.1. 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 $\partial\mathsf{X}$ is used to indicate that all fields have the same type in the newly created record. In the next two type schemes, the row $(\ell : \mathsf{X}_\ell\ ;\ \mathsf{X})$ is used to separate the type X_ℓ, which describes the contents of the field ℓ, and the row X, which collectively describes the contents of all other fields. Here, the type variable 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 1.8.25. 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 1.8.30. 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.

1.8.3 Syntax

In the following, the set of labels $\mathcal{L}$ is considered denumerable. We let L range over finite subsets of $\mathcal{L}$. When $\ell \notin \mathsf{L}$ holds, we write $\ell.\mathsf{L}$ for $\{\ell\} \uplus \mathsf{L}$. Before explaining how the syntax of types is enriched with rows, we introduce *row kinds*, whose grammar is as follows:

$$s ::= \mathit{Type} \mid \mathit{Row}(\mathsf{L})$$

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 $\Pi\,(\ell:\mathsf{int}\,;\,\partial\mathsf{bool})$, 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 $\mathit{Row}(\mathsf{L})$ denotes a family of types whose domain is $\mathcal{L} \setminus \mathsf{L}$. 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 $\Pi\,(\mathsf{int})$, which makes no sense because the argument to the record type constructor Π should be a (complete) row, or $(\ell : \mathsf{T}_1\;;\;\ell : \mathsf{T}_2\;;\partial\mathsf{bool})$, 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 1.1.14), 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 Π. In a type system equipped with extensible variant types or with object types, there might be several such type constructors; see §1.8.8 and §1.8.9. 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 $\mathsf{K}.s$ for the mapping $(\mathsf{d} \mapsto \mathsf{K}(\mathsf{d}).s)^{\mathsf{d} \in dom(\mathsf{K})}$ and $(\mathsf{K} \Rightarrow \kappa).s$ for the (composite) kind signature $\mathsf{K}.s \Rightarrow \kappa.s$. We use symmetric notations to build a composite kind signature out of a basic kind and a row kind signature.

1.8.3 DEFINITION: The signature $\mathcal{S}$ is defined as follows:

$\mathsf{F} \in dom(\mathcal{S})$	Signature	Conditions
G^s	$(\mathsf{K} \Rightarrow \kappa).s$	$(\mathsf{G} : \mathsf{K} \Rightarrow \kappa) \in \mathcal{S}_0$
H	$\mathsf{K}.Row(\varnothing) \Rightarrow \kappa.Type$	$(\mathsf{H} : \mathsf{K} \Rightarrow \kappa) \in \mathcal{S}_1$
$\partial^{\kappa,\mathsf{L}}$	$\kappa.(Type \Rightarrow Row(\mathsf{L}))$	
$\ell^{\kappa,\mathsf{L}}$	$\kappa.(Type \otimes Row(\ell.\mathsf{L}) \Rightarrow Row(\mathsf{L}))$	$\ell \notin \mathsf{L}$

We sometimes refer to $\mathcal{S}$ as the *row extension of* $\mathcal{S}_0$ *with* $\mathcal{S}_1$. □

Examples 1.8.7 and 1.8.8 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 $\mathsf{G}^{\mathit{Type}}$, 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 $\rightarrow^{\mathit{Row}(\mathsf{L})}$, whose signature is $\{\mathit{domain} \mapsto \star.\mathit{Row}(\mathsf{L}), \mathit{codomain} \mapsto \star.\mathit{Row}(\mathsf{L})\} \Rightarrow \star.\mathit{Row}(\mathsf{L})$. Thus, $\rightarrow^{\mathit{Row}(\mathsf{L})}$ is a binary type constructor whose parameters and result must have basic kind $\star$ and must have row kind $\mathit{Row}(\mathsf{L})$. 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 $\rightarrow^{\mathit{Row}(\mathsf{L})}$ maps two families of types to a family of arrow types. This is made precise in §1.8.4. One should point out that the type constructors G^s, with $s \neq \mathit{Type}$, are required only in some advanced applications of rows; Examples 1.8.28 and 1.8.39 provide illustrations. They are not used when assigning types to the usual primitive operations on records, namely creation, update, and access (Examples 1.8.25 and 1.8.30).

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 ″ $(\mathsf{int}^{\mathit{Type}})$ is invalid: indeed, $\mathsf{int}^{\mathit{Type}}$ has row kind *Type*, while ″ 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 §1.8.2. First, for every κ and L, there is a *constant row constructor* $\partial^{\kappa,\mathsf{L}}$. Its parameter must have row kind *Type*, while its result has row kind $\mathit{Row}(\mathsf{L})$: 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 ″ $(\partial^{\star,\varnothing}\,\mathsf{bool})$, while $\partial^{\star,\{\ell\}}\,\mathsf{bool}$ is an incomplete row, and may be used *e.g.* to build the type ″ $(\ell : \mathsf{int}\,;\, \partial^{\star,\{\ell\}}\,\mathsf{bool})$. Second, for every κ, L, and $\ell \notin \mathsf{L}$, there is a *row extension constructor* $\ell^{\kappa,\mathsf{L}}$. We usually write $\ell^{\kappa,\mathsf{L}} : \mathsf{T}_1 \,;\, \mathsf{T}_2$ for $\ell^{\kappa,\mathsf{L}}\ \mathsf{T}_1\ \mathsf{T}_2$ 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 T_2 has row kind $\mathit{Row}(\ell.\mathsf{L})$, then $\ell^{\kappa,\mathsf{L}} : \mathsf{T}_1 \ ;\ \mathsf{T}_2$ has row kind $\mathit{Row}(\mathsf{L})$. Thanks to this design choice, the type $(\ell^{\star,\mathsf{L}} : \mathsf{T}_1 \ ;\ \ell^{\star,\mathsf{L}} : \mathsf{T}_2 \,;\, \partial^{\star,\ell.\mathsf{L}}\,\mathsf{bool})$ is invalid: indeed, the outer ℓ expects a parameter of row kind $\mathit{Row}(\ell.\mathsf{L})$, while the inner ℓ produces a type of row kind $\mathit{Row}(\mathsf{L})$.

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 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.

1.8.4 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 ˝, whose signature is $\star \Rightarrow \star$.

The unannotated type $\mathsf{X} \rightarrow \text{˝}(\mathsf{X})$ is invalid. Indeed, because ˝'s image row kind is *Type*, the arrow must be $\rightarrow^{\mathit{Type}}$. Thus, the leftmost occurrence of X must have row kind *Type*. On the other hand, because ˝ expects a parameter of row kind $\mathit{Row}(\varnothing)$, its rightmost occurrence must have row kind $\mathit{Row}(\varnothing)$—a contradiction. The unannotated type $\mathsf{X} \rightarrow \text{˝}(\partial\mathsf{X})$ is, however, valid, provided X has kind $\star.\mathit{Type}$. In fact, it is the type of the primitive record creation operation (§1.8.2).

The unannotated type $(\ell : \mathsf{T}\ ;\ \ell : \mathsf{T}\ ;\ \mathsf{T}'')$ is also invalid: there is no way of reconstructing the missing superscripts so as to make it valid. Indeed, the row $(\ell : \mathsf{T}'\ ;\ \mathsf{T}'')$ must have row kind $\mathit{Row}(\mathsf{L})$ for some L that does not contain ℓ. However, the context where it occurs requires it to also have row kind $\mathit{Row}(\mathsf{L})$ for some L that does contain ℓ. This makes it impossible to reconstruct consistent superscripts.

Any type of the form $\text{˝}(\text{˝}(\mathsf{T}))$ is invalid, because the outer ˝ expects a parameter of row kind $\mathit{Row}(\varnothing)$, while the inner ˝ 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. □

1.8.5 EXERCISE [RECOMMENDED, ★]: Consider the unannotated type

$$\mathsf{X} \rightarrow \text{˝}(\ell : \mathsf{int}\ ;\ (\mathsf{Y} \rightarrow \partial\mathsf{X})).$$

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 $\star.\mathit{Type}$? □

1.8.6 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 $\mathcal{S}_1$.

1.8.7 EXAMPLE: Assume that there is a single basic kind $\star$, that $\mathcal{S}_0$ consists of the arrow type constructor $\rightarrow$, whose signature is $\star \otimes \star \Rightarrow \star$, and that $\mathcal{S}_1$ consists of the record type constructor Π, whose signature is $\star \Rightarrow \star$. Then, the composite kinds are $\star.\mathit{Type}$ and $\star.\mathit{Row}(\mathsf{L})$, 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:

$$\begin{array}{rcl} \mathsf{T} & ::= & \mathsf{X} \mid \mathsf{T} \rightarrow \mathsf{T} \mid \Pi\, \mathsf{R} \\ \mathsf{R} & ::= & \mathsf{X} \mid \mathsf{R} \rightarrow \mathsf{R} \mid (\ell : \mathsf{T} ; \mathsf{R}) \mid \partial \mathsf{T} \end{array}$$

Ordinary types T include ordinary type variables (that is, type variables of kind $\star.\mathit{Type}$), arrow types (where the type constructor $\rightarrow$ is really $\rightarrow^{\mathit{Type}}$), and record types, which are formed by applying the record type constructor Π to a row. Rows R include row variables (that is, type variables of kind $\star.\mathit{Row}(\mathsf{L})$ for some L), arrow rows (where the row constructor $\rightarrow$ is really $\rightarrow^{\mathit{Row}(\mathsf{L})}$ 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. □

1.8.8 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 Π, whose signature is $\circ \Rightarrow \star$. Then, the composite kinds are $\star.\mathit{Type}$, $\star.\mathit{Row}(\mathsf{L})$, $\circ.\mathit{Type}$, and $\circ.\mathit{Row}(\mathsf{L})$, 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:

$$\begin{array}{rcl} \mathsf{T} & ::= & \mathsf{X} \mid \mathsf{T} \rightarrow \mathsf{T} \mid \Pi\, \mathsf{FR} \\ \mathsf{R} & ::= & \mathsf{X} \mid \mathsf{R} \rightarrow \mathsf{R} \mid (\ell : \mathsf{T} ; \mathsf{R}) \mid \partial \mathsf{T} \\ \mathsf{FT} & ::= & \mathsf{X} \mid \mathsf{abs} \mid \mathsf{pre}\ \mathsf{T} \\ \mathsf{FR} & ::= & \mathsf{X} \mid \mathsf{abs} \mid \mathsf{pre}\ \mathsf{R} \mid (\ell : \mathsf{FT} ; \mathsf{FR}) \mid \partial \mathsf{FT} \end{array}$$

Ordinary types T are as in the previous example, except the record type constructor Π 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 $\circ.\mathit{Row}(\mathsf{L})$ for some L), applications of the row constructors abs and pre (which are really $\mathsf{abs}^{\mathit{Row}(\mathsf{L})}$ and $\mathsf{pre}^{\mathit{Row}(\mathsf{L})}$ for some L), row extension, and constant rows, where the row components are field types FT.

In many basic applications of rows, $\mathsf{abs}^{\mathit{Row}(\mathsf{L})}$ and $\mathsf{pre}^{\mathit{Row}(\mathsf{L})}$ 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 simplified setting, the syntax of types and rows becomes:

$$\begin{array}{rcl}
\mathsf{T} & ::= & \mathsf{X} \mid \mathsf{T} \rightarrow \mathsf{T} \mid \Pi\,\mathsf{FR} \\
\mathsf{FT} & ::= & \mathsf{X} \mid \mathsf{abs} \mid \mathsf{pre}\;\mathsf{T} \\
\mathsf{FR} & ::= & \mathsf{X} \mid (\ell : \mathsf{FT}\;;\;\mathsf{FR}) \mid \partial\mathsf{FT}
\end{array}$$

This is the syntax found in some introductory accounts of rows (Rémy, 1993b; Pottier, 2000). □

1.8.4 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 1.8.12 below. In this model, every type constructor whose image row kind is *Type* (that is, every type constructor of the form $\mathsf{G}^{\mathit{Type}}$ or H) is interpreted as itself, as in a free tree model. However, every application of a type constructor whose image row kind is $\mathit{Row}(\mathsf{L})$ for some L receives special treatment: it is interpreted as a family of types indexed by $\mathcal{L} \setminus \mathsf{L}$, 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 $\mathcal{L} \setminus \mathsf{L}$. More precisely,

1.8.9 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	$(\mathsf{K} \Rightarrow \kappa).\mathit{Type}$	$(\mathsf{G} : \mathsf{K} \Rightarrow \kappa) \in \mathcal{S}_0$
H	$\mathsf{K}.\mathit{Row}(\varnothing) \Rightarrow \kappa.\mathit{Type}$	$(\mathsf{H} : \mathsf{K} \Rightarrow \kappa) \in \mathcal{S}_1$
L^κ	$\kappa.(\mathit{Type}^{\mathcal{L} \setminus \mathsf{L}} \Rightarrow \mathit{Row}(\mathsf{L}))$	

□

The first two lines in this signature coincide with the definitions of $\mathsf{G}^{\mathit{Type}}$ 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 $\mathsf{t}(\epsilon)$ must be of the form G or H. If t is a ground type of kind $\kappa.\mathit{Row}(\mathsf{L})$, then its head symbol must be L^κ, and its immediate subtrees, which are indexed by $\mathcal{L} \setminus \mathsf{L}$, 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 $\mathcal{L} \setminus \mathsf{L}$. 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.

1.8.10 DEFINITION: The interpretation of the type constructors that populate $\mathcal{S}$ is defined as follows.

1. Let $(\mathsf{G} : \mathsf{K} \Rightarrow \kappa) \in \mathcal{S}_0$. Then, $\mathsf{G}^{\mathit{Type}}$ is interpreted as the function that maps $\mathsf{T} \in \mathcal{M}_{\mathsf{K}.\mathit{Type}}$ to the ground type $\mathsf{t} \in \mathcal{M}_{\kappa.\mathit{Type}}$ defined by $\mathsf{t}(\epsilon) = \mathsf{G}$ and $\mathsf{t}/\mathsf{d} = \mathsf{T}(\mathsf{d})$ for every $\mathsf{d} \in \mathit{dom}(\mathsf{K})$. This is a syntactic interpretation.

2. Let $(\mathsf{H} : \mathsf{K} \Rightarrow \kappa) \in \mathcal{S}_1$. Then, H is interpreted as the function that maps $\mathsf{T} \in \mathcal{M}_{\mathsf{K}.\mathit{Row}(\varnothing)}$ to the ground type $\mathsf{t} \in \mathcal{M}_{\kappa.\mathit{Type}}$ defined by $\mathsf{t}(\epsilon) = \mathsf{H}$ and $\mathsf{t}/\mathsf{d} = \mathsf{T}(\mathsf{d})$ for every $\mathsf{d} \in \mathit{dom}(\mathsf{K})$. (Because H is unary, there is exactly one such d.) This is also a syntactic interpretation.

3. Let $(\mathsf{G} : \mathsf{K} \Rightarrow \kappa) \in \mathcal{S}_0$. Then, $\mathsf{G}^{\mathit{Row}(\mathsf{L})}$ is interpreted as the function that maps $\mathsf{T} \in \mathcal{M}_{\mathsf{K}.\mathit{Row}(\mathsf{L})}$ to the ground type $\mathsf{t} \in \mathcal{M}_{\kappa.\mathit{Row}(\mathsf{L})}$ defined by $\mathsf{t}(\epsilon) = \mathsf{L}^\kappa$ and $\mathsf{t}(\ell) = \mathsf{G}$ and $\mathsf{t}/(\ell \cdot \mathsf{d}) = \mathsf{T}(\mathsf{d})/\ell$ for every $\ell \in \mathcal{L} \setminus \mathsf{L}$ and $\mathsf{d} \in \mathit{dom}(\mathsf{K})$. Thus, when applied to a family of rows, the type constructor $\mathsf{G}^{\mathit{Row}(\mathsf{L})}$ 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 $t_1 \in \mathcal{M}_{\kappa.\mathit{Type}}$ to the ground type $t \in \mathcal{M}_{\kappa.\mathit{Row}(L)}$ 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 $(t_1, t_2) \in \mathcal{M}_{\kappa.\mathit{Type}} \times \mathcal{M}_{\kappa.\mathit{Row}(\ell.L)}$ to the ground type $t \in \mathcal{M}_{\kappa.\mathit{Row}(L)}$ 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 (§1.2). 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 §1.8.5. 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).

1.8.11 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 1.2.9), 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 $\rightarrow$ 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 $\rightarrow$ and $''$ are usually incompatible. This property is exploited in proofs of the progress theorem. In the case of Example 1.8.7, because no type constructors other than $\rightarrow$ and $''$ are present, these conditions imply that there is no sensible way of interpreting subtyping other than equality. In the case of Example 1.8.8, two sensible interpretations of subtyping exist: one is equality, while the other is the nonstructural subtyping order obtained by letting $\mathsf{pre} \leqslant \mathsf{abs}$. □

1.8.12 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

$$(\ell_1 : \mathsf{T}_1 \,;\, \ell_2 : \mathsf{T}_2 \,;\, \mathsf{T}_3) = (\ell_2 : \mathsf{T}_2 \,;\, \ell_1 : \mathsf{T}_1 \,;\, \mathsf{T}_3) \qquad \text{(C-Row-LL)}$$
$$\partial \mathsf{T} = (\ell : \mathsf{T} \,;\, \partial \mathsf{T}) \qquad \text{(C-Row-DL)}$$
$$\mathsf{G}\ \partial \mathsf{T}_1\ \ldots\ \partial \mathsf{T}_n = \partial(\mathsf{G}\ \mathsf{T}_1\ \ldots\ \mathsf{T}_n) \qquad \text{(C-Row-GD)}$$
$$\mathsf{G}\ (\ell : \mathsf{T}_1 \,;\, \mathsf{T}'_1)\ \ldots\ (\ell : \mathsf{T}_n \,;\, \mathsf{T}'_n) = (\ell : \mathsf{G}\ \mathsf{T}_1\ \ldots\ \mathsf{T}_n \,;\, \mathsf{G}\ \mathsf{T}'_1\ \ldots\ \mathsf{T}'_n) \qquad \text{(C-Row-GL)}$$

Figure 1-12: Equational reasoning with rows

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 §1.8.5, hold in the model. □

1.8.5 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 1-12) and of constraint equivalence laws (Figure 1-13), which we now explain and prove.

1.8.13 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 (X) and type meta-variables (T) involved. Thus, each equation in Figure 1-12 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 $\partial^{\ell.\mathsf{L}}\mathsf{T} = (\ell^{\kappa,\mathsf{L}} : \mathsf{T} \,;\, \partial^{\mathsf{L}}\mathsf{T})$, where $\ell \notin \mathsf{L}$ and T has kind $\kappa.$*Type*. □

1.8.14 Exercise [Recommended, ★, ↛]: Reconstruct all of the missing kind information in the equations of Figure 1-12. □

1.8.15 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 *Row*(L) for every L, while the latter has row kind *Row*(L) for every L such that $\ell \notin \mathsf{L}$ holds. As a result, while replacing the unannotated term $(\ell : \mathsf{T} \,;\, \partial \mathsf{T})$ with $\partial \mathsf{T}$ is always valid, the converse is not: replacing the unannotated term $\partial \mathsf{T}$ with $(\ell : \mathsf{T} \,;\, \partial \mathsf{T})$ 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 $\partial\mathsf{T}$ maps every label within its domain to T, that is, $\partial^{\mathsf{L}}\mathsf{T}$ maps every label $\ell \notin \mathsf{L}$ to T. This equation holds because $\partial\mathsf{T}$ 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 $\mathsf{G}^{Row(\mathsf{L})}$ for some L, while their right-hand sides involve G^{Type}. Both equations state that it is equivalent to apply $\mathsf{G}^{Row(\mathsf{L})}$ at the level of rows or to apply G^{Type} at the level of types. Our interpretation of $\mathsf{G}^{Row(\mathsf{L})}$ was designed so as to give rise to these equations: indeed, the application of $\mathsf{G}^{Row(\mathsf{L})}$ to n ground rows (where n is the arity of G) is interpreted as a pointwise application of G^{Type} to the rows' components (item 3 of Definition 1.8.10). Their use is illustrated in Examples 1.8.28 and 1.8.39.

1.8.16 LEMMA: Each of the equations in Figure 1-12 is equivalent to true. □

The four equations in Figure 1-12 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 $\mathsf{T}_1 = \mathsf{T}_2$, where T_1 and T_2 have distinct head symbols, is not necessarily equivalent to false. In Figure 1-13, 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 1-12 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.

1.8.17 EXERCISE [RECOMMENDED, ★, ↛]: Reconstruct all of the missing kind information in the laws of Figure 1-13. □

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 T" for "the ground type $\phi(\mathsf{T})$". 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 T_1' must map ℓ_2 to the ground type T_2, while, symmetrically, the ground row T_2' must map ℓ_1 to the ground type

$$(\ell_1 : \mathsf{T}_1 \,;\, \mathsf{T}'_1) = (\ell_2 : \mathsf{T}_2 \,;\, \mathsf{T}'_2) \quad\equiv\quad \exists \mathsf{X}.(\mathsf{T}'_1 = (\ell_2 : \mathsf{T}_2 \,;\, \mathsf{X}) \wedge \mathsf{T}'_2 = (\ell_1 : \mathsf{T}_1 \,;\, \mathsf{X})) \qquad \text{(C-MUTE-LL)}$$
$$\text{if } \mathsf{X} \,\#\, \mathit{ftv}(\mathsf{T}_1, \mathsf{T}'_1, \mathsf{T}_2, \mathsf{T}'_2) \wedge \ell_1 \neq \ell_2$$

$$\partial\mathsf{T} = (\ell : \mathsf{T}' \,;\, \mathsf{T}'') \quad\equiv\quad \mathsf{T} = \mathsf{T}' \wedge \partial\mathsf{T} = \mathsf{T}'' \qquad \text{(C-MUTE-DL)}$$

$$\mathsf{G}\ \mathsf{T}_1\ \ldots\ \mathsf{T}_n = \partial\mathsf{T} \quad\equiv\quad \exists \mathsf{X}_1 \ldots \mathsf{X}_n.(\mathsf{G}\ \mathsf{X}_1\ \ldots\ \mathsf{X}_n = \mathsf{T} \wedge \textstyle\bigwedge_{i=1}^{n}(\mathsf{T}_i = \partial\mathsf{X}_i)) \qquad \text{(C-MUTE-GD)}$$
$$\text{if } \mathsf{X}_1 \ldots \mathsf{X}_n \,\#\, \mathit{ftv}(\mathsf{T}_1, \ldots, \mathsf{T}_n, \mathsf{T})$$

$$\begin{aligned}\mathsf{G}\ \mathsf{T}_1\ \ldots\ \mathsf{T}_n = (\ell : \mathsf{T} \,;\, \mathsf{T}') \quad\equiv\quad \exists \mathsf{X}_1 \ldots \mathsf{X}_n, \mathsf{X}'_1 \ldots \mathsf{X}'_n.(&\mathsf{G}\ \mathsf{X}_1\ \ldots\ \mathsf{X}_n = \mathsf{T} \wedge \\ &\mathsf{G}\ \mathsf{X}'_1\ \ldots\ \mathsf{X}'_n = \mathsf{T}' \wedge \\ &\textstyle\bigwedge_{i=1}^{n}(\mathsf{T}_i = (\ell : \mathsf{X}_i \,;\, \mathsf{X}'_i)))\end{aligned}$$
$$\text{if } \mathsf{X}_1 \ldots \mathsf{X}_n, \mathsf{X}'_1 \ldots \mathsf{X}'_n \,\#\, \mathit{ftv}(\mathsf{T}_1, \ldots, \mathsf{T}_n, \mathsf{T}, \mathsf{T}') \qquad \text{(C-MUTE-GL)}$$

Figure 1-13: Constraint equivalence laws involving rows

T_1. This may be expressed by two equations of the form $\mathsf{T}'_1 = (\ell_2 : \mathsf{T}_2 \,;\, \ldots)$ and $\mathsf{T}'_2 = (\ell_1 : \mathsf{T}_1 \,;\, \ldots)$. Furthermore, because the ground rows T'_1 and 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 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 $\partial\mathsf{T}$ maps every label to the ground type T, the ground type T' must coincide with the ground type T, while the ground row T'' must map every label in its domain to the ground type T. This is expressed by the equations $\mathsf{T} = \mathsf{T}'$ and $\partial\mathsf{T} = \mathsf{T}''$. 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 $\mathsf{G}\ \mathsf{T}_1\ \ldots\ \mathsf{T}_n$ is obtained by applying the type constructor G, pointwise, to the ground rows $\mathsf{T}_1, \ldots, \mathsf{T}_n$. If, as in C-MUTE-GD, it coincides with the constant ground row $\partial\mathsf{T}$, then every T_i must itself be a constant ground row, of the form $\partial\mathsf{X}_i$, and T must coincide with $\mathsf{G}\ \mathsf{X}_1\ \ldots\ \mathsf{X}_n$. C-MUTE-GL is obtained in a similar manner.

1.8.18 LEMMA: Each of the equivalence laws in Figure 1-13 holds. □

1.8.6 Solving equality constraints in the presence of rows

We now extend the unification algorithm given in §1.6.1 with support for rows. The extended algorithm is intended to solve unification problems where the syntax and interpretation of types are as defined in §1.8.3 and §1.8.4, re-

$$(\ell_1 : X_1 ; X_1') = (\ell_2 : T_2 ; T_2') = \epsilon \quad\to\quad \exists X.(X_1' = (\ell_2 : T_2 ; X) \wedge T_2' = (\ell_1 : X_1 ; X)) \wedge (\ell_1 : X_1 ; X_1') = \epsilon \qquad \text{(S-MUTE-LL)}$$
$$\text{if } \ell_1 \neq \ell_2$$

$$\partial X = (\ell : T ; T') = \epsilon \quad\to\quad X = T \wedge \partial X = T' \wedge \partial X = \epsilon \qquad \text{(S-MUTE-DL)}$$

$$G\ T_1\ \ldots\ T_n = \partial X = \epsilon \quad\to\quad \exists X_1 \ldots X_n.(G\ X_1\ \ldots\ X_n = X \wedge \textstyle\bigwedge_{i=1}^{n}(T_i = \partial X_i)) \wedge \partial X = \epsilon \qquad \text{(S-MUTE-GD)}$$

$$G\ T_1\ \ldots\ T_n = (\ell : X ; X') = \epsilon \quad\to\quad \exists X_1 \ldots X_n, X_1' \ldots X_n'.(G\ X_1\ \ldots\ X_n = X \wedge G\ X_1'\ \ldots\ X_n' = X' \wedge \textstyle\bigwedge_{i=1}^{n}(T_i = (\ell : X_i ; X_i'))) \wedge (\ell : X ; X') = \epsilon \qquad \text{(S-MUTE-GL)}$$

$$F\,\vec{T} = F'\,\vec{T}' = \epsilon \quad\to\quad \mathsf{false} \qquad \text{(S-CLASH')}$$
$$\text{if } F \neq F' \text{ and none of the four rules above applies}$$

Figure 1-14: Row unification (corrigendum to Figure 1-10)

spectively. Its specification consists of the original rewriting rules of Figure 1-10, minus S-CLASH, which is removed and replaced with the rules given in Figure 1-14. 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 1-13, 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 mutation 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 1-10), where they help reason about sharing.

1.8.19 EXERCISE [★, ↛]: Check that the rewriting rules in Figure 1-14 preserve well-kindedness. Conclude that, provided its input constraint is well-kinded, the unification algorithm needs not keep track of kinds. □

The properties of the unification algorithm are preserved by this extension, as witnessed by the next three lemmas. Note that the termination of reduction is ensured *only* when the initial unification problem is well-kinded. The ill-kinded unification problem $\mathsf{X} = (\ell_1 : \mathsf{T} \,;\, \mathsf{Y}) \wedge \mathsf{X} = (\ell_2 : \mathsf{T} \,;\, \mathsf{Y})$, where ℓ_1 and ℓ_2 are distinct, illustrates this point.

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

1.8.21 LEMMA: $U_1 \rightarrow U_2$ implies $U_1 \equiv U_2$. □

1.8.22 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.

1.8.23 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. □

1.8.24 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 1-12 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. □

1.8.7 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).

1.8.25 EXAMPLE [FULL RECORDS]: As in §1.8.2, 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:

$$\begin{array}{rcllr}
\{\{V; v\} \text{ with } \ell = v'\} & \xrightarrow{\delta} & \{V[\ell \mapsto v']; v\} & & (\text{R-UPDATE}) \\
\{V; v\}.\{\ell\} & \xrightarrow{\delta} & V(\ell) & (\ell \in \mathit{dom}(V)) & (\text{R-ACCESS-1}) \\
\{V; v\}.\{\ell\} & \xrightarrow{\delta} & v & (\ell \notin \mathit{dom}(V)) & (\text{R-ACCESS-2})
\end{array}$$

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 1.8.7. Let the initial environment Γ_0 contain the following bindings:

$$\begin{array}{rl}
\{\}_{\{\ell_1,\ldots,\ell_n\}} : & \forall X_1 \ldots X_n X. X_1 \to \ldots \to X_n \to X \to {''}\,(\ell_1 : X_1; \ldots; \ell_n : X_n; \partial X) \\
& \text{where } \ell_1 < \ldots < \ell_n \\
\{\cdot \text{ with } \ell = \cdot\} : & \forall X X' Y. {''}\,(\ell : X\,;\, Y) \to X' \to {''}\,(\ell : X'\,;\, Y) \\
\cdot.\{\ell\} : & \forall XY. {''}\,(\ell : X\,;\, Y) \to X
\end{array}$$

Note that, in particular, the type scheme assigned to record creation $\{\cdot\}$ is $\forall X.X \to {''}\,(\partial X)$. As a result, these bindings are exactly as announced in §1.8.2.

To illustrate how these definitions work together, let us consider the program $\{\{0\} \text{ with } \ell_1 = \mathtt{true}\}.\{\ell_2\}$, 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 $\text{int} \to {''}\,(\partial\text{int})$. Thus, assuming that 0 has type int, the expression $\{0\}$ has type $\Pi\ (\partial\mathsf{int})$. 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 $\{\cdot\ \mathsf{with}\ \ell_1 = \cdot\}$ has type $\Pi\ (\ell_1 : \mathsf{int}\ ;\ \partial\mathsf{int}) \rightarrow \mathsf{bool} \rightarrow \Pi\ (\ell_1 : \mathsf{bool}\ ;\ \partial\mathsf{int})$. May we immediately use HMX-APP to typecheck the application of $\{\cdot\ \mathsf{with}\ \ell_1 = \cdot\}$ to $\{0\}$? Unfortunately, no, because there is an apparent mismatch between the expected type $\Pi\ (\ell_1 : \mathsf{int}\ ;\ \partial\mathsf{int})$ and the effective type $\Pi\ (\partial\mathsf{int})$. To work around this problem, let us recall that, by C-ROW-DL, the equation $\Pi\ (\partial\mathsf{int}) = \Pi\ (\ell_1 : \mathsf{int}\ ;\ \partial\mathsf{int})$ is equivalent to true. Thus, HMX-SUB allows proving that $\{0\}$ has type $\Pi\ (\ell_1 : \mathsf{int}\ ;\ \partial\mathsf{int})$. Assuming that `true` has type bool, we may now apply HMX-APP and deduce

$$\mathsf{true}, \Gamma_0 \vdash \{\{0\}\ \mathsf{with}\ \ell_1 = \mathtt{true}\} : \Pi\ (\ell_1 : \mathsf{bool}\ ;\ \partial\mathsf{int}).$$

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 $\{\{0\}\ \mathsf{with}\ \ell_1 = \mathtt{true}\}.\{\ell_2\}$ has type int, provided ℓ_1 and ℓ_2 are distinct. □

1.8.26 EXERCISE [★★, ↛]: Unfold the definition of the constraint $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\{\{0\}\ \mathsf{with}\ \ell_1 = \mathtt{true}\}.\{\ell_2\} : \mathsf{X}]\!]$, which states that X is a valid type for the above program. Assuming that subtyping is interpreted as equality, simulate a run of the constraint solver (§1.6), extended with support for rows (§1.8.6), so as to solve this constraint. Check that the solved form is equivalent to $\mathsf{X} = \mathsf{int}$. □

1.8.27 EXERCISE [★★★]: Check that the definitions of Example 1.8.25 meet the requirements of Definition 1.5.5. □

1.8.28 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:

$$\mathsf{apply}\ \{\mathsf{V}; \mathsf{v}\}\ \{\mathsf{V}'; \mathsf{v}'\} \xrightarrow{\delta} \{\mathsf{V}\ \mathsf{V}'; \mathsf{v}\ \mathsf{v}'\} \qquad \text{(R-APPLY-1)}$$

$$\mathsf{apply}\ \{\mathsf{V}; \mathsf{v}\}\ \{\mathsf{V}'; \mathsf{v}'\} \xrightarrow{\delta} \mathsf{apply}\ \{\mathsf{V}; \mathsf{v}\}\ \{\mathsf{V}'[\ell \mapsto \mathsf{v}']; \mathsf{v}'\} \quad \text{if } \ell \in \mathit{dom}(\mathsf{V}) \setminus \mathit{dom}(\mathsf{V}') \qquad \text{(R-APPLY-2)}$$

$$\mathsf{apply}\ \{\mathsf{V}; \mathsf{v}\}\ \{\mathsf{V}'; \mathsf{v}'\} \xrightarrow{\delta} \mathsf{apply}\ \{\mathsf{V}[\ell' \mapsto \mathsf{v}]; \mathsf{v}\}\ \{\mathsf{V}'; \mathsf{v}'\} \quad \text{if } \ell' \in \mathit{dom}(\mathsf{V}') \setminus \mathit{dom}(\mathsf{V}) \qquad \text{(R-APPLY-3)}$$

In the first rule, $V\ V'$ is defined only if V and V' have a common domain; it is then defined as the function that maps ℓ to the expression $V(\ell)\ V'(\ell)$. 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 1.8.25. We extend the initial environment Γ_0 with the following binding:

$$\mathsf{apply} : \quad \forall XY.'' (X \rightarrow Y) \rightarrow'' X \rightarrow'' Y$$

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 $\forall XY.(X \rightarrow Y) \rightarrow X \rightarrow Y$. 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 X and Y are now row variables: their kind is $\star.\mathit{Row}(\varnothing)$. As a result, the leftmost occurrence of the arrow constructor is really $\rightarrow^{\mathit{Row}(\varnothing)}$. Thus, we are exploiting the presence of type constructors of the form G^s, with $s \neq \mathit{Type}$, 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 $\mathtt{not}$ and $\mathtt{succ}$ are assumed to have types $\mathsf{bool} \rightarrow \mathsf{bool}$ and $\mathsf{int} \rightarrow \mathsf{int}$, 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 $''\ (\ell : \mathsf{bool} \rightarrow \mathsf{bool}\ ;\ \partial(\mathsf{int} \rightarrow \mathsf{int}))$ **(1)**. In order to use apply, however, we must prove that this record has a type of the form $''\ (R_1 \rightarrow R_2)$, where R_1 and R_2 are rows. This is where C-Row-GD and C-Row-GL (Figure 1-12) come into play. Indeed, by C-Row-GD, the type $\partial(\mathsf{int} \rightarrow \mathsf{int})$ may be written $\partial\mathsf{int} \rightarrow \partial\mathsf{int}$. So, (1) may be written $''\ (\ell : \mathsf{bool} \rightarrow \mathsf{bool}\ ;\ \partial\mathsf{int} \rightarrow \partial\mathsf{int})$ **(2)**, which by C-Row-GL may be written $''\ ((\ell : \mathsf{bool}\ ;\ \partial\mathsf{int}) \rightarrow (\ell : \mathsf{bool}\ ;\ \partial\mathsf{int}))$ **(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. □

1.8.29 EXERCISE [★★★, ↛]: Check that the definitions of Example 1.8.28 meet the requirements of Definition 1.5.5. □

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 §1.8.1, they may be either encoded in terms of full records, or given a direct definition. The latter approach is illustrated below.

1.8.30 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 $\langle \ell_1 = \mathtt{t}_1; \ldots; \ell_n = \mathtt{t}_n\rangle$ and $\langle \mathsf{V}\rangle$, where V is a finite mapping of labels to values, in a manner similar to that of Example 1.8.25.

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:

$$\langle\langle \mathsf{V}\rangle \text{ with } \ell = \mathsf{v}\rangle \xrightarrow{\delta} \langle \mathsf{V}[\ell \mapsto \mathsf{v}]\rangle \qquad \text{(R-EXTEND)}$$

$$\langle \mathsf{V}\rangle.\langle\ell\rangle \xrightarrow{\delta} \mathsf{V}(\ell) \qquad (\ell \in dom(\mathsf{V})) \qquad \text{(R-ACCESS)}$$

Let us now define the syntax of types as in Example 1.8.8. Let the initial environment Γ_0 contain the following bindings:

$$\begin{array}{rl}
\langle\rangle_{\{\ell_1,\ldots,\ell_n\}} : & \forall \mathsf{X}_1 \ldots \mathsf{X}_n.\mathsf{X}_1 \to \ldots \to \mathsf{X}_n \to {}^{\prime\prime}\, (\ell_1 : \mathsf{pre}\ \mathsf{X}_1; \ldots; \ell_n : \mathsf{pre}\ \mathsf{X}_n; \partial\mathsf{abs}) \\
 & \text{where } \ell_1 < \ldots < \ell_n \\
\langle\cdot \text{ with } \ell = \cdot\rangle : & \forall \mathsf{X}\mathsf{X}'\mathsf{Y}.{}^{\prime\prime}\, (\ell : \mathsf{X}\, ;\, \mathsf{Y}) \to \mathsf{X}' \to {}^{\prime\prime}\, (\ell : \mathsf{pre}\ \mathsf{X}'\, ;\, \mathsf{Y}) \\
\cdot.\langle\ell\rangle : & \forall \mathsf{X}\mathsf{Y}.{}^{\prime\prime}(\ell : \mathsf{pre}\ \mathsf{X}\, ;\, \mathsf{Y}) \to \mathsf{X}
\end{array}$$

Note that, in particular, the type scheme assigned to the empty record $\langle\rangle$ is ${}^{\prime\prime}\, (\partial\mathsf{abs})$. □

1.8.31 EXERCISE [RECOMMENDED, ★, ↛]: Reconstruct all of the missing kind information in the type schemes given in Example 1.8.30. □

1.8.32 EXERCISE [RECOMMENDED, ★★, ↛]: Define an encoding of finite records in terms of full records, along the lines of §1.8.1. Check that the principal type schemes associated, via the encoding, with the three operations on finite records are precisely those given in Example 1.8.30. □

1.8.33 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? □

1.8.34 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. □

1.8.35 EXERCISE [★★★, ↛]: Check that the definitions of Example 1.8.30 meet the requirements of Definition 1.5.5. □

1.8.8 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 $^\circ\ (\ell_1 : \mathsf{pre}\ \mathsf{T}_1\ ;\ \ell_2 : \mathsf{pre}\ \mathsf{T}_2\ ; \partial\mathsf{abs})$ is intended to contain values of the form $\ell_1\ \mathsf{v}_1$, where v_1 has type T_1, or of the form $\ell_2\ \mathsf{v}_2$, where v_2 has type 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 $^\circ$ $(\ell_1 : \mathsf{pre}\ \mathsf{T}_1\ ;\ \ell_2 : \mathsf{abs}\ ; \partial\mathsf{abs})$ is intended to contain only values of the form $\ell_1\ \mathsf{v}_1$, where v_1 has type T_1, so it is safe to make it a subtype of the above type: in other words, it is safe to allow $\mathsf{abs} \leq \mathsf{pre}\ \mathsf{T}_2$. 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.

1.8.36 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:

$$\begin{array}{rcllr} [\,\ell : \mathsf{v} \mid \mathsf{v}'\,]\ (\ell\ \mathsf{w}) & \xrightarrow{\delta} & \mathsf{v}\ \mathsf{w} & & \text{(R-FILTER-1)} \\ [\,\ell : \mathsf{v} \mid \mathsf{v}'\,]\ (\ell'\ \mathsf{w}) & \xrightarrow{\delta} & \mathsf{v}'\ (\ell'\ \mathsf{w}) & \text{if } \ell \neq \ell' & \text{(R-FILTER-2)} \end{array}$$

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 $\rightarrow$, 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 $\bullet \Rightarrow \star$. Note the similarity with the case of records (Example 1.8.8).

Subtyping is typically interpreted in one of two ways. One is equality. The other is the nonstructural subtyping order obtained by letting $\rightarrow$ be contravariant in its domain and covariant in its codomain, $^\circ$ be covariant, $\rightarrow$ and $^\circ$ be incompatible, and letting $\mathsf{abs} \leqslant \mathsf{pre}$. Compare this definition with the case of records (Remark 1.8.11).

To complete the setup, let the initial environment Γ_0 contain the following bindings:

$$\begin{array}{rl} \ell\ \cdot : & \forall \mathsf{XY}.\mathsf{X} \rightarrow {}^\circ\,(\ell : \mathsf{pre}\ \mathsf{X}\ ;\ \mathsf{Y}) \\ [\,\ell : \cdot \mid \cdot\,]\ \cdot : & \forall \mathsf{XX'YY'}.(\mathsf{X} \rightarrow \mathsf{Y}) \rightarrow ({}^\circ\,(\ell : \mathsf{X'}\ ;\ \mathsf{Y'}) \rightarrow \mathsf{Y}) \rightarrow {}^\circ\,(\ell : \mathsf{pre}\ \mathsf{X}\ ;\ \mathsf{Y'}) \rightarrow \mathsf{Y} \\ [] : & \forall \mathsf{X}.{}^\circ\,(\partial \mathsf{abs}) \rightarrow \mathsf{X} \end{array}$$

The first binding means, in particular, that if v has type T, then a value of the form $\ell\ \mathsf{v}$ has type $^\circ\,(\ell : \mathsf{pre}\ \mathsf{T}\ ;\ \partial\mathsf{abs})$. 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 Y with rows other than $\partial\mathsf{abs}$. For instance, the value $\ell\ \mathsf{v}$ also has type $^\circ\,(\ell : \mathsf{pre}\ \mathsf{T}\ ;\ \ell' : \mathsf{pre}\ \mathsf{T'}\ ;\ \partial\mathsf{abs})$. 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 $\ell\ \mathsf{v}$ admits an infinite number of types. The point is that, if v has type T and v' has type T', then both $\ell\ \mathsf{v}$ and $\ell'\ \mathsf{v}'$ have type $^\circ\,(\ell : \mathsf{pre}\ \mathsf{T}\ ;\ \ell' : \mathsf{pre}\ \mathsf{T'}\ ;\ \partial\mathsf{abs})$, 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 $[\,\ell : \mathsf{v} \mid \mathsf{v}'\,]$ is a function that expects an argument of the form $\ell'\ \mathsf{w}$ and reduces to $\mathsf{v}\ \mathsf{w}$ if ℓ' is ℓ and to $\mathsf{v}'\ (\ell'\ \mathsf{w})$ 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 $^{\circ}\ (\partial\mathtt{abs})$, 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 $^{\circ}\ (\ell : \mathtt{pre}\ \mathtt{T}\ ;\ \mathtt{T}')$. Note that any choice of $\mathtt{T}''$ will do, including, in particular, $\mathtt{abs}$. 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. □

1.8.37 EXERCISE [★★★, $\not\rightarrow$]: Check that the definitions of Example 1.8.36 meet the requirements of Definition 1.5.5. □

1.8.38 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 1.8.30). The type of $[]$ involves the complete row $\partial\mathtt{abs}$, 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 $\langle\cdot\ \mathtt{with}\ \ell = \cdot\rangle$, provided the three continuation arrows $\rightarrow \mathtt{Y}$ 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 $\lambda\mathtt{f}.\mathtt{f}\ \langle\mathtt{v}'\ \mathtt{with}\ \ell = \mathtt{v}\rangle$, 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 1.8.36. □

1.8.39 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

$$\# \; \mathsf{v} \; (\ell \; \mathsf{w}) \quad \xrightarrow{\delta} \quad \mathsf{v}.\langle \ell \rangle \; \mathsf{w} \qquad \text{(R-SEND)}$$

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 1.8.25) 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

$$\forall \mathsf{XY}.{}'' \; (\mathsf{X} \to \partial \mathsf{Y}) \to {}^{\circ} \; \mathsf{X} \to \mathsf{Y}$$

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 ${}^{\circ}\;\mathsf{X}$ 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 $\ell\;\mathsf{w}$ 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 X stand for both of these rows. Note that the arrow in $\mathsf{X} \to \partial\mathsf{Y}$ is really $\to^{Row(\varnothing)}$: once again, we are exploiting the presence of type constructors of the form G^s, with $s \neq \mathit{Type}$, in the signature $\mathcal{S}$.

If the record of functions v is viewed as an *object*, and if the variant $\ell\;\mathsf{w}$ is viewed as a *message* ℓ carrying a parameter 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. □

1.8.40 EXERCISE [★★★, ↛]: Check that the definitions of Example 1.8.39 meet the requirements of Definition 1.5.5. □

The name *polymorphic variants* stems from the highly polymorphic type schemes assigned to the operations on variants (Example 1.8.36). 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.

1.8.9 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 1.8.30), 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.

1.8.10 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 $Row(\mathsf{L})$ 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 $(\ell : \mathsf{T}_1 \ ; \ \ell : \mathsf{T}_2 \ ; \mathsf{T}_3)$, 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 $(\ell : \mathsf{T}_1 \ ; \ \mathsf{R}_1) = (\ell : \mathsf{T}_2 \ ; \ \mathsf{R}_2)$ entails $\mathsf{T}_1 = \mathsf{T}_2$, but does *not* entail $\mathsf{R}_1 = \mathsf{R}_2$, because R_1 and 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 R_1 and 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 $(\ell : \mathsf{T}_1 \ ; \mathsf{R}_1) = (\ell : \mathsf{T}_2 \ ; \mathsf{R}_2)$ to $(\mathsf{T}_1 = \mathsf{T}_2) \wedge (\mathsf{R}_1 =_{\{\ell\}} \mathsf{R}_2)$, where the new predicate $=_L$ is interpreted as row equality *outside* L. Of course, the entire constraint solver must then be extended to deal with constraints of the form $\mathsf{T}_1 =_L \mathsf{T}_2$. 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 $(\ell : \mathsf{T} \ ; \mathsf{X})$ appears anywhere within a type derivation, then 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(X). For every label ℓ, a unary predicate $\cdot$ lacks ℓ is introduced: roughly speaking, the constraint R lacks ℓ is considered to hold if the (finite) row R does not define the label ℓ. The constrained type scheme assigned to record access is

$$\cdot.\langle\ell\rangle : \forall \mathsf{XY}[\mathsf{Y}\ \mathsf{lacks}\ \ell].'' (\ell : \mathsf{X}\ ;\ \mathsf{Y}) \rightarrow \mathsf{X}.$$

The constraint Y lacks ℓ ensures that the row $(\ell : \mathsf{X}\ ;\ \mathsf{Y})$ 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 $\{\ell_1 : \mathsf{T}_1; \ldots; \ell_n : \mathsf{T}_n\}$. Second, for every label ℓ, a binary predicate $\cdot$ has $\ell : \cdot$ is available. The idea is that the constraint T has $\ell : \mathsf{T}'$ holds if and only if T is a record type that contains the field $\ell : \mathsf{T}'$. Then, record access may be assigned the constrained type scheme

$$\cdot.\langle\ell\rangle : \forall \mathsf{XY}[\mathsf{X}\ \mathsf{has}\ \ell : \mathsf{Y}].\mathsf{X} \rightarrow \mathsf{Y}.$$

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

$\mathsf{R}_1 \leq_{\mathsf{L}} \mathsf{R}_2$, 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 1-14.

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).

A Solutions to Selected Exercises

1.1.21 SOLUTION: Within Damas and Milner's type system, we have:

$$\textsc{dm-Abs}\;\dfrac{\textsc{dm-Let}\;\dfrac{\textsc{dm-Var}\;\dfrac{}{z_1 : X \vdash z_1 : X} \qquad \dfrac{}{z_1 : X; z_2 : X \vdash z_2 : X}\;\textsc{dm-Var}}{z_1 : X \vdash \mathtt{let}\ z_2 = z_1\ \mathtt{in}\ z_2 : X}}{\varnothing \vdash \lambda z_1.\mathtt{let}\ z_2 = z_1\ \mathtt{in}\ z_2 : X \to X}$$

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.

1.1.22 SOLUTION: It is straightforward to prove that the identity function has type $\mathsf{int} \to \mathsf{int}$:

$$\dfrac{\dfrac{}{\Gamma_0; z : \mathsf{int} \vdash z : \mathsf{int}}\;\textsc{dm-Var}}{\Gamma_0 \vdash \lambda z.z : \mathsf{int} \to \mathsf{int}}\;\textsc{dm-Abs}$$

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.

$$\textsc{dm-Gen}\;\dfrac{\textsc{dm-Abs}\;\dfrac{\textsc{dm-Var}\;\dfrac{}{\Gamma_0; z : X \vdash z : X}}{\Gamma_0 \vdash \lambda z.z : X \to X} \qquad X \notin \mathit{ftv}(\Gamma_0)}{\Gamma_0 \vdash \lambda z.z : \forall X.X \to X}$$

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.

$$\textsc{dm-Abs}\;\dfrac{\textsc{dm-App}\;\dfrac{\textsc{dm-App}\;\dfrac{\Gamma_1 \vdash \hat{+} : \mathsf{int} \to \mathsf{int} \to \mathsf{int} \qquad \Gamma_1 \vdash z : \mathsf{int}}{\Gamma_1 \vdash \hat{+}\ z : \mathsf{int} \to \mathsf{int}} \qquad \Gamma_1 \vdash \hat{1} : \mathsf{int}}{\Gamma_1 \vdash z \mathbin{\hat{+}} \hat{1} : \mathsf{int}}}{\Gamma_0 \vdash \lambda z.z \mathbin{\hat{+}} \hat{1} : \mathsf{int} \to \mathsf{int}}$$

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 $\Gamma_0; f : \mathsf{int} \to$

int.

$$\dfrac{\Delta_1 \qquad \dfrac{\Gamma_2 \vdash \texttt{f} : \mathsf{int} \rightarrow \mathsf{int} \qquad \Gamma_2 \vdash \hat{2} : \mathsf{int}}{\Gamma_2 \vdash \texttt{f}\ \hat{2} : \mathsf{int}}\ \textsc{dm-App}}{\Gamma_0 \vdash \texttt{let f} = \lambda\texttt{z.z}\ \hat{+}\ \hat{1}\ \texttt{in f}\ \hat{2} : \mathsf{int}}\ \textsc{dm-Let}$$

To derive the fourth typing judgement, we re-use Δ_0, which proves that the identity function has polymorphic type $\forall\mathtt{X}.\mathtt{X} \rightarrow \mathtt{X}$. We write Γ_3 for $\Gamma_0; \texttt{f} : \forall\mathtt{X}.\mathtt{X} \rightarrow \mathtt{X}$. By DM-VAR and DM-INST, we have both $\Gamma_3 \vdash \texttt{f} : (\mathsf{int} \rightarrow \mathsf{int}) \rightarrow (\mathsf{int} \rightarrow \mathsf{int})$ and $\Gamma_3 \vdash \texttt{f} : \mathsf{int} \rightarrow \mathsf{int}$. Thus, we may build the following derivation:

$$\dfrac{\Delta_0 \qquad \textsc{dm-App}\ \dfrac{\textsc{dm-App}\ \dfrac{\begin{array}{c}\Gamma_3 \vdash \texttt{f} : (\mathsf{int} \rightarrow \mathsf{int}) \rightarrow (\mathsf{int} \rightarrow \mathsf{int}) \\ \Gamma_3 \vdash \texttt{f} : \mathsf{int} \rightarrow \mathsf{int}\end{array}}{\Gamma_3 \vdash \texttt{f f} : \mathsf{int} \rightarrow \mathsf{int}} \qquad \Gamma_3 \vdash \hat{2} : \mathsf{int}}{\Gamma_3 \vdash \texttt{f f}\ \hat{2} : \mathsf{int}}}{\Gamma_0 \vdash \texttt{let f} = \lambda\texttt{z.z in f f}\ \hat{2} : \mathsf{int}}\ \textsc{dm-Let}$$

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 $\texttt{f} : \mathsf{int} \rightarrow \mathsf{int}$ 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 $\texttt{f}$ is used at two incompatible types, namely $(\mathsf{int} \rightarrow \mathsf{int}) \rightarrow (\mathsf{int} \rightarrow \mathsf{int})$ and $\mathsf{int} \rightarrow \mathsf{int}$, inside the expression $\texttt{f f}\ \hat{2}$. Both of these types are *instances* of $\forall\mathtt{X}.\mathtt{X} \rightarrow \mathtt{X}$, the type scheme assigned to $\texttt{f}$ 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 $\lambda\texttt{f}.(\texttt{f f})$ is ill-typed in the simply-typed λ-calculus, because no type $\mathtt{T}$ may coincide with a type of the form $\mathtt{T} \rightarrow \mathtt{T}'$: 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 $\texttt{f}$ 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 $\Gamma_0; \texttt{f} : \mathtt{T} \vdash \texttt{f} : \mathtt{T}$, because all of the type variables in the type $\mathtt{T}$ appear free in the environment $\Gamma_0; \mathtt{f} : \mathsf{T}$. 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 $\mathtt{let}$-bound and is assigned the type scheme $\forall \mathsf{X}.\mathsf{X} \rightarrow \mathsf{X}$. 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 $\mathsf{T} = \mathsf{T} \rightarrow \mathsf{T}'$ 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.

1.4.1 SOLUTION: Let $\mathsf{X} \notin ftv(\Gamma)$ **(1)**. Assume that there exist a satisfiable constraint C and a type T such that $C, \Gamma \vdash \mathtt{t} : \mathsf{T}$ **(2)** holds. Thanks to (1), we find that, up to a renaming of C and T, we may further assume $\mathsf{X} \notin ftv(C, \mathsf{T})$ **(3)**. Then, applying Lemma 1.3.1 to (2), we obtain $C \wedge \mathsf{T} = \mathsf{X}, \Gamma \vdash \mathtt{t} : \mathsf{T}$, which by HMX-SUB yields $C \wedge \mathsf{T} = \mathsf{X}, \Gamma \vdash \mathtt{t} : \mathsf{X}$ **(4)**. Furthermore, by (3) and C-NAMEEQ, we have $\exists \mathsf{X}.(C \wedge \mathsf{T} = \mathsf{X}) \equiv C$. Because C is satisfiable, this implies that $C \wedge \mathsf{T} = \mathsf{X}$ is satisfiable as well. As a result, we have found a satisfiable constraint C' such that $C', \Gamma \vdash \mathtt{t} : \mathsf{X}$ holds.

Now, assume Γ is closed and X is arbitrary. Then, (1) holds, so the previous paragraph proves that, if $\mathtt{t}$ is well-typed within Γ, then there exists a satisfiable constraint C' such that $C', \Gamma \vdash \mathtt{t} : \mathsf{X}$ holds. By the completeness property, we must then have $C' \Vdash [\![\Gamma \vdash \mathtt{t} : \mathsf{X}]\!]$. Since C' is satisfiable, this implies that $[\![\Gamma \vdash \mathtt{t} : \mathsf{X}]\!]$ is satisfiable as well. Conversely, if $[\![\Gamma \vdash \mathtt{t} : \mathsf{X}]\!]$ is satisfiable, then, by the soundness property, $\mathtt{t}$ is well-typed within Γ.

1.7.2 SOLUTION: We must first ensure that R-ADD respects $\sqsubseteq$ (Definition 1.5.4). Since the rule is pure, it is sufficient to establish that $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\hat{k}_1 \hat{+} \hat{k}_2 : \mathsf{T}]\!]$ entails $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\widehat{k_1 + k_2} : \mathsf{T}]\!]$. In fact, we have

$$\begin{array}{lll}
 & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\hat{k}_1 \hat{+} \hat{k}_2 : \mathsf{T}]\!] & \\
\equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ ([\![\hat{k}_1 : \mathsf{int}]\!] \wedge [\![\hat{k}_2 : \mathsf{int}]\!] \wedge \mathsf{int} \leq \mathsf{T}) & \textbf{(1)} \\
\equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ (\mathsf{int} \leq \mathsf{int} \wedge \mathsf{int} \leq \mathsf{int} \wedge \mathsf{int} \leq \mathsf{T}) & \textbf{(2)} \\
\equiv & \mathsf{int} \leq \mathsf{T} & \textbf{(3)} \\
\equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\widehat{k_1 + k_2} : \mathsf{T}]\!] & \textbf{(4)}
\end{array}$$

where (1) and (2) are by Exercise 1.7.1; (3) is by C-IN* and by reflexivity of subtyping; (4) is by Exercise 1.7.1 again.

Second, we must check that if the configuration $\mathsf{c}\ \mathsf{v}_1\ \ldots\ \mathsf{v}_k/\mu$ (where $k \geq 0$) is well-typed, then either it is reducible, or $\mathsf{c}\ \mathsf{v}_1\ \ldots\ \mathsf{v}_k$ 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 $\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![\mathsf{v} : \mathsf{int}]\!]$ is satisfiable. Then, v cannot be a program variable, for a well-typed value must be closed. 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. v cannot be $\hat{+}$ or $\hat{+}\ \mathsf{v}'$, for otherwise $\mathsf{int} \rightarrow \mathsf{int} \rightarrow \mathsf{int} \leq \mathsf{int}$ or $\mathsf{int} \rightarrow \mathsf{int} \leq \mathsf{int}$ would be satisfiable—but the type constructors $\rightarrow$ and int are incompatible. Similarly, v cannot be a λ-abstraction. Thus, 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 $\mathsf{c}\ \mathsf{v}_1\ \ldots\ \mathsf{v}_k/\mu$ is well-typed, then a constraint of the form $\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ (\mathsf{c} \preceq \mathsf{X}_1 \rightarrow \ldots \rightarrow \mathsf{X}_k \rightarrow \mathsf{T} \wedge [\![\mathsf{v}_1 : \mathsf{X}_1]\!] \wedge \ldots \wedge [\![\mathsf{v}_k : \mathsf{X}_k]\!])$ is satisfiable. We now reason by cases on c.

∘ *Case* c is $\hat{k}$. Then, $\Gamma_0(\mathsf{c})$ is int. Because the type constructors int and $\rightarrow$ are incompatible with each other, this implies $k = 0$. Since $\hat{k}$ is a constructor, the expression is a value.

∘ *Case* c is $\hat{+}$. We may assume $k \geq 2$, because otherwise the expression is a value. Then, $\Gamma_0(\mathsf{c})$ is $\mathsf{int} \rightarrow \mathsf{int} \rightarrow \mathsf{int}$, so, by C-Arrow, the above constraint entails $\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ (\mathsf{X}_1 \leq \mathsf{int} \wedge \mathsf{X}_2 \leq \mathsf{int} \wedge [\![\mathsf{v}_1 : \mathsf{X}_1]\!] \wedge [\![\mathsf{v}_2 : \mathsf{X}_2]\!])$, which, by Lemma 1.4.5, entails $\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ ([\![\mathsf{v}_1 : \mathsf{int}]\!] \wedge [\![\mathsf{v}_2 : \mathsf{int}]\!])$. Thus, v_1 and 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.

1.7.5 Solution: We must first ensure that R-Ref, R-Deref and R-Assign respect $\sqsubseteq$ (Definition 1.5.4).

∘ *Case* R-Ref. The reduction is $\mathtt{ref}\ \mathsf{v}/\varnothing \longrightarrow m/(m \mapsto \mathsf{v})$, where $m \notin \mathit{fpi}(\mathsf{v})$ **(1)**. Let T be an arbitrary type. According to Definition 1.5.4, the goal is to show that there exist a set of type variables $\bar{\mathsf{Y}}$ and a store type M' such that $\bar{\mathsf{Y}}\ \#\ \mathit{ftv}(\mathsf{T})$ and $\mathit{ftv}(M') \subseteq \bar{\mathsf{Y}}$ and $\mathit{dom}(M') = \{m\}$ and $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathtt{ref}\ \mathsf{v} : \mathsf{T}]\!]$ entails $\exists\bar{\mathsf{Y}}.\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M'\ \mathsf{in}\ [\![m/(m \mapsto \mathsf{v}) : \mathsf{T}/M']\!]$. Now, we have

$$\begin{array}{rll}
& \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathtt{ref}\ \mathsf{v} : \mathsf{T}]\!] & \\
\equiv & \exists\mathsf{Y}.\mathsf{let}\ \Gamma_0\ \mathsf{in}\ (\mathsf{ref}\ \mathsf{Y} \leq \mathsf{T} \wedge [\![\mathsf{v} : \mathsf{Y}]\!]) & \textbf{(2)} \\
\equiv & \exists\mathsf{Y}.\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M'\ \mathsf{in}\ (m \preceq \mathsf{T} \wedge [\![\mathsf{v} : M'(m)]\!]) & \textbf{(3)} \\
\equiv & \exists\mathsf{Y}.\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M'\ \mathsf{in}\ [\![m/(m \mapsto \mathsf{v}) : \mathsf{T}/M']\!] & \textbf{(4)}
\end{array}$$

where (2) is by Exercise 1.7.1 and by C-InEx; (3) assumes M' is defined as $m \mapsto \mathsf{Y}$, and follows from (1), C-InId and C-In*; and (4) is by definition of constraint generation.

Subcase R-DEREF. The reduction is $!m/(m \mapsto \mathsf{v}) \longrightarrow \mathsf{v}/(m \mapsto \mathsf{v})$. Let T be an arbitrary type and let M be a store type of domain $\{m\}$. We have

$$\begin{array}{lll}
 & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![!m/(m \mapsto \mathsf{v}) : \mathsf{T}/M]\!] & \\
\equiv & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ \exists \mathsf{Y}.(\mathsf{ref}\ M(m) \leq \mathsf{ref}\ \mathsf{Y} \wedge \mathsf{Y} \leq \mathsf{T} \wedge [\![\mathsf{v} : M(m)]\!]) & \textbf{(1)} \\
\equiv & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ \exists \mathsf{Y}.(M(m) = \mathsf{Y} \wedge \mathsf{Y} \leq \mathsf{T} \wedge [\![\mathsf{v} : M(m)]\!]) & \textbf{(2)} \\
\equiv & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ (M(m) \leq \mathsf{T} \wedge [\![\mathsf{v} : M(m)]\!]) & \textbf{(3)} \\
\Vdash & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ ([\![\mathsf{v} : \mathsf{T}]\!] \wedge [\![\mathsf{v} : M(m)]\!]) & \textbf{(4)} \\
\equiv & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![\mathsf{v}/(m \mapsto \mathsf{v}) : \mathsf{T}/M]\!] & \textbf{(5)}
\end{array}$$

where (1) is by Exercise 1.7.1 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 1.4.5 and C-DUP; and (5) is again by definition of constraint generation.

∘ *Case* R-ASSIGN. The reduction is $m := \mathsf{v}/(m \mapsto \mathsf{v}_0) \longrightarrow \mathsf{v}/(m \mapsto \mathsf{v})$. Let T be an arbitrary type and let M be a store type of domain $\{m\}$. We have

$$\begin{array}{lll}
 & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![m := \mathsf{v}/(m \mapsto \mathsf{v}_0) : \mathsf{T}/M]\!] & \\
\Vdash & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![m := \mathsf{v} : \mathsf{T}]\!] & \textbf{(1)} \\
\equiv & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ \exists \mathsf{Z}.(\mathsf{ref}\ M(m) \leq \mathsf{ref}\ \mathsf{Z} \wedge [\![\mathsf{v} : \mathsf{Z}]\!] \wedge \mathsf{Z} \leq \mathsf{T}) & \textbf{(2)} \\
\equiv & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ \exists \mathsf{Z}.(M(m) = \mathsf{Z} \wedge \mathsf{Z} \leq \mathsf{T} \wedge [\![\mathsf{v} : \mathsf{Z}]\!]) & \textbf{(3)} \\
\equiv & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ (M(m) \leq \mathsf{T} \wedge [\![\mathsf{v} : M(m)]\!]) & \textbf{(4)} \\
\Vdash & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![\mathsf{v}/(m \mapsto \mathsf{v}) : \mathsf{T}/M]\!] & \textbf{(5)}
\end{array}$$

where (1) is by definition of constraint generation; (2) is by Exercise 1.7.1 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 $\mathsf{c}\ \mathsf{v}_1\ \ldots\ \mathsf{v}_k/\mu$ (where $k \geq 0$) is well-typed, then either it is reducible, or $\mathsf{c}\ \mathsf{v}_1\ \ldots\ \mathsf{v}_k$ is a value. We only give a sketch of this proof; see the solution to Exercise 1.7.2 for details of a similar proof.

We begin by checking that every value that is well-typed with a type of the form $\mathsf{ref}\ \mathsf{T}$ 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 $\mathsf{c}\ \mathsf{v}_1\ \ldots\ \mathsf{v}_k/\mu$ is well-typed, then a constraint of the form $\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ (\mathsf{c} \preceq \mathsf{X}_1 \rightarrow \ldots \rightarrow \mathsf{X}_k \rightarrow \mathsf{T} \wedge [\![\mathsf{v}_1 : \mathsf{X}_1]\!] \wedge \ldots \wedge [\![\mathsf{v}_k : \mathsf{X}_k]\!])$ 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 $\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ \exists \mathsf{Y}.(\mathsf{ref}\ \mathsf{Y} \rightarrow$

$\mathsf{Y} \leq \mathsf{X}_1 \to \ldots \to \mathsf{X}_k \to \mathsf{T} \wedge [\![\mathsf{v}_1 : \mathsf{X}_1]\!]$), which, by C-ARROW, Lemma 1.4.5, and C-INEX, entails $\exists \mathsf{Y}.\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![\mathsf{v}_1 : \mathsf{ref}\ \mathsf{Y}]\!]$. Thus, v_1 is well-typed with a type of the form $\mathsf{ref}\ \mathsf{Y}$. 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 $\mathtt{ref}\ \ \mathsf{v}_1\ \ldots\ \mathsf{v}_k/\mu$ 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.

1.8.5 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

$$\mathsf{X} \to^{\mathit{Type}} \text{″} \, (\ell^{\star,\mathit{Row}(\varnothing)} : \mathsf{int}^{\mathit{Type}} \,;\, (\mathsf{Y} \to^{\mathit{Row}(\{\ell\})} \partial^{\star,\mathit{Row}(\{\ell\})}\mathsf{X})).$$

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, $\partial\mathsf{int}$.

1.8.27 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 $\to$ 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* $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\{\mathsf{V}; \mathsf{v}\} : \mathsf{T}]\!]$ *is equivalent to* $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \mathsf{Z}_1 \ldots \mathsf{Z}_n \mathsf{Z}.(\bigwedge_{i=1}^{n} [\![\mathsf{V}(\ell_i) : \mathsf{Z}_i]\!] \wedge [\![\mathsf{v} : \mathsf{Z}]\!] \wedge \text{″}\,(\ell_1 : \mathsf{Z}_1; \ldots; \ell_n : \mathsf{Z}_n; \partial\mathsf{Z}) \leq \mathsf{T})$. 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 1.5.4). Because the store is not involved, the goal is to establish that $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathtt{t} : \mathsf{T}]\!]$ entails $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathtt{t}' : \mathsf{T}]\!]$, where $\mathtt{t}$ is the redex and $\mathtt{t}'$ is the reduct.

∘ *Case* R-UPDATE. We have:

$$\begin{array}{lll}
& \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\{\{\mathsf{V}; \mathsf{v}\}\ \mathsf{with}\ \ell = \mathsf{v}'\} : \mathsf{T}]\!] & \\
\equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \mathsf{X}\mathsf{X}'\mathsf{Y}.([\![\{\mathsf{V}; \mathsf{v}\} : \text{″}\,(\ell : \mathsf{X} \,;\, \mathsf{Y})]\!] \wedge [\![\mathsf{v}' : \mathsf{X}']\!] \wedge \text{″}\,(\ell : \mathsf{X}' \,;\, \mathsf{Y}) \leq \mathsf{T}) & \mathbf{(1)} \\
\equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \mathsf{X}\mathsf{X}'\mathsf{Y}\mathsf{Z}_1 \ldots \mathsf{Z}_n \mathsf{Z}.(\bigwedge_{i=1}^{n} [\![\mathsf{V}(\ell_i) : \mathsf{Z}_i]\!] \wedge [\![\mathsf{v} : \mathsf{Z}]\!] & \mathbf{(2)} \\
& \qquad \wedge \text{″}\,(\ell_1 : \mathsf{Z}_1; \ldots; \ell_n : \mathsf{Z}_n; \partial\mathsf{Z}) \leq \text{″}\,(\ell : \mathsf{X} \,;\, \mathsf{Y}) & \\
& \qquad \wedge [\![\mathsf{v}' : \mathsf{X}']\!] \wedge \text{″}\,(\ell : \mathsf{X}' \,;\, \mathsf{Y}) \leq \mathsf{T}) &
\end{array}$$

where (1) is by Exercise 1.7.1, and (2) follows from the preliminary fact and from C-ExAnd, provided $\{\ell_1, \ldots, \ell_n\}$ is the domain of V. We now distinguish two subcases:

Subcase $\ell \in \mathit{dom}(\mathsf{V})$. 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 $(\ell_2 : \mathsf{Z}_2; \ldots; \ell_n : \mathsf{Z}_n; \partial\mathsf{Z}) \leq \mathsf{Y}$, which in turn entails ˝ $(\ell_1 : \mathsf{X}'; \ell_2 : \mathsf{Z}_2; \ldots; \ell_n : \mathsf{Z}_n; \partial\mathsf{Z}) \leq$ ˝ $(\ell : \mathsf{X}' \,;\, \mathsf{Y})$. By transitivity of subtyping, the subconstraint in the second and third lines of (2) entails ˝ $(\ell_1 : \mathsf{X}'; \ell_2 : \mathsf{Z}_2; \ldots; \ell_n : \mathsf{Z}_n; \partial\mathsf{Z}) \leq \mathsf{T}$. By this remark and by C-Ex*, (2) entails

$$\begin{aligned}\mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \mathsf{X}'\mathsf{Z}_2 \ldots \mathsf{Z}_n\mathsf{Z}.(&[\![\mathsf{v}' : \mathsf{X}']\!] \wedge \textstyle\bigwedge_{i=2}^{n} [\![\mathsf{V}(\ell_i) : \mathsf{Z}_i]\!] \wedge [\![\mathsf{v} : \mathsf{Z}]\!] \\ &\wedge \text{˝}\ (\ell_1 : \mathsf{X}'; \ell_2 : \mathsf{Z}_2; \ldots; \ell_n : \mathsf{Z}_n; \partial\mathsf{Z}) \leq \mathsf{T})\end{aligned} \quad \textbf{(3)}$$

which by our preliminary fact is precisely $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\{\mathsf{V}[\ell \mapsto \mathsf{v}']; \mathsf{v}\} : \mathsf{T}]\!]$.

Subcase $\ell \notin \mathit{dom}(\mathsf{V})$. By C-Row-DL and C-Row-LL, the term $(\ell_1 : \mathsf{Z}_1; \ldots; \ell_n : \mathsf{Z}_n; \partial\mathsf{Z})$ may be replaced with $(\ell : \mathsf{Z}; \ell_1 : \mathsf{Z}_1; \ldots; \ell_n : \mathsf{Z}_n; \partial\mathsf{Z})$. Thus, reasoning as in the previous subcase, we find that (2) entails

$$\begin{aligned}\mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \mathsf{X}'\mathsf{Z}_1 \ldots \mathsf{Z}_n\mathsf{Z}.(&[\![\mathsf{v}' : \mathsf{X}']\!] \wedge \textstyle\bigwedge_{i=1}^{n} [\![\mathsf{V}(\ell_i) : \mathsf{Z}_i]\!] \wedge [\![\mathsf{v} : \mathsf{Z}]\!] \\ &\wedge \text{˝}\ (\ell_1 : \mathsf{X}'; \ell_1 : \mathsf{Z}_1; \ldots; \ell_n : \mathsf{Z}_n; \partial\mathsf{Z}) \leq \mathsf{T})\end{aligned} \quad \textbf{(4)}$$

which by our preliminary fact is precisely $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\{\mathsf{V}[\ell \mapsto \mathsf{v}']; \mathsf{v}\} : \mathsf{T}]\!]$.

○ *Cases* R-Access-1, R-Access-2. We have:

$$\begin{aligned} & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\{\mathsf{V}; \mathsf{v}\}.\{\ell\} : \mathsf{T}]\!] \\ \equiv\ & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \mathsf{XY}.([\![\{\mathsf{V}; \mathsf{v}\} : \text{˝}\ (\ell : \mathsf{X} \,;\, \mathsf{Y})]\!] \wedge \mathsf{X} \leq \mathsf{T}) & \textbf{(1)} \\ \equiv\ & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \mathsf{XYZ}_1 \ldots \mathsf{Z}_n\mathsf{Z}.(\textstyle\bigwedge_{i=1}^{n} [\![\mathsf{V}(\ell_i) : \mathsf{Z}_i]\!] \wedge [\![\mathsf{v} : \mathsf{Z}]\!] & \textbf{(2)} \\ & \qquad \wedge \text{˝}\ (\ell_1 : \mathsf{Z}_1; \ldots; \ell_n : \mathsf{Z}_n; \partial\mathsf{Z}) \leq \text{˝}\ (\ell : \mathsf{X} \,;\, \mathsf{Y}) \\ & \qquad \wedge \mathsf{X} \leq \mathsf{T}) \end{aligned}$$

where (1) is by Exercise 1.7.1, and (2) follows from the preliminary fact and from C-ExAnd, provided $\{\ell_1, \ldots, \ell_n\}$ is the domain of V. We now distinguish two subcases:

Subcase $\ell \in \mathit{dom}(\mathsf{V})$, *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 $\mathsf{Z}_1 \leq \mathsf{X}$. By transitivity of subtyping, by Lemma 1.4.5, and by C-Ex*, we find that (2) entails $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathsf{V}(\ell) : \mathsf{T}]\!]$.

Subcase $\ell \notin \mathit{dom}(\mathsf{V})$, *i.e.*, (R-Access-2). By C-Row-DL and C-Row-LL, the term $(\ell_1 : \mathsf{Z}_1; \ldots; \ell_n : \mathsf{Z}_n; \partial\mathsf{Z})$ may be replaced with $(\ell : \mathsf{Z}; \ell_1 : \mathsf{Z}_1; \ldots; \ell_n : \mathsf{Z}_n; \partial\mathsf{Z})$. Thus, reasoning as in the previous subcase, we find that (2) entails $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathsf{v} : \mathsf{T}]\!]$.

Before attacking the proof of the progress property, let us briefly check that every value v that is well-typed with type ˝ T must be a record value, that is, must be of the form $\{\mathsf{V}; \mathsf{w}\}$. Indeed, assume that $\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![\mathsf{v} : ″\ \mathsf{T}]\!]$ is satisfiable. Then, v cannot be a program variable, for a well-typed value must be closed. Furthermore, v cannot be a memory location m, because $\mathsf{ref}\ M(m) \leq ″\ \mathsf{T}$ is unsatisfiable: indeed, the type constructors ref and $″$ are incompatible (recall that $″$ is isolated). Similarly, v cannot be a partially applied constant or a λ-abstraction, because $\mathsf{T}' \to \mathsf{T}'' \leq ″\ \mathsf{T}$ is unsatisfiable. Thus, v must be a fully applied constructor. Since the only constructors in the language are the record constructors $\{\}_L$, 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 $\mathsf{c}\ \mathsf{v}_1\ \ldots\ \mathsf{v}_k/\mu$ is is well-typed, then either it is reducible, or $\mathsf{c}\ \mathsf{v}_1\ \ldots\ \mathsf{v}_k$ is a value. By the well-typedness hypothesis, a constraint of the form $\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![\mathsf{c}\ \mathsf{v}_1\ \ldots\ \mathsf{v}_k : \mathsf{T}]\!]$ is satisfiable.

∘ *Case* c is $\{\}_L$. If k is less than or equal to $n + 1$, where n is the cardinal of L, then $\mathsf{c}\ \mathsf{v}_1\ \ldots\ \mathsf{v}_k$ 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* c is $\{\cdot\ \mathsf{with}\ \ell = \cdot\}$. Analogous to the next case.

∘ *Case* c is $\cdot.\{\ell\}$. If $k = 0$, then $\mathsf{c}\ \mathsf{v}_1\ \ldots\ \mathsf{v}_k$ is a value. Assume $k \geq 1$. Then, the constraint $\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![\mathsf{c}\ \mathsf{v}_1 : \mathsf{T}]\!]$ is satisfiable. By Exercise 1.7.1, this implies that $\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![\mathsf{v}_1 : ″\ (\ell : \mathsf{X}\ ;\ \mathsf{Y})]\!]$ is satisfiable. Thus, v_1 must be a record value, and the configuration is reducible by R-ACCESS-1 or R-ACCESS-2.

1.8.33 SOLUTION: To make extension strict, it suffices to restrict its binding in the initial environment Γ_0, as follows:

$$\langle \cdot\ \mathsf{with}\ \ell = \cdot \rangle : \forall \mathsf{XY}.″\ (\ell : \mathsf{abs}\ ;\ \mathsf{Y}) \to \mathsf{X} \to ″\ (\ell : \mathsf{pre}\ \mathsf{X}\ ;\ \mathsf{Y}).$$

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:

$$\cdot \setminus \langle \ell \rangle : \forall \mathsf{XY}.″\ (\ell : \mathsf{X}\ ;\ \mathsf{Y}) \to ″\ (\ell : \mathsf{abs}\ ;\ \mathsf{Y})$$

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 1.8.32). Strict restriction, which requires the field to be present in the input record, may be assigned the following type scheme:

$$\cdot \setminus \langle \ell \rangle : \forall \mathsf{XY}.″\ (\ell : \mathsf{pre}\ \mathsf{X}\ ;\ \mathsf{Y}) \to ″\ (\ell : \mathsf{abs}\ ;\ \mathsf{Y})$$

1.8.34 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 $\Pi\,(\ell : \mathsf{pre}\ \mathsf{T}\ ;\ \mathsf{T}')$ to a function whose domain type is $\Pi\,(\ell : \mathsf{abs}\ ;\ \mathsf{T}')$," or, more generally, as "writing a program whose well-typedness requires some constraint of the form $\Pi\,(\ell{:}\mathsf{pre}\ \mathsf{T}\ ;\ \mathsf{T}') \leq \Pi\,(\ell{:}\mathsf{abs}\ ;\ \mathsf{T}')$ 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 $\mathsf{pre}\ \mathsf{T} \leq \mathsf{abs} \equiv \mathsf{true}$ 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 $\{\ell_1 : \mathsf{T}_1; \ldots; \ell_n : \mathsf{T}_n\}$. Let us forget about the types $\mathsf{T}_1, \ldots, \mathsf{T}_n$, 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 $\mathsf{T}_1, \ldots, \mathsf{T}_n$, 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).

1.3.2 SOLUTION: Our hypotheses are $\mathsf{C}, \Gamma \vdash \mathsf{t} : \forall\bar{\mathsf{X}}[\mathsf{D}].\mathsf{T}$ **(1)** and $\mathsf{C} \Vdash [\vec{\mathsf{X}} \mapsto \vec{\mathsf{T}}]\mathsf{D}$ **(2)**. We may also assume, *w.l.o.g.*, $\bar{\mathsf{X}}\ \#\ \mathit{ftv}(\mathsf{C}, \Gamma, \vec{\mathsf{T}})$ **(3)**. By HMX-INST and (1), we have $\mathsf{C}\wedge\mathsf{D}, \Gamma \vdash \mathsf{t} : \mathsf{T}$, which by Lemma 1.3.1 yields $\mathsf{C}\wedge\mathsf{D}\wedge\vec{\mathsf{X}} = \vec{\mathsf{T}}, \Gamma \vdash \mathsf{t} : \mathsf{T}$ **(4)**. Now, we claim that $\vec{\mathsf{X}} = \vec{\mathsf{T}} \Vdash \mathsf{T} \leq [\vec{\mathsf{X}} \mapsto \vec{\mathsf{T}}]\mathsf{T}$ **(5)** holds; the proof appears in the next paragraph. Applying HMX-SUB to (4) and to (5), we obtain $\mathsf{C} \wedge \mathsf{D} \wedge \vec{\mathsf{X}} = \vec{\mathsf{T}}, \Gamma \vdash \mathsf{t} : [\vec{\mathsf{X}} \mapsto \vec{\mathsf{T}}]\mathsf{T}$ **(6)**. By C-EQ and by (2), we have $\mathsf{C} \wedge \vec{\mathsf{X}} = \vec{\mathsf{T}} \Vdash \mathsf{D}$, so (6) may be written $\mathsf{C} \wedge \vec{\mathsf{X}} = \vec{\mathsf{T}}, \Gamma \vdash \mathsf{t} : [\vec{\mathsf{X}} \mapsto \vec{\mathsf{T}}]\mathsf{T}$ **(7)**. Last, (3) implies $\bar{\mathsf{X}}\ \#\ \mathit{ftv}(\Gamma, [\vec{\mathsf{X}} \mapsto \vec{\mathsf{T}}]\mathsf{T})$ **(8)**. Applying rule HMX-EXISTS to (7) and (8), we get $\exists\bar{\mathsf{X}}.(\mathsf{C}\wedge\vec{\mathsf{X}} = \vec{\mathsf{T}}), \Gamma \vdash \mathsf{t} : [\vec{\mathsf{X}} \mapsto \vec{\mathsf{T}}]\mathsf{T}$ **(9)**. By C-NAMEEQ and by (3), $\exists\bar{\mathsf{X}}.(\mathsf{C}\wedge\vec{\mathsf{X}} = \vec{\mathsf{T}})$ is equivalent to C, hence (9) is the goal $\mathsf{C}, \Gamma \vdash \mathsf{t} : [\vec{\mathsf{X}} \mapsto \vec{\mathsf{T}}]\mathsf{T}$.

There now remains to establish (5). One possible proof method is to unfold the definition of $\Vdash$ and reason by structural induction on T. Here is another, axiomatic approach. Let Z be fresh for T, $\vec{\mathsf{X}}$, and $\vec{\mathsf{T}}$. By reflexivity of subtyping and by C-EXTRANS, we have $\mathsf{true} \equiv \mathsf{T} \leq \mathsf{T} \equiv \exists\mathsf{Z}.(\mathsf{T} \leq \mathsf{Z} \wedge \mathsf{Z} \leq \mathsf{T})$, which by congruence of $\equiv$ and by C-EXAND implies $\vec{\mathsf{X}} = \vec{\mathsf{T}} \equiv \exists\mathsf{Z}.(\mathsf{T} \leq \mathsf{Z} \wedge \vec{\mathsf{X}} = \vec{\mathsf{T}} \wedge \mathsf{Z} \leq \mathsf{T})$ **(10)**. Furthermore, by C-EQ, we have $(\vec{\mathsf{X}} = \vec{\mathsf{T}} \wedge \mathsf{Z} \leq \mathsf{T}) \equiv (\vec{\mathsf{X}} = \vec{\mathsf{T}} \wedge \mathsf{Z} \leq [\vec{\mathsf{X}} \mapsto \vec{\mathsf{T}}]\mathsf{T}) \Vdash (\mathsf{Z} \leq [\vec{\mathsf{X}} \mapsto \vec{\mathsf{T}}]\mathsf{T})$ **(11)**. Combining (10) and (11) yields

$\vec{X} = \vec{T} \Vdash \exists Z.(T \leq Z \wedge Z \leq [\vec{X} \mapsto \vec{T}]T)$, which by C-ExTrans may be read $\vec{X} = \vec{T} \Vdash T \leq [\vec{X} \mapsto \vec{T}]T$.

1.3.3 Solution: The simplest possible derivation of $\mathsf{true}, \varnothing \vdash \lambda z.z : \mathsf{int} \to \mathsf{int}$ is syntax-directed. It closely resembles the Damas-Milner derivation given in Exercise 1.1.22.

$$\dfrac{\dfrac{}{\mathsf{true}, z : \mathsf{int} \vdash z : \mathsf{int}}\ \textsc{hmx-Var}}{\mathsf{true}, \varnothing \vdash \lambda z.z : \mathsf{int} \to \mathsf{int}}\ \textsc{hmx-Abs}$$

As in Exercise 1.1.22, we may use a type variable X instead of the type int, then employ hmx-Gen to quantify universally over X.

$$\dfrac{\dfrac{\dfrac{}{\mathsf{true}, z : X \vdash z : X}\ \textsc{hmx-Var}}{\mathsf{true}, \varnothing \vdash \lambda z.z : X \to X}\ \textsc{hmx-Abs} \qquad X \# \mathit{ftv}(\mathsf{true}, \varnothing)}{\mathsf{true}, \varnothing \vdash \lambda z.z : \forall X[\mathsf{true}].X \to X}\ \textsc{hmx-Gen}$$

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:

$$\dfrac{\mathsf{true}, \varnothing \vdash \lambda z.z : \forall X[\mathsf{true}].X \to X \qquad \mathsf{true} \Vdash [X \mapsto \mathsf{int}]\mathsf{true}}{\mathsf{true}, \varnothing \vdash \lambda z.z : \mathsf{int} \to \mathsf{int}}\ \textsc{hmx-Inst}'$$

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 1.3.1 and the solution of Exercise 1.3.2 are constructive, it is possible to exhibit the HM(X) derivation that underlies it. We find:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{X = \mathsf{int}, z : X \vdash z : X}\ \textsc{hmx-Var}}{X = \mathsf{int}, \varnothing \vdash \lambda z.z : X \to X}\ \textsc{hmx-Abs}}{X = \mathsf{int}, \varnothing \vdash \lambda z.z : \forall X.X \to X}\ \textsc{hmx-Gen}}{X = \mathsf{int}, \varnothing \vdash \lambda z.z : X \to X}\ \textsc{hmx-Inst} \quad X = \mathsf{int} \Vdash X \to X \leq \mathsf{int} \to \mathsf{int}}{X = \mathsf{int}, \varnothing \vdash \lambda z.z : \mathsf{int} \to \mathsf{int}}\ \textsc{hmx-Sub}}{\exists X.(X = \mathsf{int}), \varnothing \vdash \lambda z.z : \mathsf{int} \to \mathsf{int}}\ \textsc{hmx-Exists}$$

Since $\exists X.(X = \mathsf{int})$ is equivalent to true, the conclusion is indeed the desired judgement.

1.7.1 SOLUTION: We have

$$\begin{array}{lll}
 & \text{let } \Gamma_0 \text{ in } [\![\mathtt{c}\ \mathtt{t}_1\ \ldots\ \mathtt{t}_n : \mathsf{T}']\!] & \\
\equiv & \text{let } \Gamma_0 \text{ in } \exists \mathsf{Z}_1 \ldots \mathsf{Z}_n.(\bigwedge_{i=1}^{n} [\![\mathtt{t}_i : \mathsf{Z}_i]\!] \wedge \mathtt{c} \preceq \mathsf{Z}_1 \rightarrow \ldots \rightarrow \mathsf{Z}_n \rightarrow \mathsf{T}') & \textbf{(1)} \\
\equiv & \text{let } \Gamma_0 \text{ in } \exists \mathsf{Z}_1 \ldots \mathsf{Z}_n \bar{\mathsf{X}}.(\bigwedge_{i=1}^{n} [\![\mathtt{t}_i : \mathsf{Z}_i]\!] & \textbf{(2)} \\
 & \qquad \wedge\ \mathsf{T}_1 \rightarrow \ldots \rightarrow \mathsf{T}_n \rightarrow \mathsf{T} \leq \mathsf{Z}_1 \rightarrow \ldots \rightarrow \mathsf{Z}_n \rightarrow \mathsf{T}') & \\
\equiv & \text{let } \Gamma_0 \text{ in } \exists \bar{\mathsf{X}}.(\bigwedge_{i=1}^{n} [\![\mathtt{t}_i : \mathsf{T}_i]\!] \wedge \mathsf{T} \leq \mathsf{T}') & \textbf{(3)}
\end{array}$$

where (1) is by definition of constraint generation; (2) is by C-INID; (3) is by C-ARROW, C-EXAND, and by Lemma 1.4.6.

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.

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.

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.

Davide Ancona and Elena Zucca. A theory of mixin modules: Basic and derived operators. *Mathematical Structures in Computer Science*, 8(4):401–446, August 1998.

Davide Ancona and Elena Zucca. A calculus of module systems. *Journal of Functional Programming*, 12(2):91–132, March 2002.

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`.

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.

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.

Gilad Bracha and William R. Cook. Mixin-based inheritance. In *Conference on Object-Oriented Programming Languages, Systems, and Applications / European Conference on Object-Oriented Programming*, pages 303–311, Ottawa, Canada, October 1990.

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, 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.

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.

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.

Luca Cardelli. Phase distinctions in type theory. unpublished manuscript, 1988.

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 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.

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.

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.

Hubert Comon. Constraints in term algebras (short survey). In *Conference on Algebraic Methodology and Software Technology (AMAST)*, Workshops in Computing. Springer-Verlag, 1993.

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, 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.

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.

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.

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.

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.

Kathleen Fisher and John H. Reppy. The design of a class mechanism for moby. In *Proc. of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 37–49, Atlanta, Georgia, May 1999.

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.

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.

Giorgio Ghelli and Benjamin Pierce. Bounded existentials and minimal typing. *Theoretical Computer Science*, 193:75–96, 1998. Circulated in manuscript form in 1992.

Jean-Yves Girard. *Interprétation Fonctionnelle et Élimination des Coupures dans l'Arithmétique d'Ordre Supérieure*. PhD thesis, Université Paris VII, 1972.

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 2002.

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 2001.

Jr. Guy L. Steele. *Common Lisp the Language*. Digital Press, 1990.

Thérèse Hardin, Luc Maranget, and Bruno Pagano. Functional runtimes within the lambda-sigma calculus. *Journal of Functional Programming*, 8(2), March 1998.

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 1990.

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, 2000.

Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, 1993.

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.

Gérard Huet. *Résolution d'equations dans les langages d'ordre 1,2, ...,ω*. Thèse de Doctorat d'Etat, Université de Paris 7 (France), 1976.

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*.

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.

Trevor Jim. What are principal typings and what are they good for? In ACM, editor, *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, pages 42–53, 1996.

Trevor Jim and Jens Palsberg. Type inference in systems of recursive types with subtyping. Manuscript, 1999.

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.

Assaf J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. The undecidability of the semi-unification problem. *Information and Computation*, 102(1):83–101, January 1993. 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.

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.

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.

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.

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 1994.

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 1995.

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. 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.

Barbara Liskov. A history of CLU. *ACM SIGPLAN Notices*, 28(3):133–147, 1993.

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.

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.

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.

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.

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, 1997.

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 1991.

John C. Mitchell and Gordon Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988a.

John C. Mitchell and Gordon D. Plotkin. Abstract types have existential types. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502, 1988b. Summary in *ACM Symposium on Principles of Programming Languages (POPL), New Orleans, Louisiana*, 1985.

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.

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, 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`.

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.

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, 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.

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.

Jens Palsberg. Efficient inference of object types. *Information and Computation*, 123(2): 198–209, 1995.

Jens Palsberg, Mitchell Wand, and Patrick M. O'Keefe. Type inference with non-structural subtyping. *Formal Aspects of Computing*, 9:49–67, 1997.

David Parnas. The criteria to be used in decomposing systems into modules. *Communications of the ACM*, 14(1):221–227, 1972.

Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, April 2003a.

Simon Peyton Jones. Special issue: Haskell 98 language and libraries. *Journal of Functional Programming*, 13, January 2003b.

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.

Jakob Rehof. Minimal typings in atomic subtyping. In *ACM Symposium on Principles of Programming Languages (POPL), Paris, France*, pages 278–291, January 1997.

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.

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, 1974.

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.

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.

Amr Sabry. What is a purely functional language? *Journal of Functional Programming*, 8(1):1–22, January 1998.

Peter Sestoft. Moscow ml, 2003. URL `http://www.dina.dk/~sestoft/mosml.html`.

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.

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 $\mathsf{HM}(X)$. In *Workshop on Types in Programming (TIP'02)*, volume 75 of *Electronic Notes in Theoretical Computer Science*, July 2002.

Geoffrey S. Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23(2–3):197–226, December 1994.

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 *Twenty SeventhACM Symposium on Principles of Programming Languages (POPL)*, pages 214–227, Boston, January 2000.

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.

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.

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.

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 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.

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.

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 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.

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.

J. B. Wells. The essence of principal typings. In *International Colloquium on Automata, Languages and Programming*, volume 2380 of *Lecture Notes in Computer Science*, pages 913–925. Springer-Verlag, 2002.

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.

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 1994.