Semantics of Types for Mutable State

Amal Jamil Ahmed

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
By the Department of
Computer Science

November 2004

Abstract

Proof-carrying code (PCC) is a framework for mechanically verifying the safety of machine language programs. A program that is successfully verified by a PCC system is guaranteed to be safe to execute, but this safety guarantee is contingent upon the correctness of various trusted components. For instance, in traditional PCC systems the trusted computing base includes a large set of low-level typing rules. Foundational PCC systems seek to minimize the size of the trusted computing base. In particular, they eliminate the need to trust complex, low-level type systems by providing machine-checkable proofs of type soundness for real machine languages.

In this thesis, I demonstrate the use of logical relations for proving the soundness of type systems for mutable state. Specifically, I focus on type systems that ensure the safe allocation, update, and reuse of memory. For each type in the language, I define logical relations that explain the meaning of the type in terms of the operational semantics of the language. Using this model of types, I prove each typing rule as a lemma.

The major contribution is a model of System F with general references — that is, mutable cells that can hold values of any closed type including other references, functions, recursive types, and impredicative quantified types. The model is based on ideas from both possible worlds and the indexed model of Appel and McAllester.

I show how the model of mutable references is encoded in higher-order logic. I also show how to construct an indexed possible-worlds model for a von Neumann machine. The latter is used in the Princeton Foundational PCC system to prove type safety for a full-fledged low-level typed assembly language. Finally, I present a semantic model for a region calculus that supports type-invariant references as well as memory reuse.

Acknowledgments

My work was funded by DARPA grant F30602-99-1-0519, NSF grant DGE-9972930, and by the George Van Ness Lothrop Fellowship in Engineering for 2002-2003, a Princeton University honorific fellowship.

First and foremost I would like to thank my advisor, Andrew Appel, for his encouragement, support, and guidance throughout my graduate career. I am especially grateful for his energy and contagious enthusiasm which always made research a pleasure. His door was always open, and his insights immensely valuable. I am indebted to him for all that he has taught me, both technical and not.

Thanks also to my thesis readers, David Walker and Peter O'Hearn, for all the time spent reading my thesis and for many helpful comments. David Walker taught me a lot about logic and was both a mentor and friend. His influence on my work and research interests is and, I expect, will remain evident. Peter O'Hearn is a pool of immense wisdom and insight when it comes to research on mutable state. I consider myself extremely fortunate to have had the benefit of his intuition. It has undoubtedly helped me improve the quality of the work described in this thesis.

I've had the privilege of working with many talented people at Princeton. I have learnt a great deal during my discussions with Juan Chen, Limin Jia, Neophytos Michael, Xinming Ou, Chris Richards, Kedar Swadi, Gang Tan, and Dinghao Wu. I would like to thank Roberto Virga and Xinming Ou for their contributions to this thesis in terms of proof implementation.

Many people provided support and friendship during my time in graduate school. I would especially like to thank Patrick Min and Iannis Tourlakis who made Office 417 a happy, fun, and special place. Thanks also to all the other grad students who made my stay at the department more enjoyable, especially (in no particular order)

Allison Klein, Sanjeev Kumar, Rudro Samanta, George Tzanetakis, Dan Dantas, Jay Ligatti, and Georg Essl.

A special thanks to Melissa Lawson for taking care of countless numbers of things without a hitch, including all the last-minute requests that I've dumped at her door. Thanks also to the administrative and technical staff at the department who were always friendly, always helpful, and amazing at making sure that everything ran smoothly.

I would like to thank my parents for their endless love and support. Without a doubt, I wouldn't be here today if it weren't for their belief that their daughter's aspirations need never be different from their son's. This was certainly unorthodox thinking for the part of the world I grew up in. Thanks also to my brother, Omar, who has been a very important part of my life. I am especially grateful to him for the interest and patience with which he has listened to my rambling and complaints, and then known just what to say.

This thesis is dedicated to my father who passed away a few months before I started at Princeton. His absence makes reaching this milestone bittersweet. But it is comforting to know that he would have been happy and proud.

Finally, I would like to thank Ijlal, my husband and soulmate, for more love, encouragement, and support than I could have ever thought possible.

For my Dadoo.

Contents

List of Figures

Chapter 1

Introduction

This thesis investigates the use of logical relations for proving the soundness of type systems for mutable state. In particular, it focuses on type systems that ensure the safe allocation, update, and reuse of a computer's memory. For each type in the language, I shall define logical relations that explain the meaning of the type in terms of the operational semantics of the language. This proof technique can be used to build machine-checkable safety proofs for real assembly languages. Such proofs are crucial for building flexible and secure systems for mechanically verifying the safety of machine-language programs.

1.1 Mechanically Verifying Safety

Over the past decade, it has become increasingly common for networked devices to receive and execute programs from various sources. Personal computers, PDAs, and mobile phones can download and run new applications. Web browsers download Java applets that provide customized user interfaces. Servers upload Java servlets so that data-intensive computation can be performed closer to the repository of the data. Scientists can upload software to satellites and space stations. And PC owners can donate idle time on their PCs to projects that use distributed computing to search for extraterrestrial intelligence,[1] or to predict the Earth's climate fifty years from now.[2] (The advent of *grid computing* [FKT01, Par02] has made it easier to donate CPU cycles; rather than sign up with specific projects, we can simply add our computers to a computational grid that pools together resources to be shared by a number of projects.)

In each of the above scenarios, we may not always know or may not completely trust the producer of the code that we download and run. Nonetheless, we would

[1] `http://setiathome.berkeley.edu/`

[2] `http://www.climateprediction.net/`

like to be sure that the downloaded code will not *misbehave* — for instance, a tax calculator should correctly calculate my tax liability or refund, or at the very least, it should not crash my PC or make it hang, or corrupt my financial data or leak it to others. Downloaded code may misbehave thanks to a malicious producer or simply due to bugs that went undetected despite the best intentions and efforts of its producer. Therefore, regardless of who the producer is, it is desirable to *mechanically check* that the behavior of the code matches our expectations. For that we need two things:

- First, we have to specify precisely what we expect.
- Second, we need a mechanism that guarantees that the expectations are met.

Section 1.1.1 discusses a particularly effective way accomplishing both of the above. But first, I shall digress and talk about the degrees and types of well-behavedness that one might expect from (downloaded) code.

Correctness: Ideally, we would like be sure that the code we've received is *correct* — that is, it does the *right* thing and computes what we *expect*. To verify that the code is correct, we must formally specify what we expect. This is possible for very small programs but for large, complex software, it can be a daunting and error-prone task. Consider, for example, the tax calculator I mentioned above. To verify that this program correctly computes the 2003 tax liability for U.S. residents, one would have to formally encode the entire 2003 U.S. Individual Income Tax Code. Even after one has such an encoding, there is no guarantee that the encoding is correct. The problem is that to formally specify what we expect a program to compute, we must essentially write the program all over again, though perhaps in a more deductive style. Such a task is as prone to errors as the task of writing the original software.

Security: For most applications it is usually sufficient to prove weaker properties about the code's behavior than partial or total correctness. We may, for instance, be content with a guarantee that the downloaded code is *secure*. Unfortunately, there is no universally accepted definition of security, so we must be more specific about what security policies we want the code to satisfy. Some well-known concerns addressed by security policies are access control, information flow, and availability. *Access control* policies limit *who* can perform *what* actions on *which* objects. For instance, we may have an access control policy that says if Alice runs the tax calculation program (using security terminology, Alice is the *principal*), then the program may read, but not write, Alice's financial files, but it may neither read nor write Bob's financial files. *Information flow* policies restrict what principals can infer about objects by observing system behavior. Specifically, *secrecy* guarantees that low-security (observable) data is not influenced by high-security (secret) data, while *integrity* guarantees that high-security (trustworthy) data is not influenced, or tainted, by low-security (unreliable) data. *Availability* policies restrict principals from denying others the use of precious resources such as memory and CPU cycles.

Schneider [Sch00a] defined a security policy as a binary partition on sets of executions, that is, as a predicate $\mathcal{P}$ that divides the set of executions into those that satisfy the policy $\mathcal{P}$ and those that do not. Security policies may be classified based on the kinds of mechanisms that may be used to enforce them. Broadly speaking, enforcement mechanisms may place restrictions on what the code is allowed to do (perhaps even by modifying the code), or simply check that the the code does not violate the security policy.

Schneider specified a class of enforcement mechanisms called EM for *execution monitoring*. An execution monitor intercepts security relevant events that occur as the untrusted program executes, and intervenes (terminates the program [Sch00a] or takes some corrective action [HMS03]) upon seeing an event that would lead to a violation of the security policy being enforced. Hamlen, Morrisett, and Schneider [HMS03] present a taxonomy of enforceable security policies that shows how classes of security policies enforceable by static analysis [LY99, MWCG99, Mye99, Nac97], execution monitoring [And72, Lam71, LY99, RC91, Vis00], and program rewriting [DG71, ES00a, ES00b, ET99, Sma97, WLAG93] relate to each other and to various computational complexity classes.

Notice that access control policies are EM-enforceable; an execution monitor can terminate a program that is about to perform an operation it is not authorized to. Information flow policies are enforceable by program rewriting, but they are not EM-enforceable. Informally, a set of executions satisfies an information flow policy $\mathcal{P}$ if all the executions in that set are observably equivalent — that is, one cannot look at a single execution and decide whether it belongs to $\mathcal{P}$. Execution monitors are predicates on individual executions, hence, they cannot enforce information flow policies. Program rewriters, meanwhile, can modify programs that violate the policy being enforced such that the observable outputs that cause the violation are suppressed.

Safety: At a bare minimum, we would like to have the assurance that the code we are about to execute is *safe*. Lamport [Lam77] described a *safety property* as one that stipulates that no "bad thing" happens during any execution.[3] This implies that all EM-enforceable security policies are actually safety properties. Well-known examples of "bad" things (i.e., violations of safety properties) include dereferencing a dangling pointer and using an integer as a pointer. The next section discusses type systems, which are particularly well-suited to specifying and enforcing safety properties.

[3] Also, Lamport [Lam77] describes a *liveness property* as one that guarantees that some "good thing" must happen during any execution. Availability, which says that *eventually* a resource will become available, is an example of a liveness property.

1.1.1 Type Systems

A *type system* is a collection of syntactic rules that specifies the set of acceptable program behaviors. A *type checker* verifies that a program obeys the rules of a particular type system. It does so by checking that types are assigned correctly and consistently to all program subexpressions. Type checking guarantees that a program is well-typed which, in turn, guarantees that the program is well-behaved (as long as the type system correctly rules out "bad" behavior).

Cardelli [Car97] defined *type safety* in very general terms as the property that programs do not cause untrapped errors (i.e., execution errors that are not immediately detected). Let us look at some specific examples of properties a type system can enforce. Type systems typically guarantee a number of interrelated safety properties, for instance, *memory safety* (programs can only access appropriate memory locations) and *control safety* (programs can only transfer control to appropriate program points). Type systems that support *type abstraction* can provide some powerful assurances about program behavior. Such systems can guarantee that the behavior of a program is independent of the representation of an abstract type [Rey83]. One consequence of this *representation independence* (or *parametricity*) principle is that type checking can be modular — that is, one can swap the implementation of an abstraction with another well-typed one, without having to type check the entire program. Another important consequence is that simply by examining the type of a function, it is possible (in a purely functional setting) to infer a number of its computational properties [Wad89]. Hence, just by assigning an appropriate type to a function, it is possible to ensure that the function does not perform any unauthorized operations on values of abstract type. Types also make it easy to construct logical relations that can be used to verify adherence to policies that rely on some form of observational equivalence, for instance, encryption protocols [SP01] and secure information flow [HR98].

There is a trade-off between the expressive power of a type system and automated, efficient type checking. While it is possible to construct arbitrarily expressive type systems with the power of any logic, the downside is that such type systems (e.g., NuPRL [CAB$^+$86]) generally require sophisticated theorem provers and programmer guidance to construct a proof of type safety. For example, recent work on dependent type systems [XP98, XP99] extends type checking to include the verification of value-range restrictions which allows many array bounds checks to be eliminated, but the programmer must add additional typing annotations (e.g., loop invariants) to aid the type checker. Notice, however, that while a code producer must do additional work to prove the program safe, there is no added burden on the code receiver, as long as the code she receives is accompanied by enough typing annotations (or perhaps even a full-fledged proof of type-safety in an appropriate logic) for the checker to mechanically verify that the code is safe.

In practice, it is common to restrict attention to type systems for which checking is computable within a reasonable complexity bound so that code producers don't have to do additional work to prove type safety. Such type systems give us an effective way of specifying what we expect, while type checking allows us to mechanically verify that the expectations are met.

1.1.2 Type-Preserving Compilation

The programs we download are typically machine language programs rather than high-level source code with type annotations. Fortunately, there has been a great deal of research over the past decade on how to propagate type information present in source language programs into compiler intermediate languages [TMC+96, SA95, PJHH+93, LY99, DMTW97]. The fruits of this research, namely *type-preserving compilers* and *typed intermediate languages*, have made it possible for type systems to be used for mechanically verifying the safety of programs written in low-level languages. At each stage of the compilation process, a type-preserving compiler must correctly propagate typing information to subsequent stages. Type-preserving compilers are also *type-directed*, that is, at each stage of compilation, they may use typing information to guide program transformations or optimizations. The output of each compilation phase is a program in an appropriate typed intermediate language.

Type-checkers for compiler intermediate languages are important for two reasons. First, they allow compiler writers to (mechanically) detect bugs in their compilers. Specifically, if a progam type-checks before but not after an optimization, that signals an error in the compiler. Second, they allow code receivers to mechanically verify the safety of lower level code, a feature that's successfully exploited in the Java Virtual Machine [LY99].

The Java platform consists of a type-preserving compiler that translates a Java program into a Java Virtual Machine Language (JVML) program (Java bytecode) that may be shipped to a code receiver. The code receiver type checks the untrusted JVML program (using the Java bytecode verifier) before running it. If the program is well-typed the code receiver can be sure that the JVML program is safe to execute, as long as the JVML type system is sound and the JVML type checker and run-time system (which includes a just-in-time compiler or a JVML interpreter) are free of bugs.

In their work on Typed Assembly Language (TAL) [MWCG98, MWCG99], Morrisett *et al.* showed that types could be propagated all the way to the target language. Specifically, their compiler produces assembly language programs with type annotations for the Intel IA32 architecture [MCG+99]. A type checker verifies that these programs are well-typed with respect to the type annotations before assembling them. The TAL program is safe to execute as long as the type checker and

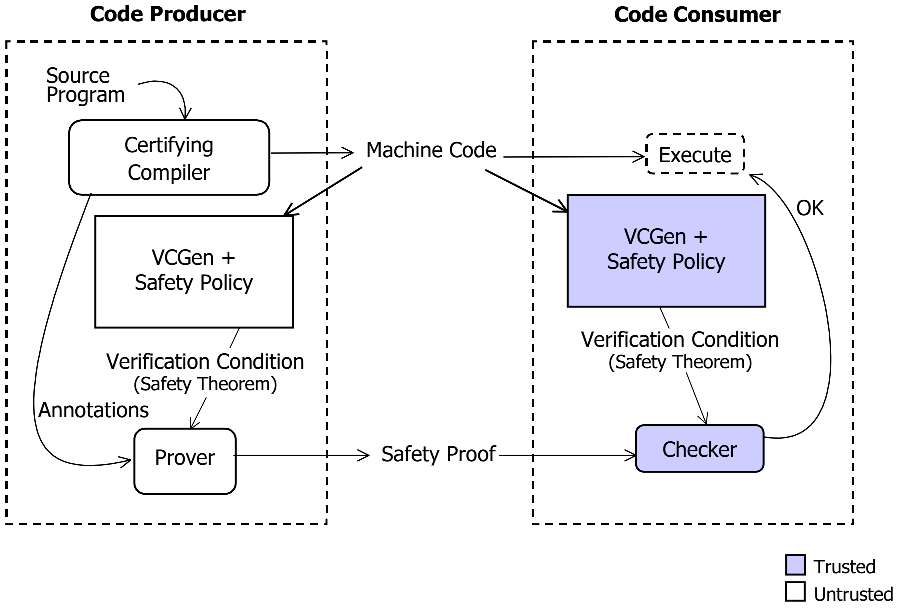


Figure 1.1: Proof-Carrying Code Architecture

assembler have been implemented correctly.

1.1.3 Proof-Carrying Code

Proof-carrying code [Nec97, NL98a] goes one step further than TAL by generating proofs of safety for machine language programs. Figure 1.1 gives a simplified illustration of the PCC architecture as described by Necula [Nec97]. In this architecture, the code producer compiles her source program using a *certifying compiler* (essentially a type-preserving compiler) which produces machine code together with a set of type annotations. The machine code is then sent to a Verification Condition Generator (VCGen) which consults the safety policy that must be satisfied and infers the safety theorem for the given machine code — that is, the theorem that must be proved in order to show that the code adheres to the safety policy. A prover, using the type annotations generated by the compiler as hints, generates a proof of the safety theorem. The code producer then sends the machine code as well as the safety proof to the code consumer. The consumer independently generates her own safety theorem using the same VCGen and safety policy as the producer. Note that this means she need only trust her own VCGen and safety policy, and not the producer's. This safety theorem is then checked against the proof supplied by the producer. Proof checking is computationally very easy and if the safety theorem is indeed proved by the supplied proof, then the consumer is assured that the code is safe to execute, as long as the trusted components (i.e., the proof checker and the consumer's VCGen and safety policy) are implemented or specified correctly. The Touchstone [NL98a] compiler implements this framework.

1.1.4 Foundational Proof-Carrying Code

A classic security principle, formulated by Saltzer and Schroeder [SS75] almost thirty years ago, is as follows.

***Minimal Trusted Computing Base*:**

> Assurance that an enforcement mechanism behaves as intended is greatest when the mechanism is small and simple.

In each of the safety mechanisms I described above, the safety guarantee is meaningful only if certain (trusted) components are implemented or specified correctly. An important feature of the research advances described in the last two sections is that they each reduced the size and complexity of the components that had to be trusted. For instance, type checking source code guarantees safety only if the type checker, compiler, and runtime are implemented correctly; successfully verified Java bytecode is safe only if the bytecode verifier, JIT compiler, and Java runtime are free of bugs; well-typed TAL code is safe only if the type-checker and assembler are correct; and machine code certified by a PCC system is safe only if the safety policy has no holes and if the VCGen is implemented correctly. So we have come from a world where we had to trust entire compilers to a world where we must trust much smaller components. But how do we build a system where the trusted computing base is the smallest possible?

The goal of the Foundational Proof-Carrying Code (FPCC) project at Princeton is to mechanically verify the safety of machine language programs using a minimal trusted computing base. Foundational PCC [App01, AF00] addresses two areas of particular concern in Necula's PCC. The first is Necula's VCGen which is a complex and large program, approximately 23,000 lines of C [AW02]. The VCGen traverses the machine language program and extracts a formula (in first-order logic) which is true only if the given program obeys the safety policy. A bug in the VCGen would imply that we are demanding a proof of the wrong theorem, i.e., a theorem that does *not* guarantee that the given machine code is safe. Appel and Felty [AF00] showed how to eliminate the VCGen and reason directly about machine code in higher-order logic, instead of using the two-step process of generating a verification condition and then proving it safe.

The second area of concern has to do with Necula's safety policy which has axioms of the type system (i.e., the typing rules for the machine language) built in.

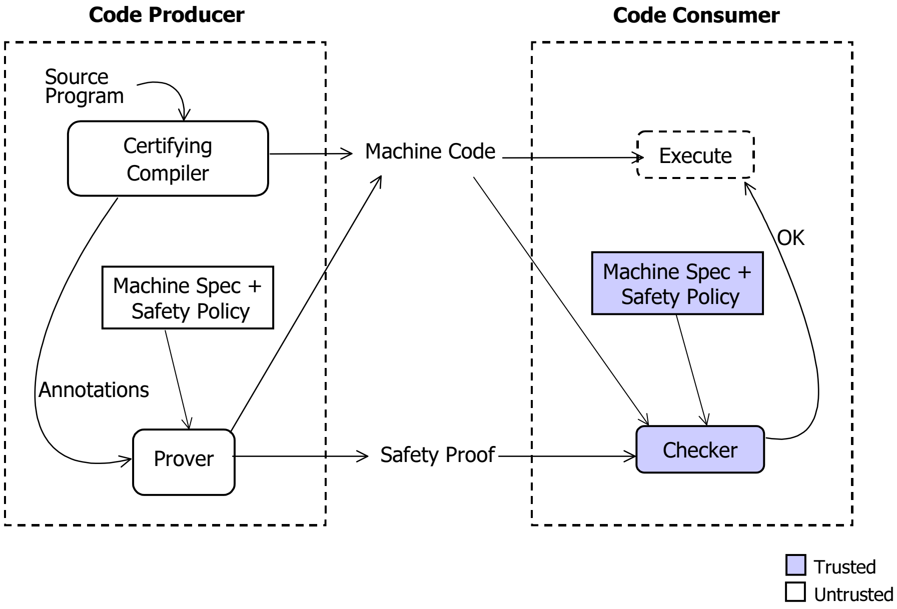


Figure 1.2: Foundational Proof-Carrying Code Architecture

If the type system is not sound, then unsafe programs will be accepted. In fact, League *et al.* [LST03] have shown that one of the SpecialJ [CLN+00] typing rules is unsound. Appel *et al.* write the rules of their type system as machine-checkable lemmas, instead of axioms. This means that the soundness of the type system can be mechanically verified by the proof checker along with the rest of the safety proof.

Figure 1.2 illustrates the FPCC architecture; the trusted components are shaded. In this framework, the VCGen has been replaced by a machine specification which specifies the encoding and instruction semantics for a real machine (such as the Sparc) in higher-order logic. For the Sparc, Michael and Appel [MA00] have shown how to do this in just 1,600 lines of higher-order logic (encoded in Twelf [PS99], an implementation of the LF logical framework [HHP93]). Also, the safety policy shown in Figure 1.2 is much smaller and simpler than Necula's safety policy as it does not have the type system built in. The FPCC safety theorem (also part of the TCB) says simply that if a program can step from a computation state s to a computation state s' (in accordance with the machine instruction semantics and without violating the safety policy), then it is possible to safely take one more step.

Not having the type system built into the safety policy makes a foundational PCC architecture more flexible than Necula's PCC framework because the code producer can "explain" a novel type system or safety argument to the code consumer.

That is, the code producer does not have to program in a specific source language or use a specific compiler as long as she sends the consumer a type soundness proof for her compiler's TAL along with the machine code and the rest of the proof. In a foundational PCC system, the code consumer can mechanically verify the safety of the machine language program as well as the soundness of the type system used to justify the program's safety.

Appel and Felty [AF00] prove the type system sound by defining the semantics of each type in terms of the operational semantics of the real machine. This allows them prove the validity of each typing rule (i.e., each typing rule becomes a lemma). They then prove that if a machine language expression is well-typed, then it cannot get stuck. These proofs are *foundational* in that they rely only on the axioms of higher-order logic and arithmetic, the operational semantics of the target architecture, and the safety policy. Appel and Felty demonstrated how to prove type soundness for a language with dynamically allocated immutable cells. Appel and McAllester [AM01] extended the approach to languages with (co- and contravariant) recursive types. This thesis shows how to extend the approach to languages with mutable references.

1.2 Mutable State and Type Safety

In a language with mutable memory cells, should a cell update be permitted to change the type of the memory cell, or must all memory updates be type-preserving? The former are known as *strong updates* and the latter as *weak updates*. Type-safe languages typically require that updates be type-preserving because such a constraint — I'll call it the *type invariance principle* — makes it easier to establish type safety in the presence of aliasing.

More precisely, type safety guarantees that a well-typed program will not get stuck. To prove type safety — whether using syntactic subject-reduction [WF94] or operationally-based logical relations (the method used by Appel and Felty [AF00]) — we have to show that well-typed programs (or computation states) step to well-typed programs. In particular, since the memory (or *heap* or *store*) is part of the computation state, we have to show that a well-typed memory steps to a well-typed memory. This implies that if an update at some location ℓ in a well-typed memory m changes the type of ℓ, then we will have to correctly propagate the change to the types of all of ℓ's aliases (and to the types of all aliases of ℓ's aliases, and so on) in order to show that the updated memory m' (as a whole) is well-typed. Alternatively, we could devise a type system that tracks aliasing [SWM00, WM00, Wal01] such that the types of aliases do not have to be explicitly modified when the type of the aliased location changes.

A much simpler strategy, employed by practical languages like ML and Java, is to require that the types of allocated memory cells never change. Thus, there are no type changes to propagate; a type-preserving (or weak) update in a well-typed memory m results in a well-typed memory m'.

***Type Invariance Principle*:**

> The type of every allocated memory location must remain unchanged for the duration of program execution.

Enforcing the type invariance principle is tricky in a semantic setting; in fact, naïve attempts to enforce it lead to circularity. To ensure that every update is type-preserving, we need to keep track of the type τ of values that may be written into each allocated location. We can do this using a memory typing Ψ that maps allocated locations to types. But now, consider the semantics of mutable reference types (denoted $\mathsf{ref}\ \tau$). According to the type invariance principle, an allocated memory location has type $\mathsf{ref}\ \tau$ iff it contains a value of type τ *and* it will *always* contain a value of type τ. Therefore, in the definition of $\mathsf{ref}\ \tau$, to check whether a location ℓ will always contain a value of type τ, we need to check that Ψ maps ℓ to τ — that is, the definition of $\mathsf{ref}\ \tau$ must refer to Ψ. To accommodate the latter, we must model all types τ so that they take store typings Ψ as an argument. But this means that *types* must be predicates on partial functions from locations to *types*. A simple diagonalization argument will show that this set of types has an inconsistent cardinality.

1.3 Contributions and Dissertation Outline

In this thesis, I will show how to construct machine-checkable foundational proofs of safety for programs that mutate state. Foundational proofs rely only on the operational semantics of the language and the axioms of the higher-order logic (or any other suitable logic that may be used to encode the proofs). In particular, I'll show how to prove type soundness for languages that support dynamically allocated, (weakly) updatable memory cells that may contain values of any type, including functions, mutable references, co- and contravariant recursive types, and even impredicative quantified types. I'll also describe how to extend these results to permit memory reuse.

In chapter two, I will introduce the basic technique that I use throughout this thesis to construct foundational safety proofs. This technique may be summarized as follows: first, one defines the meaning of types and typing judgments based on the operational semantics of the language; then, based on the meanings of types and typing judgments, one proves the typing rules as lemmas; finally, one proves that if a program type-checks then, based on the meaning of a typing judgment, the program is safe. I demonstrate the technique in both a pure and an impure setting

— specifically, I apply it to a purely functional λ-calculus and to a λ-calculus with dynamically-allocated immutable references. I show that in the pure setting this technique is precisely the logical relations proof technique (though my terminology and notation are different) and in the impure setting it corresponds to Kripke (possible-worlds) logical relations.

In chapter three, I describe how to model a language with general references — that is, mutable memory cells that may contain values of any closed type, including other references and impredicative quantified types. While Java and ML do not support impredicative polymorphism, the ability to store values of impredicative quantified types in mutable cells is crucial in a typed target language for a certifying compiler for Java or ML (for instance, to represent ML function closures without defunctionalizing). Because existing models of general references [AHM98, Lev02] do not model impredicative polymorphism, they are not expressive enough to represent the semantics of the ML or Java type systems. Our goal is to model ML and Java types.

Chapter three explains how to avoid the circularity I alluded to in the previous section. Rather than model types as predicates on memory typings Ψ, I model types as an infinite sequence of increasingly refined approximations; each approximation in the sequence is a predicate on memory typings Ψ whose codomains contain only less refined type approximations.

Also in chapter three, I point out a correspondence between the possible-worlds model of mutable references and that of the modal logic S4. Intuitively, in languages that support the type invariance principle, which requires that something *always* be true, reference types correspond to the modal logic connective $\Box$ which is interpreted in S4 as "always."

Chapter four discusses some changes and extensions to the language considered in chapter three. In particular, I show how to modify the semantics of quantified types to address the fact that, unlike the operational semantics in chapter three, real machines do not have explicit instructions for type application and existential unpacking. I also present some examples and discuss related work.

A foundational PCC system demands that all safety proofs be *machine-checkable*. In chapter five, I explain how the model of mutable references may be encoded in CiC (briefly) and in higher-order logic. As mentioned above, I construct a model of mutable references by stratifying types into a series of approximations. Higher-order logic, unlike CiC, does not provide a convenient mechanism for defining stratified metalogical types; I'll explain how we can construct the stratification ourselves.

For foundational PCC, we need proofs of safety for real machine language programs. In chapter six, I describe the von Neumann model we've built for the FPCC system being developed at Princeton.

In chapter seven, I show how to extend the mutable references model to support the reuse of memory cells (possibly at different types), without violating the type-invariance principle. Specifically, I present a semantics for a calculus with primitives for region-based memory management.

Finally, chapter eight examines an important direction for future research which is how to extend the model I've described to a partial equivalence relation (or PER) model. Using a PER model one can reason about observational equivalence in the presence of mutable state. This would permit foundational proofs of noninterference (a program's observable behavior is independent of certain program inputs) and the correctness of compiler optimizations (an unoptimized program's observable behavior is equivalent to that of its optimized version).

Chapter 2

Foundational Proofs of Safety

A proof-carrying code (PCC) system consists of a type-preserving compiler that produces programs in a typed assembly language (TAL). These programs are safe *provided* that the TAL is sound *and* the translation of TAL programs to machine code preserves safety. A foundational PCC system seeks to eliminate these caveats by giving a semantics to TAL expressions and types in terms of the operational semantics of the underlying von Neumann machine. I shall refer to such semantics as *denotational-operational* since the denotation of types is based on the operation of terms. The (untyped) machine code is then proved safe by showing that it satisfies the TAL type annotations produced by the compiler.

The goal of this chapter is to introduce denotational-operational models in a very simple setting. Imagine that the TAL mentioned above corresponds to the simply-typed λ-calculus, while the machine language corresponds to the untyped λ-calculus, and the semantics of a simply-typed λ-calculus term is given via type-erasure. I shall explain how to define the semantics of types of the simply-typed λ-calculus based on the operational semantics of the untyped λ-calculus and how to use this semantic model to prove the safety of untyped λ-calculus programs.

I first consider a pure functional language (Section 2.1) and then a simple language with mutable state, specifically, with dynamically allocated immutable references (Section 2.2). I specify safety for programs in each language, describe how to prove them safe, and discuss how the proofs correspond to proofs based on the logical relations technique. The presentation in Sections 2.1 and 2.2 resembles the exposition by Appel and McAllester [AM01] of a model of types for the λ-calculus (specifically, the purely functional λ-calculus with recursive types).

2.1 A Pure Language

Syntax The language I consider in this section is the lambda calculus with booleans. I shall call this pure functional language λ^P. The syntax of λ^P terms is given by

$$\frac{e_1 \longmapsto_P e_1'}{e_1\, e_2 \longmapsto_P e_1'\, e_2}\ (\texttt{PO-app1})$$

$$\frac{e_2 \longmapsto_P e_2'}{(\lambda x.e_1)\, e_2 \longmapsto_P (\lambda x.e_1)\, e_2'}\ (\texttt{PO-app2})$$

$$\frac{}{(\lambda x.e)\, v \longmapsto_P e[v/x]}\ (\texttt{PO-app3})$$

Figure 2.1: Pure Functional Language (λ^P): Operational Semantics

the following grammar.

$$\begin{array}{llcl} \textit{Values} & v & ::= & \texttt{true} \mid \texttt{false} \mid \lambda x.e \\ \textit{Expressions} & e & ::= & x \mid v \mid (e_1\, e_2) \end{array}$$

I use the metavariable x to range over a countably infinite set Var of variables. A term v is a value if it is `true`, `false` or a lambda abstraction that contains no free variables x. Notice that terms in this language are untyped, just as machine instructions in a foundational PCC system would be untyped. I will show how to give purely semantic (not syntactic) typings to the untyped lambda calculus.

2.1.1 Operational Semantics and Safety

The formal operational semantics for λ^P (shown in Figure 2.1) is an entirely conventional call-by-value semantics. It is specified by a relation $\longmapsto_P$ on closed terms. The notation $e \longmapsto_P e'$ denotes a single operational step from e to e' and $e \longmapsto_P^* e'$ denotes a chain of the form $e \longmapsto_P e_1 \longmapsto_P \ldots \longmapsto_P e_j$ where e_j is e' and $j \geq 0$ (i.e., e evaluates to e' in zero or more steps).

A term e is *irreducible* if it has no successor in the step relation, that is, $\mathrm{irred}(e)$ if e is a value or if e is a "stuck" expression (such as $\texttt{true}(e')$) to which no operational rule applies. I write $\mathrm{val}(e)$ to denote that term e is a closed value. An expression e is considered *safe* if it never evaluates to a "stuck" expression.

Definition 2.1 (Safe)
A term e is safe *if whenever e evaluates to e', either e' is a value or another step is*

possible.

$$\mathrm{safe}(e) \;\stackrel{\mathrm{def}}{=}\; \forall e'.\; e \longmapsto_P^* e' \implies (\mathrm{val}(e') \;\vee\; \exists e''.\; e' \longmapsto_P e'')$$

To show that a given term is safe I will construct safety proofs based on type systems. Such proofs typically consist of a typing derivation that proves that the given term typechecks, but since I am interested in constructing *foundational* safety proofs, I must also prove a theorem stating that typability implies safety. Proofs that typability implies safety are typically based on syntactic subject reduction. While it is possible to construct foundational safety proofs that rely on syntactic subject reduction, in this thesis I will take a *semantic* approach, as pioneered in the NuPrl system [CAB+86] and as applied to proof-carrying code by Appel and Felty [AF00]. In a semantic proof, one assigns a meaning (a semantic truth value) to type judgments. One then proves that if a type judgment is true, then the typed program is safe. One also proves that the type inference rules are sound, that is, if the premises are true then the conclusion is true. This ensures that derivable type judgments are true, hence typable programs are safe. The rest of this section explains how to construct such semantic safety proofs for λ^P programs.

2.1.2 Semantics of Types and Judgments

Values in λ^P may be booleans (of type bool) or functions (of type $\tau_1 \to \tau_2$). I treat types as *sets of values* rather than as syntactic type expressions. Hence, the type bool is a set that consists of just two values, `true` and `false`. To describe the semantics of type $\tau_1 \to \tau_2$, I first have to specify what it means for a closed term e to have type τ (denoted by judgments of the form $e : \tau$). Informally, e has type τ if it evaluates to a value of type τ.

Definition 2.2 (Expr : Type)
For any closed expression e and type τ I write $e : \tau$ if whenever $e \longmapsto_P^* e'$ and e' is irreducible, then e' is a value in set τ; that is,

$$e : \tau \;\stackrel{\mathrm{def}}{=}\; \forall e'.\; (e \longmapsto_P^* e' \;\wedge\; \mathrm{irred}(e')) \implies e' \in \tau$$

I can now specify the semantics of function types. The type $\tau_1 \to \tau_2$ is the set of all values of the form $\lambda x.e$ such that for any value v of type τ_1 the result of substituting v for x in e (written $e[v/x]$) is an expression of type τ_2. The boolean and function types are specified as follows.

$$\begin{array}{lcl} \mathsf{bool} & \stackrel{\mathrm{def}}{=} & \{\, \mathtt{true}, \mathtt{false} \,\} \\ \tau_1 \to \tau_2 & \stackrel{\mathrm{def}}{=} & \{\, \lambda x.e \mid \forall v.\; v \in \tau_1 \implies e[v/x] : \tau_2 \,\} \end{array}$$

$$\frac{}{\Gamma \vDash_P x : \Gamma(x)}\ \texttt{(P-var)}$$

$$\frac{}{\Gamma \vDash_P \texttt{true} : \mathsf{bool}}\ \texttt{(P-true)} \qquad \frac{}{\Gamma \vDash_P \texttt{false} : \mathsf{bool}}\ \texttt{(P-false)}$$

$$\frac{\Gamma\,[x \mapsto \tau_1] \vDash_P e : \tau_2}{\Gamma \vDash_P \lambda x.e : \tau_1 \longrightarrow \tau_2}\ \texttt{(P-abs)}$$

$$\frac{\Gamma \vDash_P e_1 : \tau_1 \longrightarrow \tau_2 \qquad \Gamma \vDash_P e_2 : \tau_1}{\Gamma \vDash_P (e_1\, e_2) : \tau_2}\ \texttt{(P-app)}$$

Figure 2.2: Pure Functional Language (λ^P): Type-checking Lemmas

Up to now I have only dealt with closed λ^P expressions, as these are the ones that are evaluated at run time. Now I turn to expressions with free variables, upon which the static type-checking rules must operate. The notation $FV(e)$ denotes the set of variables that occur free in e.

Definition 2.3 (Semantics of Judgment)
A type environment *is a mapping from lambda calculus variables to types. A* value environment *(also known as a ground substitution) is a mapping from lambda calculus variables to values. For any type environment Γ and value environment σ, I write $\sigma : \Gamma$ if for all variables $x \in \mathrm{dom}(\Gamma)$ we have $\sigma(x) : \Gamma(x)$; that is,*

$$\sigma : \Gamma \quad \overset{\text{def}}{=} \quad \forall x \in \mathrm{dom}(\Gamma).\ \sigma(x) : \Gamma(x)$$

I write $\Gamma \vDash_P e : \tau$ iff $FV(e) \subseteq \mathrm{dom}(\Gamma)$ and

$$\forall \sigma.\, \sigma : \Gamma \implies \sigma(e) : \tau$$

where $\sigma(e)$ is the result of replacing the free variables in e with their values under σ. Finally, I write $\vDash_P e : \tau$ to mean $\Gamma_0 \vDash_P e : \tau$ for the empty environment Γ_0.

Note that $\Gamma \vDash_P e : \tau$ can be viewed as a three place relation that holds on the type environment Γ, the term e, and the type τ.

The type-checking rules for λ^P are given in Figure 2.2. I write $\Gamma\,[x \mapsto \tau]$ to denote the type environment that is identical to Γ except that it maps the variable

x (where $x \notin \mathrm{dom}(\Gamma)$) to τ. Each of the type inference lemmas in Figure 2.2 states that if certain instances of the relation $\Gamma \models_P e : \tau$ hold, then certain other instances hold. Once I have proved the type inference lemmas in Figure 2.2, these lemmas can be used in the same manner as standard type inference rules to prove statements of the form $\Gamma \models_P e : \tau$.

2.1.3 Typability Implies Safety

A "program" in λ^P is simply a closed term e. I must now prove that if a type judgment is true then the typed program is safe. In a conventional syntactic type theory, the safety theorem (typability implies safety) is difficult (or at least tedious) to prove. Here it follows directly from the definitions.

Theorem 2.4 (Safety)
If $\models_P e : \tau$ *and* τ *is a type, then* $\mathrm{safe}(e)$.

Proof: To prove $\mathrm{safe}(e)$ we must show that for any term e', if e evaluates to e', either e' is a value or another step is possible. Suppose $e \longmapsto_P^* e'$. If e' is not irreducible, then there must exist some e'' such that $e' \longmapsto_P e''$. Otherwise, e' is irreducible. By Definition 2.3, $\models_P e : \tau$ denotes $\Gamma_0 \models_P e : \tau$ where Γ_0 is the empty context. From $\Gamma_0 \models_P e : \tau$, again by Definition 2.3, it follows that e is closed. Choose the empty value environment σ_0 and by the semantics of $\Gamma_0 \models_P e : \tau$ (Definition 2.3) we have $\sigma_0 : \Gamma_0 \implies \sigma_0(e) : \tau$. The premise $\sigma_0 : \Gamma_0$ is trivially satisfied; applying the trivial substitution we have $e : \tau$. Since $e \longmapsto_P^* e'$ and $\mathrm{irred}(e')$ it follows that $e' \in \tau$. Since τ is a type, that is, a set of values, it follows that $\mathrm{val}(e')$. □

2.1.4 Typing Rules as Lemmas

It remains for us to show that the type inference rules shown in Figure 2.2 are sound.

Theorem 2.5 (P-var)
If Γ *is a type environment and* x *is a variable such that* $x \in \mathrm{dom}(\Gamma)$ *then* $\Gamma \models_P x : \Gamma(x)$.

Proof: For any σ such that $\sigma : \Gamma$ we must show that $\sigma(x) : \Gamma(x)$. This is immediate from the definition of $\sigma : \Gamma$. □

Theorem 2.6 (P-true)
If Γ *is a type environment then* $\Gamma \models_P \texttt{true} : \mathsf{bool}$.

Proof: For any σ such that $\sigma : \Gamma$ we must show that $\sigma(\texttt{true}) : \mathsf{bool}$. Since `true` is a closed term it suffices to prove that $\texttt{true} : \mathsf{bool}$. This follows immediately from the definition of bool. □

Theorem 2.7 (P-false)
If Γ is a type environment then $\Gamma \vDash_P \texttt{false} : \mathsf{bool}$.

Proof: By the same argument as for theorem `P-true`. □

Theorem 2.8 (P-abs)
Let Γ be a type environment, let τ_1 and τ_2 be types, and let $\Gamma\,[x \mapsto \tau_1]$ be a type environment that is identical to Γ except that it maps x to τ_1. If $\Gamma\,[x \mapsto \tau_1] \vDash_P e : \tau_2$ then $\Gamma \vDash_P \lambda x.e : \tau_1 \to \tau_2$.

Proof: We must prove that under the premises of the theorem for any σ such that $\sigma : \Gamma$ we have $\sigma(\lambda x.e) : \tau_1 \to \tau_2$. Suppose $\sigma : \Gamma$ and that v is some value such that $v \in \tau_1$. Notice that the premise $\Gamma\,[x \mapsto \tau_1] \vDash_P e : \tau_2$ guarantees that $x \notin \mathrm{dom}(\Gamma)$ (since, as mentioned above, the notation $\Gamma\,[x \mapsto \tau_1]$ assumes that $x \notin \mathrm{dom}(\Gamma)$). By the definition of $\tau_1 \to \tau_2$ it now suffices to show that $\sigma(e[v/x]) : \tau_2$. Let $\sigma\,[x \mapsto v]$ be the value environment identical to σ except that it maps x to v. From $v : \tau_1$ it follows that $\sigma\,[x \mapsto v] : \Gamma\,[x \mapsto \tau_1]$, which together with the premise $\Gamma\,[x \mapsto \tau_1] \vDash_P e : \tau_2$ (by Definition 2.3) implies $\sigma\,[x \mapsto v](e) : \tau_2$. But this implies $\sigma(e[v/x]) : \tau_2$. □

Theorem 2.9 (P-app)
If Γ is a type environment, e_1 and e_2 are (possibly open) terms, and τ_1 and τ_2 are types such that $\Gamma \vDash_P e_1 : \tau_1 \to \tau_2$ and $\Gamma \vDash_P e_2 : \tau_1$ then $\Gamma \vDash_P (e_1\,e_2) : \tau_2$.

Proof: We must prove that under the premises of the theorem for any σ such that $\sigma : \Gamma$ we have $\sigma(e_1\,e_2) : \tau_2$. Suppose $\sigma : \Gamma$. Let $\sigma(e_1\,e_2) \longmapsto_P^* e'$ and $\mathrm{irred}(e')$; we must show that $e' \in \tau_2$. By the operational semantics, it follows that:

1. $\sigma(e_1) \longmapsto_P^* e_1'$ and $\mathrm{irred}(e_1')$. By the premises of the theorem we have $\sigma(e_1) : \tau_1 \to \tau_2$ so by Definition 2.2 (Expr : Type) we have $e_1' \in \tau_1 \to \tau_2$. From the definition of $\to$ it follows that e_1' is of the form $\lambda x.e_{11}$.

2. $\sigma(e_2) \longmapsto_P^* e_2'$ and $\mathrm{irred}(e_2')$. By the premises of the theorem we have $\sigma(e_2) : \tau_1$. Hence, by Definition 2.2 (Expr : Type) we have $e_2' \in \tau_1$.

3. $(\lambda x.e_{11})e_2' \longmapsto_P e_{11}[e_2'/x] \longmapsto_P^* e'$. Since $\lambda x.e_{11} \in \tau_1 \to \tau_2$ and $e_2' \in \tau_1$, by the definition of $\tau_1 \to \tau_2$ it follows that $e_{11}[e_2'/x] : \tau_2$. By Definition 2.2 (Expr : Type) and $\mathrm{irred}(e')$ it follows that $e' \in \tau_2$ as we wanted to show.

□

2.1.5 Discussion: Trusted Computing Base

I have shown that one can construct a foundational safety proof for any term e that is well-typed with respect to the typing rules in Figure 2.2. The proof is foundational because these typing rules are not treated as axioms, that is, they do not have to be trusted. In fact, the only axioms that the proof relies on are:

- the operational semantics of the language (Figure 2.1) — including, for instance, the definition of substitution (which rule `PO-app3` relies on);
- the specification of safety (Definition 2.1);
- the axioms of higher-order logic (or any other suitable logic) in which the specifications and proofs are formalized.[1]

All other definitions are built on top of these axioms and are not part of the trusted computing base.

One can construct machine-checkable proofs by encoding the specifications and proofs described above in a suitable logic and developing a proof checker for the logic. The proof checker would also be part of the trusted computing base.

2.1.6 Discussion: Logical Relations

The semantic approach to proving safety that I have described above corresponds exactly to the proof technique known as logical relations. To illustrate the correspondence, I will sketch out how one would prove that well-typed λ^P programs are safe using logical relations. As much as possible, I will use the terminology one would expect to see in a proof based on logical relations.

The λ^P operational semantics is exactly as before, that is, the semantics given in Figure 2.1. Type judgments have the form $\Gamma \vdash_P e : \tau$; I use $\vdash_P$ rather than $\models_P$ to indicate that we are now dealing with a syntactic type theory — though, in this case, the difference is entirely cosmetic, as we shall see — where Γ is a mapping from variables to syntactic type expressions and that τ is a syntactic type expression. The typing rules are exactly the rules shown in Figure 2.2 except that each occurrence of $\models_P$ should be replaced by $\vdash_P$.

To prove that well-typed terms are safe using logical relations, one begins by defining, for each type τ, a unary relation (i.e., a set) R^τ of closed terms e of type τ. I regard these sets as predicates and write $R^\tau(e)$ for $e \in R^\tau$. A logical relation $\mathcal{R} = \{R^\tau\}$ is a family of relations R^τ for each type τ in the language. The relations for λ^P may be defined as follows.

[1] Note that the definitions use logical connectives such as $\forall$, $\Longrightarrow$, $\wedge$, etc., and the proofs use logical inference rules to operate on the definitions — the latter is not immediately obvious since the proofs, for simplicity of presentation, use informal reasoning in place of formal proof steps such as "apply the $\forall$ introduction rule ..." or "using $\wedge$ elimination we have ..." and so on.

- $R^{\mathsf{bool}}(e)$ iff for all e' such that $e \longmapsto_P^* e'$ and irred(e'), either $e' = \texttt{true}$ or $e' = \texttt{false}$.
- $R^{\tau_1 \to \tau_2}(e)$ iff for all e' such that $e \longmapsto_P^* e'$ and irred(e'), $e' = \lambda x.e_2$, and whenever $R^{\tau_1}(e_1)$, we have $R^{\tau_2}(e\,e_1)$.

Hence, the relation R^τ consists of all expressions e such that if $e \longmapsto_P^* e'$ then $e' \in R^\tau$. Notice that $R^\tau(e)$ corresponds to $e : \tau$ (Definition 2.2) which I defined as part of the (denotational-operational) semantic approach above.

There are now two parts to the proof of safety. First, one proves that every element of every set R^τ is safe. This follows easily from the definition of R^τ. Notice that this lemma corresponds to Theorem 2.4.

Lemma 2.10
If $R^\tau(e)$, then safe(e).

Next, one shows that every well-typed term e of type τ is an element of R^τ. For the lambda abstraction case (i.e., when e is of the form $\lambda x.e'$), to conclude that $\lambda x.e'$ is an element of $R^{\tau_1 \to \tau_2}$ we will need to apply the induction hypothesis in order to conclude that e' is an element of R^{τ_2}. But here we have a problem: R^{τ_2} is a set of closed terms, while e' may contain free occurrences of the variable x. To deal with this situation, we have to generalize the induction hypothesis to open terms. Hence, we must prove the following lemma.

Lemma 2.11
If $x_1 : \tau_1, x_2 : \tau_2, \ldots, x_n : \tau_n \vdash_P e : \tau$ and $v_1, \ldots, v_n$ are closed values of types $\tau_1, \ldots, \tau_n$ such that $R^{\tau_i}(v_i)$ for each i, then $R^\tau(e[v_1/x_1] \cdots [v_n/x_n])$.

The proof is by induction on the type derivation $x_1 : \tau_1, x_2 : \tau_2, \ldots, x_n : \tau_n \vdash_P e : \tau$. Notice that the cases of this proof, namely for typing rules `P-var`, `P-true`, `P-false`, `P-abs`, and `P-app`, mirror the proofs of the typing rule lemmas that I proved as part of the semantic approach, that is, Theorems 2.5 through 2.9, respectively.

Given a closed program e, one can now easily prove the final theorem which says that if $\vdash_P e : \tau$ then safe(e). The proof is immediate using Lemmas 2.10 and 2.11. This proof is foundational: the typing rules do not have to be trusted since the property of the typing rules crucial to our proof (that well-typed terms are safe) has been captured by the logical relation $\mathcal{R} = \{R^\tau\}$ via Lemma 2.11. Hence, the TCB is identical to that described in Section 2.1.5.

The presentation here resembles that in Pierce [Pie02] (Chapter 12), where logical relations are used to prove that every well-typed term of the simply typed λ-calculus is normalizable. The logical relations proof technique is also described in Mitchell [Mit96] and Gunter [Gun92].

2.2 An Impure Language: Immutable References

In this section I consider a language with side-effects, specifically, an extension of the pure language of Section 2.1 (λ^P) with dynamically allocated immutable cells. Programs written in this language — which I call λ^I — can allocate new reference cells on the heap (or *store*) but can never update existing cells. My aim in this section is to explain what changes we must make to the semantic model as we move from a pure language (without side-effects) to an impure language (with side-effects). The language is restricted to immutable references for simplicity; the semantics of mutable references is much trickier and will be the focus of Chapter 3.

Syntax The syntax of λ^I terms is given by the following grammar.

$$\begin{array}{llcl} \textit{Values} & v & ::= & \ell \mid \texttt{true} \mid \texttt{false} \mid \lambda x.e \\ \textit{Expressions} & e & ::= & x \mid v \mid (e_1\, e_2) \mid \texttt{new}(e) \mid \texttt{!}\, e \end{array}$$

I use the metavariable x to range over a countably infinite set *Var* of variables and the metavariable ℓ to range over a countably infinite set of *locations Loc*. A term e is a value (written $\mathrm{val}(e)$) if it is a location ℓ, a constant (`true` or `false`), or a lambda abstraction with no free variables x. The language includes terms for allocating (and simultaneously initializing) a new cell on the heap ($\texttt{new}(e)$) and for reading the contents of an allocated cell ($\texttt{!}\,e$).

Operational Semantics A *store* S is a finite map from locations to closed values. The small-step semantics for λ^I is given as a step relation between machine states. The state of the abstract machine is described by a pair (S, e) of a store and a closed term. The notation $(S, e) \longmapsto_I (S', e')$ denotes a single operational step from machine state (S, e) to state (S', e'), while $(S, e) \longmapsto_I^* (S', e')$ denotes a chain of j steps, for some $j \geq 0$, of the form $(S, e) \longmapsto_I (S_1, e_1) \longmapsto_I \ldots \longmapsto_I (S_j, e_j)$ where S_j is S' and e_j is e'. The operational semantics is given in Figure 2.3 and is completely standard. Notice that the only rule in the operational semantics that produces a side-effect — a *store effect*, to be precise — is the rule that evaluates expressions of the form $\texttt{new}(v)$ (see `IO-new2`); it extends the store with a previously unallocated location ℓ initialized to v. The operational rule for dereferencing a cell ℓ checks that ℓ is an allocated cell (see `IO-deref2`).

In a language with side-effects, the order in which terms are evaluated is important. In examples that follow, I shall write $\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2$ in place of $(\lambda x.e_2)e_1$ to make it readily obvious that e_1 is evaluated before e_2.

Safety A state (S, e) is *irreducible* if it has no successor in the step relation, that is, $\mathrm{irred}(S, e)$ if e is a value or (S, e) is a "stuck" state such as $(S, \texttt{true}(e))$ or $(S, \texttt{!}\,\ell)$

$$\frac{(S, e_1) \longmapsto_I (S', e_1')}{(S, e_1\, e_2) \longmapsto_I (S', e_1'\, e_2)} \;\texttt{(IO-app1)}$$

$$\frac{(S, e_2) \longmapsto_I (S', e_2')}{(S, (\lambda x.e_1)\, e_2) \longmapsto_I (S', (\lambda x.e_1)\; e_2')} \;\texttt{(IO-app2)}$$

$$\frac{}{(S, (\lambda x.e)\, v) \longmapsto_I (S, e[v/x])} \;\texttt{(IO-app3)}$$

$$\frac{(S, e) \longmapsto_I (S', e')}{(S, \texttt{new}(e)) \longmapsto_I (S', \texttt{new}(e'))} \;\texttt{(IO-new1)}$$

$$\frac{\ell \notin \mathrm{dom}(S)}{(S, \texttt{new}(v)) \longmapsto_I (S\,[\ell \mapsto v], \ell)} \;\texttt{(IO-new2)}$$

$$\frac{(S, e) \longmapsto_I (S', e')}{(S, !\, e) \longmapsto_I (S', !\, e')} \;\texttt{(IO-deref1)}$$

$$\frac{\ell \in \mathrm{dom}(S)}{(S, !\, \ell) \longmapsto_I (S, S(\ell))} \;\texttt{(IO-deref2)}$$

Figure 2.3: Immutable References (λ^I): Operational Semantics

where $\ell \notin \mathrm{dom}(S)$. A machine state (S, e) is *safe* if it never reaches a "stuck" state.

Definition 2.12 (Safe)
A state (S, e) is safe *if whenever (S, e) evaluates to a state (S', e'), either e' is a value or another step is possible.*

$$\begin{aligned} \mathrm{safe}(S, e) \;\overset{\mathrm{def}}{=}\; & \forall S', e'.\ (S, e) \longmapsto_I^* (S', e') \\ & \qquad \Longrightarrow\ (\mathrm{val}(e') \ \vee\ \exists S'', e''.\ (S', e') \longmapsto_I (S'', e'')) \end{aligned}$$

I wish to show that a program e is safe to execute given an initial store S, i.e., $\mathrm{safe}(S, e)$. To prove that an initial machine state (S, e) is safe using the semantic approach, I will first specify the semantics of types and type judgments; I will then prove that if a type judgment is true, then the typed program is safe; and finally, I will prove that the λ^I typing rules are sound.

2.2.1 Semantics of Types using Possible Worlds

Values in λ^I may be booleans of type bool, functions of type $\tau_1 \to \tau_2$, or store locations ℓ of type $\mathsf{box}\ \tau$ (the type ascribed to immutable cells that contain values of type τ). In Section 2.1, I modeled types of the purely functional λ-calculus as sets of values, or equivalently, as predicates on values. For λ^I, the model of types as sets of values is inadequate. To see why, let us consider the semantics of the immutable reference type $\mathsf{box}\ \tau$. A location ℓ has type $\mathsf{box}\ \tau$ if (1) ℓ is an *allocated* location, that is, if it is in the domain of the current store S, and (2) the value stored at location ℓ in store S has type τ. Hence, to determine if location ℓ has type $\mathsf{box}\ \tau$ we need to have access to the current store. The solution is to model types as predicates on stores as well as values. I say a type τ is a set of pairs of the form $\langle S, v\rangle$. Informally, $\langle S, v\rangle \in \tau$ means that value v has type τ with respect to store S. The semantics of immutable references can now be defined as follows.[2]

$$\mathsf{box}\ \tau \ \stackrel{\text{def}}{=}\ \{\, \langle S, \ell\,\rangle \mid \ell \in \mathrm{dom}(S) \ \wedge\ \langle S, S(\ell)\rangle \in \tau \,\}$$

A type, however, is not just *any* set of pairs $\langle S, v\rangle$; there are certain properties that these sets must satisfy as I shall illustrate through an example. Consider the following program which allocates and initializes a cell, executes several instructions (elided), and then dereferences the original cell. (The comments that follow each line in the program describe aspects of the state immediately after that line has been evaluated.)

```
                                % $S_0$
let $x_1$ = new(true) in        % $S_1 = S_0\,[\ell_1 \mapsto \texttt{true}],\ \ell_1 \notin \mathrm{dom}(S_0),\ x_1 \mapsto \ell_1$
let $x_2$ = ...  in             % $S_2 =\ \ldots,\ x_1 \mapsto \ell_1$
      ⋮
let $x_n$ = ...  in             % $S_n =\ \ldots,\ x_1 \mapsto \ell_1$
!$x_1$                          % $S_{n+1} = S_n$
```

Suppose that we execute the above program with initial store S_0. By the λ^I operational semantics, after evaluating the first line of the program we have store

[2] Notice that the second condition in the definition of $\mathsf{box}\ \tau$ (i.e., $\langle S, S(\ell)\rangle \in \tau$) essentially subsumes the first ($\ell \in \mathrm{dom}(S)$), so I could have simply defined $\mathsf{box}\ \tau$ as $\{\, \langle S, \ell\,\rangle \mid \langle S, S(\ell)\rangle \in \tau \,\}$.

$S_1 = S_0\,[\ell_1 \mapsto \texttt{true}]$ for some location $\ell_1 \notin \mathrm{dom}(S_0)$ and x_1 is bound to ℓ_1. According to the semantics of box above, $\langle S_1, \ell_1\rangle \in \mathsf{box\ bool}$ holds. Next, we evaluate the elided terms; suppose that these include one or more `new` expressions, and suppose that the net effect of these terms is to transform the store from S_1 to S_n. Hence, when we reach the final instruction $(!\,x_1)$, x_1 is bound to ℓ_1 and we have store S_n. At this point, from a semantic perspective, dereferencing ℓ_1 given store S_n is safe as long as:

1. ℓ_1 is an allocated location, i.e., $\ell_1 \in \mathrm{dom}(S_n)$
2. $S_n(\ell_1)$ is a value of type bool

Together these two conditions imply that we need to prove $\langle S_n, \ell_1\rangle \in \mathsf{box\ bool}$ in order to show that the last instruction is safe to execute. We already know that $\langle S_1, \ell_1\rangle \in \mathsf{box\ bool}$, but we need to account for the fact that S_n is different from S_1. An examination of the operational semantics suggests that condition (1) must hold since there are no terms in λ^I that *shrink* the store — hence, $\mathrm{dom}(S_n)$ must be a superset of $\mathrm{dom}(S_1)$. Similarly, condition (2) must be true since there are no λ^I terms that *update* allocated locations — hence, if ℓ is an allocated location in S_1 then $S_1(\ell) = S_n(\ell)$. More formally, for any two machine states (S, e) and (S', e'), if $(S, e) \longmapsto_I^* (S', e')$, then the relationship between S and S' is specified by $S \sqsubseteq S'$ (read "S' is a *valid extension* of S"):

Definition 2.13 (Store Extension)
A valid store extension *is defined as follows:*

$$S \sqsubseteq S' \;\stackrel{\text{def}}{=}\; \forall \ell.\ \ell \in \mathrm{dom}(S) \implies (\ell \in \mathrm{dom}(S') \;\wedge\; S(\ell) = S'(\ell))$$

The concept of a valid store extension $S \sqsubseteq S'$ is useful for restricting attention to only those stores S' that are *possible* in the future given the operational semantics of λ^I. Returning to the above example, given $\langle S_1, \ell_1\rangle \in \mathsf{box\ bool}$ and $S_1 \sqsubseteq S_n$, we can show that $\langle S_n, \ell_1\rangle \in \mathsf{box\ bool}$. In general, we want all of the types λ^I to have this property exhibited by box bool, that is, we want type sets in λ^I to be closed under valid store extension.

Definition 2.14 (Type)
A type *is a set τ of tuples of the form $\langle S, v\rangle$, where S is a store and v is a value, such that if $\langle S, v\rangle \in \tau$ and $S \sqsubseteq S'$ then $\langle S', v\rangle \in \tau$; that is,*

$$\mathrm{type}(\tau) \;\stackrel{\text{def}}{=}\; \forall S, S', v.\ (\langle S, v\rangle \in \tau \;\wedge\; S \sqsubseteq S') \implies \langle S', v\rangle \in \tau$$

I have shown that the absence of cell deallocation and update in λ^I (captured by the definition of $\sqsubseteq$) is crucial for proving that box types are preserved under evaluation. Now let us take a look at boolean and function types.

A value v has type bool as long as v is either $\mathtt{true}$ or $\mathtt{false}$. Though types are predicates on stores as well as values, the store is irrelevant when determining if a value v has type bool. It trivially follows that bool is closed under store extension.

$$\mathsf{bool} \;\stackrel{\text{def}}{=}\; \{\, \langle S, v\rangle \mid v = \mathtt{true} \;\vee\; v = \mathtt{false} \,\}$$

When deciding if a value $\lambda x.e$ has type $\tau_1 \rightarrow \tau_2$ with respect to a current store S we have to keep in mind that the program may not utilize (apply) $\lambda x.e$ until some point in the future, a point when the store may be different from S thanks to the evaluation of $\mathtt{new}$ terms in the interim. Let S' be that future store; since λ^I does not allow cell deallocation or update, clearly it should be the case that $S \sqsubseteq S'$. For any value v that has type τ_1 with respect to S', if evaluating the expression $e[v/x]$ with store S' gives us a value of type τ_2 (along with a store that is a valid extension of S') then we may conclude that $\lambda x.e$ has type $\tau_1 \rightarrow \tau_2$ with respect to store S. Before I can formally define $\tau_1 \rightarrow \tau_2$, I must specify what it means for a closed expression e to have type τ with respect to store S (written $e :_S \tau$). Informally, I say $e :_S \tau$ if e evaluates to a value of type τ and if the evaluation results in a store S' that is a valid extension of store S.

Definition 2.15 (Expr : Type)
For any closed expression e and type τ I write $e :_S \tau$ if whenever $(S, e) \longmapsto_I^* (S', e')$ and (S', e') is irreducible, then $\langle S, e'\rangle \in \tau$; that is,

$$\begin{aligned} e :_S \tau \;\stackrel{\text{def}}{=}\; & \forall S', e'.\; ((S, e) \longmapsto_I^* (S', e') \;\wedge\; \mathrm{irred}(S', e')) \\ & \Longrightarrow\; S \sqsubseteq S' \;\wedge\; \langle S', e'\rangle \in \tau \end{aligned}$$

Note that for a value v, the statements $v :_S \tau$ and $\langle S, v\rangle \in \tau$ are equivalent. The semantics of function types can now be defined as follows.

$$\tau_1 \rightarrow \tau_2 \;\stackrel{\text{def}}{=}\; \{\, \langle S, \lambda x.e\rangle \mid \forall v, S'.\; (S \sqsubseteq S' \;\wedge\; \langle S', v\rangle \in \tau_1) \;\Longrightarrow\; e[v/x] :_{S'} \tau_2 \,\}$$

In order to prove that function types are closed under store extension I will first need to show that valid store extension ($\sqsubseteq$) is transitive (see Lemma 2.18 for details). Store extension states that allocated cells cannot be freed or updated; hence, stores can only grow, which means that $\sqsubseteq$ relates monotonically increasing stores, so it is easy to show that $\sqsubseteq$ is transitive.

A Modal Interpretation The semantics that I have presented thus far is actually a *possible-worlds semantics*. Possible-worlds models are often called Kripke models in honor of Saul Kripke who introduced the notion of possible worlds [Kri63] when he developed a model theory for propositional modal logic based on this concept. I use possible worlds to capture the modal (temporal) nature of types in a language with side-effects. In order to further motivate some of the properties of stores and types in our model, I will now examine the connections between possible-worlds models of modal logic and our model for λ^I.

Possible-worlds models are specified by defining the following (see Huth and Ryan [HR00]):

- A set W, whose elements are called *worlds*. In Kripke's multiple-world interpretation of modal logic, a proposition is true or false relative to a world. Similarly, in our model, a value v has type τ relative to a world, that is, relative to a state of computation during evaluation. While one could use machine states (S, e) to model worlds, all the information that our model needs from a world may be found in the store component of a machine state (see discussion of the labeling function below). Hence, a world in our model corresponds to a store, so we have $W = \mathit{Loc} \overset{\text{fin}}{\to} \mathit{Val}$, where Loc is the countably infinite set of locations and Val is the set of λ^I values.

- A relation $\mathit{Acc} \subseteq W \times W$ called the *accessibility* relation. Given worlds w and w', $\mathit{Acc}(w, w')$ says that w' is accessible from w. In our model, this corresponds to the store extension relation $(\sqsubseteq)$ and $S \sqsubseteq S'$ says that store S' is accessible (or possible) from store S.

- A *labeling function* $L : W \to \mathcal{P}(\texttt{Atoms})$ that, given a world w, yields the set of atomic propositions that hold in that world. The atomic propositions p that we are interested in are of the form: "location l is allocated and contains value v." Therefore, in our model,

$$L(S) = \{\, (\ell, v) \mid S(\ell) = v \,\}$$

- The properties that the accessibility relation Acc should satisfy. Imposing different laws on the accessibility relation leads to different modal logics. Our model is constructed to ensure that the $\sqsubseteq$ relation is reflexive and transitive. This suggests a connection with the modal logic S4 since models of S4 also require that the accessibility relation be reflexive and transitive.

In Kripke's semantics of modal logic, modal operators allow us to reason about the truth of a proposition in all worlds accessible (or possible) from the current world. The modal operator that we are particularly interested in is $\Box$ which may be pronounced "always" or "necessarily" or "everywhere" (among other possibilities) depending on the desired interpretation. We shall interpret $\Box$ as "always" and interpret $\Box p$ as "It will always be true that p." This corresponds to interpreting $\mathit{Acc}(w, w')$ as "w' is in the future of w." To enforce this interpretation, a model for this logic must satisfy $\Box p \to \Box\Box p$ — informally, if p will always be true then it will

$$\begin{array}{lcl}
\mathsf{bool} & \overset{\text{def}}{=} & \{\, \langle S, v\rangle \mid v = \texttt{true} \ \lor\ v = \texttt{false} \,\} \\
\mathsf{box}\ \tau & \overset{\text{def}}{=} & \{\, \langle S, \ell\,\rangle \mid \ell \in \mathrm{dom}(S) \ \land\ \langle S, S(\ell)\rangle \in \tau \,\} \\
\tau_1 \rightarrow \tau_2 & \overset{\text{def}}{=} & \{\, \langle S, \lambda x.e\rangle \mid \forall v, S'.\ (S \sqsubseteq S' \ \land\ \langle S', v\rangle \in \tau_1) \implies e[v/x] :_{S'} \tau_2 \,\}
\end{array}$$

Figure 2.4: Immutable References (λ^I): Type Definitions

always be true that p will always be true. This is equivalent to requiring that Acc be transitive. Also, we assume that the future includes the present — that is, if p will always be true then p is true right now — so the model must satisfy $\Box p \rightarrow p$. This is equivalent to requiring that Acc be reflexive. The resulting modal logic is known as KT4 or S4 in the literature. Reading the proposition p as "location ℓ is allocated and contains value v" we can see why the reflexivity and transitivity of $\sqsubseteq$ captures the semantics of immutable references that we had wanted to describe.

2.2.2 Validity of Types

The type definitions developed in Section 2.2.1 are summarized in Figure 2.4. They rely on Definitions 2.13 (Store Extension) and 2.15 (Expr : Type). Next, I shall prove that each of the type constructors shown in Figure 2.4 is a type or produces a type when applied to valid arguments. In each case, I have to show that types are preserved under store extension (as specified by Definition 2.14). I shall first prove some properties of store extension that we will need later, namely that store extension is both reflexive and transitive.

Lemma 2.16 (Store Extension Reflexive)
$S \sqsubseteq S$.

Proof: Immediate. □

Lemma 2.17 (Store Extension Transitive)
If $S_1 \sqsubseteq S_2$ and $S_2 \sqsubseteq S_3$ then $S_1 \sqsubseteq S_3$.

Proof: For any location $\ell \in \mathrm{dom}(S_1)$ we must show that $\ell \in \mathrm{dom}(S_3)$ and $S_1(\ell) = S_3(\ell)$. Suppose $\ell \in \mathrm{dom}(S_1)$. From $S_1 \sqsubseteq S_2$ we have $\ell \in \mathrm{dom}(S_2)$ and that $S_1(\ell) = S_2(\ell)$. Since $\ell \in \mathrm{dom}(S_2)$, from $S_2 \sqsubseteq S_3$ it follows that $\ell \in \mathrm{dom}(S_3)$ and that $S_2(\ell) = S_3(\ell)$. By the transitivity of equality it follows that $S_1(\ell) = S_3(\ell)$. □

The fact that bool is a type follows immediately from its definition: whenever $\langle S, v\rangle \in \mathsf{bool}$ and $S \sqsubseteq S'$, we have $\langle S', v\rangle \in \mathsf{bool}$ since whether or not v is a boolean is independent of the current store. To prove that $\tau_1 \to \tau_2$ is a type (assuming τ_1 and τ_2 are types) we will need to make use of Lemma 2.17 which says that store extension is transitive.

Lemma 2.18 (Type $\tau_1 \to \tau_2$)
If τ_1 and τ_2 are types then $\tau_1 \to \tau_2$ is also a type.

PROOF: We have to show that $\tau_1 \to \tau_2$ is closed under store extension. Suppose $\langle S, v\rangle \in \tau_1 \to \tau_2$ and that $S \sqsubseteq S'$. Note that v must be a closed abstraction of the form $\lambda x.e$. We have to show that $\langle S', \lambda x.e\rangle \in \tau_1 \to \tau_2$. Suppose $S' \sqsubseteq S''$ and $\langle S'', v_1\rangle \in \tau_1$ for some store S'' and value v_1 — by the definition of $\to$ we have to show that $e[v_1/x] :_{S''} \tau_2$. By Lemma 2.17 ($\sqsubseteq$ transitive) we have $S \sqsubseteq S''$. By the definition of $\to$, $\langle S, \lambda x.e\rangle \in \tau_1 \to \tau_2$ together with $S \sqsubseteq S''$ and $\langle S'', v_1\rangle \in \tau_1$ implies $e[v_1/x] :_{S''} \tau_2$. □

Next, we prove that $\mathsf{box}\ \tau$ is a type as long as τ is a type.

Lemma 2.19
If τ is a type, then $\mathsf{box}\ \tau$ is a type.

PROOF: To show that $\mathsf{box}\ \tau$ is closed under store extension, suppose $\langle S, v\rangle \in \mathsf{box}\ \tau$ and $S \sqsubseteq S'$. We have to show that $\langle S', v\rangle \in \mathsf{box}\ \tau$. Note that v must be some store location ℓ. From $\langle S, \ell\,\rangle \in \mathsf{box}\ \tau$ we have $\ell \in \mathrm{dom}(S)$ and $\langle S, S(\ell)\rangle \in \tau$. Since τ is a type (i.e., it is closed under store extension), we have $\langle S', S(\ell)\rangle \in \tau$. From $\ell \in \mathrm{dom}(S)$ and $S \sqsubseteq S'$ it follows that $\ell \in \mathrm{dom}(S')$ and $S(\ell) = S'(\ell)$. Hence, $\langle S', S'(\ell)\rangle \in \tau$. Now, by the definition of box, we have $\langle S', \ell\,\rangle \in \mathsf{box}\ \tau$. □

2.2.3 Judgments, Typing Rules, and Safety

Up to now I have only dealt with closed λ^I terms, as these are the ones that are evaluated at run time. Now I turn to terms with free variables, upon which the static type-checking rules must operate. Type judgments in λ^I have the form $\Gamma \vDash_I e : \tau$ where Γ is a type environment, that is, a mapping from variables to types. As before, a value environment (ground substitution) is a mapping from lambda calculus variables to values and the judgment $\Gamma \vDash_I e : \tau$ can be viewed as a three place relation that holds on the type environment Γ, the term e, and the type τ.

Definition 2.20 (Semantics of Judgment)
For any type environment Γ and value environment σ I write $\sigma :_S \Gamma$ if for all variables $x \in \mathrm{dom}(\Gamma)$ we have $\sigma(x) :_S \Gamma(x)$; that is,

$$\sigma :_S \Gamma \quad \stackrel{\text{def}}{=} \quad \forall x \in \mathrm{dom}(\Gamma).\ \sigma(x) :_S \Gamma(x)$$

I write $\Gamma \vDash_I e : \tau$ *iff* $FV(e) \subseteq \mathrm{dom}(\Gamma)$ *and*

$$\forall \sigma, S.\ \sigma :_S \Gamma \implies \sigma(e) :_S \tau$$

where $\sigma(e)$ *is the result of replacing the free variables in* e *with their values under* σ*.*

Finally, I write $\vDash_I e : \tau$ *to mean* $\Gamma_0 \vDash_I e : \tau$ *for the empty type environment* Γ_0*.*

The typing rules for λ^I are given in Figure 2.5. As in the previous section, I write $\Gamma\,[x \mapsto \tau]$ to denote the type environment that extends Γ by mapping x to τ where $x \notin \mathrm{dom}(\Gamma)$. There are two ways in which the type-checking rules in Figure 2.5 differ from the typing rules one is accustomed to (in syntactic type theories) for a language with references — that is, in addition to the fact that I use $\vDash_I$ rather than $\vdash_I$. First, the type judgment does not contain a store typing Ψ that maps locations in the store to the types of the values that they may contain — that is, type judgments have the form $\Gamma \vDash_I e : \tau$ rather than $\Gamma; \Psi \vDash_I e : \tau$. Second, there is no typing rule for locations, that is, with a conclusion of form $\Gamma \vDash_I \ell : \mathtt{box}\ \tau$. These differences can be explained as follows. Our type-checking rules are used to determine if a "program" is well-typed. A program in λ^I is a closed term that does not contain any location symbols ℓ, though, once a program begins executing it may step (by means of `new`) to expressions that do contain location symbols. A conventional subject-reduction proof requires that the type system be able to type-check programs *during* execution, which means that there must be a way to type-check locations ℓ. Since we are not doing subject reduction, we do not need to type-check executing programs. Hence, we do not need to type-check locations, which means that we need neither a store typing Ψ, nor a typing rule for locations.[3]

I shall prove each of the type inference rules in Figure 2.5 as lemmas (see Section 2.2.4). These lemmas can then be used as standard type inference rules to prove statements of the form $\Gamma \vDash_I e : \tau$.

Our goal is to prove that a program is safe to execute given an initial store S. There are no additional constraints on S, in fact, S might as well be the empty store because the program e — since it has no free variables or location symbols — cannot access any locations in S. Hence, we need to prove that given an arbitrary store S and a well-typed program e (with some type τ), machine state (S, e) is

[3] The fact that type judgments in λ^I do not require a store typing Ψ may also be explained as a benefit of using a possible-worlds model (where types are sets of store-value pairs) rather than a model in which types are simply sets of values. A model in which types are simply sets of values can neither keep track of which locations are allocated at different points during the computation, nor specify how the store is allowed to change over time (as the $\sqsubseteq$ relation does). In such a model, type judgments would need a store typing Ψ in order to track allocated locations and to somehow ensure that their types are preserved over the course of evaluation. The semantics of such type judgments would have to quantify over stores S whose contents are of the types specified by Ψ.

$$\frac{}{\Gamma \vDash_I x : \Gamma(x)} \; \texttt{(I-var)}$$

$$\frac{}{\Gamma \vDash_I \texttt{true} : \mathsf{bool}} \; \texttt{(I-true)} \qquad \frac{}{\Gamma \vDash_I \texttt{false} : \mathsf{bool}} \; \texttt{(I-false)}$$

$$\frac{\Gamma\,[x \mapsto \tau_1] \vDash_I e : \tau_2}{\Gamma \vDash_I \lambda x.e : \tau_1 \longrightarrow \tau_2} \; \texttt{(I-abs)}$$

$$\frac{\Gamma \vDash_I e_1 : \tau_1 \longrightarrow \tau_2 \qquad \Gamma \vDash_I e_2 : \tau_1}{\Gamma \vDash_I (e_1\, e_2) : \tau_2} \; \texttt{(I-app)}$$

$$\frac{\Gamma \vDash_I e : \tau}{\Gamma \vDash_I \texttt{new}(e) : \mathsf{box}\ \tau} \; \texttt{(I-new)} \qquad \frac{\Gamma \vDash_I e : \mathsf{box}\ \tau}{\Gamma \vDash_I \,!\,e : \tau} \; \texttt{(I-deref)}$$

Figure 2.5: Immutable References (λ^I): Type-checking Lemmas

safe. Notice that the safety theorem requires that τ be a valid type. One can prove type(τ) using the lemmas in Section 2.2.2.

Theorem 2.21 (Safety)
If $\vDash_I e : \tau$, τ is a type, and S is a store, then (S, e) is safe.

Proof: To prove safe(S, e) we must show that for any state (S', e') reachable in some number of steps from (S, e), either e' is a value or another step is possible. Suppose $(S, e) \longmapsto_I^* (S', e')$. If (S', e') is not irreducible, then there must exist some (S'', e'') such that $(S', e') \longmapsto_I (S'', e'')$. Otherwise, (S', e') is irreducible and we must prove that e' is a value. From $\vDash_I e : \tau$ we have $\Gamma_0 \vDash_I e : \tau$ and it follows that e is closed (since Γ_0 is the empty type environment). Choose the empty value environment σ_0 and store S. By the definition of $\vDash_I$ we have $\sigma_0 :_S \Gamma_0 \implies \sigma_0(e) :_S \tau$. The premise $\sigma_0 :_S \Gamma_0$ is trivially satisfied; applying the trivial substitution we have $e :_S \tau$. Since $(S, e) \longmapsto_I^* (S', e')$ and irred(S', e') it follows by Definition 2.15 (Expr : Type) that $S \sqsubseteq S'$ and $\langle S', e' \rangle \in \tau$. Since τ is a type it follows that val(e'). $\square$

2.2.4 Validity of Typing Rules

It only remains for us to prove each of the typing rules in Figure 2.4 as lemmas. The lemma for variables follows immediately from the definition of $\models_I$. The lemmas for `I-true` and `I-false` are immediate from the definition of $\models_I$ and type bool.

Theorem 2.22 (Abstraction)
Let Γ be a type environment, let τ_1 and τ_2 be types, and let $\Gamma\,[x \mapsto \tau_1]$ be the type environment that is identical to Γ except that it maps x to τ_1 (where $x \notin \mathrm{dom}(\Gamma)$). If $\Gamma\,[x \mapsto \tau_1] \models_M e : \tau_2$ then $\Gamma \models_M \lambda x.e : \tau_1 \to \tau_2$.

Proof: We must show that under the premises of the theorem, for any σ and S such that $\sigma :_S \Gamma$, we have $\sigma(\lambda x.e) :_S \tau_1 \to \tau_2$. Suppose $\sigma :_S \Gamma$. Let v and S' be such that $S \sqsubseteq S'$ and $\langle S', v\rangle \in \tau_1$. By the definition of $\to$ it now suffices to show that $\sigma(e[v/x]) :_{S'} \tau_2$. Let $\sigma\,[x \mapsto v]$ be the substitution identical to σ except that it maps x to v. Since the codomain of Γ contains types (which are closed under store extension), and since $v :_{S'} \tau_1$, we now have that $\sigma\,[x \mapsto v] :_{S'} \Gamma\,[x \mapsto \tau_1]$. This, together with the premise $\Gamma\,[x \mapsto \tau_1] \models_I e : \tau_2$, allows us to conclude that $\sigma\,[x \mapsto v](e) :_{S'} \tau_2$. But this implies $\sigma(e[v/x]) :_{S'} \tau_2$. □

Theorem 2.23 (Application)
If Γ is a type environment, e_1 and e_2 are (possibly open) terms, and τ_1 and τ_2 are types such that $\Gamma \models_I e_1 : \tau_1 \to \tau_2$ and $\Gamma \models_I e_2 : \tau_2$ then $\Gamma \models_I (e_1\, e_2) : \tau_2$.

Proof: We must show that under the premises of the theorem, for any value environment σ and store S such that $\sigma :_S \Gamma$, we have $\sigma(e_1\, e_2) :_S \tau_2$. Suppose $\sigma :_S \Gamma$. Also, suppose $(S, \sigma(e_1\, e_2)) \longmapsto_I^* (S', e')$ and $\mathrm{irred}(S', e')$ — we must show that $S \sqsubseteq S'$ and $\langle S', e'\rangle \in \tau_2$. The proof is in three parts. Informally, these correspond to (1) reducing e_1 to a value (say e_1'), (2) reducing e_2 to a value (say e_2'), and (3) applying the function value e_1' to the argument value e_2' and reducing the beta-reduced expression to a value.

1. By the operational semantics it follows that $(S, \sigma(e_1)) \longmapsto_I^* (S_1', e_1')$ and $\mathrm{irred}(S_1', e_1')$. From the premise $\Gamma \models_I e_1 : \tau_1 \to \tau_2$ and $\sigma :_S \Gamma$ we have $\sigma(e_1) :_S \tau_1 \to \tau_2$, so by Definition 2.15 (Expr : Type) we have $\langle S_1', e_1'\rangle \in \tau_1 \to \tau_2$. From the definition of $\to$ it follows that e_1' is of the form $\lambda x.e_{11}$.

2. By the operational semantics we also have $(S_1', \sigma(e_2)) \longmapsto_I^* (S_2', e_2')$ and $\mathrm{irred}(S_2', e_2')$. From $\sigma :_S \Gamma$, $S \sqsubseteq S_1'$, and the fact that the codomain of Γ contains types (which are closed under store extension) it follows that $\sigma :_{S_1'} \Gamma$. This, together with the premise $\Gamma \models_I e_2 : \tau_1$ gives us $\sigma(e_2) :_{S_1'} \tau_1$. Hence, by Definition 2.15 (Expr : Type) we have $S_1' \sqsubseteq S_2'$ and $\langle S_2', e_2'\rangle \in \tau_1$.

3. $(S_2', (\lambda x.e_{11})e_2') \longmapsto_I (S_2', e_{11}[e_2'/x]) \longmapsto_I^* (S', e')$. By the definition of $\rightarrow$, from $\langle S_1', \lambda x.e_{11}\rangle \in \tau_1 \rightarrow \tau_2$, $S_1' \sqsubseteq S_2'$, and $\langle S_2', e_2'\rangle \in \tau_2$ we have $e_{11}[e_2'/x] :_{S_2'} \tau_2$. Then, by Definition 2.15 (Expr : Type) it follows that $S_2' \sqsubseteq S'$ and $\langle S', e'\rangle \in \tau_2$.

Finally, we have $S \sqsubseteq S_1' \sqsubseteq S_2' \sqsubseteq S'$ and store extension is transitive (Lemma 2.17). Hence, we have $S \sqsubseteq S'$ and also $\langle S', e'\rangle \in \tau_2$ as we wanted to show. □

Lemma 2.24 (Closed New)
If e is a closed term and τ is a type such that $e :_S \tau$ then $\texttt{new}(e) :_S \textsf{box}\ \tau$.

Proof: Suppose $(S, \texttt{new}(e)) \longmapsto_I^* (S', \ell)$ where $\mathrm{irred}(S', \ell)$; we must show that $S \sqsubseteq S'$ and $\langle S', \ell\rangle \in \textsf{box}\ \tau$. The proof is in two parts. Informally, these correspond to (1) reducing e to a value and (2) allocating a new store location initialized to that value.

1. By the operational semantics it follows that $(S, e) \longmapsto_I^* (S_1, e_1)$ and $\mathrm{irred}(S_1, e_1)$. From the premise $e :_S \tau$ we have $S \sqsubseteq S_1$ and $\langle S_1, e_1\rangle \in \tau$.

2. By the operational semantics we also have $(S_1, \texttt{new}(e_1)) \longmapsto_I (S_1 [\ell \mapsto e_1], \ell)$ where $\ell \notin \mathrm{dom}(S_1)$. Since ℓ is a value we have $\mathrm{irred}(S_1 [\ell \mapsto e_1], \ell)$. Note that $S' \equiv S_1 [\ell \mapsto e_1]$. Clearly, $S_1 \sqsubseteq S'$; then, by the transitivity of $\sqsubseteq$ (Lemma 2.17), $S \sqsubseteq S'$ as we wanted to show.

   Since τ is a type, from $\langle S_1, e_1\rangle \in \tau$ and $S_1 \sqsubseteq S'$ it follows that $\langle S', e_1\rangle \in \tau$. Since $S'(\ell) = e_1$, we have $\langle S', S'(\ell)\rangle \in \tau$ which, together with $\ell \in S'$, implies $\langle S', \ell\rangle \in \textsf{box}\ \tau$ as we wanted to show.

□

Theorem 2.25 (New)
If Γ is a type environment, e is a (possibly open) term, and τ is a type such that $\Gamma \models_I e : \tau$, then $\Gamma \models_I \texttt{new}(e) : \textsf{box}\ \tau$.

Proof: We must prove that for any value environment σ and store S such that $\sigma :_S \Gamma$ we have $\sigma(\texttt{new}(e)) :_S \textsf{box}\ \tau$, or equivalently, $\texttt{new}(\sigma(e)) :_S \textsf{box}\ \tau$. Suppose $\sigma :_S \Gamma$. From the premise $\Gamma \models_I e : \tau$ we have $\sigma(e) :_S \tau$. Since $\sigma(e)$ is a closed term the result now follows from Lemma 2.24. □

Lemma 2.26 (Closed Deref)
If e is a closed term and τ is a type such that $e :_S \textsf{box}\ \tau$ then $!\,e :_S \tau$.

Proof: Suppose $(S, !\,e) \longmapsto_I^* (S', e')$ such that $\mathrm{irred}(S', e')$. We must prove that $S \sqsubseteq S'$ and $\langle S', e' \rangle \in \tau$. Informally, the proof reasoning involves (1) reducing e to a location ℓ and (2) reading the contents at location ℓ in the store. More formally, by the operational semantics we have $(S, e) \longmapsto_I^* (S', \ell)$ and $\mathrm{irred}(S', \ell)$. Hence, from the premise $e :_S \mathsf{box}\ \tau$ we may conclude that $S \sqsubseteq S'$ and $\langle S', \ell \rangle \in \mathsf{box}\ \tau$. By the definition of $\mathsf{box}\ \tau$ we have $\ell \in \mathrm{dom}(S')$ and $\langle S', S'(\ell) \rangle \in \tau$. By the operational semantics we also have $(S', !\,\ell) \longmapsto_I (S', e')$ where $e' = S'(\ell)$. Hence, $\langle S', e' \rangle \in \tau$. □

Theorem 2.27 (Deref)
If Γ *is a type environment,* e *is a (possibly open) term, and* τ *is a type such that* $\Gamma \vDash_I e : \mathsf{box}\ \tau$*, then* $\Gamma \vDash_I !\,e : \tau$*.*

Proof: Follows from Lemma 2.26 in the same manner that Theorem 2.25 (New) follows from Lemma 2.24 (Closed New). □

2.2.5 Discussion: Kripke Logical Relations

In Section 2.1.6, I discussed the correspondence between logical relations and our semantic model of the λ-calculus with booleans. Analogously, there is a correspondence between Kripke logical relations and the possible-worlds model I have developed for λ^I. Recall that a logical relation $\mathcal{R} = \{R^\tau\}$ is a family of relations R^τ for each type τ in the language. For a Kripke logical relation, instead of having a relation for each type, one must have a relation for each type and possible world. Specifically, a (unary) Kripke logical relation over a set A is defined as follows. Suppose that we have a set of worlds W, an accessibility relation $Acc \subseteq W \times W$, and a family $\{i^\tau_{w\,w'}\}$ of *transition functions* where w and w' are worlds such that $Acc(w, w')$. A transition function maps some element $a \in A$ in world w to some element $a' \in A$ in world w'. Then a Kripke logical relation is a family $\mathcal{R} = \{R^\tau_w\}$ of relations R^τ_w indexed by types τ and worlds $w \in W$, satisfying the following condition:

(Monotonicity) $R^\tau_w(a)$ implies $R^\tau_{w'}(i^\tau_{w\,w'}(a))$ for all w' such that $Acc(w, w')$.

The monotonicity condition says that when $Acc(w, w')$, the relation R^τ_w is contained in $R^\tau_{w'}$, modulo the transition function. (See Mitchell [Mit96] for a detailed description of Kripke logical relations.)

To prove that well-typed λ^I terms are safe one would define a Kripke logical relation as follows.

- Define a Kripke logical relation over the set $A = \{\, e \mid e \text{ is a } \lambda^I \text{ term} \,\}$.

- Pick the set of worlds $W = \mathit{Store}$, that is, worlds w correspond to stores S.

- Let the accessibility relation $\mathit{Acc} = \sqsubseteq$.

- Let the transition functions correspond to the $\longmapsto_I$ relation, that is, $i^{\tau}_{S\,S'}(e) = e'$ whenever $(S, e) \longmapsto^*_I (S', e')$.

Next, for each type τ and store S, one defines a set (i.e., a unary relation) R^{τ}_S of closed terms e that have type τ with respect to some store S:

- $R^{\mathsf{bool}}_S(e)$ iff for all S' and e' such that $(S, e) \longmapsto^*_I (S', e')$ and $\mathrm{irred}(S', e')$, we have $S \sqsubseteq S'$ and either $e' = \texttt{true}$ or $e' = \texttt{false}$.

- $R^{\tau_1 \to \tau_2}_S(e)$ iff for all S' and e' such that $(S, e) \longmapsto^*_I (S', e')$ and $\mathrm{irred}(S', e')$, we have $S \sqsubseteq S'$ and $e' = \lambda x.e_2$, and for all stores S_1 such that $S \sqsubseteq S_1$ whenever $R^{\tau_1}_{S_1}(e_1)$, we have $R^{\tau_2}_{S_1}(e\, e_1)$.

- $R^{\mathsf{box}\,\tau}_S(e)$ iff , for all S' and e' such that $(S, e) \longmapsto^*_I (S', e')$ and $\mathrm{irred}(S', e')$, we have $S \sqsubseteq S'$ and $e' = \ell$, and for all stores S_1 such that $S \sqsubseteq S_1$, $R^{\tau}_{S_1}(!\, e)$.

Notice that $R^{\tau}_S(e)$ corresponds to $e :_S \tau$ (Definition 2.15) in our semantic model. The monotonicity condition for our Kripke logical relation is as follows:

$$\text{If } R^{\tau}_S(e) \text{ and } (S, e) \longmapsto^*_I (S', e'), \text{ then } R^{\tau}_{S'}(e') \text{ for all } S' \text{ such that } S \sqsubseteq S'.$$

If the monotonicity condition holds, the above definition implies that the relation R^{τ}_S consists of all expressions e such that if $(S, e) \longmapsto^*_I (S', e')$, then $S \sqsubseteq S'$ and $e' \in R^{\tau}_{S'}$.

As in Section 2.1.6, the proof is in two parts. First, we prove that if $R^{\tau}_S(e)$ then $\mathrm{safe}(S, e)$ (which corresponds to Theorem 2.21). The proof follows easily from the definition of R^{τ}_S. Next, we show that every well-typed term e of type τ is an element of R^{τ}_S for some store S. The proof is by induction on the type derivation and (as for Lemma 2.11 in Section 2.1.6) the cases of this proof, namely for typing rules `I-var`, `I-true`, `I-false`, `I-abs`, `I-app`, `I-new`, and `I-deref`, bear a close resemblance to the proofs of the typing rule lemmas we proved in Section 2.2.4.

In order to prove this last lemma, we must prove that the Kripke monotonicity condition holds. Informally, the proof follows from the fact that monotonicity over the course of evaluation plays a fundamental role at every "layer" of our model. If S is the current store and S' is some possible future store then:

- The lowest layer of the model, relying on the absence of cell deallocation and update, specifies that if ℓ maps to v in S then ℓ maps to v in S'. This is captured by valid store extension ($\sqsubseteq$) which establishes the monotonicity of stores.

- The next layer of the model shows that if $\langle S, v\rangle \in \tau$ then $\langle S', v\rangle \in \tau$. For each type τ, we prove the monotonicity of sets of values of that type. To prove this property (specified as type(τ)) we must rely on the monotonicity of stores.

- The next layer establishes type preservation, that is, if e has type τ with respect to store S and we reach a state (S', e'), then e' has type τ with respect S'. This corresponds precisely to Kripke monotonicity. We prove the monotonicity (over the course of evaluation) of sets of expressions of each type. The proof relies on the monotonicity of sets of values of each type.

Hence, in λ^I we ultimately rely on the absence of deallocation and update to establish monotonicity at each "layer" of the model. Both cell updates and deallocation are features that make monotonicity (and hence, type invariance) much harder to establish. In subsequent chapters, I will describe how to prove type invariance in the presence of features that make monotonicity harder to establish, namely cell updates (Chapter 3) and deallocation (Chapter 7).

Chapter 3

Mutable References: Mutation & Quantified Types

Almost all practical programming languages use mutable references.[1] Object-oriented languages such as Java and functional languages such as ML permit general references — that is, mutable memory cells that may contain values of any (statically-checked) closed type, including recursive types, functions, and even other references. Therefore, general references are essential in our plans to build foundational PCC systems for practical languages. In a foundational PCC system we want to prove the safety of low-level programs, that is, programs written in the *target* language of a type-preserving compiler. Target languages for type-preserving compilation of languages like ML and Java must not only support mutable references, but also quantified types. Universal types, for instance, are useful for encoding ML parametric polymorphism and Java inheritance [Lea02], while existential types are essential for encoding ML function closures [MMH96, MWCG99] and abstract data types [MP88], and also for Java object encoding [BCP99, Lea02].

Quantified types may be *predicative* or *impredicative*. Let type $\tau = \exists\alpha.\tau'$ (where α may occur free in τ'). We say that τ is an impredicative existential type if α may be instantiated with any type, *including* τ *itself*. The definition of impredicative universal types is analogous. More generally,

> *a definition (of a set, a type, etc.) is called "impredicative" if it involves a quantifier whose domain includes the very thing being defined* [Pie02].

While high-level languages such as ML and Java do not themselves support impredicative polymorphism, this feature is essential in a target language for type-preserving compilation of these languages. Consider, for instance, type-preserving compilers for higher-order functional languages (such as ML). As part of the type-preserving translation of higher-order functions into function closures — known as

[1] Haskell [HPW91] is an exception.

typed closure conversion — functions with free variables are replaced by code abstracted on an extra environment parameter; free variables in the body of the function are replaced by references to the environment; and the function is converted into a "closure", that is, a data structure that contains the code and a representation of the code's environment. An important property of closure conversion is that the representation of the environment is *private* to the closure. Hence, typed closure conversion uses an existential quantifier to *hide* the type of the function environment [MMH96, MWCG99]. An ML function of type $\tau_1 \to \tau_2$ would be translated to an existential package with two components as shown below — I write $\mathcal{C}[\![\tau]\!]$ to denote the translation of a source language type τ. The first component is the code which has type $\mathtt{code}(\tau_{env}, \mathcal{C}[\![\tau_1]\!], \mathcal{C}[\![\tau_2]\!])$ indicating that it takes an environment of type τ_{env} and an argument of type $\mathcal{C}[\![\tau_1]\!]$, and returns a result of type $\mathcal{C}[\![\tau_2]\!]$. The second is the environment which has type τ_{env}.

$$\mathcal{C}[\![\tau_1 \to \tau_2]\!] \quad = \quad \exists \tau_{env}.\, \langle \mathtt{code}(\tau_{env}, \mathcal{C}[\![\tau_1]\!], \mathcal{C}[\![\tau_2]\!]) \times \tau_{env} \rangle$$

The environment of a function closure f can contain other function closures, including f itself — that is, the existential can be instantiated with other existential types, including itself — so that the quantification must be impredicative. Furthermore, since ML has mutable references, the environment of a function closure can contain a mutable reference. A mutable reference can, in turn, store a function closure. In the target language, therefore, mutable references must be able to contain impredicative existentials and vice versa.[2]

In Chapter 2 (Section 2.2) I described a model for a language with dynamic allocation of immutable cells. To build a foundational PCC system (based on the semantic approach) for a practical high-level language such as ML or Java, we have to construct a semantic model of a language that supports:

- dynamic heap allocation of mutable cells;
- impredicative polymorphism and general references (mutable cells that can store functions, other mutable references, impredicative quantified types, etc.);
- recursive types (both covariant and contravariant).

Semantic models of general references have posed a challenge to semanticists for years [FJM$^+$96, TG00]. Building semantic models of impredicative polymorphism is also difficult [CGW89], but modeling the combination of the two features seems especially tricky. Recently, two models of mutable references have emerged, built

[2] There may be other, as yet undiscovered, ways of representing function closures, but for now it seems we needs mutable references to impredicative quantified types, regardless of whether we adopt a *type-passing* interpretation of polymorphism [MMH96] or one based on *type-erasure* [MWCG99].

$$\begin{array}{llll}
\textit{Expressions} & e & ::= & x \mid \ell \mid \texttt{unit} \mid \lambda x.e \mid (e_1\, e_2) \mid \texttt{new}(e) \mid {!}\, e \mid e_1 := e_2 \\
 & & & \mid \Lambda.e \mid e[\,] \mid \texttt{pack}\, e \mid \texttt{unpack}\, e_1 \,\texttt{as}\, x \,\texttt{in}\, e_2 \\
\textit{Values} & v & ::= & \ell \mid \texttt{unit} \mid \lambda x.e \mid \Lambda.e \mid \texttt{pack}\, v
\end{array}$$

Figure 3.1: Mutable References (λ^M): Syntax

using game semantics [AHM98] and category theory [Lev02] respectively, but neither of these supports the storage of polymorphic values in mutable cells.

This chapter examines the challenges of modeling mutable references that may store values of any closed type (including impredicative quantified types). To simplify the presentation, instead of directly describing the von Neumann model that we have built for our foundational PCC system (see Chapter 6), I will first explain how to construct a semantic model of the polymorphic λ-calculus augmented with mutable references. The material in this chapter is based on research done jointly with Andrew Appel and Roberto Virga [AAV02, AAV03].

3.1 Polymorphic λ-calculus with Mutable Cells

3.1.1 Syntax

The language I consider is the polymorphic lambda-calculus (also known as System F) augmented with mutable references, existentials, and the constant `unit`. I call this language λ^M. I have omitted booleans and immutable references from the language, but these types along with cartesian products, intersection and union types, and other standard constructions are easy to add to the model I shall present. The syntax of lambda terms is given by the grammar shown in Figure 3.1. I use the metavariable x to range over a countably infinite set *Var* of variables and the metavariable ℓ to range over a countably infinite set *Loc* of locations. A term v is a *value* if it is a location ℓ, the constant `unit`, a term or type abstraction, or an existential package, and if it contains no free term variables x.

This syntax is slightly unconventional: in most presentations the polymorphic operators $\Lambda\alpha.e$ and $e[\tau]$ and the existential operators $\texttt{pack}[\tau, e]\,\texttt{as}\,\exists\alpha.\tau'$ and $\texttt{unpack}\, e\, \texttt{as}\, [\tau, x]\, \texttt{in}\, e'$ mention types syntactically. I want to give purely semantic (not syntactic) typings to *untyped* lambda calculus, so I omit all types from the syntax. In general, I let the vestigial operators remain in the untyped syntax in order to preserve the operational semantics. For instance, the term $\Lambda.e$ is a suspended computation (normally written $\Lambda\alpha.e$); $e[\,]$ runs the suspended computation.

3.1.2 Operational Semantics

A *store* S is a finite map from locations to closed values. The domain of S includes all locations already allocated; locations not in $\mathrm{dom}(S)$ are available to be allocated by `new` in future computation steps. The small-step semantics for λ^M is given by an abstract machine whose state is described by a pair (S, e) of a store and a closed term. The notation $(S, e) \longmapsto_M (S', e')$ denotes a single operational step. I write $(S, e) \longmapsto_M^j (S', e')$ to denote that there exists a chain of j steps of the form $(S, e) \longmapsto_M (S_1, e_1) \longmapsto_M \ldots \longmapsto_M (S_j, e_j)$ where S_j is S' and e_j is e'. I write $(S, e) \longmapsto_M^* (S', e')$ if $(S, e) \longmapsto_M^j (S', e')$ for some $j \geq 0$.

The operational semantics is given in Figure 3.2 and is completely standard.[3] There are only two rules in the operational semantics that produce side-effects. The first is the rule that evaluates terms of the form `new`(v) (see `MO-new2`); it extends the store with a previously unallocated location ℓ initialized to v. The second is the rule that evaluates terms of the form $\ell := v$ (see `MO-assign3`); it checks that ℓ is an allocated cell and updates the store so ℓ maps to v.

In the examples that follow, I use the notation `let` x `=` e_1 `in` e_2 as syntactic sugar for $(\lambda x.e_2)e_1$. I write `let _ =` e_1 `in` e_2 in place of $(\lambda x.e_2)e_1$ when the variable x does not occur free in e_2.

3.2 Towards a Model of Mutable References

This section examines the challenges of modeling general references and gradually closes in on a semantics that supports such references.

3.2.1 Need for a New Model

The model that I built for λ^I (λ-calculus with immutable references) in Section 2.2 cannot be used to reason about the safety of programs written in an imperative language — that model prohibits updates to allocated cells. I could change the model for λ^I so that it permits updates to store locations, but such updates may lead to inconsistency in the presence of aliasing. To see why, consider the λ^M program in Figure 3.3(a), in particular, the instruction $x := y$ — the variable x points to an allocated cell and the assignment $x := y$ updates the cell with the value y. (The comments, which I shall explain momentarily, describe invariants

[3] The reader may not consider the rules `MO-tapp2` (type application) and `MO-unpack2` (unpacking an existential) completely standard. On a real machine, there are no such instructions. In Chapter 4 (see Section 4.1) I will describe how the semantic model must be changed to accommodate an operational semantics that treats type application and existential unpacking as virtual instructions, or coercions.

$$\frac{(S,\, e_1) \longmapsto_M (S',\, e_1')}{(S,\, e_1\, e_2) \longmapsto_M (S',\, e_1'\, e_2)}\ \texttt{(MO-app1)}$$

$$\frac{(S,\, e_2) \longmapsto_M (S',\, e_2')}{(S,\, (\lambda x.e_1)\, e_2) \longmapsto_M (S',\, (\lambda x.e_1)\, e_2')}\ \texttt{(MO-app2)}$$

$$\frac{}{(S,\, (\lambda x.e)\, v) \longmapsto_M (S,\, e[v/x])}\ \texttt{(MO-app3)}$$

$$\frac{(S,\, e) \longmapsto_M (S',\, e')}{(S,\, \texttt{new}(e)) \longmapsto_M (S',\, \texttt{new}(e'))}\ \texttt{(MO-new1)}$$

$$\frac{\ell \notin \mathrm{dom}(S)}{(S,\, \texttt{new}(v)) \longmapsto_M (S\,[\ell \mapsto v],\, \ell)}\ \texttt{(MO-new2)}$$

$$\frac{(S,\, e) \longmapsto_M (S',\, e')}{(S,\, !\,e) \longmapsto_M (S',\, !\,e')}\ \texttt{(MO-deref1)} \qquad \frac{\ell \in \mathrm{dom}(S)}{(S,\, !\,\ell) \longmapsto_M (S,\, S(\ell))}\ \texttt{(MO-deref2)}$$

$$\frac{(S,\, e_1) \longmapsto_M (S',\, e_1')}{(S,\, e_1 := e_2) \longmapsto_M (S',\, e_1' := e_2)}\ \texttt{(MO-assign1)}$$

$$\frac{(S,\, e_2) \longmapsto_M (S',\, e_2')}{(S,\, v_1 := e_2) \longmapsto_M (S',\, v_1 := e_2')}\ \texttt{(MO-assign2)}$$

$$\frac{\ell \in \mathrm{dom}(S)}{(S,\, \ell := v) \longmapsto_M (S\,[\ell \mapsto v],\, \texttt{unit})}\ \texttt{(MO-assign3)}$$

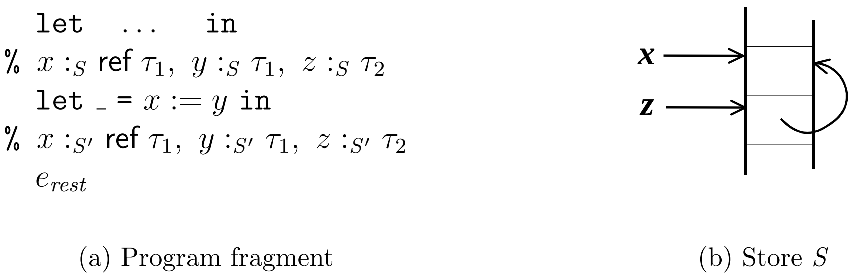
$$\frac{(S,\, e) \longmapsto_M (S',\, e')}{(S,\, e[\,]) \longmapsto_M (S',\, e'[\,])}\ \texttt{(MO-tapp1)} \qquad \frac{}{(S,\, (\Lambda.e)[\,]) \longmapsto_M (S,\, e)}\ \texttt{(MO-tapp2)}$$

$$\frac{(S,\, e) \longmapsto_M (S',\, e')}{(S,\, \texttt{pack}\, e) \longmapsto_M (S',\, \texttt{pack}\, e')}\ \texttt{(MO-pack)}$$

$$\frac{(S,\, e_1) \longmapsto_M (S',\, e_1')}{(S,\, \texttt{unpack}\, e_1\, \texttt{as}\, x\, \texttt{in}\, e_2) \longmapsto_M (S',\, \texttt{unpack}\, e_1'\, \texttt{as}\, x\, \texttt{in}\, e_2)}\ \texttt{(MO-unpack1)}$$

$$\frac{}{(S,\, \texttt{unpack}\, (\texttt{pack}\, v)\, \texttt{as}\, x\, \texttt{in}\, e_2) \longmapsto_M (S,\, e_2[v/x])}\ \texttt{(MO-unpack2)}$$

Figure 3.2: Mutable References (λ^M): Operational Semantics



$\texttt{let} \quad \ldots \quad \texttt{in}$

$\texttt{\%}\ x :_S \mathsf{ref}\ \tau_1,\ \ y :_S \tau_1,\ \ z :_S \tau_2$

$\texttt{let \_ =}\ x := y\ \texttt{in}$

$\texttt{\%}\ x :_{S'} \mathsf{ref}\ \tau_1,\ \ y :_{S'} \tau_1,\ \ z :_{S'} \tau_2$

e_{rest}

(a) Program fragment

(b) Store S

Figure 3.3: Store Update in the Presence of Aliasing

that should hold before and after the update.) Suppose that the typing derivation for the program establishes the following:

1. $\Gamma \vDash_M \texttt{let \_ =}\ x := y\ \texttt{in}\ e_{rest} : \tau'$

2. $\Gamma \vDash_M e_{rest} : \tau'$

where type environment $\Gamma = \{x \mapsto \mathsf{ref}\ \tau_1,\ y \mapsto \tau_1,\ z \mapsto \tau_2\}$.

Let S and S' denote the stores immediately before and after the evaluation of $x := y$. By the semantics of λ^I judgments, judgment (1) implies that $x :_S \mathsf{ref}\ \tau_1$ (variable x has type $\mathsf{ref}\ \tau_1$ with respect to store S), $y :_S \tau_1$, and $z :_S \tau_2$ (see the comments in Figure 3.3(a)). Judgment (2) implies that x, y, and z should have the same types as before, but now with respect to store S'. Hence, we need a semantic model that allows us to prove the following Hoare triple.

$$\begin{array}{l}\{x :_S \mathsf{ref}\ \tau_1 \ \wedge\ y :_S \tau_1 \ \wedge\ z :_S \tau_2\} \\ x := y \\ \{x :_{S'} \mathsf{ref}\ \tau_1 \ \wedge\ y :_{S'} \tau_1 \ \wedge\ z :_{S'} \tau_2\}\end{array}$$

Consider the scenario when S is the store depicted in Figure 3.3(b) — the assignment $x := y$ updates the location that x points to (thereby modifying the data structure that z points to), so that we cannot know if z has type τ_2 with respect to the modified store S'. We do not want to rule out situations such as this one where an aliased location is being updated. However, allowing updates of aliased locations while guaranteeing consistency is not an easy task. We need to construct a semantic model that ensures that even when we update an allocated location, type judgments made with respect to the old store continue to be valid with respect to the new store. This suggests that writing to an allocated location should be permitted only if the update is *type-preserving*.

Languages like ML and Java also deal with the aliasing problem by allowing only type-preserving updates, or *weak updates*. To permit reuse of memory, ML and Java rely on garbage collection — hence, *strong updates* are dealt with outside of the type system. In this chapter, I shall only consider weak updates. The issue of strong updates and memory reuse is discussed in Chapter 7.

Type-preserving updates imply that only values of a certain type may be written at each allocated location. Hence, we require a model that, for each allocated location, keeps track of this type. Unfortunately, tracking permissible store updates is tricky.

3.2.2 Modeling Permissible Store Updates

In the model for immutable references described in Section 2.2 types are predicates on stores and values. We say $\ell :_S \mathsf{box}\ \tau$ if location ℓ is allocated and if the value stored at location ℓ is of type τ. To allow for the type-preserving update of allocated locations we might think of introducing a *store typing* Ψ, a finite map from locations to closed types: for each allocated location ℓ, we keep track of the type τ of updates allowed at ℓ. A type would then be a predicate on three arguments: a store S, a store typing Ψ, and a value v. Then our *desired* definition of $\mathsf{ref}\,\tau$ (the type ascribed to mutable references) is as follows: we say $\ell :_{S,\Psi} \mathsf{ref}\ \tau$ if location ℓ is allocated, if the store typing Ψ says that the permissible update type for location ℓ is τ, and if the value in the store at location ℓ is of type τ.

$$\mathsf{ref}\ \tau \ \stackrel{\text{def}}{=}\ \{\, \langle S, \Psi, \ell\,\rangle \mid \ell \in \mathrm{dom}(\Psi) \ \wedge\ \Psi(\ell) = \tau \ \wedge\ \langle S, \Psi, S(\ell)\rangle \in \tau \,\}$$

Since the second condition in the above definition (i.e., $\Psi(\ell) = \tau$) essentially subsumes the first (i.e., $\ell \in \mathrm{dom}(\Psi)$), we can simplify the definition of ref as follows.

$$\mathsf{ref}\ \tau \ \stackrel{\text{def}}{=}\ \{\, \langle S, \Psi, \ell\,\rangle \mid \Psi(\ell) = \tau \ \wedge\ \langle S, \Psi, S(\ell)\rangle \in \tau \,\}$$

Notice that implicit in the above definition is the assumption that the current store S *satisfies* the store typing Ψ (or S is well-typed with respect to Ψ) — that is, we assume that if $\ell \in \mathrm{dom}(\Psi)$ then $\ell \in \mathrm{dom}(S)$ and $S(\ell)$ has the type $\Psi(\ell)$, or more precisely, for all locations $\ell \in \mathrm{dom}(\Psi)$, $\langle S, \Psi, S(\ell)\rangle \in \Psi(\ell)$. But the latter implies that the definition of $\mathsf{ref}\ \tau$ can be simplified to the following.

$$\mathsf{ref}\ \tau \ \stackrel{\text{def}}{=}\ \{\, \langle S, \Psi, \ell\,\rangle \mid \Psi(\ell) = \tau \,\}$$

Notice that the semantics of $\mathsf{ref}\ \tau$ no longer *directly* depends upon the store S. In a sense, Ψ contains all the information that we need about the "current" store S in order to decide if a location has type $\mathsf{ref}\ \tau$. Furthermore, there are no other type predicates in our language that directly make use of the current store in their definitions. Hence, to simplify the specification of our model, we can try to model types as predicates on just store typings Ψ and values v, that is, as sets of pairs $\langle \Psi, v \rangle$.

3.2.3 An Inconsistent Model

The semantics of immutable references (Section 2.2) modeled a type as a predicate on a store S (a finite map from locations to values) and a value v. I write the types of these logical objects as,

$$\begin{array}{lcl} \mathit{Store} & = & \mathit{Loc} \overset{\text{fin}}{\to} \mathit{Val} \\ \mathit{Type} & = & \mathit{Store} \times \mathit{Val} \to o \end{array}$$

where o is the type of propositions (*true* or *false*).

For the current model, we would like to model types as predicates on store typings Ψ (a finite map from locations to closed types) and values v. The types of these logical objects are as follows.

$$\begin{array}{lcl} \mathit{StoreType} & = & \mathit{Loc} \overset{\text{fin}}{\to} \mathit{Type} \\ \mathit{Type} & = & \mathit{StoreType} \times \mathit{Val} \to o \end{array}$$

But there is a problem with this specification: notice that the metalogical type of *Type* is recursive, and, furthermore, that it has an inconsistent cardinality — the set of types must be bigger than itself.

3.2.4 A Hierarchy of Types

To better understand the problem, let us take a closer look at our latest *desired* definition of ref. We say that a location ℓ has type $\mathsf{ref}\,\tau$ if it will *always* contain a value of type τ (where “always” includes the present and all possible futures) — that is, if the current store typing Ψ (such that the current store is well-typed with respect to Ψ) says that the permissible update type for location ℓ is τ.

$$\mathsf{ref}\,\tau \;\overset{\text{def}}{=}\; \{\, \langle \Psi, \ell \,\rangle \mid \Psi(\ell) = \tau \,\}$$

Notice that τ is a “smaller” type than $\mathsf{ref}\,\tau$ and that to determine the members of $\mathsf{ref}\,\tau$ we, in fact, only consider those locations in the store typing whose permissible update types are “smaller” than $\mathsf{ref}\,\tau$. This suggests a well-foundedness ordering: types in our model should be stratified so that a type at level k relies not on the *entire* store typing, but only on that subset of the store typing that maps locations to types at level j for $j < k$. This leads us to the following *Type* hierarchy:

$$\begin{array}{lcl} \mathit{Type}_0 & = & \mathit{Unit} \\ \mathit{StoreType}_k & = & \mathit{Loc} \overset{\text{fin}}{\to} \mathit{Type}_k \\ \mathit{Type}_{k+1} & = & \mathit{StoreType}_k \times \mathit{Val} \to o \end{array}$$

By stratifying types we have eliminated the circularity. The types and store typings in our desired definition of $\mathsf{ref}\ \tau$ may now be annotated with levels to reflect the existence of a type hierarchy. The type $\mathsf{ref}\tau$ at level k in the hierarchy ($\mathsf{ref}\tau \in \mathit{Type}_k$) may be defined as a predicate on a store typing $\Psi \in \mathit{StoreType}_{k-1}$ and a value v as shown below. In the interest of brevity, I shall write τ^k and Ψ^k in place of "τ such that $\tau \in \mathit{Type}_k$" and "Ψ such that $\Psi \in \mathit{StoreType}_k$" respectively. Type levels and store typing levels should always be assumed to be nonnegative integers.

$$(\mathsf{ref}\ \tau)^k \overset{\text{is something like}}{=} \{\langle \Psi^{k-1}, \ell \rangle \mid \Psi^{k-1}(\ell) = \tau^{k-1}\}$$

We still have to show that for each λ^M type τ there exists some $k \geq 0$ such that $\tau \in \mathit{Type}_k$. In the next section I shall try to specify a method for stratifying the full set of λ^M types so that it fits into the type hierarchy shown above.

3.2.5 Stratifying Types Based on Syntax

From the hypothetical definition of $\mathsf{ref}\ \tau$ above, we concluded that τ is a "smaller" (i.e., less complex) type than $\mathsf{ref}\ \tau$ — informally, τ has fewer nested occurrences of ref than $\mathsf{ref}\ \tau$. In terms of our type hierarchy then, $\mathsf{ref}\ \tau$ is a level k type if and only if τ is a level j type where $j < k$. But what does it mean for a type τ to belong to some level k in the type hierarchy? Informally, it means that at level k we have sufficient "information" to conclude whether or not a value v has type τ. This information comes in the form of a store typing Ψ^{k-1} — that is, a store typing that tells us the permissible update types of only those locations that map to types at levels $k-1$ and below. The notion of having *sufficient* information at some level suggests that for each (non-empty) type τ there exists a finite level k_{min} such that $\tau \in \mathit{Type}_{k_{min}}$ and $\tau \notin \mathit{Type}_j$ where $0 \leq j < k_{min}$.

Suppose for the moment that the only types in our language are the empty type $\perp$, the primitive types (say unit and bool) and mutable references ref. Then, to determine whether a type τ belongs to Type_k for some $k \geq 0$ we use the following set of rules.

- $\tau = \perp \implies \tau \in \mathit{Type}_0$ (where $\perp$ is the empty type)
- τ is a primitive type (i.e., unit or bool) $\implies \tau \in \mathit{Type}_1$
- $\tau \in \mathit{Type}_k \implies \mathsf{ref}\ \tau \in \mathit{Type}_{k+1}$

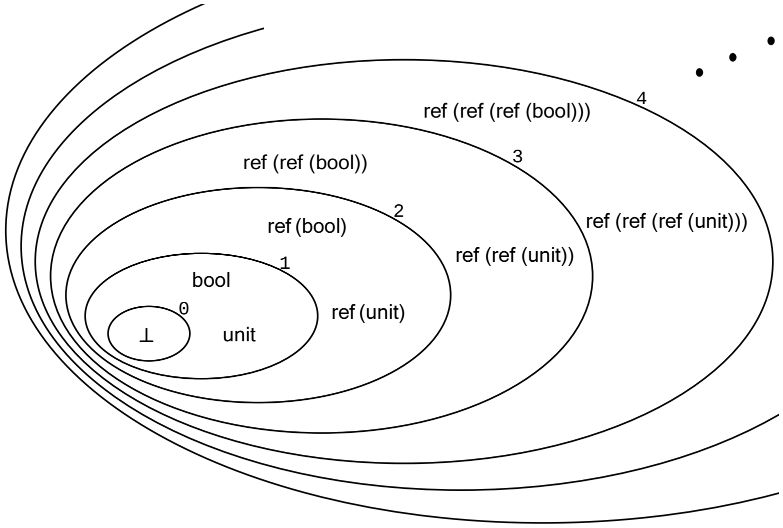


Figure 3.4: Type Hierarchy Based on Syntax of Types

- $\tau \in \mathit{Type}_k \implies \tau \in \mathit{Type}_{k+1}$

The last of the above rules deserves some further explanation. Informally, the last rule stems from the fact that a type at some level k has more "information" than a type at some level $j < k$ — the former is a predicate on Ψ^{k-1} whereas the latter is a predicate on Ψ^{j-1} and $\Psi^{j-1} \subseteq \Psi^{k-1}$ since $j < k$. Therefore, if we have sufficient information at level k to conclude that v has type τ, then at level $k+1$ we still have sufficient information to conclude that v has type τ.

Figure 3.4 illustrates the first few levels of the hierarchy for the types $\bot$, unit, bool, and ref. Level 0 contains only the empty type $\bot$. Level 1 consists of the primitive types unit and bool, as well as all the level 0 types. Level 2 consists of the types $\mathsf{ref\,unit}$, $\mathsf{ref\,bool}$, and all of the level 1 types. In general, for all types τ in level k, level $k+1$ consists of all types $\mathsf{ref}\,\tau$, as well as the level k types τ. Hence, the third level in our type hierarchy contains types such as $\mathsf{ref}(\mathsf{ref}(\mathsf{bool}))$ but not $\mathsf{ref}(\mathsf{ref}(\mathsf{ref}(\mathsf{bool})))$. For any (finite) type expression there is some level in the hierarchy powerful enough to contain it. Since the type expressions in a well-typed monomorphic program are finite, we can find some level of the hierarchy strong enough to type each program.

The Level of a Quantified Type Suppose we now add quantified types to the language. Consider, for example, the type $\exists\alpha.\mathsf{ref}\,\alpha$. How do we determine if $\exists\alpha.\mathsf{ref}\,\alpha$ belongs to some level k in the type hierarchy? If we know that α belongs to Type_k

then we can conclude that $\mathsf{ref}\,\alpha$ belongs to Type_{k+1} and $\exists\alpha.\mathsf{ref}\,\alpha$ belongs to Type_{k+1} (and the type $\mathsf{ref}(\exists\alpha.\mathsf{ref}\,\alpha)$ belongs to Type_{k+2} and so on). But α is a type variable, so we cannot predict how complex the type that instantiates α will be. For instance, suppose we pick k to be 43; now if α is instantiated with $\tau = \mathsf{ref}^{50}(\mathsf{bool})$, then the 43rd level of the hierarchy won't contain τ — only the 51st level (and above) will. Furthermore, we wish to model impredicative quantified types, which means that α — which we assumed to be a level k type — may be instantiated with $\exists\alpha.\mathsf{ref}\ \alpha$, which we just concluded is a level $k+1$ type — but this implies that α should be a level $k+1$ type. Hence, there is no finite level of the hierarchy that is guaranteed to be powerful enough to contain $\exists\alpha.\mathsf{ref}\ \alpha$.

Imagine, for a moment, that we *do know* of a finite level k of the hierarchy that is guaranteed to contain a type such as $\exists\alpha.\mathsf{ref}\ \alpha$ or $\forall\alpha.(\mathsf{ref}\ \alpha) \to \alpha$. Hence, we know that level k of the hierarchy must contain any type τ that instantiates α. But this violates the essence of impredicative quantified types. In the existential case, knowing that the witness type τ is contained in level k of the hierarchy violates the abstraction guaranteed by an existential type — some information about the hidden type is revealed. In the case of universal types, knowing that level k of the hierararchy must contain any τ that instantiates α leaves us with a form of bounded (rather than impredicative) polymorphism.

In earlier work [AAV02] we relied on the syntactic complexity of a type τ in order to determine a level of the type hierarchy that is guaranteed to contain τ. As I have just explained, this approach cannot accommodate references to quantified types since we cannot know the level of a type variable. However, this does not mean that we should abandon the idea of a type hierarchy — what we need is a different (non-syntactic) rationale for stratifying the set of types. In the next section, I will explain an alternate interpretation of type levels that makes it possible to stratify the set of λ^M types.

3.2.6 Stratifying Types Based on Semantic Approximation

Instead of requiring that levels in the type hierarchy correspond to the syntactic complexity of type expressions, we shall treat levels as an indication of how many more steps the program can safely execute. Informally, we want $\left\langle \Psi^{k-1}, v \right\rangle \in \tau^k$ to mean that v "looks" like it belongs to type τ for k steps — perhaps v is not "really" a member of type τ, but any program of type $\tau \to \tau'$ must execute for at least k steps on v before getting to a stuck state. The statement $\left\langle \Psi^{k-1}, v \right\rangle \in \tau^k$ may also be read as "the assumption that v has type τ with respect to Ψ cannot be proved wrong within k steps of execution". Hence, levels in the type hierarchy correspond to approximations of a type's behavior.

I call k the *approximation index* following Appel and McAllester's indexed model of types [AM01]. The latter gave a semantics of recursive types for the purely functional λ-calculus, specifically, a semantics of covariant as well as contravariant equirecursive types. In the indexed model, we say a value v has type τ to approximation k if, in any computation running for no more than k steps, the value v behaves as if it were an element of the type τ. I wish to make use of this intuition to stratify types in our model of mutable references.

Recall that the original observation that led us to a type hierarchy (see Section 3.2.4) was that τ is a "smaller" type than $\mathsf{ref}\ \tau$ and that to determine the members of the set $\mathsf{ref}\ \tau$ we need only consider those locations in the store typing whose permissible update types are "smaller" than $\mathsf{ref}\ \tau$. Suppose that location ℓ contains a value v. The assumption that ℓ has type $\mathsf{ref}\ \tau$ cannot be proved wrong within k steps of execution if and only if the assumption that v has type τ cannot be proved wrong within $k-1$ steps since the former requires an extra dereferencing step. Hence, in terms of the number of steps that can be safely executed, τ is "smaller" type than $\mathsf{ref}\ \tau$. To determine the members of $\mathsf{ref}\ \tau$ to approximation k we only need a store typing that tells us each store location's permissible update type to approximation $k-1$.

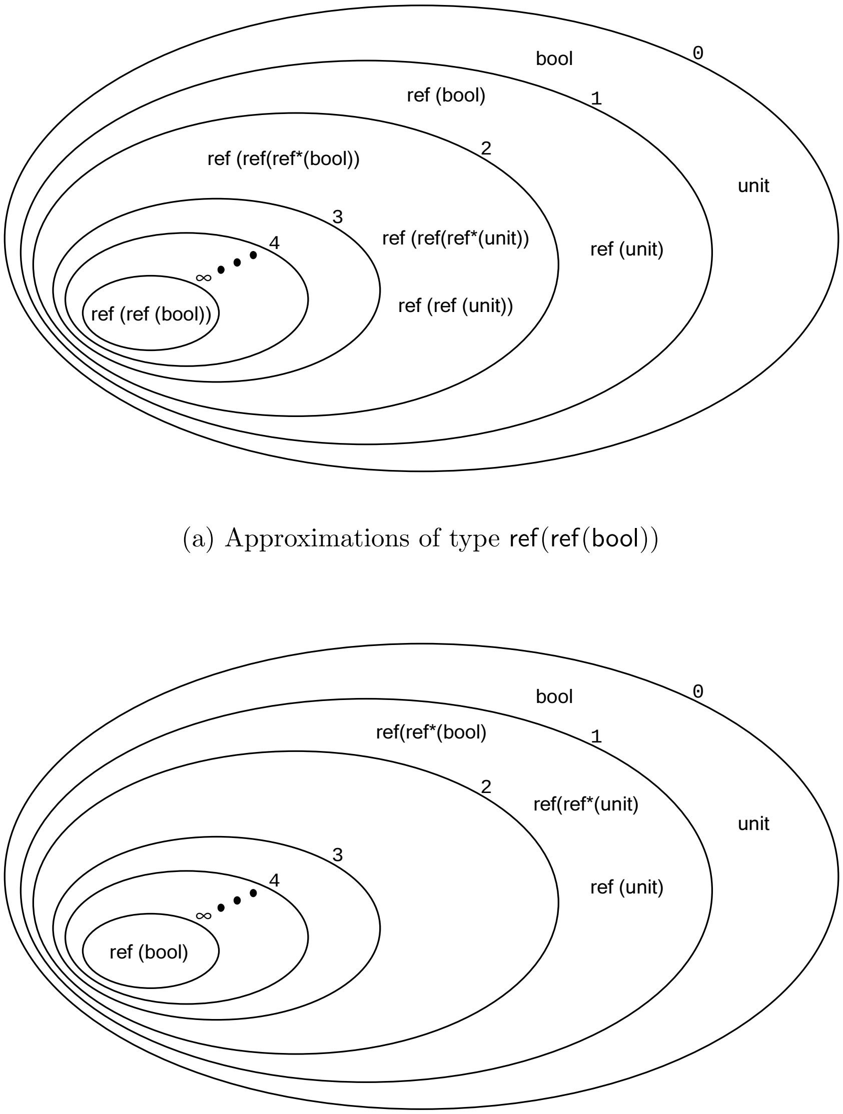
Consider, for instance, the type $\mathsf{ref}(\mathsf{ref}(\mathsf{bool}))$. Figure 3.5(a) illustrates the stratification of $\mathsf{ref}(\mathsf{ref}(\mathsf{bool}))$, assuming that the only types in the language are unit, bool, and ref. Note that every type that appears in the figure denotes a *set* of pairs of store typings and values. (In the figure, types of the form $\mathsf{ref}\ ^*(\tau)$ denote zero or more applications of ref to τ.) Level 0 contains *all* the types in the language because at level 0 (i.e., if we intend to run the program for 0 more steps) a value v of *any type* behaves as if it belongs to $\mathsf{ref}(\mathsf{ref}\ (\mathsf{bool}))$. Intuitively, the assumption that v has type $\mathsf{ref}\ (\mathsf{ref}(\mathsf{bool}))$ cannot be proved wrong in zero steps.

Level 1 contains a subset of the level 0 types, specifically, all types of the form $\mathsf{ref}\ \tau$, where τ is *any* type. At level 1, a value v of any type of the form $\mathsf{ref}\ \tau$ behaves as if it belongs to $\mathsf{ref}(\mathsf{ref}(\mathsf{bool}))$. Intuitively, a program that executes just one step can dereference v without getting stuck (since v has type $\mathsf{ref}\ \tau$). Hence, the assumption that v has type $\mathsf{ref}(\mathsf{ref}(\mathsf{bool}))$ cannot be proved wrong in one step.

Level 2 contains a subset of the level 1 types, specifically, all types of the form $\mathsf{ref}(\mathsf{ref}(\tau))$ (for *any* τ). At level 2, a value v of any type $\mathsf{ref}(\mathsf{ref}\,\tau)$ "looks" like a value of type $\mathsf{ref}(\mathsf{ref}(\mathsf{bool}))$. Intuitively, given v the program can perform two dereferences without getting stuck, so the assumption that v has type $\mathsf{ref}(\mathsf{ref}(\mathsf{bool}))$ cannot be proved wrong in two steps.

Finally, level 3 contains only one type, namely $\mathsf{ref}(\mathsf{ref}\ (\mathsf{bool}))$. At level 3 and above (i.e., in the limit) the only values that behave as if they belong to $\mathsf{ref}(\mathsf{ref}(\mathsf{bool}))$ are the ones that actually do. Intuitively, a program that executes for three or more steps can tell precisely whether or not a value v has the type $\mathsf{ref}(\mathsf{ref}(\mathsf{bool}))$. For comparison, Figure 3.5(b) illustrates the stratification of type $\mathsf{ref}(\mathsf{bool})$.

The levels of the hierarchy provide approximations of a type τ — every value that behaves as if it belongs to τ for k steps of execution is contained in the kth



(a) Approximations of type ref(ref(bool))

(b) Approximations of type ref(bool)

Figure 3.5: Type Hierarchy Based on Semantic Approximation

approximation of the type. Level 0 provides the least precise approximation of a type; any type τ to approximation 0 is equivalent to $\top$. The precision increases with the level, that is, with the number of steps the program may execute. Accordingly, in Figures 3.5(a) and 3.5(b), as we go from outside to inside (i.e., as the levels increase), the number of types contained in each circle decrease. Hence, $\tau \in \mathit{Type}_{k+1}$ implies $\tau \in \mathit{Type}_k$ — informally, if the assumption that v has type τ cannot be proved wrong within k steps, then it cannot be proved wrong within $j < k$ steps. As a result, the hierarchy in Figure 3.5 is "inside-out" as compared to the syntax-based type hierarchy in Figure 3.4.

The Level of a Quantified Type The approximation power of the indexed model helps us model mutable references to quantified types by allowing us to "pick a level" for a type variable while ensuring safety and abstraction. Suppose in some execution we are about to instantiate α in $\exists\alpha.\mathsf{ref}\ \alpha$ with $\mathsf{ref}^{50}(\mathsf{bool})$, but we intend to run the program for only 30 more steps. Then it's all the same whether we instantiate α with $\mathsf{ref}^{50}(\mathsf{bool})$ or $\mathsf{ref}^{30}(\perp)$, since in 30 execution steps the program cannot dereference more than 30 references. Therefore, if we intend to run the program for only k steps, it suffices to use the kth level of the hierarchy.

The next section more rigorously describes the ideas introduced in this section and formalizes an indexed model of impredicative polymorphism and mutable references.

3.3 An Indexed Possible-Worlds Model

In this section I will describe how to construct a model of general references by adapting the ideas underlying Appel and McAllester's indexed model to a setting with mutable state.

A rough outline of the section is as follows. I start by giving a definition of safety suited to the indexed model. I then show how to model the hierarchy of types as a set and explain what properties these type sets must possess. Next, I present the semantics of various types in λ^M, followed by the semantics of λ^M judgments and the typing rules for the language. Finally, I explain how our model is a possible-worlds model.

3.3.1 Safety

A state (S, e) is *irreducible* if it has no successor in the step relation, that is, $\mathrm{irred}(S, e)$ if e is a value or (S, e) is a "stuck" state such as $(S, \mathtt{unit}(e'))$ or $(S, \ell := v)$ where $\ell \notin \mathrm{dom}(S)$. I say that a machine state (S, e) is safe for k steps if it takes at least k steps for it to reach a "stuck" state. Note that any state is safe for 0 steps. A state (S, e) is safe if execution starting with (S, e) does not reach a stuck state in *any* number of steps.

Definition 3.1 (Safe)
A state (S, e) is safe for k steps if for any reduction $(S, e) \longmapsto_M^j (S', e')$ of $j < k$ steps, either e' is a value or another step is possible.

$$\begin{array}{rcl} \text{safen}(k, S, e) & \stackrel{\text{def}}{=} & \forall j, S', e'.\ (j < k \ \wedge\ (S, e) \longmapsto_M^j (S', e')) \\ & & \qquad \Longrightarrow\ (\text{val}(e')\ \vee\ \exists S'', e''.\ (S', e') \longmapsto_M (S'', e'')) \end{array}$$

A state (S, e) is called safe if it is safe for all $k \geq 0$ steps.

$$\text{safe}(S, e) \ \stackrel{\text{def}}{=}\ \forall k \geq 0.\ \text{safen}(k, S, e)$$

To show that a program e is safe to execute given an initial store S (i.e., safe(S, e)), we have to prove that the initial machine state is safe for all $k \geq 0$ steps (i.e., safen(k, S, e) for all $k \geq 0$). To this end, I will first specify the semantics of λ^M types and type judgments; I will then prove, based on these definitions, that typability implies safety — that is, if a type judgment is true, then the typed program is safe; and finally, I will prove that the λ^M typing rules are sound.

3.3.2 Modeling Stratified Types as Sets

I am interested in modeling types as sets of tuples, but as explained in Section 3.2.3 a type cannot simply be a set of tuples of the form $\langle \Psi, v \rangle$ since that leads to an inconsistent model. The inconsistency can be eliminated by stratifying types, which results in a type hierarchy as discussed in Section 3.2.4. I shall incorporate the notion of a type hierarchy into our semantics as follows. First, I model types as sets of tuples $\langle k, \Psi, v \rangle$, where k (the *approximation index*) is a nonnegative integer, Ψ is a mapping from locations to types (that is, a mapping from locations to sets of tuples of the form $\langle k, \Psi, v \rangle$), and v is a value — we say that v has type τ to approximation k with respect to store typing Ψ. The intuitive idea is that in any computation running for no more than k steps, the value v behaves as if it were an element of the type τ.

Next, I define the notion of a k-approximation of a type (which corresponds to the kth level of the type in the hierarchy), and extend the notion pointwise to store typings.

Definition 3.2 (Approx)
The k-approximation of a set is the subset of its elements whose index is less than k; I also extend this notion pointwise to store typings:

$$\begin{array}{rcl} \lfloor \tau \rfloor_k & \stackrel{\text{def}}{=} & \{\langle j, \Psi, v \rangle \mid j < k \ \wedge\ \langle j, \Psi, v \rangle \in \tau\} \\ \lfloor \Psi \rfloor_k & \stackrel{\text{def}}{=} & \{(\ell \mapsto \lfloor \tau \rfloor_k) \mid \Psi(\ell) = \tau\} \end{array}$$

Finally, since a naïve construction of types as collections of tuples $\langle k, \Psi, v\rangle$ (where Ψ is a relation on locations and types) can lead to paradoxes, I construct a stratified semantics by requiring that all type definitions obey the following invariant: for all $k \geq 0$, the definition of the $(k+1)$-approximation of a type τ cannot consider any type beyond approximation k. I shall refer to this as the *stratification invariant*. To comply with this invariant, when considering a tuple $\langle k, \Psi, v\rangle$, I do not require Ψ to be defined beyond $\lfloor \Psi \rfloor_k$. Intuitively, if we intend to run the program for no more than k more steps, then typechecking with $\ell \mapsto \lfloor \tau \rfloor_k$ is just as safe as typechecking with $\ell \mapsto \tau$.

Consider, for instance, the semantics of mutable reference types $\mathsf{ref}\ \tau$. The *hypothetical* definition of $\mathsf{ref}\ \tau$ from Section 3.2.4 (shown with level annotations) is as follows.

$$(\mathsf{ref}\ \tau)^k \quad \overset{\text{is something like}}{=} \quad \{\left\langle \Psi^{k-1}, \ell \right\rangle \mid \Psi^{k-1}(\ell) = \tau^{k-1}\}$$

The *actual* definition of $\mathsf{ref}\ \tau$ is shown below. I say that a location ℓ has type $\mathsf{ref}\ \tau$ for k steps with respect to store typing Ψ (written $\langle k, \Psi, \ell\,\rangle \in \mathsf{ref}\ \tau$) if ℓ *will* contain a value of type τ for at least $k-1$ steps — that is, if the store typing Ψ says that ℓ may only be updated with values that "behave" like values of type τ for at least $k-1$ steps. Hence, I define $\mathsf{ref}\ \tau$ so that it does not rely on Ψ and τ beyond $\lfloor \Psi \rfloor_k$ and $\lfloor \tau \rfloor_k$, respectively.

$$\mathsf{ref}\ \tau \quad \overset{\text{def}}{=} \quad \{\langle k, \Psi, \ell\,\rangle \mid \lfloor \Psi \rfloor_k(\ell) = \lfloor \tau \rfloor_k\}$$

As I will show, *all* our definitions obey the stratification invariant and hence, our types do indeed form a set, which can be constructed using recursively defined sets Type_k and $\mathit{StoreType}_k$ as follows.

$$\begin{array}{l}
\tau \in \mathit{Type}_0 \text{ iff } \tau = \{\} \\
\tau \in \mathit{Type}_{k+1} \text{ iff } \forall \langle j, \Psi, v\rangle \in \tau.\ j \leq k \ \wedge\ \Psi \in \mathit{StoreType}_j \\[1ex]
\Psi \in \mathit{StoreType}_k \text{ iff } \forall \ell \in \mathrm{dom}(\Psi).\ \Psi(\ell) \in \mathit{Type}_k \\[2ex]
\tau \in \mathit{Type} \text{ iff } \forall k.\ \lfloor \tau \rfloor_k \in \mathit{Type}_k \\
\Psi \in \mathit{StoreType} \text{ iff } \forall k.\ \lfloor \Psi \rfloor_k \in \mathit{StoreType}_k
\end{array}$$

3.3.3 Properties of Types

As we can see from the definition above, for τ to be a type (written $\tau \in \mathit{Type}$), given $\langle k, \Psi, v\rangle \in \tau$ and a location $\ell \in \mathrm{dom}(\Psi)$ such that $\Psi(\ell) = \tau'$, it must be the case that every element $\langle k', \Psi', v'\rangle \in \tau'$ has $k' < k$. This, however, is not the only property that a type set τ must satisfy; there are other ways in which τ must be well-behaved as I shall illustrate through an example. Consider the program given below which allocates and initializes a cell, executes several instructions (elided), then dereferences the original cell before executing the rest of the program e_{rest}.

The comments that follow each line of the program describe aspects of the state immediately after that line has been evaluated. Aspects of the computation state that we must consider when reasoning about mutable references include not just the current store S but also the current store typing Ψ, and the number of future steps k that the program can safely take. In a state with store S, store typing Ψ, and k steps left to execute, I shall assume:

1. that $\mathrm{dom}(\Psi)$ contains all the locations that the rest of the program might access (except for those not yet allocated); and

2. that S satisfies Ψ for k steps.[4] I shall formalize this notion shortly (see Definition 3.5), but roughly, S satisfies Ψ for k steps — or equivalently, S is well-typed to approximation k with respect to Ψ — if, for each location $\ell \in \mathrm{dom}(\Psi)$, $\ell \in \mathrm{dom}(S)$ and $S(\ell)$ has type $\Psi(\ell)$ to approximation k.

Later, I shall describe how each of these assumptions is formally built into our model (see Definitions 3.7 and 3.6 for assumptions (1) and (2), respectively).

```
                              % $S_0 = \Psi_0 = \{\}$
let $x_1$ = new(true) in      % $S_1 = S_0\,[\ell_1 \mapsto \texttt{true}]$, $\Psi_1 = \lfloor\Psi_0\rfloor_{10}\,[\ell_1 \mapsto \lfloor\mathsf{bool}\rfloor_{10}]$,
                              % $x_1 \mapsto \ell_1$, 10 more steps
let $x_2$ = ... in            % $S_2 =$ ..., $\Psi_2 =$ ..., $x_1 \mapsto \ell_1$
    ⋮
let $x_n$ = ... in            % $S_n =$ ..., $\Psi_n =$ ..., $x_1 \mapsto \ell_1$, 6 more steps
let $y$ = !$x_1$ in           % $S_{n+1} =$ ..., $\Psi_{n+1} =$ ...
$e_{rest}$
```

Suppose that we execute the above program with an initial empty store S_0. Since there are no allocated locations, the domain of the initial store typing Ψ_0 is empty. By the λ^M operational semantics, after evaluating the first line of the program we have store $S_1 = S_0\,[\ell_1 \mapsto \texttt{true}]$ for some location $\ell_1 \notin \mathrm{dom}(S_0)$ and x_1 is bound to ℓ_1. Suppose, at this point, that the program must execute 10 more steps. This means that we only need to know the types that the contents of allocated locations must have for another 9 steps (since an update or dereference would use up one step). Hence, the store typing $\Psi_1 = \lfloor\Psi_0\rfloor_{10}\,[\ell_1 \mapsto \lfloor\mathsf{bool}\rfloor_{10}]$ describes the "state" of the store S_1 *with enough precision* for us to be able to conclude that location ℓ_1 has type $\mathsf{ref\,bool}$ for 10 steps — that is, to conclude that $\langle 10, \Psi_1, \ell_1\rangle \in \mathsf{ref\,bool}$ according to the semantics of ref above.

[4] The notion that S should satisfy Ψ was briefly discussed in Section 3.2.2.

Next, we evaluate the elided terms; suppose that these include one or more `new` and assignment expressions, and suppose that the net effect of these terms is to transform the store from S_1 to S_n. Just before we evaluate the expression dereferencing x_1, we have x_1 bound to ℓ_1, store S_n, and 6 more steps to execute. At this point, from a semantic perspective, it is safe to dereference ℓ_1 given store S_n as long as:

1. ℓ_1 is an allocated location, i.e., $\ell_1 \in \text{dom}(S_n)$
2. $S_n(\ell_1)$ looks like it belongs to type bool for 5 more steps

Together these two conditions imply that we need to prove that the store typing Ψ_n, which captures the "state" of the store S_n with 6 more steps to execute, maps ℓ_1 to $\lfloor \mathsf{bool} \rfloor_6$. Equivalently, we need to prove $\langle 6, \Psi_n, \ell_1 \rangle \in \mathsf{ref\ bool}$ in order to show that the instruction $!\,x_1$ is safe to execute.

We already know that $\langle 10, \Psi_1, \ell_1 \rangle \in \mathsf{ref\ bool}$, but we need to account for the fact that Ψ_n is different from Ψ_1. This difference is related to the fact that S_n is different from S_1 due to the allocation of new cells or the update of existing cells. An examination of the λ^M operational semantics suggests that condition (1) must hold since there are no terms in λ^M that *shrink* the store — hence $\text{dom}(S_n)$ must be a superset of $\text{dom}(S_1)$. Since the store does not shrink, the store typing should not shrink — hence, $\text{dom}(\Psi_n)$ must be a superset of $\text{dom}(\Psi_1)$.

The reason why condition (2) must hold is not as straightforward. Since assignment expressions in λ^M update allocated locations, $S_n(\ell_1)$ may *not* be equal to $S_1(\ell_1)$. But λ^M permits only *type-preserving* updates and this is where the store typings come in. Intuitively, if Ψ_1 says that ℓ_1 has type τ then Ψ_n should also say that ℓ_1 has type τ. To be precise, though, we must take the number of future execution steps into account as follows. If Ψ_1 says that some location ℓ_1 will have type τ for 10 steps, and we then execute 4 of those 10 steps to get to the state described by Ψ_n, then Ψ_n should say that ℓ_1 will have type τ for 6 steps.

The above notion of *approximately type-preserving* updates may be formalized as follows. For any two machine states (S, e) and (S', e'), and store typings Ψ and Ψ' such that S satisfies Ψ for k steps and S' satisfies Ψ' for $j \leq k$ steps, if $(S, e) \longmapsto_M^{k-j} (S', e')$, then the relationship between Ψ and Ψ' is specified by $(k, \Psi) \sqsubseteq (j, \Psi')$. I call $\sqsubseteq$ the extend-state relation.

Definition 3.3 (State Extension)
A valid state extension *is defined as follows:*

$$(k, \Psi) \sqsubseteq (j, \Psi') \;\stackrel{\text{def}}{=}\; j \leq k \;\wedge\; (\forall \ell \in \text{dom}(\Psi).\ \lfloor \Psi' \rfloor_j(\ell) = \lfloor \Psi \rfloor_j(\ell))$$

State extension specifies how a store typing may change during zero or more computation steps. Computation steps might allocate new reference cells, in which case dom(Ψ') will be a strict superset of dom(Ψ). The computation might choose to forget some information, in which case j will be strictly less than k and Ψ' will be less precise (have a lower approximation level) than Ψ. This means, however, that the program may now run for at most j more steps rather than k more steps. Finally, the computation might update an allocated location ℓ with a new value v. Since updates must be approximately type-preserving, if we have j steps left to execute then, since the update itself takes up one step, v must have type $\Psi(\ell)$ to approximation j — therefore, $\lfloor\Psi'\rfloor_j(\ell) = \lfloor\Psi\rfloor_j(\ell)$.

The concept of a valid state extension $(k, \Psi) \sqsubseteq (j, \Psi')$ is useful for restricting attention to only those store typings that are *possible* in the future given the operational semantics of λ^M. Returning to the above example, given $\langle 10, \Psi_1, \ell_1\rangle \in \mathsf{ref\,bool}$ and $(10, \Psi_1) \sqsubseteq (6, \Psi_n)$, we can show that $\langle 6, \Psi_n, \ell_1\rangle \in \mathsf{ref\ bool}$. In general, I want all of the types in our language to have this property exhibited by ref bool, that is, I want type sets in λ^M to be closed under valid state extension.

Definition 3.4 (Type)
A type *is a set τ of tuples of the form $\langle k, \Psi, v\rangle$ where v is a value, k is a nonnegative integer, and Ψ is a store typing, and where the set τ is closed under state extension; that is,*

$$\begin{aligned}\mathrm{type}(\tau) \;\stackrel{\mathrm{def}}{=}\; & \forall k, j, \Psi, \Psi', v.\\ & (\langle k, \Psi, v\rangle \in \tau \;\wedge\; (k, \Psi) \sqsubseteq (j, \Psi')) \implies \langle j, \Psi', v\rangle \in \tau\end{aligned}$$

I have shown that the absence of cell deallocation in λ^M together with the requirement that updates be approximately type-preserving (captured by the definition of $\sqsubseteq$) are crucial for proving that ref types are preserved under evaluation. Now let us consider each of the remaining types in λ^M.

3.3.4 Base Types

A value v has type unit if and only if $v = \mathtt{unit}$. Though types are predicates on approximation indices k and store typings Ψ as well as values, the approximation index and store typing are irrelevant when determining if a value v has type unit — the same is true for other base types such as bool, int, char, and so on. It trivially follows that unit is closed under state extension.

$$\mathsf{unit} \;\stackrel{\mathrm{def}}{=}\; \{\langle k, \Psi, \mathtt{unit}\rangle\}$$

3.3.5 Function Types

When deciding if a value $\lambda x.e$ has type $\tau_1 \to \tau_2$ for k steps (i.e., to approximation k) with respect to store typing Ψ we have to keep in mind that the program may not utilize (apply) $\lambda x.e$ until some point in the future, a point when the store typing may be different from Ψ thanks to the evaluation of `new` terms in the interim. Also, we must account for the number of execution steps we use up to get to that future point. Hence, for $\lambda x.e$ in $\tau_1 \to \tau_2$ to be safe for k steps, since *beta-reduction uses up one step*, the resulting (beta-reduced) expression must be safe for some $j < k$ steps. Let Ψ' be the future store typing to approximation j. Since λ^M does not allow cell deallocation and requires approximately type-preserving updates, clearly it should be the case that $(k, \Psi) \sqsubseteq (j, \Psi')$.

Next, let S' be a store that satisfies Ψ' for j steps (i.e., S' is well-typed to approximation j with respect to Ψ'). Let v be a value that has type τ_1 for j steps with respect to Ψ'. If evaluating the expression $e[v/x]$ with store S' gives us a value of type τ_2 in less than j steps (along with a store that satisfies some store typing that is a valid extension of Ψ') then we may conclude that $\lambda x.e$ has type $\tau_1 \to \tau_2$ for k steps with respect to Ψ. The formal definition makes this clearer, but before I can formally define $\tau_1 \to \tau_2$, I need two auxiliary definitions.

First, I formally state what it means for store S to satisfy store typing Ψ for k steps (i.e., for S to be well-typed to approximation k with respect to Ψ).

Definition 3.5 (Well-typed Store)
A store S is well-typed to approximation k with respect to a store typing Ψ iff $\mathrm{dom}(\Psi) \subseteq \mathrm{dom}(S)$ *and the contents of each location $\ell \in \mathrm{dom}(\Psi)$ has type $\Psi(\ell)$ to approximation k:*

$$\begin{array}{rcl} S :_k \Psi & \stackrel{\text{def}}{=} & \mathrm{dom}(\Psi) \subseteq \mathrm{dom}(S) \;\wedge \\ & & \forall j < k.\, \forall \ell \in \mathrm{dom}(\Psi).\ \langle j, \lfloor \Psi \rfloor_j, S(\ell) \rangle \in \lfloor \Psi \rfloor_k(\ell) \end{array}$$

There are two things about the above definition that deserve comment. First, it is important to note that all the tuples required to be in $\Psi(\ell)$ have index strictly less than k; this helps avoid circularity. Second, note that store S is allowed to have "extra" allocated locations that are not typed by Ψ. We do not need to know the types of these "extra" locations because, as we shall see, whenever our model makes use of $S :_k \Psi$ we already know that $\mathrm{dom}(\Psi)$ contains all the locations that the rest of the program may access, except for those not yet allocated.[5] Hence, the requirement that $\mathrm{dom}(\Psi) \subseteq \mathrm{dom}(S)$ is sufficient for our purposes. The requirement $\mathrm{dom}(\Psi) = \mathrm{dom}(S)$, though not incorrect, would have been overly restrictive.

Next, I specify what it means for a closed expression e to have type τ for k steps with respect to store typing Ψ (written $e :_{k,\Psi} \tau$). Intuitively, $e :_{k,\Psi} \tau$ says that in a state (S, e) such that $S :_k \Psi$, term e behaves like an element of τ for k steps of computation. If (S, e) reaches a state (S', v) (i.e., if e evaluates to a value

[5] We "already know" that $\mathrm{dom}(\Psi)$ contains all "relevant" allocated locations thanks to the semantics of judgments — see Definition 3.7.

v) in less than k steps, then it must be the case that v has type τ and S' must be a store that is well-typed with respect to a valid extension of Ψ. Otherwise, (S, e) can take k steps without getting stuck — that is, there exist S' and e' such that $(S, e) \longmapsto_M^k (S', e')$.

Definition 3.6 (Expr : Type)
For any closed expression e and type τ, $e :_{k,\Psi} \tau$ if whenever $S :_k \Psi$, $(S, e) \longmapsto_M^j (S', e')$ for $j < k$, and (S', e') is irreducible, then there exists a store typing Ψ' such that $(k, \Psi) \sqsubseteq (k - j, \Psi')$, $S' :_{k-j} \Psi'$, and $\langle k - j, \Psi', e' \rangle \in \tau$; that is,

$$\begin{aligned} e :_{k,\Psi} \tau \quad \stackrel{\text{def}}{=} \quad & \forall j, S, S', e'.\ (0 \leq j < k \ \wedge\ S :_k \Psi \\ & \qquad\qquad\qquad \wedge\ (S, e) \longmapsto_M^j (S', e') \ \wedge\ \mathrm{irred}(S', e')) \\ & \qquad \Longrightarrow \exists \Psi'.\ (k, \Psi) \sqsubseteq (k - j, \Psi') \\ & \qquad\qquad\quad \wedge\ S' :_{k-j} \Psi' \ \wedge\ \langle k - j, \Psi', e' \rangle \in \tau \end{aligned}$$

Note that if $e :_{k,\Psi} \tau$ and $0 \leq j \leq k$ then $e :_{j,\lfloor\Psi\rfloor_j} \tau$. This is because τ is closed under valid state extension and $(k, \Psi) \sqsubseteq (j, \lfloor\Psi\rfloor_j)$ is a valid state extension (one that "forgets" some information, or retains information in a less precise form). Also note that, if v is a value and $k > 0$, the statements $v :_{k,\Psi} \tau$ and $\langle k, \Psi, v \rangle \in \tau$ are equivalent.

The semantics of function types can now be defined as follows.

$$\begin{aligned} \tau_1 \rightarrow \tau_2 \quad \stackrel{\text{def}}{=} \quad & \{ \langle k, \Psi, \lambda x.e \rangle \mid \forall v, \Psi', j < k. \\ & \qquad ((k, \Psi) \sqsubseteq (j, \Psi') \ \wedge\ \langle j, \Psi', v \rangle \in \tau_1) \\ & \qquad\qquad \Longrightarrow\ e[v/x] :_{j,\Psi'} \tau_2 \ \} \end{aligned}$$

In order to prove that function types are closed under state extension we will first need to show that valid state extension ($\sqsubseteq$) is transitive (see Lemma 3.13). State extension specifies that the approximation index can either decrease or remain the same, that allocated cells cannot be freed, and that the designated type of each location can only become less precise (which means that the type set becomes larger). Hence, $\sqsubseteq$ relates monotonically decreasing approximation indices and monotonically increasing store typings, so it is easy to show that $\sqsubseteq$ is transitive.

3.3.6 Quantified Types

For simplicity of presentation, I avoid the use of type variables in the representation of quantified types. Specifically, instead of writing $\forall\alpha.\tau$ and $\exists\alpha.\tau$ where α is the *only* type variable that occurs free in the type expression τ, I write $\forall F$ and $\exists F$ where F is a closed function from types to types (also called a type functional). The two representations are equivalent in terms of expressive power, but they lead to different λ^M type judgments. To accommodate quantified types of the form

$\forall\alpha.\tau$ and $\exists\alpha.\tau$, a language needs type-checking rules that explicitly manage a set of type variables. This means that λ^M type judgments would have to have the form $\Delta; \Gamma \vDash_M e : \tau$ where Δ is the set of type variables that may appear in τ. By comparison, for quantified types of the form $\forall F$ and $\exists F$, it would suffice to have λ^M type judgments of the form $\Gamma \vDash_M e : \tau$ (as I shall describe in Section 3.3.7). Hence, though avoiding the use of type variables makes the representation of quantified types (as well as the type-checking rules for quantified types — see Figure 3.7) slightly unconventional, it nonetheless leads to a *simpler* semantics of λ^M — that is, a semantics not cluttered by the bookkeeping of type variables.[6]

The use of *closed* type functionals F in the representation of quantified types — rather than type functionals F with free type variables — limits the set of types that we are able to express. I shall discuss this limitation and how to get around it in Section 4.4.1.

Universal Types When deciding if a value $\Lambda.e$ has type $\forall F$ for k steps (i.e., to approximation k) with respect to a store typing Ψ, we have to keep in mind that $\Lambda.e$ may not be instantiated with a type until some point in the future. Let Ψ' be the store typing at that future point, and let $j \leq k$ be the number of execution steps left — that is, we use up $k - j$ steps to get to that future point. As discussed in Section 3.3.5 (for function types), it must be the case that $(k, \Psi) \sqsubseteq (j, \Psi')$.

If $\Lambda.e$ is instantiated with the type τ, we must make sure that τ is a valid type. But since we have j steps left to execute and *type instantiation uses up one step*, we only need to ensure that τ is a valid type for $j - 1$ steps, that is, $\text{type}(\lfloor\tau\rfloor_j)$. Hence, if we have τ such that $\text{type}(\lfloor\tau\rfloor_j)$, and e has type $F(\tau)$ for all $i < j$ steps with respect to store typing $\lfloor\Psi'\rfloor_i$, then we may conclude that $\Lambda.e$ has type $\forall F$ for k steps with respect to Ψ. Note that it suffices to check that τ is a valid type to approximation j since we only check that e has type $F(\tau)$ to approximation $i < j$.[7] The semantics of $\forall F$ can be defined as follows.

$$\begin{aligned} \forall F \ \stackrel{\text{def}}{=}\ \{\langle k, \Psi, \Lambda.e\rangle \mid \forall j, \Psi', \tau.\ ((k, \Psi) \sqsubseteq (j, \Psi')\ \wedge\ \text{type}(\lfloor\tau\rfloor_j)) \\ \Longrightarrow\ \forall i < j.\ e :_{i, \lfloor\Psi'\rfloor_i} F(\tau)\} \end{aligned}$$

It is important to note that above definition satisfies the stratification invariant discussed in Section 3.3.2, which helps avoid circularity. To determine whether the tuple $\langle k, \Psi, \Lambda.e\rangle$ belongs to $\forall F$, first, we do not require Ψ to be defined beyond $\lfloor\Psi\rfloor_k$, and second, we only require $\text{type}(\lfloor\tau\rfloor_j)$ (where $j \leq k$) rather than $\text{type}(\tau)$.

[6] Though I do not describe it here, it should be possible to support the use of type variables in the representation of types by modifying the model I shall present so that it defines a semantics of type-checking judgments $\Delta; \Gamma \vDash_M e : \tau$ rather than $\Gamma \vDash_M e : \tau$.

[7] To be precise, the truth of this statement is contingent upon F having a property that I have not discussed yet, namely upon F being *nonexpansive*. I shall explain what it means for a type functional to be nonexpansive in Section 3.3.9.

Hence, to define the $(k+1)$-approximation of $\forall F$, we do not consider any type beyond approximation k as required by the stratification invariant.

The proof that universal types are closed under state extension relies on the transitivity of state extension (see Lemma 3.16 in Section 3.4).

Existential Types To determine whether a value $\texttt{pack}\, v$ has type $\exists F$ for k steps with respect to store typing Ψ, we must make sure that two conditions are satisfied. First, we must check that the witness type τ is a valid type; since *unpacking the existential uses up one step*, we only have to ensure that τ is a valid type for $k-1$ steps, that is, $\text{type}(\lfloor\tau\rfloor_k)$. Second, we must make sure that v has type $F(\tau)$ for all $j < k$ steps (since unpacking uses up one step) with respect to store typing $\lfloor\Psi\rfloor_j$. It suffices to check that τ is a valid type to approximation k since we only check that v has type $F(\tau)$ to approximation $j < k$.[8] The semantics of $\exists F$ is as follows.

$$\exists F \quad \stackrel{\text{def}}{=} \quad \{\langle k, \Psi, \texttt{pack}\, v\rangle \mid \exists\tau.\ (\text{type}(\lfloor\tau\rfloor_k)\ \wedge\ \forall j < k.\ \langle j, \lfloor\Psi\rfloor_j, v\rangle \in F(\tau))\}$$

To decide whether $\langle k, \Psi, \texttt{pack}\, v\rangle$ belongs to $\exists F$, we do not require Ψ to be defined beyond $\lfloor\Psi\rfloor_k$. Also, as in the definition of $\forall F$ above, we only require $\text{type}(\lfloor\tau\rfloor_k)$ rather than $\text{type}(\tau)$. Hence, the definition of $\exists F$ adheres to the stratification invariant, helping us avoid circularity.

The λ^M type definitions developed thus far are collected in Figure 3.6 (along with the definition of $\bot$, the empty type). Proofs that these types are closed under state extension are given in Section 3.4.

3.3.7 Judgments, Typing Rules, and Safety

Up to now I have only dealt with closed terms, as these are the ones that "step" at run time. Now I turn to terms with free variables, upon which the static type-checking rules must operate. Type judgments in λ^M have the form $\Gamma \vDash_M e : \tau$ where Γ is a type environment, that is, a mapping from term variables to types. A value environment σ (also called a ground substitution) is a mapping from term variables to values.

Definition 3.7 (Semantics of Judgment)
For any type environment Γ and value environment σ I write $\sigma :_{k,\Psi} \Gamma$ (read "σ approximately obeys Γ") if for all variables $x \in \text{dom}(\Gamma)$ we have $\sigma(x) :_{k,\Psi} \Gamma(x)$; that is,

$$\sigma :_{k,\Psi} \Gamma \quad \stackrel{\text{def}}{=} \quad \forall x \in \text{dom}(\Gamma).\ \sigma(x) :_{k,\Psi} \Gamma(x)$$

[8] To be precise, this statement holds only if F is nonexpansive (see Section 3.3.9).

$$
\begin{array}{lll}
\bot & \stackrel{\text{def}}{=} & \{\} \\[1ex]
\mathsf{unit} & \stackrel{\text{def}}{=} & \{\langle k, \Psi, \mathtt{unit}\rangle\} \\[1ex]
\tau_1 \rightarrow \tau_2 & \stackrel{\text{def}}{=} & \{\langle k, \Psi, \lambda x.e\rangle \mid \forall v, \Psi', j < k.\ ((k, \Psi) \sqsubseteq (j, \Psi') \ \wedge\ \langle j, \Psi', v\rangle \in \tau_1) \\
 & & \qquad\qquad\qquad\qquad\qquad \implies e[v/x] :_{j,\Psi'} \tau_2\} \\[1ex]
\mathsf{ref}\ \tau & \stackrel{\text{def}}{=} & \{\langle k, \Psi, \ell\,\rangle \mid \lfloor \Psi \rfloor_k(\ell) = \lfloor \tau \rfloor_k\} \\[1ex]
\forall F & \stackrel{\text{def}}{=} & \{\langle k, \Psi, \Lambda.e\rangle \mid \forall j, \Psi', \tau.\ ((k, \Psi) \sqsubseteq (j, \Psi') \ \wedge\ \text{type}(\lfloor \tau \rfloor_j)) \\
 & & \qquad\qquad\qquad\qquad \implies \forall i < j.\ e :_{i, \lfloor \Psi' \rfloor_i} F(\tau)\} \\[1ex]
\exists F & \stackrel{\text{def}}{=} & \{\langle k, \Psi, \mathtt{pack}\, v\rangle \mid \exists \tau.\ (\text{type}(\lfloor \tau \rfloor_k) \ \wedge\ \forall j < k.\ \langle j, \lfloor \Psi \rfloor_j, v\rangle \in F(\tau))\}
\end{array}
$$

Figure 3.6: Mutable References (λ^M): Type Definitions

I write $\Gamma \vDash_M^k e : \tau$ *iff* $FV(e) \subseteq \text{dom}(\Gamma)$ *and*

$$\forall \sigma, \Psi.\ \sigma :_{k,\Psi} \Gamma \implies \sigma(e) :_{k,\Psi} \tau$$

where $\sigma(e)$ *is the result of replacing the free variables in e with their values under* σ*.*

I write $\Gamma \vDash_M e : \tau$ *if for all* $k \geq 0$ *we have* $\Gamma \vDash_M^k e : \tau$*. Finally, I write* $\vDash_M e : \tau$ *to mean* $\Gamma_0 \vDash_M e : \tau$ *for the empty context* Γ_0*.*

Hence, the meaning of the judgment $\Gamma \vDash_M^k e : \tau$ on an open expression e and type τ can be obtained from our semantics of a similar judgment on closed expressions, so long as we quantify over all legitimate substitutions of values for variables. Note that $\Gamma \vDash_M e : \tau$ can be viewed as a three place relation that holds on the context Γ, the term e, and the type τ.

The typing rules for λ^M are given in Figure 3.7. As before, I write $\Gamma\,[x \mapsto \tau]$ to denote the type environment that extends Γ by mapping x to τ where $x \notin \text{dom}(\Gamma)$. I shall prove each of the type-checking rules in Figure 3.7 as lemmas (see Section 3.5). These lemmas can then be used in the same manner as standard typing rules to prove statements of the form $\Gamma \vDash_M e : \tau$.

A "program" in λ^M is a closed expression that does not contain any location symbols ℓ. When a program begins executing, it steps (by means of `new`) to expressions that may contain location symbols. A conventional subject-reduction proof requires that the static type system be able to type-check programs during execution, which means that there must be a way to type-check locations ℓ. However, our judgment $\Gamma \vDash_M e : \tau$ has no provision to type-check locations (for instance, there is no store typing Ψ to the left of the $\vDash_M$ symbol). Since we are not doing subject

$$\frac{}{\Gamma \vDash_M x : \Gamma(x)}\ \texttt{(M-var)} \qquad \frac{}{\Gamma \vDash_M \texttt{unit} : \mathsf{unit}}\ \texttt{(M-unit)}$$

$$\frac{\Gamma\,[x \mapsto \tau_1] \vDash_M e : \tau_2}{\Gamma \vDash_M \lambda x.e : \tau_1 \to \tau_2}\ \texttt{(M-abs)}$$

$$\frac{\Gamma \vDash_M e_1 : \tau_1 \to \tau_2 \quad \Gamma \vDash_M e_2 : \tau_1}{\Gamma \vDash_M (e_1\, e_2) : \tau_2}\ \texttt{(M-app)}$$

$$\frac{\Gamma \vDash_M e : \tau}{\Gamma \vDash_M \texttt{new}(e) : \mathsf{ref}\ \tau}\ \texttt{(M-new)} \qquad \frac{\Gamma \vDash_M e : \mathsf{ref}\ \tau}{\Gamma \vDash_M\, !\,e : \tau}\ \texttt{(M-deref)}$$

$$\frac{\Gamma \vDash_M e_1 : \mathsf{ref}\ \tau \qquad \Gamma \vDash_M e_2 : \tau}{\Gamma \vDash_M e_1 := e_2 : \mathsf{unit}}\ \texttt{(M-assign)}$$

$$\frac{\forall \tau.\, \mathrm{type}(\tau) \implies \Gamma \vDash_M e : F(\tau)}{\Gamma \vDash_M \Lambda.e : \forall F}\ \texttt{(M-tabs)} \qquad \frac{\mathrm{type}(\tau) \qquad \Gamma \vDash_M e : \forall F}{\Gamma \vDash_M e[\,] : F(\tau)}\ \texttt{(M-tapp)}$$

$$\frac{\mathrm{type}(\tau) \qquad \Gamma \vDash_M e : F(\tau)}{\Gamma \vDash_M \texttt{pack}\, e : \exists F}\ \texttt{(M-pack)}$$

$$\frac{\Gamma \vDash_M e_1 : \exists F \qquad \forall \tau.\, \mathrm{type}(\tau) \implies \Gamma\,[x \mapsto F(\tau)] \vDash_M e_2 : \tau_2}{\Gamma \vDash_M \texttt{unpack}\, e_1\, \mathsf{as}\, x\, \texttt{in}\, e_2 : \tau_2}\ \texttt{(M-unpack)}$$

Figure 3.7: Mutable References (λ^M): Type-checking Lemmas

reduction, we do not need to type-check executing programs — hence, we do not need to type-check location symbols.

I wish to prove that a λ^M program is safe to execute given an initial store S. There are no additional constraints on S. In fact, S might as well be the empty store because a program e — since it has no free variables or location symbols — cannot access any locations in S. Hence, I would like to prove that given an arbitrary store S and a well-typed program e, machine state (S, e) is safe.

Theorem 3.8 (Safety)
If $\vDash_M e : \tau$, τ is a type, and S is a store, then (S, e) is safe.

Proof: To prove safe(S, e) we must show that for any $k \geq 0$, it is safe to execute (S, e) for k steps. Hence, we must show that for any state (S', e') reachable from (S, e) in less than k steps, either e' is a value or another step is possible. Suppose $(S, e) \longmapsto_M^j (S', e')$ where $j < k$. If (S', e') is not irreducible, then there must exist some (S'', e'') such that $(S', e') \longmapsto_M (S'', e'')$. Otherwise, (S', e') is irreducible and we must prove that e' is a value. By Definition 3.7, from $\models_M e : \tau$ we have $\Gamma_0 \models_M^k e : \tau$, where Γ_0 is the empty context, and it follows that e is closed. Choose the empty value environment σ_0 and the empty store typing Ψ_0. By the definition of $\Gamma_0 \models_M^k e : \tau$ (Definition 3.7) we have the following.

$$\sigma_0 :_{k,\Psi_0} \Gamma_0 \implies \sigma_0(e) :_{k,\Psi_0} \tau$$

The premise is trivially satisfied; applying the trivial substitution we obtain $e :_{k,\Psi_0} \tau$. Since Ψ_0 is the empty store typing, it trivially follows that $S :_k \Psi_0$. Then, from $e :_{k,\Psi_0} \tau$ (by Definition 3.6) — since $S :_k \Psi_0$, $(S, e) \longmapsto_M^j (S', e')$ for $j < k$, and irred(S', e') — it follows that there exists a store typing Ψ' such that $(k, \Psi_0) \sqsubseteq (k - j, \Psi')$, $S' :_{k-j} \Psi'$, and $\langle k - j, \Psi', e' \rangle \in \tau$. Since τ is a type it follows that val(e'). □

3.3.8 Components of Possible-Worlds Model

In Section 2.2.1 I described a model of λ^I (λ-calculus with immutable references) and discussed connections between it and Kripke models of modal logic. Those observations continue to hold for our indexed model of λ^M which is also a possible-worlds model, though the set of worlds, the accessibility relation, and the labeling function for our current possible-worlds semantics are different from what we saw for λ^I. Let us look at each of the things one must specify when building a possible-worlds model.

First, one specifies a set W, whose elements are called *worlds*. In Kripke's multiple-world interpretation of modal logic, a proposition is true or false relative to a world. In our model, a value v has type τ relative to a world, that is, relative to a state of computation during evaluation. All the information that we need from a world — that is, in order to decide whether v has type τ — may be given by pairs of the form (k, Ψ) where k is the number of steps left to execute and Ψ is the store typing that describes the store component of the current machine state. Hence, in our model the set of worlds W is as follows:

$$W = \{ (k, \Psi) \mid k \geq 0 \ \wedge \ \forall \ell \in \text{dom}(\Psi). \ (\forall \langle k', \Psi', v \rangle \in \Psi(\ell). \ k' < k) \}$$

Note that we could have used tuples (k, Ψ, S) as worlds, but instead, we have constructed the model in such a way that S does not need to be a part of the world (for reasons that were first discussed in Section 3.2.2). Informally, all the information that we might need from S (in order to decide whether some v has type τ at some state of computation) is available from $\lfloor \Psi \rfloor_k$.

Second, one specifies a relation $Acc \subseteq W \times W$ called the *accessibility* relation. Given worlds w and w', $Acc(w, w')$ says that w' is accessible from w. In our model, this corresponds to the state extension relation ($\sqsubseteq$) and $(k, \Psi) \sqsubseteq (j, \Psi')$ says that state (j, Ψ') is accessible (or possible) from state (k, Ψ).

Third, one specifies a *labeling function* $L : W \to \mathcal{P}(\mathtt{Atoms})$ that, given a world w, yields the set of atomic propositions that hold in that world. The atomic propositions p that we are interested in are of the form: "location l is allocated and its contents belong to τ." Therefore, in our model,

$$L(k, \Psi) = \{\, (\ell, \tau) \mid \lfloor \Psi \rfloor_k(\ell) = \tau \,\}$$

Finally, one specifies the properties that the accessibility relation Acc should satisfy. Our model is constructed to ensure that the $\sqsubseteq$ relation is reflexive and transitive. As for λ^I, this suggests a connection with the modal logic S4 since models of S4 also require that the accessibility relation be reflexive and transitive.

3.3.9 Well Founded and Nonexpansive Type Functions

In the next two sections, I shall prove the validity of all λ^M types (Section 3.4) and typing rules (Section 3.5). In order to prove some of the lemmas for quantified types, I must first define the notion of a *nonexpansive* type functional.[9]

Informally, a type functional F is nonexpansive if, in order to determine whether or not e has type $F(\tau)$ to approximation k, we do not need to know the entire set τ, but rather, it suffices to know the set τ to approximation k.

Definition 3.9 (Nonexpansive)
A nonexpansive functional *is a function F from types to types such that for any type τ and $k \geq 0$ we have*

$$\lfloor F(\tau) \rfloor_k \;=\; \lfloor F(\lfloor \tau \rfloor_k) \rfloor_k$$

A related notion is that of a *well founded* type functional. Informally, a type functional F is well founded if, in order to determine whether or not e has type $F(\tau)$ to approximation k, it suffices to know the set τ to approximation $k - 1$.

[9] Specifically, the lemmas for the validity of existential types $\exists F$ (Lemma 3.19) and the typing rules for type abstraction ($\forall F$ introduction, Theorem 3.28) and unpack ($\exists F$ elimination, Theorem 3.33) require that F be a nonexpansive type functional.

Definition 3.10 (Well Founded)
A well founded functional *is a function F from types to types such that for any type τ and $k \geq 0$ we have*

$$\lfloor F(\tau) \rfloor_{k+1} \;=\; \lfloor F(\lfloor \tau \rfloor_k) \rfloor_{k+1}$$

Let us consider some examples of nonexpansive and well founded functionals. I write $\lambda\alpha.\alpha$ for the identity type function F (where λ should not be confused with the notation for lambda abstraction in our syntax of terms). The type function $\lambda\alpha.\alpha$ is nonexpansive but not well founded. Meanwhile the type functions $\lambda\alpha.\mathsf{ref}\ \alpha$ and $\lambda\alpha.\alpha \to \alpha$ are well founded since the type constructors ref and → are well founded. Informally, e has type $\mathsf{ref}\ \tau$ to approximation k as long as e "contains" some value that has type τ to approximation $k-1$ — that is, it suffices to only know the set τ to approximation $k-1$. Intuitively, ref is well founded because dereferencing takes up one step. Similarly, → is well founded because beta-reduction takes up one step. Note that every well founded constructor is nonexpansive.

The definitions of both well founded and nonexpansive functionals are due to Appel and McAllester [AM01] who used the term "nonexpansive" by analogy with the metric-space semantics of MacQueen, Plotkin, and Sethi [MPS86]. Appel and McAllester proved that all the constructors that can be built by compositions of their type constructors are nonexpansive. The same result holds in our model, though the semantics of *our* type constructors is different. I shall state the lemma here without proof. The proofs are similar to those by Appel and McAllester.

Lemma 3.11 (Nonexpansive Constructors)

1. *Every well founded constructor is nonexpansive.*

2. $\lambda\alpha.\alpha$ *is nonexpansive.*

3. $\lambda\alpha.\tau$*, where* α *is not free in* τ*, is well founded.*

4. *The composition of nonexpansive constructors is nonexpansive.*

5. *The composition of a nonexpansive constructor with a well founded constructor (in either order) is well founded.*

6. *If F and G are nonexpansive, then $\lambda.F\alpha \to G\alpha$ is well founded.*

7. *If F is nonexpansive, then $\lambda\alpha.\mathsf{ref}(F\alpha)$ is well founded.*

In Chapter 4 (Section 4.1) I shall explain how to model quantified types $\forall F$ and $\exists F$ in a language that treats type application and the unpacking of existentials as coercions rather than explicit "run time" steps. To accommodate such an operational semantics, the model requires that the type functionals F be well founded. The notion of well founded functionals is also necessary for proving the validity of recursive types μF (as I shall explain in Chapter 4, Section 4.2.1).

3.4 Validity of Types

The safety theorem (Theorem 3.8) states that a program e with type τ is safe to execute (Theorem 3.8) as long as τ is a type (Definition 3.4). In this section, I shall prove that each of the type constructors shown in Figure 3.6 is a type, or produces a type when applied to valid arguments. In each case I have to show that types are preserved under valid state extension ($\sqsubseteq$). I shall first prove some properties of state extension that we will need later, namely that valid state extension is both reflexive and transitive.

Lemma 3.12 (State Extension Reflexive)
$(k, \Psi) \sqsubseteq (k, \Psi)$.

PROOF: Trivial. □

Lemma 3.13 (State Extension Transitive)
If $(k_1, \Psi_1) \sqsubseteq (k_2, \Psi_2)$ *and* $(k_2, \Psi_2) \sqsubseteq (k_3, \Psi_3)$ *then* $(k_1, \Psi_1) \sqsubseteq (k_3, \Psi_3)$.

PROOF: The proof is in two parts. First, from the premises of the lemma and the definition of $\sqsubseteq$ we have $k_2 \leq k_1$ and $k_3 \leq k_2$. It follows that $k_3 \leq k_1$. Second, suppose $\ell \in \mathrm{dom}(\Psi_1)$. Then the first premise of the lemma allows us to conclude that $\lfloor \Psi_2 \rfloor_{k_2}(\ell) = \lfloor \Psi_1 \rfloor_{k_2}(\ell)$. It follows that $\ell \in \mathrm{dom}(\Psi_2)$. Then, from the second premise of the lemma we have $\lfloor \Psi_3 \rfloor_{k_3}(\ell) = \lfloor \Psi_2 \rfloor_{k_3}(\ell)$. From $\lfloor \Psi_2 \rfloor_{k_2}(\ell) = \lfloor \Psi_1 \rfloor_{k_2}(\ell)$ and $k_3 \leq k_2$ it follows that $\lfloor \Psi_2 \rfloor_{k_3}(\ell) = \lfloor \Psi_1 \rfloor_{k_3}(\ell)$ (by Definition 3.2 (Approx)). Hence we may conclude that $\lfloor \Psi_3 \rfloor_{k_3}(\ell) = \lfloor \Psi_1 \rfloor_{k_3}(\ell)$ by transitivity of set equality. □

The fact that $\bot$ and unit are types follows immediately from their definitions. Next, we prove that $\tau_1 \to \tau_2$ is a type as long as τ_1 and τ_2 are types — the proof relies on the transitivity of state extension.

Lemma 3.14 (Type $\tau_1 \to \tau_2$)
If τ_1 *and* τ_2 *are types then* $\tau_1 \to \tau_2$ *is also a type.*

PROOF: We must prove that $\tau_1 \to \tau_2$ is closed under valid state extension. Suppose that $\langle k, \Psi, v \rangle \in \tau_1 \to \tau_2$ and $(k, \Psi) \sqsubseteq (j, \Psi')$. Note that $\langle k, \Psi, v \rangle \in \tau_1 \to \tau_2$ implies that v is of the form $\lambda x.e$. We must prove that $\langle j, \Psi', \lambda x.e \rangle \in \tau_1 \to \tau_2$. Suppose $(j, \Psi') \sqsubseteq (i, \Psi'')$ and $\langle i, \Psi'', v'' \rangle \in \tau_1$ for $i < j$ and some value v'' — we have to show $e[v''/x] :_{i,\Psi''} \tau_2$. Since $i < j$ and by the definition of $\sqsubseteq$ we have $j \leq k$, it follows that $i < k$; by Lemma 3.13 we have $(k, \Psi) \sqsubseteq (i, \Psi'')$; and we already have $\langle i, \Psi'', v'' \rangle \in \tau_1$. These three statements together with $\langle k, \Psi, \lambda x.e \rangle \in \tau_1 \to \tau_2$ and the definition of $\to$ allow us to conclude that $e[v''/x] :_{i,\Psi''} \tau_2$. □

Lemma 3.15 (Type ref)
If τ is a type then $\mathsf{ref}\ \tau$ *is a type.*

Proof: To show that $\mathsf{ref}\,\tau$ is closed under state extension, suppose that $\langle k, \Psi, v\rangle \in \mathsf{ref}\ \tau$ and $(k, \Psi) \sqsubseteq (j, \Psi')$. We must prove that $\langle j, \Psi', v\rangle \in \mathsf{ref}\ \tau$. By the definition of ref, since $\langle k, \Psi, v\rangle \in \mathsf{ref}\ \tau$, it follows that v is some location ℓ and $\lfloor \Psi \rfloor_k(\ell) = \lfloor \tau \rfloor_k$. The latter implies that $\ell \in \mathrm{dom}(\Psi)$. Then, from $(k, \Psi) \sqsubseteq (j, \Psi')$ it follows that $\lfloor \Psi' \rfloor_j(\ell) = \lfloor \Psi \rfloor_j(\ell)$, and hence, $\ell \in \mathrm{dom}(\Psi')$. In addition, from $\lfloor \Psi \rfloor_k(\ell) = \lfloor \tau \rfloor_k$ and $j \leq k$, it follows by Definition 3.2 (Approx) that $\lfloor \Psi \rfloor_j(\ell) = \lfloor \tau \rfloor_j$, and so we may conclude that $\lfloor \Psi' \rfloor_j(\ell) = \lfloor \tau \rfloor_j$ by transitivity of set equality. □

Lemma 3.16 (Type $\forall F$)
If F is a function from types to types then $\forall F$ is a type.

Proof: To prove that $\forall F$ is closed under state extension, suppose that $\langle k, \Psi, v\rangle \in \forall F$ and $(k, \Psi) \sqsubseteq (j, \Psi')$. Note that $\langle k, \Psi, v\rangle \in \forall F$ implies that v is of the form $\Lambda.e$. We must show that $\langle j, \Psi', \Lambda.e\rangle \in \forall F$. Suppose $(j, \Psi') \sqsubseteq (i, \Psi'')$ and $\mathrm{type}(\lfloor \tau \rfloor_i)$ for some set τ — we must show that $e :_{i', \lfloor \Psi'' \rfloor_{i'}} F(\tau)$ for all $i' < i$. Since state extension is transitive (Lemma 3.13), we have $(k, \Psi) \sqsubseteq (i, \Psi'')$. Now, from $\langle k, \Psi, \Lambda.e\rangle \in \forall F$ we may conclude that for any $i' < i$, $e :_{i', \lfloor \Psi'' \rfloor_{i'}} F(\tau)$. □

Lemma 3.17
If $\mathrm{type}(\lfloor \tau \rfloor_k)$ *and $j \leq k$ then* $\mathrm{type}(\lfloor \tau \rfloor_j)$.

Proof: Immediate from definitions 3.4 (Type) and 3.2 (Approx). □

Lemma 3.18
If $(k, \Psi) \sqsubseteq (j, \Psi')$, $i < k$, and $i < j$, then $(i, \lfloor \Psi \rfloor_i) \sqsubseteq (i, \lfloor \Psi' \rfloor_i)$.

Proof: Immediate from definitions 3.3 (State Extension) and 3.2 (Approx). □

Lemma 3.19 (Type $\exists F$)
If F is a nonexpansive functional then $\exists F$ is a type.

Proof: To prove that $\exists F$ is closed under memory extension, suppose that $\langle k, \Psi, v\rangle \in \exists F$ and $(k, \Psi) \sqsubseteq (j, \Psi')$. Note that $\langle k, \Psi, v\rangle \in \exists F$ implies that v is of the form $\mathtt{pack}\, v'$ where v' is a value. We must show that $\langle j, \Psi', \mathtt{pack}\, v'\rangle \in \exists F$. The proof has two parts. First, we have to prove the existence of some set τ such that $\mathrm{type}(\lfloor \tau \rfloor_j)$. From $\langle k, \Psi, \mathtt{pack}\, v'\rangle \in \exists F$ it follows by the definition of $\exists F$ that there exists some τ such that $\mathrm{type}(\lfloor \tau \rfloor_k)$. Since $j \leq k$ it follows by Lemma 3.17 that $\mathrm{type}(\lfloor \tau \rfloor_j)$.

For the second part, let $i < j$; we must show $\langle i, \lfloor \Psi' \rfloor_i, v'\rangle \in F(\tau)$. Since $i < j$ and $j \leq k$ we have $i < k$, and from $\langle k, \Psi, \mathtt{pack}\, v'\rangle \in \exists F$ we have $\langle i, \lfloor \Psi \rfloor_i, v'\rangle \in F(\tau)$.

Since F is a function from types to types and $\lfloor\tau\rfloor_j$ is a type, $F(\lfloor\tau\rfloor_j)$ is a type. It follows that $\lfloor F(\lfloor\tau\rfloor_j)\rfloor_j$ is a type (by Definition 3.2 (Approx)). Since F is nonexpansive, we have that $\lfloor F(\lfloor\tau\rfloor_j)\rfloor_j = \lfloor F(\tau)\rfloor_j$. Hence $\lfloor F(\tau)\rfloor_j$ is a type. Now, since $\langle i, \lfloor\Psi\rfloor_i, v'\rangle \in F(\tau)$ and $i < j$, it follows that $\langle i, \lfloor\Psi\rfloor_i, v'\rangle \in \lfloor F(\tau)\rfloor_j$. Furthermore, since $(k, \Psi) \sqsubseteq (j, \Psi')$, $i < j$, and $i < k$, by Lemma 3.18 we may conclude that $(i, \lfloor\Psi\rfloor_i) \sqsubseteq (i, \lfloor\Psi'\rfloor_i)$. Since types are closed under state extension, from $\langle i, \lfloor\Psi\rfloor_i, v'\rangle \in \lfloor F(\tau)\rfloor_j$ and $\text{type}(\lfloor F(\tau)\rfloor_j)$ we may conclude that $\langle i, \lfloor\Psi'\rfloor_i, v'\rangle \in \lfloor F(\tau)\rfloor_j$. But since $i < j$ it follows that $\langle i, \lfloor\Psi'\rfloor_i, v'\rangle \in F(\tau)$. □

3.5 Validity of Typing Rules

I shall now prove each of the typing rules in Figure 3.7 as lemmas. The lemma for variables, stating that $\Gamma \vDash_M x : \Gamma(x)$, follows immediately from the definition of $\vDash_M$. The lemma for `M-unit` is immediate from the definition of `unit`.

3.5.1 Lambda Abstraction and Application

Theorem 3.20 (Abstraction)
Let Γ be a type environment, let τ_1 and τ_2 be types, and let $\Gamma[x := \tau_1]$ be the type environment that is identical to Γ except that it maps x to τ_1. If $\Gamma[x := \tau_1] \vDash_M e : \tau_2$ then $\Gamma \vDash_M \lambda x.e : \tau_1 \to \tau_2$.

Proof: We must show that under the premises of the theorem for any $k \geq 0$ we have $\Gamma \vDash_M^k \lambda x.e : \tau_1 \to \tau_2$. More specifically, for any σ and Ψ such that $\sigma :_{k,\Psi} \Gamma$ we must show that $\sigma(\lambda x.e) :_{k,\Psi} \tau_1 \to \tau_2$. Suppose $\sigma :_{k,\Psi} \Gamma$. Let $j < k$ and v, Ψ', be such that $(k, \Psi) \sqsubseteq (j, \Psi')$ and $\langle j, \Psi', v\rangle \in \tau_1$. By definition of $\to$ it now suffices to show that $\sigma(e[v/x]) :_{j,\Psi'} \tau_2$. Let $\sigma\,[x \mapsto v]$ be the substitution identical to σ except that it maps x to v. Since the codomain of Γ contains types (which are closed under state extension), and since $v :_{j,\Psi'} \tau_1$, we now have that $\sigma\,[x \mapsto v] :_{j,\Psi'} \Gamma\,[x \mapsto \tau_1]$. To show $\sigma(e[v/x]) :_{j,\Psi'} \tau_2$, suppose $S' :_j \Psi'$. By the premise $\Gamma\,[x \mapsto \tau_1] \vDash_M e : \tau_2$, together with $S' :_j \Psi'$ and $\sigma\,[x \mapsto v] :_{j,\Psi'} \Gamma\,[x \mapsto \tau_1]$ we have $\sigma\,[x \mapsto v](e) :_{j,\Psi'} \tau_2$. But this implies $\sigma(e[v/x]) :_{j,\Psi'} \tau_2$. □

Theorem 3.21 (Application)
If Γ is a type environment, e_1 and e_2 are (possibly open) terms, and τ_1 and τ_2 are types such that $\Gamma \vDash_M e_1 : \tau_1 \to \tau_2$ and $\Gamma \vDash_M e_2 : \tau_1$ then $\Gamma \vDash_M (e_1\, e_2) : \tau_2$.

Proof: We must prove that under the premises of the theorem, for any $k \geq 0$, we have $\Gamma \vDash_M^k (e_1\, e_2) : \tau_2$. More specifically, for any σ and Ψ such that $\sigma :_{k,\Psi} \Gamma$ we must show $\sigma(e_1\, e_2) :_{k,\Psi} \tau_2$. From the premise $\Gamma \vDash_M e_1 : \tau_1 \to \tau_2$ we have

$\sigma(e_1) :_{k,\Psi} \tau_1 \to \tau_2$. To show $\sigma(e_1\, e_2) :_{k,\Psi} \tau_2$, suppose $S :_k \Psi$ for some store S. Then, from $\sigma(e_1) :_{k,\Psi} \tau_1 \to \tau_2$ it follows that $(S, \sigma(e_1))$ is safe for k steps. Either $(S, \sigma(e_1))$ reduces for k steps without reaching a state (S', v_1) where v_1 is a value — in which case $(S, \sigma(e_1\, e_2))$ does not generate a value in less than k steps and hence $\sigma(e_1\, e_2) :_{k,\Psi} \tau_2$ (for any τ_2) — or the value v_1 is a lambda expression $\lambda x.e$. In the latter case, since $\sigma(e_1) :_{k,\Psi} \tau_1 \to \tau_2$ and $(S, \sigma(e_1)) \longmapsto^j_M (S', \lambda x.e)$, where $j < k$ and $\text{irred}(S', \lambda x.e)$, it follows that there exists a Ψ' such that $(k, \Psi) \sqsubseteq (k - j, \Psi')$ and $S' :_{k-j} \Psi'$ and $\langle k - j, \Psi', \lambda x.e\rangle \in \tau_1 \to \tau_2$.

From $\sigma :_{k,\Psi} \Gamma$ it follows that $\sigma(x) :_{k,\Psi} \Gamma(x)$ for all variables $x \in \text{dom}(\Gamma)$. A type environment Γ is a mapping from variables to types. Hence, since $(k, \Psi) \sqsubseteq (k - j, \Psi')$, it follows that $\sigma(x) :_{k-j,\Psi'} \Gamma(x)$ for all $x \in \text{dom}(\Gamma)$ — that is, we have $\sigma :_{k-j,\Psi'} \Gamma$. Now, from premise $\Gamma \vDash_M e_2 : \tau_1$, since $k - j \geq 0$, $S' :_{k-j} \Psi'$, and $\sigma :_{k-j,\Psi'} \Gamma$, we have $\sigma(e_2) :_{k-j,\Psi'} \tau_1$. It follows that $(S', \sigma(e_2))$ is safe for $k - j$ steps, i.e., either $(S', \sigma(e_2))$ does not generate a value in fewer than $k - j$ steps — in which case, $(S, \sigma(e_1\, e_2))$ does not generate a value in fewer than k steps so we have $\sigma(e_1\, e_2) :_{k,\Psi} \tau_2$ (for any τ_2) — or $(S', \sigma(e_2)) \longmapsto^i_M (S'', v)$ where $i < k - j$. In the latter case, $(S, \sigma(e_1\, e_2)) \longmapsto^{j+i}_M (S'', (\lambda x.e)v)$ where $j + i < k$. Also, since $\sigma(e_2) :_{k-j,\Psi'} \tau_1$ and $(S', \sigma(e_2)) \longmapsto^i_M (S'', v)$ for $i < k - j$ and $\text{irred}(S'', v)$, it follows that there exists a Ψ'' such that $(k - j, \Psi') \sqsubseteq (k - j - i, \Psi'')$, $S'' :_{k-j-i} \Psi''$, and $\langle k - j - i, \Psi'', v\rangle \in \tau_1$.

Pick memory typing $\Psi^* = \lfloor \Psi'' \rfloor_{k-j-i-1}$. Let $k^* = k - j - i - 1$. Then the following information-forgetting state extension holds: $(k - j - i, \Psi'') \sqsubseteq (k^*, \Psi^*)$. Since $\langle k - j - i, \Psi'', v\rangle \in \tau_1$ and τ_1 is a type, we have $\langle k^*, \Psi^*, v\rangle \in \tau_1$. The definition of $\to$ then implies that $e[v/x] :_{k^*,\Psi^*} \tau_2$. But we now have $(S, \sigma(e_1\, e_2)) \longmapsto^{j+i+1}_M (S'', e[v/x])$, $(k, \Psi) \sqsubseteq (k^*, \Psi^*)$, $S'' :_{k^*} \Psi^*$, and $e[v/x] :_{k^*,\Psi^*} \tau_2$. By Definition 3.6 (Expr : Type), this means that if $(S'', e[v/x])$ generates a value in fewer than k^* steps then that value will be of type τ_2. Hence, we may conclude that $\sigma(e_1\, e_2) :_{k,\Psi} \tau_2$ as we wanted to show. □

3.5.2 Allocation, Assignment, Dereferencing

I use the notation $\mathsf{aprx\text{-}extend}(k, \Psi, \ell, \tau)$ to denote a store typing that extends $\lfloor \Psi \rfloor_k$ by mapping ℓ to $\lfloor \tau \rfloor_k$. $\mathsf{aprx\text{-}extend}$ serves as a useful abbreviation in the proof of the `M-new` typing rule.

Definition 3.22 (Approximately Extend Store Typing)
The approximate extension of a store typing Ψ with location ℓ mapped to type τ is defined as follows:

$$\mathsf{aprx\text{-}extend}(k, \Psi, \ell, \tau) \;=\; \lfloor \Psi \rfloor_k \;\cup\; (\ell \mapsto \lfloor \tau \rfloor_k)$$

Lemma 3.23 (Closed New)
If e is a closed term and τ is a type such that $e :_{k,\Psi} \tau$ then $\texttt{new}(e) :_{k,\Psi} \mathsf{ref}\ \tau$.

PROOF: Since τ is a type by Lemma 3.15 $\mathsf{ref}\,\tau$ is a type. We must show $\texttt{new}(e) :_{k,\Psi} \mathsf{ref}\,\tau$. Suppose $S :_k \Psi$ for some store S. Then we must show that $(S, \texttt{new}(e))$ is safe for k steps.

From $e :_{k,\Psi} \tau$ and $S :_k \Psi$ it follows that (S, e) is safe for k steps. Hence, if the state $(S, \texttt{new}(e))$ reduces for k steps without reaching a state of the form $(S', \texttt{new}(v))$, it follows that $\texttt{new}(e) :_{k,\Psi} \mathsf{ref}\ \tau$. So we assume without loss of generality that $(S, \texttt{new}(e)) \longmapsto^{j}_{M} (S', \texttt{new}(v))$ for some $j < k$, store S', and value v. Then $e :_{k,\Psi} \tau$ implies that there exists a Ψ' such that we have

1. $(k, \Psi) \sqsubseteq (k - j, \Psi')$
2. $S' :_{k-j} \Psi'$
3. $\langle k - j, \Psi', v\rangle \in \tau$
4. Pick some location ℓ such that $\ell \notin \mathrm{dom}(S')$. Suppose that $(S', \texttt{new}(v)) \longmapsto_M (S'\,[\ell \mapsto v], \ell)$
5. Pick $\Psi'' = \mathsf{aprx\text{-}extend}(k - j - 1, \lfloor\Psi'\rfloor_{k-j-1}, \ell, \tau)$. Note that $\lfloor\Psi''\rfloor_{k-j-1} = \Psi''$ by the definition of $\mathsf{aprx\text{-}extend}$.
6. The following information-forgetting store extension holds: $(k - j, \Psi') \sqsubseteq (k - j - 1, \lfloor\Psi'\rfloor_{k-j-1})$.
7. We must show that $(k - j - 1, \lfloor\Psi'\rfloor_{k-j-1}) \sqsubseteq (k - j - 1, \Psi'')$. Suppose $\ell_1 \in \mathrm{dom}(\lfloor\Psi'\rfloor_{k-j-1})$ — we must show that $\lfloor\Psi'\rfloor_{k-j-1}(\ell_1) = \lfloor\Psi''\rfloor_{k-j-1}(\ell_1) = \Psi''(\ell_1)$. Since $\ell_1 \in \mathrm{dom}(\lfloor\Psi'\rfloor_{k-j-1})$ and since we have $\ell \notin \mathrm{dom}(\lfloor\Psi'\rfloor_{k-j-1}$, it follows that $\ell_1 \neq \ell$. Then $\Psi''(\ell_1) = \lfloor\lfloor\Psi'\rfloor_{k-j-1}\rfloor_{k-j-1}(\ell_1) = \lfloor\Psi'\rfloor_{k-j-1}(\ell_1)$ as needed.
8. We must show that $S'\,[\ell \mapsto v] :_{k-j-1} \Psi''$. First, from (2) we have $\mathrm{dom}(\Psi') \subseteq \mathrm{dom}(S')$, which implies that $\mathrm{dom}(\lfloor\Psi'\rfloor_{k-j-1}) \subseteq \mathrm{dom}(S')$. Then $\mathrm{dom}(\Psi'') = \mathrm{dom}(\lfloor\Psi'\rfloor_{k-j-1}) \cup \{\,\ell\,\} \subseteq \mathrm{dom}(S'\,[\ell \mapsto v])$. Second, pick $i < k - j - 1$ and $\ell_1 \in \mathrm{dom}(\Psi'')$; we have to show $\langle i, \lfloor\Psi''\rfloor_i, S'\,[\ell \mapsto v](\ell_1)\rangle \in \lfloor\Psi''\rfloor_{k-j-1}(\ell_1)$.
   - Suppose $\ell = \ell_1$. Then we must show $\langle i, \lfloor\Psi''\rfloor_i, v\rangle \in \Psi''(\ell) = \lfloor\tau\rfloor_{k-j-1}$. Since $\langle k - j, \Psi', v\rangle \in \tau$, type($\tau$), and by (6) and (7) we have $(k - j, \Psi') \sqsubseteq (k - j - 1, \Psi'') \sqsubseteq (i, \lfloor\Psi''\rfloor_i)$, it follows that $\langle i, \lfloor\Psi''\rfloor_i, v\rangle \in \tau$. From $i < k - j - 1$ it follows that $\langle i, \lfloor\Psi''\rfloor_i, v\rangle \in \lfloor\tau\rfloor_{k-j-1}$.

- Suppose $\ell \neq \ell_1$. Then we must show that $\langle i, \lfloor\Psi''\rfloor_i, S'(\ell_1)\rangle \in \lfloor\Psi'\rfloor_{k-j-1}(\ell_1)$. From $S' :_{k-j} \Psi'$ and $\ell_1 \in \mathrm{dom}(\Psi')$ we have $\langle k-j-1, \lfloor\Psi'\rfloor_{k-j-1}, S'(\ell_1)\rangle \in \lfloor\Psi'\rfloor_{k-j}(\ell_1)$. From (7) and the fact that $\lfloor\Psi'\rfloor_{k-j}(\ell_1)$ is a type, we have $\langle k-j-1, \Psi'', S'(\ell_1)\rangle \in \lfloor\Psi'\rfloor_{k-j}(\ell_1)$. Then, since $i < k-j-1$ and $(k-j-1, \Psi'') \sqsubseteq (i, \lfloor\Psi''\rfloor_i)$, we have $\langle i, \lfloor\Psi''\rfloor_i, S'(\ell_1)\rangle \in \lfloor\Psi'\rfloor_{k-j}(\ell_1)$. Since $i < k-j-1$, we have $\langle i, \lfloor\Psi''\rfloor_i, S'(\ell_1)\rangle \in \lfloor\Psi'\rfloor_{k-j-1}(\ell_1)$ as needed.

9. We must show that $\langle k-j-1, \Psi'', \ell\rangle \in \mathsf{ref}\ \tau$. By the definition of ref, it suffices to show $\lfloor\Psi''\rfloor_{k-j-1}(\ell) = \lfloor\tau\rfloor_{k-j-1}$, that is, $\Psi''(\ell) = \lfloor\tau\rfloor_{k-j-1}$. Since $\Psi'' = \mathsf{aprx\text{-}extend}(k-j-1, \lfloor\Psi'\rfloor_{k-j-1}, \ell, \tau)$, by the definition of $\mathsf{aprx\text{-}extend}$ we have $\Psi''(\ell) = \lfloor\tau\rfloor_{k-j-1}$ as we needed to show.

Finally, we have the following:

- $(S, \mathtt{new}(e)) \longmapsto_M^{j+1} (S'[\ell \mapsto v], \ell)$ (from $(S, \mathtt{new}(e)) \longmapsto_M^{j} (S', \mathtt{new}(v))$ and (3));

- $(k, \Psi) \sqsubseteq (k-j-1, \Psi'')$ (from (1), (6), (7), and transitivity of $\sqsubseteq$);

- $S'[\ell \mapsto v] :_{k-j-1} \Psi''$ (from (8);

- $\langle k-j-1, \Psi'', \ell\rangle \in \mathsf{ref}\ \tau$ (from (9)).

Together these statements imply $\mathtt{new}(e) :_{k,\Psi,S} \mathsf{ref}\ \tau$. □

Theorem 3.24 (New)
If Γ is a type environment, e is a (possibly open) term, and τ is a type such that $\Gamma \vDash_M e : \tau$ then $\Gamma \vDash_M \mathtt{new}(e) : \mathsf{ref}\ \tau$

Proof: We must prove that for any $k \geq 0$ we have $\Gamma \vDash_M^k \mathtt{new}(e) : \mathsf{ref}\ \tau$. More specifically, for any σ and Ψ such that $\sigma :_{k,\Psi} \Gamma$, we must show $\sigma(\mathtt{new}(e)) :_{k,\Psi} \mathsf{ref}\ \tau$. Suppose $\sigma :_{k,\Psi} \Gamma$. By the premise of the theorem we have $\sigma(e) :_{k,\Psi} \tau$. The result now follows from Lemma 3.23. □

Lemma 3.25 (Closed Dereferencing)
If e is a closed term and τ is a type set such that $e :_{k,\Psi} \mathsf{ref}\ \tau$ then $!\,e :_{k,\Psi} \tau$.

Proof: Since τ is a type by Lemma 3.15 $\mathsf{ref}\ \tau$ is a type. To show $!\,e :_{k,\Psi} \tau$, let $S :_k \Psi$ for some store S. Then we must show that $(S, !\,e)$ is safe for k steps. Since $e :_{k,\Psi} \mathsf{ref}\ \tau$ it follows that (S, e) is safe for k steps. Hence, the state $(S, !\,e)$ either reduces for k steps without reaching a state of the form $(S', !\,v)$, in which case $!\,e :_{k,\Psi} \tau$ (for any τ), or $(S, !\,e) \longmapsto_M^j (S', !\,\ell)$ where $\ell \in \mathrm{dom}(S')$ and $j < k$. In the latter case, since $e :_{k,\Psi} \mathsf{ref}\ \tau$, Definition 3.6 (Expr : Type) implies that there exists a Ψ' such that $(k, \Psi) \sqsubseteq (k-j, \Psi')$, $S' :_{k-j} \Psi'$, and $\langle k-j, \Psi', \ell \rangle \in \mathsf{ref}\ \tau$. By the definition of ref it follows that $\lfloor \Psi' \rfloor_{k-j}(\ell) = \lfloor \tau \rfloor_{k-j}$. From $S' :_{k-j} \Psi'$, since $\ell \in \mathrm{dom}(\Psi')$, we have $\langle k-j-1, \lfloor \Psi' \rfloor_{k-j-1}, S'(\ell) \rangle \in \lfloor \Psi' \rfloor_{k-j}(\ell)$. Hence, we have $\langle k-j-1, \lfloor \Psi' \rfloor_{k-j-1}, S'(\ell) \rangle \in \lfloor \tau \rfloor_{k-j}$. Now, the following information-forgetting store extension holds: $(k-j, \Psi') \sqsubseteq (k-j-1, \lfloor \Psi' \rfloor_{k-j-1})$. Then, we have

1. $(S, !\, e) \longmapsto_{M}^{j+1} (S', S'(\ell))$ where $\mathrm{irred}(S', S'(\ell))$;
2. $(k, \Psi) \sqsubseteq (k-j-1, \lfloor \Psi' \rfloor_{k-j-1})$;
3. $S' :_{k-j-1} \lfloor \Psi' \rfloor_{k-j-1}$;
4. $\langle k-j-1, \lfloor \Psi' \rfloor_{k-j-1}, S'(\ell) \rangle \in \tau$

These four statements imply $!\, e :_{k,\Psi} \tau$. □

Theorem 3.26 (Dereferencing)
If Γ is a type environment, e is a (possibly open) term, and τ is a type such that $\Gamma \vDash_M e : \mathsf{ref}\ \tau$ then $\Gamma \vDash_M !\, e : \tau$

Proof: As for theorem 3.24 we must show that under the premises of the theorem for any $k \geq 0$, σ and Ψ such that $\sigma :_{k,\Psi} \Gamma$ we have $\sigma(!\, e) :_{k,\Psi} \tau$. Suppose $\sigma :_{k,\Psi} \Gamma$. By the premise of the theorem we have $\sigma(e) :_{k,\Psi} \mathsf{ref}\ \tau$. The result now follows from Lemma 3.25. □

Theorem 3.27 (Assignment)
Let Γ be a type environment, let e_1 and e_2 be (possibly open) terms and τ be a type. If $\Gamma \vDash_M e_1 : \mathsf{ref}\ \tau$ and $\Gamma \vDash_M e_2 : \tau$ then $\Gamma \vDash_M e_1 := e_2 : \mathsf{unit}$.

Proof: We must show that under the premises of the theorem, for any $k \geq 0$, we have $\Gamma \vDash_M^k e_1 := e_2 : \mathsf{unit}$. More specifically, for any σ and Ψ such that $\sigma :_{k,\Psi} \Gamma$ we must show $\sigma(e_1 := e_2) :_{k,\Psi} \mathsf{unit}$. Let S be a store such that $S :_k \Psi$; then we must show that $(S, \sigma(e_1 := e_2))$ is safe for k steps. From the premise $\Gamma \vDash_M e_1 : \mathsf{ref}\ \tau$ together with $\sigma :_{k,\Psi} \Gamma$, it follows that $\sigma(e_1) :_{k,\Psi} \mathsf{ref}\ \tau$. The latter, by Definition 3.6, implies that $(S, \sigma(e_1))$ is safe for k steps. Suppose, without loss of generality, that $(S, \sigma(e_1))$ reduces to (S, ℓ) in j steps where $j < k$. Then there exists Ψ' such that

1. $(k, \Psi) \sqsubseteq (k-j, \Psi')$
2. $S' :_{k-j} \Psi'$
3. $\langle k-j, \Psi', \ell \rangle \in \mathsf{ref}\ \tau$

Since the codomain of Γ contains types (which are closed under state extension), from $\sigma :_{k,\Psi} \Gamma$ and (1), it follows that $\sigma :_{k-j,\Psi'} \Gamma$. Then, from $\Gamma \vDash_M e_2 : \tau$ we have $\sigma(e_2) :_{k-j,\Psi'} \tau$ from which it follows that $(S', \sigma(e_2))$ is safe for $k-j$ steps. Suppose, without loss of generality, that $(S', \sigma(e_2))$ reduces to (S'', v) (where v is a value) in $i < k-j$ steps. Then there exists a Ψ'' such that

4. $(k-j, \Psi') \sqsubseteq (k-j-i, \Psi'')$
5. $S'' :_{k-j-i} \Psi''$
6. $\langle k-j-i, \Psi'', v\rangle \in \tau$
7. From (3) we have $\ell \in \mathrm{dom}(\Psi')$. From (4) we have $\ell \in \mathrm{dom}(\Psi'')$. Then, from (5) it follows that $\ell \in \mathrm{dom}(S'')$. Therefore, we have $(S, \sigma(e_1 := e_2)) \longmapsto_M^{j+1} (S'', \ell := v) \longmapsto_M (S''[\ell \mapsto v], \mathtt{unit})$ since $\ell \in \mathrm{dom}(S'')$.
8. Let $k' = k-j-i-1$. We also have $(k, \Psi) \sqsubseteq (k-j-i, \Psi'') \sqsubseteq (k', \lfloor \Psi'' \rfloor_{k'})$.
9. By the definition of unit we have $\langle k', \lfloor \Psi'' \rfloor_{k'}, \mathtt{unit}\rangle \in \mathsf{unit}$.
10. It remains for us to show $S''[\ell \mapsto v] :_{k'} \lfloor \Psi'' \rfloor_{k'}$:

    First, from (5) we have $\mathrm{dom}(\Psi'') \subseteq \mathrm{dom}(S'')$. Then, $\mathrm{dom}(\Psi'') \subseteq \mathrm{dom}(S''[\ell \mapsto v])$, and since $\mathrm{dom}(\Psi'') = \mathrm{dom}(\lfloor S'' \rfloor_{k'})$, it follows that $\mathrm{dom}(\lfloor S'' \rfloor_{k'}) \subseteq \mathrm{dom}(S''[\ell \mapsto v])$.

    Second, pick $i' < k'$ and $\ell_1 \in \mathrm{dom}(\lfloor \Psi'' \rfloor_{k'})$. We have to show that $\langle i', \lfloor \lfloor \Psi'' \rfloor_{k'} \rfloor_{i'}, S''[\ell \mapsto v](\ell_1)\rangle \in \lfloor \Psi'' \rfloor_{k'}(\ell_1)$, or equivalently, $\langle i', \lfloor \Psi'' \rfloor_{i'}, S''[\ell \mapsto v](\ell_1)\rangle \in \lfloor \Psi'' \rfloor_{k'}(\ell_1)$.

    - Suppose $\ell_1 = \ell$. Then we must show $\langle i', \lfloor \Psi'' \rfloor_{i'}, v\rangle \in \lfloor \Psi'' \rfloor_{k'}(\ell)$. From (6) and the information-forgetting state extension $(k-j-i, \Psi'') \sqsubseteq (i', \lfloor \Psi'' \rfloor_{i'})$ and type(τ) we have $\langle i', \lfloor \Psi'' \rfloor_{i'}, v\rangle \in \tau$. From (3), by the definition of ref, we have $\lfloor \Psi' \rfloor_{k-j}(\ell) = \lfloor \tau \rfloor_{k-j}$. And from (4) and (8) we have $(k-j, \Psi') \sqsubseteq (k', \lfloor \Psi'' \rfloor_{k'})$. Then, since $\ell \in \mathrm{dom}(\Psi')$ (from (3)), we have $\lfloor \Psi'' \rfloor_{k'}(\ell) = \lfloor \Psi' \rfloor_{k'}(\ell) = \lfloor \tau \rfloor_{k'}$. Since $i' < k'$ we have $\langle i', \lfloor \Psi'' \rfloor_{i'}, v\rangle \in \lfloor \tau \rfloor_{k'}$. Hence, $\langle i', \lfloor \Psi'' \rfloor_{i'}, v\rangle \in \lfloor \Psi'' \rfloor_{k'}(\ell)$ as we wanted to show.
    - Suppose $\ell_1 \neq \ell$. Then we must show $\langle i', \lfloor \Psi'' \rfloor_{i'}, S''(\ell_1)\rangle \in \lfloor \Psi'' \rfloor_{k'}(\ell_1)$. From $S'' :_{k-j-i} \Psi''$ and $\ell_1 \in \mathrm{dom}(\lfloor \Psi'' \rfloor_{k'})$, we have $\langle k', \lfloor \Psi'' \rfloor_{k'}, S''(\ell_1)\rangle \in \lfloor \Psi'' \rfloor_{k-j-i}(\ell_1)$. Then, since $\lfloor \Psi'' \rfloor_{k-j-i}(\ell_1)$ is a type and $(k', \lfloor \Psi'' \rfloor_{k'}) \sqsubseteq (i', \lfloor \Psi'' \rfloor_{i'})$ (information-forgetting state extension since $i' < k'$), it follows that $\langle i', \lfloor \Psi'' \rfloor_{i'}, S''(\ell_1)\rangle \in \lfloor \Psi'' \rfloor_{k-j-i}(\ell_1)$. Since $i' < k'$, we have $\langle i', \lfloor \Psi'' \rfloor_{i'}, S''(\ell_1)\rangle \in \lfloor \Psi'' \rfloor_{k'}(\ell_1)$ as we wanted to show.

Finally we have the following:

- $(S, \sigma(e_1 := e_2)) \longmapsto_M^{j+i+1} (S''[\ell \mapsto v], \mathtt{unit})$ (from (7));

- $(k, \Psi) \sqsubseteq (k', \lfloor\Psi''\rfloor_{k'})$ (from (8), by transitivity of $\sqsubseteq$);
- $S''[\ell \mapsto v] :_{k'} \lfloor\Psi''\rfloor_{k'}$ (from (10));
- $\langle k', \lfloor\Psi''\rfloor_{k'}, \texttt{unit}\rangle \in \textsf{unit}$ (from (9)).

These four statements imply $\sigma(e_1 := e_2) :_{k,\Psi} \textsf{unit}$. □

3.5.3 Type Abstraction and Application

Theorem 3.28 (Type Abstraction)
Let Γ be a type environment and let F be a nonexpansive type functional. If $\Gamma \vDash_M e : F(\tau)$ for any type set τ, then $\Gamma \vDash_M \Lambda.e : \forall F$.

Proof: We must show that, for any $k \geq 0$ and σ and Ψ such that $\sigma :_{k,\Psi} \Gamma$, we have $\sigma(\Lambda.e) :_{k,\Psi} \forall F$. Suppose $\sigma :_{k,\Psi} \Gamma$. Let j, v, Ψ', and τ be such that $(k, \Psi) \sqsubseteq (j, \Psi')$ and type$(\lfloor\tau\rfloor_j)$. By the definition of $\forall F$ it suffices to show that for any $i < j$ we have $\sigma(e) :_{i,\lfloor\Psi'\rfloor_i} F(\tau)$. Since $\lfloor\tau\rfloor_j$ is a type, given the premises of the theorem it follows that $\Gamma \vDash_M e : F(\lfloor\tau\rfloor_j)$. Hence, $\sigma(e) :_{k,\Psi} F(\lfloor\tau\rfloor_j)$. Also, since F is a function from types to types, $F(\lfloor\tau\rfloor_j)$ is a type.

The following information-forgetting state extension holds: $(j, \Psi') \sqsubseteq (i, \lfloor\Psi'\rfloor_i)$. Then by Lemma 3.13 ($\sqsubseteq$ transitive) it follows that $(k, \Psi) \sqsubseteq (i, \lfloor\Psi'\rfloor_i)$. Hence, from $\sigma(e) :_{k,\Psi} F(\lfloor\tau\rfloor_j)$ and type$(F(\lfloor\tau\rfloor_j))$ we can conclude that $\sigma(e) :_{i,\lfloor\Psi'\rfloor_i} F(\lfloor\tau\rfloor_j)$. Now since $i < j$, by definitions 3.2, 3.4, and 3.6 (Approx, Type, and Expr : Type) it follows that $\sigma(e) :_{i,\lfloor\Psi'\rfloor_i} \lfloor F(\lfloor\tau\rfloor_j)\rfloor_j$. Using the premise that F is nonexpansive we have that $\sigma(e) :_{i,\lfloor\Psi'\rfloor_i} \lfloor F(\tau)\rfloor_j$. But since $i < j$, Definition 3.2 (Approx) implies $\sigma(e) :_{i,\lfloor\Psi'\rfloor_i} F(\tau)$ as we needed to show. □

Lemma 3.29 (Closed Type Application)
If e is a closed term, τ is a type, and F is a function from types to types such that $e :_{k,\Psi} \forall F$, then $e[\,] :_{k,\Psi} F(\tau)$.

Proof: We must prove $e[\,] :_{k,\Psi} F(\tau)$ under the premises of the lemma. Since F is a function from types to types, by Lemma 3.16 $\forall F$ is a type. To show $e[\,] :_{k,\Psi} F(\tau)$, let S be a store such that $S :_k \Psi$. From $e :_{k,\Psi} \forall F$ we have that (S, e) is safe for k steps and if (S, e) reduces to (S', v) (where v is a value) in fewer than k steps, then v must be of the form $\Lambda.e'$. Hence, the state $(S, e[\,])$ either reduces for k steps without reaching a state of the form $(S', v[\,])$ or there exist e' and S' such that $(S, e[\,]) \longmapsto^j_M (S', (\Lambda.e')[\,])$ with $j < k$. In the first case we have that $(S, e[\,])$ is safe for k steps and (S, e) does not reduce to a value in fewer than k steps and hence $e[\,] :_{k,\Psi} F(\tau)$. In the second case, it follows from the operational semantics in Figure 3.2 that $(S, e) \longmapsto_M^j (S', \Lambda.e')$. Since $e :_{k,\Psi} \forall F$, by Definition 3.6 (Expr : Type) there exists a store typing Ψ' such that:

1. $(k, \Psi) \sqsubseteq (k - j, \Psi')$;
2. $S' :_{k-j} \Psi'$;
3. $\langle k - j, \Psi', \Lambda.e' \rangle \in \forall F$.

We also have the following:

4. $(k - j, \Psi') \sqsubseteq (k - j, \Psi')$ since state extension is reflexive (Lemma 3.12)
5. $\text{type}(\lfloor \tau \rfloor_{k-j})$ (follows from $\text{type}(\tau)$)

By the definition of $\forall F$, from (3), together with (4), (5), and $k - j - 1 < k - j$, we can conclude that $e' :_{k-j-1, \lfloor \Psi' \rfloor_{k-j-1}} F(\tau)$.
Finally, we have the following:

- $(S, e[\,]) \longmapsto_M^{j+1} (S', e')$;
- $(k, \Psi) \sqsubseteq (k - j - 1, \lfloor \Psi' \rfloor_{k-j-1})$ (valid information-forgetting state extension);
- $S' :_{k-j-1} \lfloor \Psi' \rfloor_{k-j-1}$;
- $e' :_{k-j-1, \lfloor \Psi' \rfloor_{k-j-1}} F(\tau)$ where $F(\tau)$ is a type.

These four statements imply $e[\,] :_{k,\Psi} F(\tau)$. □

Theorem 3.30 (Type Application)
Let Γ be a type environment, let F be a function from types to types, and let τ be a type. If $\Gamma \vDash_M e : \forall F$ then $\Gamma \vDash_M e[\,] : F(\tau)$.

Proof: We must prove that for any $k \geq 0$ we have $\Gamma \vDash_M^k e[\,] : F(\tau)$. More specifically, for any σ and Ψ such that $\sigma :_{k,\Psi} \Gamma$, we must show $\sigma(e[\,]) :_{k,\Psi} F(\tau)$. Suppose $\sigma :_{k,\Psi} \Gamma$. By the premise of the theorem we have $\sigma(e) :_{k,\Psi} \forall F$ and $\text{type}(\tau)$. The result now follows from Lemma 3.29. □

3.5.4 Pack and Unpack

Lemma 3.31 (Closed Pack)
If e is a closed term, τ is type set and F is a function from types to types such that $e :_{k,\Psi} F(\tau)$ then `pack` *$e :_{k,\Psi} \exists F$.*

PROOF: Since τ is type and F is a function from types to types, $F(\tau)$ is a type. To show $\texttt{pack}\, e :_{k,\Psi} \exists F$, suppose $S :_k \Psi$ for some store S. We must show that $(S, \texttt{pack}\, e)$ is safe for k steps. Assume without loss of generality that $(S, \texttt{pack}\, e) \longmapsto_M^j (S', \texttt{pack}\, v)$ where $j < k$. Then, by the operational semantics of λ^M we have $(S, e) \longmapsto_M^j (S', v)$. Hence, from $e :_{k,\Psi} F(\tau)$ we have that there exists Ψ' such that $(k, \Psi) \sqsubseteq (k - j, \Psi')$, $S' :_{k-j} \Psi'$, and $\langle k - j, \Psi', v\rangle \in F(\tau)$.

Next, we must show that $\langle k - j, \Psi', \texttt{pack}\, v\rangle \in \exists F$. This follows (by the definition of $\exists F$) in two steps as follows.

1. From type(τ) we have type($\lfloor \tau \rfloor_{k-j}$).

2. Suppose $i < k - j$. We must show that $\langle i, \lfloor \Psi' \rfloor_i, v\rangle \in F(\tau)$. Note that $(k - j, \Psi') \sqsubseteq (i, \lfloor \Psi' \rfloor_i)$ is a valid information-forgetting state extension (since $i < k - j$). Since $F(\tau)$ is a type and we already have $\langle k - j, \Psi', v\rangle \in F(\tau)$, it follows that $\langle i, \lfloor \Psi' \rfloor_i, v\rangle \in F(\tau)$ as needed.

Now we have the following:

- $(S, \texttt{pack}\, e) \longmapsto_M^j (S', \texttt{pack}\, v)$
- $(k, \Psi) \sqsubseteq (k - j, \Psi')$
- $S' :_{k-j} \Psi'$
- $\langle k - j, \Psi', \texttt{pack}\, v\rangle \in \exists F$

These statements imply $\texttt{pack}\, e :_{k,\Psi} \exists F$. □

Theorem 3.32 (Pack)
Let Γ be a type environment, let τ be a type, and let F be a function from types to types. If e is a (possibly open) term such that $\Gamma \models_M e : F(\tau)$ the $\Gamma \models_M \texttt{pack}\, e : \exists F$.

PROOF: Follows from Lemma 3.31 in the same manner that Theorem 3.30 (Tapp) follows from Lemma 3.29 (Closed Tapp). □

Theorem 3.33 (Unpack)
Let Γ be a type environment and let F be a nonexpansive type functional. If τ_2 is a type and e_1 and e_2 are (possibly open) terms such that $\Gamma \models_M e_1 : \exists F$ and $\Gamma\,[x \mapsto F(\tau)] \models_M e_2 : \tau_2$ for any type τ, then $\Gamma \models_M \texttt{unpack}\, e_1\, \texttt{as}\, x\, \texttt{in}\, e_2 : \tau_2$.

PROOF: We must prove under the premises of the theorem for any $k \geq 0$ and σ and Ψ such that $\sigma :_{k,\Psi} \Gamma$ that $\sigma(\texttt{unpack}\, e_1\, \texttt{as}\, x\, \texttt{in}\, e_2) :_{k,\Psi} \tau_2$. From $\Gamma \models_M e_1 : \exists F$ we have $\sigma(e_1) :_{k,\Psi} \exists F$. Now, since F is a nonexpansive functional, by Lemma 3.19 $\exists F$

is a type. To show $\sigma(\texttt{unpack}\, e_1\, \texttt{as}\, x\, \texttt{in}\, e_2) :_{k,\Psi} \tau_2$, assume that $S :_k \Psi$ for some store S. We must show that $(S, \sigma(\texttt{unpack}\, e_1\, \texttt{as}\, x\, \texttt{in}\, e_2))$ is safe for k steps.

From $\sigma(e_1) :_{k,\Psi} \exists F$ and $S :_k \Psi$ we have that $(S, \sigma(e_1))$ is safe for k steps, and that if $(S, \sigma(e_1))$ generates a value in fewer than k steps that value must be of the form $\texttt{pack}\, v$ where v is a value. Hence, assume without loss of generality that $(S, \sigma(\texttt{unpack}\, e_1\, \texttt{as}\, x\, \texttt{in}\, e_2)) \longmapsto_M^j (S', \sigma(\texttt{unpack}\,(\texttt{pack}\, v)\, \texttt{as}\, x\, \texttt{in}\, e_2))$ for $j < k$ and store S'. Then, by the operational semantics of λ^M (Figure 3.2) we can conclude that $(S, \sigma(e_1)) \longmapsto_M^j (S', \sigma(\texttt{pack}\, v))$ (where $\sigma(\texttt{pack}\, v) = \texttt{pack}\, v$ since value v has no free variables). Since $\sigma(e_1) :_{k,\Psi} \exists F$, by Definition 3.6 (Expr : Type) there exists Ψ' such that:

1. $(k, \Psi) \sqsubseteq (k - j, \Psi')$
2. $S' :_{k-j} \Psi'$
3. $\langle k - j, \Psi', \texttt{pack}\, v\rangle \in \exists F$

From (3), by the definition of $\exists F$, we have that:

4. there exists a set τ such that $\lfloor \tau \rfloor_{k-j}$ is a type, and
5. $\langle k - j - 1, \lfloor \Psi' \rfloor_{k-j-1}, v\rangle \in F(\tau)$.

Since $\lfloor \tau \rfloor_{k-j}$ is a type, by the premises of the theorem we have $\Gamma\,[x \mapsto F(\lfloor \tau \rfloor_{k-j})] \vDash_M e_2 : \tau_2$. To make further use of this statement, we will first have to show that $\langle k - j - 1, \lfloor \Psi' \rfloor_{k-j-1}, v\rangle \in F(\lfloor \tau \rfloor_{k-j})$, which we do as follows:

6. From (5) and Definition 3.2 (Approx) we can conclude that $\langle k - j - 1, \lfloor \Psi' \rfloor_{k-j-1}, v\rangle \in \lfloor F(\tau) \rfloor_{k-j}$ (since $k - j - 1 < k - j$). By the premises of the theorem, F is nonexpansive, which implies that $\lfloor F(\tau) \rfloor_{k-j} = \lfloor F(\lfloor \tau \rfloor_{k-j}) \rfloor_{k-j}$. Hence, we have $\langle k - j - 1, \lfloor \Psi' \rfloor_{k-j-1}, v\rangle \in \lfloor F(\lfloor \tau \rfloor_{k-j}) \rfloor_{k-j}$. But this implies that $\langle k - j - 1, \lfloor \Psi' \rfloor_{k-j-1}, v\rangle \in F(\lfloor \tau \rfloor_{k-j})$ as we wanted to show.

Let $\sigma\,[x \mapsto v]$ be the substitution that is identical to σ except that it maps x to v. Recall that the codomain of Γ contains types (which are closed under state extension) and that we have $\sigma :_{k,\Psi} \Gamma$. Note that $(k-j, \Psi') \sqsubseteq (k-j-1, \lfloor \Psi' \rfloor_{k-j-1})$ is a valid information-forgetting state extension. From the latter and (1), by the transitivity of $\sqsubseteq$ (Lemma 3.13), it follows that that $(k, \Psi) \sqsubseteq (k-j-1, \lfloor \Psi' \rfloor_{k-j-1})$. This, together with (6), allows us to conclude that $\sigma\,[x \mapsto v] :_{k-j-1, \lfloor \Psi' \rfloor_{k-j-1}} \Gamma\,[x \mapsto F(\lfloor \tau \rfloor_{k-j})]$. Hence, from $\Gamma\,[x \mapsto F(\lfloor \tau \rfloor_{k-j})] \vDash_M e_2 : \tau_2$ it follows that $\sigma\,[x \mapsto v](e_2) :_{k-j-1, \lfloor \Psi' \rfloor_{k-j-1}} \tau_2$. But this implies $\sigma(e_2[v/x]) :_{k-j-1, \lfloor \Psi' \rfloor_{k-j-1}} \tau_2$.

Finally we have the following:

- $(S, \sigma(\texttt{unpack}\, e_1\, \texttt{as}\, x\, \texttt{in}\, e_2)) \longmapsto_M^{j+1} (S', \sigma(e_2[v/x]))$;

- $(k, \Psi) \sqsubseteq (k - j - 1, \lfloor \Psi' \rfloor_{k-j-1})$ (by transitivity of $\sqsubseteq$);
- $S' :_{k-j-1} \lfloor \Psi' \rfloor_{k-j-1}$;
- $\sigma(e_2[v/x]) :_{k-j-1, \lfloor \Psi' \rfloor_{k-j-1}} \tau_2$.

Hence, we can conclude that $\sigma(\texttt{unpack}\, e_1\, \texttt{as}\, x\, \texttt{in}\, e_2) :_{k,\Psi} \tau_2$. □

3.6 Summary

In this chapter I described the challenges of modeling general references, that is, mutable references that may store values of any closed type, including function types, other references, and even impredicative polymorphic types. Specifically, I showed how to construct a denotational-operational model of λ^M (i.e., System F + mutable references + existentials) that permits foundational proofs of safety of λ^M programs. The “vanilla” language and model presented in this chapter can be modified and extended in various ways. The next chapter describes some of these extensions and discusses related work.

Chapter 4

Mutable References: Extensions & Discussion

We want to build a foundational proof-carrying code system that compiles programs written in a practical language, like ML or Java, down to machine code that runs on a von Neumann machine, such as the Sparc or Pentium. But real machines do not have instructions for type application or for unpacking values of existential type as the λ^M abstract machine did in the last chapter. Section 4.1 modifies the λ^M operational semantics to address this fact and looks at how such a change affects the semantics of universal and existential types. Also, practical languages require support for recursive types — e.g., ML supports both covariant and contravariant recursive datatypes — as well as product types, union and intersection types, immutable references and so on. In Section 4.2 I will show how to add such types to the model I constructed for λ^M. In the remainder of the chapter I shall describe some examples of how our semantic model allows us to prove the safety of programs that mutate the store, offer some observations, and discuss related work.

4.1 Eliminating Noncomputational Steps

The λ^M operational semantics given in Chapter 3 (see Figure 3.2) has explicit "run-time" steps for unpacking an existential ($\texttt{unpack}\,(\texttt{pack}\,v)\,\texttt{as}\,x\,\texttt{in}\,e$) and applying a universal ($(\Lambda.e)[\,]$). A real machine, however, has no such instructions. This section looks at the ramifications of changing the λ^M operational semantics so that unpacking an existential and applying a universal are treated as *virtual* instructions, or coercions on type annotations, that may be erased (along with the types) just before the program is run — that is, they have no effect on the *number* of steps a program has executed. All of the rules in Figure 3.2 remain unchanged, with the exception of rules `MO-tapp2` and `MO-unpack2`. The latter are replaced by the two rules `MO-tapp2'` and `MO-unpack2'` given below. The *virtual step* or coercion relation $\longmapsto^0_M$ indicates that the number of real steps executed while performing the coercion is zero. Hence, according to our modified operational semantics, applying a universal or unpacking an existential requires zero steps. I shall treat the old unannotated step relation $\longmapsto_M$ as $\longmapsto^1_M$.

$$\frac{}{(S,\ (\Lambda.e)[\,])\longmapsto^0_M (S,\ e)}\ \texttt{(MO-tapp2')}$$

$$\frac{}{(S,\ \texttt{unpack}\,(\texttt{pack}\,v)\,\texttt{as}\,x\,\texttt{in}\,e_2)\longmapsto^0_M (S,\ e_2[v/x])}\ \texttt{(MO-unpack2')}$$

The typing rules for universal and existential types remain unchanged, but it turns out that these rules are no longer provable given our semantic definitions of $\forall F$ and $\exists F$ in Section 3.3.6 (see Figure 3.6). To see why, consider a state of computation where one can safely execute k more steps and one has a value $\texttt{pack}\,v$ such that $\langle k, \Psi, \texttt{pack}\,v\rangle \in \exists F$. If the next "step" taken by the program is to unpack the value $\texttt{pack}\,v$, then according to the new operational semantics the program should still be able to safely execute k more steps. Hence, after unpacking the existential one needs to be able to prove that $\langle k, \Psi, v\rangle \in F(\tau)$ where τ is the witness type. However, the semantics of $\exists F$ (which I've shown below for easy reference) only allows one to conclude that $\langle j, \lfloor\Psi\rfloor_j, v\rangle \in F(\tau)$ for all $j < k$. This, in turn, is due to the fact that the definition requires only the k-approximation of τ to be a valid type (i.e., $\text{type}(\lfloor\tau\rfloor_k)$).

$$\exists F \quad \stackrel{\text{def}}{=} \quad \{\langle k, \Psi, \texttt{pack}\,v\rangle \mid \exists\tau.\ (\text{type}(\lfloor\tau\rfloor_k)\ \wedge\ \forall j < k.\ \langle j, \lfloor\Psi\rfloor_j, v\rangle \in F(\tau))\}$$

In order to conclude that $\langle j, \lfloor\Psi\rfloor_j, v\rangle \in F(\tau)$ for $j \leq k$, one would have to know that the $(k+1)$-approximation of τ is a valid type — that is, one would have to define $\exists F$ as follows:

$$\exists F \quad \stackrel{\text{bad\_def}}{=} \quad \{\langle k, \Psi, \texttt{pack}\,v\rangle \mid \exists\tau.\ (\text{type}(\lfloor\tau\rfloor_{k+1})\ \wedge\ \forall j \leq k.\ \langle j, \lfloor\Psi\rfloor_j, v\rangle \in F(\tau))\}$$

But the above definition *violates the stratification invariant* (see Section 3.3.2) by considering the type τ to approximation $k+1$. It seems, therefore, that we have a conundrum: the current semantics of $\exists F$ makes the typing rule for unpack impossible to prove, while the modified semantics of $\exists F$ leads to an inconsistent model.

The solution to this problem is to require that F be well founded (Section 3.3.9). Intuitively, if F is well founded then one can check that the witness type τ is valid to approximation k (i.e., require $\text{type}(\lfloor\tau\rfloor_k)$), but still be able to decide that v has type $F(\tau)$ to approximation k since the wellfoundedness of F allows us to execute one more step. Recall that type constructors like ref and → are well founded. The reason for their wellfoundedness, intuitively, is the extra dereferencing step for ref and the beta-reduction step for →. Hence, if the function F is well founded then using a value of type $\forall F$ or $\exists F$ will require at least one *real* runtime step. The strategy, then, is to piggyback the coercion onto this real runtime step.

To specify the new semantics of $\forall F$ and $\exists F$, I need to first formalize an approximate notion of wellfoundedness.

Definition 4.1 (Well Founded Upto)
A type functional F is well founded upto k *(written* $\mathrm{wfupto}(k, F)$*) if for any type τ and any j such that $0 \leq j \leq k$ we have*

$$\lfloor F(\tau) \rfloor_{j+1} \;=\; \lfloor F(\lfloor \tau \rfloor_j) \rfloor_{j+1}$$

Note that it follows that F is well founded if and only if $\mathrm{wfupto}(k, F)$ for all $k \geq 0$.

Universal Types The new semantics of $\forall F$ is given below. When deciding if $\langle k, \Psi, \Lambda.e \rangle \in \forall F$, we require that F be well founded upto k — as explained above, this guarantees that there will be a real step that the type application coercion can piggyback on. Now, suppose that $\Lambda.e$ is instantiated with the type τ at some point in the future when there are $j \leq k$ real execution steps left and the store typing is Ψ'. First, one requires that $(k, \Psi) \sqsubseteq (j, \Psi')$ holds. Second, one must make sure that τ is a valid type for $j - 1$ steps, that is, $\mathrm{type}(\lfloor \tau \rfloor_j)$. Now if e has type $F(\tau)$ for j (and fewer) real steps with respect to store typing Ψ', then it must be the case that $\Lambda.e$ has type $\forall F$ for k steps with respect to store typing Ψ.

$$\begin{array}{rcl} \forall F & \stackrel{\mathrm{def}}{=} & \{\langle k, \Psi, \Lambda.e \rangle \mid \mathrm{wfupto}(k, F) \;\wedge \\ & & \qquad (\forall j, \Psi', \tau.\ ((k, \Psi) \sqsubseteq (j, \Psi') \;\wedge\; \mathrm{type}(\lfloor \tau \rfloor_j)) \\ & & \qquad\qquad \Longrightarrow\; e :_{j, \Psi'} F(\tau))\} \end{array}$$

Note that the above definition does not violate the stratification invariant. First, it requires that F be well founded only upto k. Second, it requires that only the j-approximation of τ be a valid type, where $j \leq k$. Finally, it does not require that Ψ be defined beyond $\lfloor \Psi \rfloor_k$.

Existential Types When deciding whether $\langle k, \Psi, \mathtt{pack}\, v \rangle \in \exists F$ one has to check that F is well founded upto k (which guarantees that the unpack coercion will have a real step to piggyback on). In addition, one must check for the existence of a witness type τ such that $\lfloor \tau \rfloor_k$ is a valid type, and make sure that v has type $F(\tau)$ for k steps. The semantics of $\exists F$ is as follows. The definition below adheres to the stratification invariant.

$$\begin{aligned}\exists F \;\stackrel{\text{def}}{=}\; \{\langle k, \Psi, \mathsf{pack}\, v\rangle \mid\ & \text{wfupto}(k, F) \;\wedge \\ & (\exists \tau.\, \text{type}(\lfloor \tau \rfloor_k) \;\wedge\; \langle k, \Psi, v\rangle \in F(\tau))\}\end{aligned}$$

Requiring Well-Founded Functionals in Practice We have seen that when type application and existential unpacking are virtual instructions, types of the form $\forall F$ and $\exists F$ only make sense if F is a well-founded functional.[1] Let us consider how this restriction might affect us in practice in a target language for type-preserving compilation. The identity type function $\lambda\alpha.\alpha$ is nonexpansive but not well founded (as noted in Section 3.3.9), but fortunately, we never need to construct quantified types $\forall F$ and $\exists F$ such that F is the identity type function since the former is equivalent to $\bot$ and the latter to $\top$.

We shall see that the intersection and union (or untagged sum) type constructors are also nonexpansive but not well founded. Hence, for instance, the type function $\lambda\alpha.(\alpha \cup \mathsf{unit})$ is not well founded, but the type function $\lambda\alpha.(F(\alpha) \cup \mathsf{unit})$ *is* well founded as long as F is well founded (and similarly for $\cap$).

In a typed low-level language, universal types are commonly used to describe the type of polymorphic functions. For instance, the polymorphic identity function would have type $\forall F$ where $F = \lambda\alpha.(\alpha \rightarrow \alpha)$. Notice that F is well founded because $\rightarrow$ is well founded. Meanwhile, in a typed low-level language, the most common use of existential types is to describe function closures and objects. For instance, a closure would have type $\exists F$ where $F = \lambda\alpha.(\tau_{\mathsf{code}} \times \alpha)$ (where τ_{code} is the type of the code, α is the type of the closure's environment, and "$\times$" denotes the cartesian product type). Here, F is well founded because the type constructor for cartesian products (see Section 4.2) is well founded (since projection uses up a step). In general, therefore, the quantified types $\forall F$ and $\exists F$ used in the output of a type-preserving compiler are such that F is well founded.

Validity of Types and Typing Rules In the remainder of this section I shall prove that the new $\forall F$ and $\exists F$ are valid types and that the typing rules for type abstraction, type application, pack, and unpack are sound given the new semantics.

Lemma 4.2 (Type $\forall F$)
If F is a function from types to types then $\forall F$ is a type.

Proof: To prove that $\forall F$ is closed under state extension, suppose that $\langle k, \Psi, v\rangle \in \forall F$ and $(k, \Psi) \sqsubseteq (j, \Psi')$. Note that $\langle k, \Psi, v\rangle \in \forall F$ implies that v is of the form $\Lambda.e$. We must show that $\langle j, \Psi', \Lambda.e\rangle \in \forall F$. The proof is in two parts.

[1] Note that one can still construct types $\forall F$ and $\exists F$ where F is nonexpansive, but rather than erasing the corresponding type application and unpack coercions from the code before it runs, one would have to replace each one with a `nop`.

1. We must show wfupto(j, F). From $\langle k, \Psi, \Lambda.e\rangle \in \forall F$ we have wfupto(k, F). Since $j \leq k$ it follows that wfupto(j, F).

2. Suppose $(j, \Psi') \sqsubseteq (i, \Psi'')$ and type$(\lfloor\tau\rfloor_i)$ for some set τ — we must show that $e :_{i,\Psi''} F(\tau)$. Since state extension is transitive (Lemma 3.13), we have $(k, \Psi) \sqsubseteq (i, \Psi'')$. Now, from $\langle k, \Psi, \Lambda.e\rangle \in \forall F$ we may conclude that $e :_{i,\Psi''} F(\tau)$.

□

Lemma 4.3 (Type $\exists F$)
If F is a function from types to types then $\exists F$ is a type.

Proof: To prove that $\exists F$ is closed under memory extension, suppose that $\langle k, \Psi, v\rangle \in \exists F$ and $(k, \Psi) \sqsubseteq (j, \Psi')$. Note that $\langle k, \Psi, v\rangle \in \exists F$ implies that v is of the form $\mathtt{pack}\, v'$ where v' is a value. We must show that $\langle j, \Psi', \mathtt{pack}\, v'\rangle \in \exists F$. The proof has three parts. First, we have to prove wfupto(j, F). From $\langle k, \Psi, \mathtt{pack}\, v'\rangle \in \exists F$ we have wfupto(k, F). Since $j \leq k$, we have wfupto(j, F).

Second, we have to prove the existence of some set τ such that type$(\lfloor\tau\rfloor_j)$. From $\langle k, \Psi, \mathtt{pack}\, v'\rangle \in \exists F$ it follows by the definition of $\exists F$ that there exists some τ such that type$(\lfloor\tau\rfloor_k)$. Since $j \leq k$ it follows by Lemma 3.17 that type$(\lfloor\tau\rfloor_j)$.

For the third part, we must show $\langle j, \Psi', v'\rangle \in F(\tau)$. From $\langle k, \Psi, \mathtt{pack}\, v'\rangle \in \exists F$ we have $\langle k, \Psi, v'\rangle \in F(\tau)$. Since F is a function from types to types and $\lfloor\tau\rfloor_k$ is a type, $F(\lfloor\tau\rfloor_k)$ is a type. It follows that $\lfloor F(\lfloor\tau\rfloor_k)\rfloor_{k+1}$ is a type (by Definition 3.2 (Approx)). Since wfupto(k, F), we have $\lfloor F(\lfloor\tau\rfloor_k)\rfloor_{k+1} = \lfloor F(\tau)\rfloor_{k+1}$. Hence, $\lfloor F(\tau)\rfloor_{k+1}$ is a type. Now, since $\langle k, \Psi, v'\rangle \in F(\tau)$ and $k < k+1$, it follows that $\langle k, \Psi, v'\rangle \in \lfloor F(\tau)\rfloor_{k+1}$. Furthermore, since $(k, \Psi) \sqsubseteq (j, \Psi')$ and type$(\lfloor F(\tau)\rfloor_{k+1})$, we have $\langle j, \Psi', v'\rangle \in \lfloor F(\tau)\rfloor_{k+1}$. But since $j < k+1$ it follows that $\langle j, \Psi', v'\rangle \in F(\tau)$.
□

Theorem 4.4 (Type Abstraction)
Let Γ be a type environment and let F be a well founded type functional. If $\Gamma \vDash_M e : F(\tau)$ for any type set τ, then $\Gamma \vDash_M \Lambda.e : \forall F$.

Proof: We must show that, for any $k \geq 0$ and σ and Ψ such that $\sigma :_{k,\Psi} \Gamma$, we have $\sigma(\Lambda.e) :_{k,\Psi} \forall F$. Suppose $\sigma :_{k,\Psi} \Gamma$. The proof is in two parts. First we must show wfupto(k, F), which follows easily from the fact that F is well founded and $k \geq 0$.

For the second part, let j, v, Ψ', and τ be such that $(k, \Psi) \sqsubseteq (j, \Psi')$ and type$(\lfloor\tau\rfloor_j)$ — we must show that $\sigma(e) :_{j,\Psi'} F(\tau)$. Since $\lfloor\tau\rfloor_j$ is a type, given the premises of the theorem it follows that $\Gamma \vDash_M e : F(\lfloor\tau\rfloor_j)$. Hence, $\sigma(e) :_{k,\Psi} F(\lfloor\tau\rfloor_j)$. Also, since F is a function from types to types, $F(\lfloor\tau\rfloor_j)$ is a type.

From $\text{type}(F(\lfloor\tau\rfloor_j))$, $\sigma(e) :_{k,\Psi} F(\lfloor\tau\rfloor_j)$, and $(k,\Psi) \sqsubseteq (j,\Psi')$ we can conclude that $\sigma(e) :_{j,\Psi'} F(\lfloor\tau\rfloor_j)$. Now since $j < j+1$, by Definitions 3.2, 3.4, and 3.6 (Approx, Type, and Expr : Type) it follows that $\sigma(e) :_{j,\Psi'} \lfloor F(\lfloor\tau\rfloor_j)\rfloor_{j+1}$. Since $\text{wfupto}(k,F)$ and $j \leq k$ we have $\lfloor F(\tau)\rfloor_{j+1} = \lfloor F(\lfloor\tau\rfloor_j)\rfloor_{j+1}$. Hence, $\sigma(e) :_{j,\Psi'} \lfloor F(\tau)\rfloor_{j+1}$. But since $j < j+1$, Definition 3.2 (Approx) implies $\sigma(e) :_{j,\Psi'} F(\tau)$ as we needed to show. □

Lemma 4.5 (Closed Type Application)
If e is a closed term, τ is a type, and F is a function from types to types such that $e :_{k,\Psi} \forall F$, then $e[\,] :_{k,\Psi} F(\tau)$.

Proof: We must prove $e[\,] :_{k,\Psi} F(\tau)$ under the premises of the lemma. Since F is a function from types to types, by Lemma 3.16 $\forall F$ is a type. To show $e[\,] :_{k,\Psi} F(\tau)$, let S be a store such that $S :_k \Psi$. From $e :_{k,\Psi} \forall F$ we have that (S,e) is safe for k steps and if (S,e) reduces to (S',v) (where v is a value) in fewer than k steps, then v must be of the form $\Lambda.e'$. Hence, assume without loss of generality that there exist e' and S' such that $(S,e[\,]) \longmapsto^j_M (S',(\Lambda.e')[\,])$ with $j < k$. It follows from the operational semantics in Figure 3.2 that $(S,e) \longmapsto^j_M (S',\Lambda.e')$. Since $e :_{k,\Psi} \forall F$, by Definition 3.6 (Expr : Type) there exists a store typing Ψ' such that:

1. $(k,\Psi) \sqsubseteq (k-j,\Psi')$;
2. $S' :_{k-j} \Psi'$;
3. $\langle k-j,\Psi',\Lambda.e'\rangle \in \forall F$.

We also have the following:

4. $(k-j,\Psi') \sqsubseteq (k-j,\Psi')$ since state extension is reflexive (Lemma 3.12)
5. $\text{type}(\lfloor\tau\rfloor_{k-j})$ (follows from $\text{type}(\tau)$)

By the definition of $\forall F$, from (3), together with (4) and (5), we can conclude that $e' :_{k-j,\Psi'} F(\tau)$. Finally, we have $(S,e[\,]) \longmapsto^j_M (S',e')$, $(k,\Psi) \sqsubseteq (k-j,\Psi')$, $S' :_{k-j} \Psi'$, and $e' :_{k-j,\Psi'} F(\tau)$ (where $F(\tau)$ is a type). These four statements imply $e[\,] :_{k,\Psi} F(\tau)$. □

Theorem 4.6 (Type Application)
Let Γ be a type environment, let F be a function from types to types, and let τ be a type. If $\Gamma \vDash_M e : \forall F$ then $\Gamma \vDash_M e[\,] : F(\tau)$.

Proof: Follows from Lemma 4.5 in the same manner that Theorem 3.30 (Tapp) follows from Lemma 3.29 (Closed Tapp). □

Lemma 4.7 (Closed Pack)
Let e be a closed term, let τ be a type, and let F be a type functional well founded upto k. If $e :_{k,\Psi} F(\tau)$ then $\texttt{pack}\, e :_{k,\Psi} \exists F$.

Proof: Since τ is type and F is a function from types to types, $F(\tau)$ is a type. To show $\texttt{pack}\, e :_{k,\Psi} \exists F$, suppose $S :_k \Psi$ for some store S. We must show that $(S, \texttt{pack}\, e)$ is safe for k steps. Assume without loss of generality that $(S, \texttt{pack}\, e) \longmapsto^j_M (S', \texttt{pack}\, v)$ where $j < k$. Then, by the operational semantics of λ^M we have $(S, e) \longmapsto^j_M (S', v)$. Hence, from $e :_{k,\Psi} F(\tau)$ we have that there exists Ψ' such that $(k, \Psi) \sqsubseteq (k-j, \Psi')$, $S' :_{k-j} \Psi'$, and $\langle k-j, \Psi', v\rangle \in F(\tau)$.

Next, we must show that $\langle k-j, \Psi', \texttt{pack}\, v\rangle \in \exists F$. Following the definition of $\exists F$ the proof is in three parts. First, from the premise $\text{wfupto}(k, F)$ it follows that $\text{wfupto}(k-j, F)$ since $k-j \leq k$. Second, from $\text{type}(\tau)$ we have $\text{type}(\lfloor\tau\rfloor_{k-j})$. Third, we have $\langle k-j, \Psi', v\rangle \in F(\tau)$ from above.

Hence, we have the following: $(S, \texttt{pack}\, e) \longmapsto^j_M (S', \texttt{pack}\, v)$, $(k, \Psi) \sqsubseteq (k-j, \Psi')$, $S' :_{k-j} \Psi'$, and $\langle k-j, \Psi', \texttt{pack}\, v\rangle \in \exists F$. These four statements imply $\texttt{pack}\, e :_{k,\Psi} \exists F$. □

Theorem 4.8 (Pack)
Let Γ be a type environment, let τ be a type, and let F be a well founded function from types to types. If e is a (possibly open) term such that $\Gamma \models_M e : F(\tau)$ the $\Gamma \models_M \texttt{pack}\, e : \exists F$.

Proof: We must prove that for any $k \geq 0$ we have $\Gamma \models^k_M \texttt{pack}\, e : \exists F$. More specifically, for any σ and Ψ such that $\sigma :_{k,\Psi} \Gamma$, we must show $\sigma(\texttt{pack}\, e) :_{k,\Psi} \exists F$. Suppose $\sigma :_{k,\Psi} \Gamma$. By the premise of the theorem we have $\sigma(e) :_{k,\Psi} F(\tau)$ and $\text{type}(\tau)$. Since F is well founded, it follows that $\text{wfupto}(k, F)$. The result now follows from Lemma 4.7. □

Theorem 4.9 (Unpack)
Let Γ be a type environment and let F be a function from types to types. If τ_2 is a type and e_1 and e_2 are (possibly open) terms such that $\Gamma \models_M e_1 : \exists F$ and $\Gamma\,[x \mapsto F(\tau)] \models_M e_2 : \tau_2$ for any type τ, then $\Gamma \models_M \texttt{unpack}\, e_1\, \texttt{as}\, x\, \texttt{in}\, e_2 : \tau_2$.

Proof: We must prove under the premises of the theorem for any $k \geq 0$ and σ and Ψ such that $\sigma :_{k,\Psi} \Gamma$ that $\sigma(\texttt{unpack}\, e_1\, \texttt{as}\, x\, \texttt{in}\, e_2) :_{k,\Psi} \tau_2$. From $\Gamma \models_M e_1 : \exists F$ we have $\sigma(e_1) :_{k,\Psi} \exists F$. Now, since F is a nonexpansive functional, by Lemma 3.19 $\exists F$ is a type. To show $\sigma(\texttt{unpack}\, e_1\, \texttt{as}\, x\, \texttt{in}\, e_2) :_{k,\Psi} \tau_2$, assume that $S :_k \Psi$ for some store S. We must show that $(S, \sigma(\texttt{unpack}\, e_1\, \texttt{as}\, x\, \texttt{in}\, e_2))$ is safe for k steps.

From $\sigma(e_1) :_{k,\Psi} \exists F$ and $S :_k \Psi$ we have that $(S, \sigma(e_1))$ is safe for k steps, and that if $(S, \sigma(e_1))$ generates a value in fewer than k steps that value must be of the form $\texttt{pack}\, v$ where v is a value. Hence, assume without loss of generality that

$(S, \sigma(\texttt{unpack}\, e_1 \,\texttt{as}\, x \,\texttt{in}\, e_2)) \longmapsto^{j}_{M} (S', \sigma(\texttt{unpack}\,(\texttt{pack}\, v) \,\texttt{as}\, x \,\texttt{in}\, e_2))$ for $j < k$ and store S'. Then, by the operational semantics of λ^M (Figure 3.2) we can conclude that $(S, \sigma(e_1)) \longmapsto^{j}_{M} (S', \sigma(\texttt{pack}\, v))$ (where $\sigma(\texttt{pack}\, v) = \texttt{pack}\, v$ since value v has no free variables). Since $\sigma(e_1) :_{k,\Psi} \exists F$, by Definition 3.6 (Expr : Type) there exists Ψ' such that:

1. $(k, \Psi) \sqsubseteq (k - j, \Psi')$
2. $S' :_{k-j} \Psi'$
3. $\langle k - j, \Psi', \texttt{pack}\, v\rangle \in \exists F$

From (3), by the definition of $\exists F$, we have that:

4. there exists a set τ such that $\lfloor \tau \rfloor_{k-j}$ is a type, and
5. $\langle k - j - 1, \lfloor \Psi' \rfloor_{k-j-1}, v\rangle \in F(\tau)$.

Since $\lfloor \tau \rfloor_{k-j}$ is a type, by the premises of the theorem we have $\Gamma\,[x \mapsto F(\lfloor \tau \rfloor_{k-j})] \models_M e_2 : \tau_2$. To make further use of this statement, we will first have to show that $\langle k - j - 1, \lfloor \Psi' \rfloor_{k-j-1}, v\rangle \in F(\lfloor \tau \rfloor_{k-j})$, which we do as follows:

6. From (5) and Definition 3.2 (Approx) we can conclude that $\langle k - j - 1, \lfloor \Psi' \rfloor_{k-j-1}, v\rangle \in \lfloor F(\tau) \rfloor_{k-j}$ (since $k - j - 1 < k - j$). By the premises of the theorem, F is nonexpansive, which implies that $\lfloor F(\tau) \rfloor_{k-j} = \lfloor F(\lfloor \tau \rfloor_{k-j}) \rfloor_{k-j}$. Hence, we have $\langle k - j - 1, \lfloor \Psi' \rfloor_{k-j-1}, v\rangle \in \lfloor F(\lfloor \tau \rfloor_{k-j}) \rfloor_{k-j}$. But this implies that $\langle k - j - 1, \lfloor \Psi' \rfloor_{k-j-1}, v\rangle \in F(\lfloor \tau \rfloor_{k-j})$ as we wanted to show.

Let $\sigma\,[x \mapsto v]$ be the substitution that is identical to σ except that it maps x to v. Recall that the codomain of Γ contains types (which are closed under state extension) and that we have $\sigma :_{k,\Psi} \Gamma$. Note that $(k{-}j, \Psi') \sqsubseteq (k{-}j{-}1, \lfloor \Psi' \rfloor_{k-j-1})$ is a valid information-forgetting state extension. From the latter and (1), by the transitivity of $\sqsubseteq$ (Lemma 3.13), it follows that that $(k, \Psi) \sqsubseteq (k{-}j{-}1, \lfloor \Psi' \rfloor_{k-j-1})$. This, together with (6), allows us to conclude that $\sigma\,[x \mapsto v] :_{k-j-1, \lfloor \Psi' \rfloor_{k-j-1}} \Gamma\,[x \mapsto F(\lfloor \tau \rfloor_{k-j})]$. Hence, from $\Gamma\,[x \mapsto F(\lfloor \tau \rfloor_{k-j})] \models_M e_2 : \tau_2$ it follows that $\sigma\,[x \mapsto v](e_2) :_{k-j-1, \lfloor \Psi' \rfloor_{k-j-1}} \tau_2$. But this implies $\sigma(e_2[v/x]) :_{k-j-1, \lfloor \Psi' \rfloor_{k-j-1}} \tau_2$.

Finally we have the following:

- $(S, \sigma(\texttt{unpack}\, e_1 \,\texttt{as}\, x \,\texttt{in}\, e_2)) \longmapsto^{j+1}_{M} (S', \sigma(e_2[v/x]))$;
- $(k, \Psi) \sqsubseteq (k - j - 1, \lfloor \Psi' \rfloor_{k-j-1})$ (by transitivity of $\sqsubseteq$);
- $S' :_{k-j-1} \lfloor \Psi' \rfloor_{k-j-1}$;
- $\sigma(e_2[v/x]) :_{k-j-1, \lfloor \Psi' \rfloor_{k-j-1}} \tau_2$.

Hence, we can conclude that $\sigma(\texttt{unpack}\, e_1 \,\texttt{as}\, x \,\texttt{in}\, e_2) :_{k,\Psi} \tau_2$. □

4.2 Modeling Additional Types

This section looks at how to extend λ^M with recursive types, union and intersection types, cartesian products, and immutable references. These are all features one needs in a target language for type-preserving compilation of ML or Java. Section 4.3 presents examples that make use of some of these types.

4.2.1 Recursive Types

I extend λ^M with recursive types of the form μF, where F is a function from types to types. The indexed model I presented in Chapter 3 can accommodate recursive types without any changes to the underlying semantics. This is not surprising given the fact that recursive types were the main motivation behind Appel and McAllester's indexed model [AM01]. The material in this section is based on that work.

I will show how to model equi-recursive types (μF is *equal* to $F(\mu F)$) — as opposed to iso-recursive types which require explicit fold and unfold terms (μF isomorphic to $F(\mu F)$ via fold/unfold). Because equi-recursive types do not require explicit fold and unfold terms, the syntax of λ^M terms remains exactly as shown in Figure 3.1. The semantics of μF is given below — F is a function from types to types and the notation $F^k(\tau)$ denotes k applications of F.

$$\mu F \quad \stackrel{\text{def}}{=} \quad \{\langle k, \Psi, v\rangle \mid \langle k, \Psi, v\rangle \in F^{k+1}(\bot)\}$$

Note that if F is a function from types to types and τ is a type then it follows that $F^k(\tau)$ is a type for any $k \geq 0$. I extend the set of λ^M typing rules shown in Figure 3.7 with the following two rules for recursive types.

$$\frac{\Gamma \vDash_M e : \mu F}{\Gamma \vDash_M e : F(\mu F)}\ \texttt{(M-unfold)} \qquad\qquad \frac{\Gamma \vDash_M e : F(\mu F)}{\Gamma \vDash_M e : \mu F}\ \texttt{(M-fold)}$$

In order to prove the validity of recursive types μF I need the following lemma about well-founded functionals (which I defined in Section 3.3.9, see Definition 3.10). The lemmas says that the type sets produced by j or more applications of a well-founded functional to *any* type are identical to approximation j.

Lemma 4.10
For F well founded and $j \leq k$, for any τ, τ_1, τ_2,

$$\begin{array}{ll} (1) & \lfloor F^j(\tau_1)\rfloor_j = \lfloor F^j(\tau_2)\rfloor_j \\ (2) & \lfloor F^j(\tau)\rfloor_j = \lfloor F^k(\tau)\rfloor_j \end{array}$$

Proof: (1) By induction.

$$\begin{array}{l} \lfloor F^j(\tau_1)\rfloor_0 = \bot = \lfloor F^j(\tau_2)\rfloor_0. \\ \lfloor F^{j+1}(\tau_1)\rfloor_{j+1} = \\ \lfloor F(F^j(\tau_1))\rfloor_{j+1} = \\ \lfloor F(\lfloor F^j(\tau_1)\rfloor_j)\rfloor_{j+1} = \\ \lfloor F(\lfloor F^j(\tau_2)\rfloor_j)\rfloor_{j+1} = \\ \lfloor F(F^j(\tau_2))\rfloor_{j+1} = \\ \lfloor F^{j+1}(\tau_2)\rfloor_{j+1}. \end{array}$$

(2) Using (1), taking $\tau_2 = F^{k-j}(\tau_1)$. □

Now I can show that if F is a well-founded functional then μF is closed under state extension.

Lemma 4.11 (Type μF)
If F is well founded, then μF is a type.

Proof: We must show that μF is closed under valid state extension. Suppose that $\langle k, \Psi, v\rangle \in \mu F$ and that $(k, \Psi) \sqsubseteq (j, \Psi')$.

$$\begin{array}{ll} \langle k, \Psi, v\rangle \in \mu F & \\ \langle k, \Psi, v\rangle \in F^{k+1}(\bot) & \text{by definition of } \mu F \\ \langle j, \Psi', v\rangle \in F^{k+1}(\bot) & \text{by Definition 3.4 (Type)} \\ \langle j, \Psi', v\rangle \in \lfloor F^{k+1}(\bot)\rfloor_{j+1} & \text{by Definition 3.2 (Approx)} \\ \langle j, \Psi', v\rangle \in \lfloor F^{j+1}(\bot)\rfloor_{j+1} & \text{by Lemma 4.10} \\ \langle j, \Psi', v\rangle \in F^{j+1}(\bot) & \text{by Definition 3.2 (Approx)} \\ \langle j, \Psi', v\rangle \in \mu F & \text{by definition of } \mu F \end{array}$$

□

Before I can prove the typing rules, I need a few auxiliary lemmas.

Lemma 4.12
$\lfloor\lfloor\tau\rfloor_{k+1}\rfloor_k \;=\; \lfloor\tau\rfloor_k.$

Lemma 4.13
If F is well founded,

(a) $\lfloor\mu F\rfloor_k \;=\; \lfloor F^k\bot\rfloor_k$

(b) $\lfloor F(\mu F)\rfloor_{k+1} \;=\; \lfloor F^{k+1}\bot\rfloor_{k+1}$

Proof: (a) For $k = 0$, each side is equivalent to $\bot$. For $k > 0$, each of the following lines is equivalent:

$$\begin{array}{ll}
\langle j, \Psi, v\rangle \in \lfloor \mu F \rfloor_k & \\
j < k \wedge \langle j, \Psi, v\rangle \in \mu F & \text{by Definition 3.2 (Approx)} \\
j < k \wedge \langle j, \Psi, v\rangle \in F^{j+1}\bot & \text{by definition of } \mu F \\
j < k \wedge \langle j, \Psi, v\rangle \in \lfloor F^{j+1}\bot \rfloor_{j+1} & \text{by Definition 3.2 (Approx)} \\
j < k \wedge \langle j, \Psi, v\rangle \in \lfloor F^{k}\bot \rfloor_{j+1} & \text{by Lemma 4.10} \\
j < k \wedge \langle j, \Psi, v\rangle \in F^{k}\bot & \text{by Definition 3.2 (Approx)} \\
\langle j, \Psi, v\rangle \in \lfloor F^{k}\bot \rfloor_{k} & \text{by Definition 3.2 (Approx)}
\end{array}$$

(b) Each of the following sets is equivalent.

$$\begin{array}{ll}
\lfloor F^{k+1}\bot \rfloor_{k+1} & \\
\lfloor F(F^{k}\bot) \rfloor_{k+1} & \\
\lfloor F(\lfloor F^{k}\bot \rfloor_{k}) \rfloor_{k+1} & \text{by well-foundedness of } F \\
\lfloor F(\lfloor \mu F \rfloor_{k}) \rfloor_{k+1} & \text{by } (a) \\
\lfloor F(\mu F) \rfloor_{k+1} & \text{by well-foundedness of } F
\end{array}$$ □

Lemma 4.14
If F is well founded, $\lfloor \mu F \rfloor_k = \lfloor F(\mu F) \rfloor_k$.

Proof: Each of the following sets is equivalent.

$$\begin{array}{ll}
\lfloor \mu F \rfloor_{k} & \\
\lfloor F^{k}\bot \rfloor_{k} & \text{by Lemma 4.13}(a) \\
\lfloor F^{k+1}\bot \rfloor_{k} & \text{by Lemma 4.10} \\
\lfloor \lfloor F^{k+1}\bot \rfloor_{k+1} \rfloor_{k} & \text{by Lemma 4.12} \\
\lfloor \lfloor F(\mu F) \rfloor_{k+1} \rfloor_{k} & \text{by Lemma 4.13}(b) \\
\lfloor F(\mu F) \rfloor_{k} & \text{by Lemma 4.12}
\end{array}$$ □

The lemmas for typing rules `M-unfold` and `M-fold` can now easily be proved.

Theorem 4.15 (Unfold and Fold)
If F is well founded, then $\mu F = F(\mu F)$. Hence, the typing rules for μF (`M-unfold` and `M-fold`) hold for any well-founded functional F.

Proof: We have that $\langle k, \Psi, v\rangle \in \mu F$ iff $\langle k, \Psi, v\rangle \in \lfloor \mu F \rfloor_{k+1}$ iff $\langle k, \Psi, v\rangle \in \lfloor F(\mu F) \rfloor_{k+1}$ iff $\langle k, \Psi, v\rangle \in F(\mu F)$. □

4.2.2 Union, Intersection, and Product Types

Union (or untagged sum) types of the form $\tau_1 \cup \tau_2$ and intersection types $\tau_1 \cap \tau_2$ may be added to λ^M as follows. A value v has type $\tau_1 \cup \tau_2$ for k steps with respect to store typing Ψ if v has type τ_1 *or* if v has type τ_2 for k steps with respect to Ψ. Note that the "or" in the preceding sentence should not be interpreted as "exclusive or" — I am modeling non-disjoint union types here. Analogously, a value v has type $\tau_1 \cap \tau_2$ for k steps with respect to store typing Ψ if v has both type τ_1 *and* type τ_2 for k steps with respect to Ψ.

$$\begin{array}{lcl} \tau_1 \cup \tau_2 & \overset{\text{def}}{=} & \{ \langle k, \Psi, v \rangle \mid \langle k, \Psi, v \rangle \in \tau_1 \ \vee \ \langle k, \Psi, v \rangle \in \tau_2 \} \\ \tau_1 \cap \tau_2 & \overset{\text{def}}{=} & \{ \langle k, \Psi, v \rangle \mid \langle k, \Psi, v \rangle \in \tau_1 \ \wedge \ \langle k, \Psi, v \rangle \in \tau_2 \} \end{array}$$

Notice that both union and intersection types are nonexpansive but not well founded.

To add product types to λ^M, I extend the λ^M syntax shown in Figure 3.1 as follows.

$$\begin{array}{llll} \textit{Expressions} & e & ::= & \ldots \mid \langle e_1, e_2 \rangle \mid \pi_1(e) \mid \pi_2(e) \\ \textit{Values} & v & ::= & \ldots \mid \langle v_1, v_2 \rangle \end{array}$$

A value $\langle v_1, v_2 \rangle$ has type $\tau_1 \times \tau_2$ for k steps with respect to store typing Ψ if v_1 and v_2 each have types τ_1 and τ_2 (respectively) for upto $k-1$ steps with respect to the appropriate approximation of Ψ.

$$\tau_1 \times \tau_2 \ \overset{\text{def}}{=} \ \{ \langle k, \Psi, \langle v_1, v_2 \rangle \rangle \mid \forall j < k. \ \langle j, \lfloor \Psi \rfloor_j, v_1 \rangle \in \tau_1 \ \wedge \ \langle j, \lfloor \Psi \rfloor_j, v_2 \rangle \in \tau_2 \}$$

The typing rules for union, intersection, and product types are given in Figure 4.1. The validity of the types defined in this section is easy to prove, as is the validity of the typing rules shown in Figure 4.1. I shall omit the proofs here.

4.2.3 Singleton Types and Immutable References

A value v has singleton type $\mathsf{S}(v)$ (which identifies it exactly) for any number of steps k and with respect to any store typing Ψ. The semantics of singleton types is given below. (Notice the difference in fonts: S denotes a singleton type while S denotes a store.)

$$\mathsf{S}(v) \ \overset{\text{def}}{=} \ \{ \langle k, \Psi, v \rangle \}$$

Next, I extend λ^M with immutable reference types. The model of λ^I presented in Section 2.2 supported immutable references by disallowing updates. The model of λ^M, however, permits approximately type-preserving updates. Hence, to add immutable references to λ^M I use a trick involving singleton types.

The first step is to extend the λ^M syntax with terms of the form `inew`(e) which allocates an immutable reference cell initialized to the result of evaluating e. In

$$\frac{\Gamma \models_M e : \tau_1}{\Gamma \models_M e : \tau_1 \cup \tau_2}\ \texttt{(M-}\cup\texttt{i1)} \qquad \frac{\Gamma \models_M e : \tau_2}{\Gamma \models_M e : \tau_1 \cup \tau_2}\ \texttt{(M-}\cup\texttt{i2)}$$

$$\frac{\Gamma \models_M e : \tau_1 \cup \tau_2 \qquad \Gamma\,[x \mapsto \tau_1] \models_M e' : \tau \qquad \Gamma\,[x \mapsto \tau_2] \models_M e' : \tau}{\Gamma \models_M e'[e/x] : \tau}\ \texttt{(M-}\cup\texttt{e)}$$

$$\frac{\Gamma \models_M e : \tau_1 \quad \Gamma \models_M e : \tau_2}{\Gamma \models_M e : \tau_1 \cap \tau_2}\ \texttt{(M-}\cap\texttt{i)}$$

$$\frac{\Gamma \models_M e : \tau_1 \cap \tau_2}{\Gamma \models_M e : \tau_1}\ \texttt{(M-}\cap\texttt{e1)} \qquad \frac{\Gamma \models_M e : \tau_1 \cap \tau_2}{\Gamma \models_M e : \tau_2}\ \texttt{(M-}\cap\texttt{e2)}$$

$$\frac{\Gamma \models_M e_1 : \tau_1 \quad \Gamma \models_M e_2 : \tau_2}{\Gamma \models_M \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}\ \texttt{(M-prod)}$$

$$\frac{\Gamma \models_M e : \tau_1 \times \tau_2}{\Gamma \models_M \pi_1(e) : \tau_1}\ \texttt{(M-fst)} \qquad \frac{\Gamma \models_M e : \tau_1 \times \tau_2}{\Gamma \models_M \pi_2(e) : \tau_2}\ \texttt{(M-snd)}$$

Figure 4.1: Union, Intersection, and Product Typing Rules

terms of the operational semantics, `inew`(e) is identical to `new`(e). However, they are handled quite differently by the underlying semantic model. Recall the handling of `new`: upon allocation of a mutable cell ℓ of type $\mathsf{ref}\,\tau$, the λ^M semantics requires that store typing Ψ be extended with $\ell \mapsto \lfloor \tau \rfloor_k$ (where k is the number of steps left to execute). Meanwhile, the instruction `inew` is handled as follows: upon allocation of an immutable cell ℓ of type $\mathsf{box}\,\tau$, initialized to the value v (where v must have type τ to approximation k), the λ^M semantics requires that the store typing Ψ be extended with $\ell \mapsto \lfloor \tau \cap \mathsf{S}(v) \rfloor_k$. Intuitively, the contents of cell ℓ must have the type τ as well as the type $\mathsf{S}(v)$ for the next k steps. This means that the program is free to update the contents of cell ℓ but only with the exact value written into the cell upon allocation. Hence, as required for an immutable reference cell, the contents of ℓ remain unchanged. The semantics of immutable reference types is as follows.

$$\mathsf{box}\,\tau \ \stackrel{\text{def}}{=}\ \{\, \langle k, \Psi, \ell \rangle \mid \exists v.\ \lfloor \Psi \rfloor_k = \lfloor \tau \cap \mathsf{S}(v) \rfloor_k \,\}$$

Notice that the above definition requires only that there *exist some value* v such that Ψ (approximately) maps ℓ to $\tau \cap \mathsf{S}(v)$. As I informally explained above, that value v should be equal to the contents of cell ℓ in the current store S. However, the definition of $\mathsf{box}\ \tau$ cannot state this requirement because types do not have access to the current store S — they only have access to the current store typing Ψ. Fortunately, the requirement is made explicit elsewhere in the model. Specifically, given a current store S that is approximately well-typed with respect to store typing Ψ (see Definition 3.5 in Chapter 3, Section 3.3.5), it follows that $S(\ell)$ must be equal to v.[2]

Finally, I extend the set of λ^M typing rules in Figure 3.7 with the following typing rules for box types. It is relatively straightforward to prove that $\mathsf{box}\ \tau$ is a type and that the typing rules shown below are sound.

$$\frac{\Gamma \vDash_M e : \tau}{\Gamma \vDash_M \texttt{inew}(e) : \mathsf{box}\ \tau}\ \texttt{(M-inew)} \qquad \frac{\Gamma \vDash_M e : \mathsf{box}\ \tau}{\Gamma \vDash_M \,!\,e : \tau}\ \texttt{(M-ideref)}$$

One would expect box and ref to behave as immutable and mutable references (respectively) are supposed to — that is, box should be covariant and ref should be neither covariant nor contravariant. These properties can be proved as lemmas.

4.3 Examples

In this section I shall present two program fragments and show how they can be proved safe using the indexed model of λ^M. I shall assume that the compiler supplies the typing derivation for each program. In each example, I shall assume that the program (or program fragment) is well-typed with respect to some type environment Γ_0, and that the expression e_{rest} (which denotes the rest of the program) is well-typed with respect to some type environment Γ_{rest}, that is, $\Gamma_{rest} \vDash_M e_{rest} : \tau'$ for some type τ'.

4.3.1 Allocation, Update, and Aliases

I begin with a rather simple example. The program fragment shown below allocates a new reference cell and initializes it with a pointer to an existing cell (pointed to by variable x). Hence, x and $!\,y$ alias the same location — let us call that location

[2] In our foundational PCC implementation, types are (roughly) sets of tuples of the form $\langle k, \Psi, S, v\rangle$ — that is, types have access to the store. Hence, the semantics of immutable reference types in our implementation is closer to the following:

$$\mathsf{box}\ \tau \ \stackrel{\text{def}}{=}\ \{\, \langle k, \Psi, S, \ell\,\rangle \mid \exists v.\ \lfloor\Psi\rfloor_k = \lfloor \mathsf{S}(v)\rfloor_k \ \wedge\ \forall j < k.\ \langle j, \lfloor\Psi\rfloor_j, S, S(\ell)\rangle \in \lfloor\tau\rfloor_k \,\}$$

$\{\, \exists\Psi.\ x :_{k,\Psi} \mathsf{ref}\ \tau \ \wedge\ z :_{k,\Psi} \tau \ \wedge S :_k \Psi \,\}$

let y = new(x) in

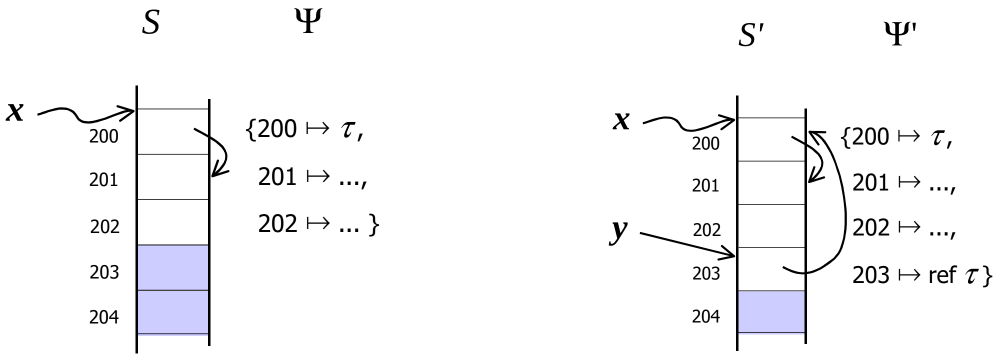
$\{\, \exists\Psi'.\ x :_{k-1,\Psi'} \mathsf{ref}\ \tau \ \wedge\ z :_{k-1,\Psi'} \tau \ \wedge\ y :_{k-1,\Psi'} \mathsf{ref}(\mathsf{ref}\ \tau)$
$\qquad \wedge\ (k, \Psi) \sqsubseteq (k-1, \Psi') \ \wedge\ S' :_{k-1} \Psi' \,\}$

let _ = $!\,y := z$ in

$\{\, \exists\Psi''.\ x :_{k-3,\Psi''} \mathsf{ref}\ \tau \ \wedge\ z :_{k-3,\Psi''} \tau \ \wedge\ y :_{k-3,\Psi''} \mathsf{ref}(\mathsf{ref}\ \tau)$
$\qquad \wedge\ (k-1, \Psi') \sqsubseteq (k-3, \Psi'') \ \wedge\ S'' :_{k-3} \Psi'' \,\}$

e_{rest}

(a) Program fragment



(b) Store S, where $S :_k \Psi$

(c) Store S', where $S' :_{k-1} \Psi'$

Figure 4.2: Example 1: Allocation & Update in the Presence of Aliasing

ℓ. The program then updates the contents of ℓ via the alias $!\,y$.

% $\Gamma_0 = \{\, x : \mathsf{ref}\ \tau,\ z : \tau \,\}$
  let y = new(x) in
% $\Gamma_1 = \{\, x : \mathsf{ref}\ \tau,\ z : \tau,\ y : \mathsf{ref}(\mathsf{ref}\ \tau) \,\}$
  let _ = $!\,y := z$ in
% $\Gamma_{rest} = \{\, x : \mathsf{ref}\ \tau,\ z : \tau,\ y : \mathsf{ref}(\mathsf{ref}\ \tau) \,\}$
  e_{rest}

The assignment to $!y$ is safe because it is approximately type-preserving: $!y$ has type $\mathsf{ref}\ \tau$ and the program updates it with a value of type τ. The assignment has no effect on the type of x which remains unchanged. Proving the program safe amounts to proving each of the Hoare triples in the program fragment shown in Figure 4.2(a) for all $k \geq 0$.[3] In the program fragment (Figure 4.2(a)), S is any store that satisfies some initial store typing Ψ such that x and z have the types $\mathsf{ref}\ \tau$ and τ (to approximation k) with respect to Ψ. Stores S' and S'' denote the stores obtained by evaluating, respectively, $\mathtt{new}(x)$ in store S and $!y := z$ in S'. Figures 4.2(b) and 4.2(c) illustrate the stores S and S' which are approximately well-typed with respect to store typings Ψ and Ψ' respectively; unallocated store cells are shaded. Using the indexed model of λ^M that I have presented, one can prove the validity of each of the Hoare triples in Figure 4.2(a). The program is safe for any number of steps because the proof holds for all $k \geq 0$.

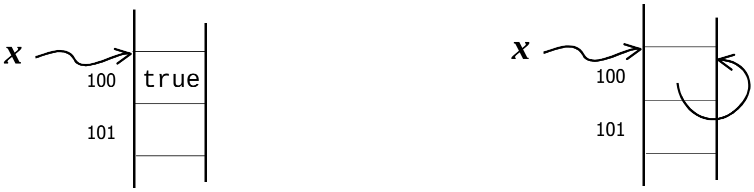
4.3.2 Cycles in Memory

The λ^M model can handle cycles in the memory graph as I shall now illustrate. Consider the following program which creates a reference cell initialized to `true`, binds the new cell to variable x, and then, creates a cycle in memory by assigning x to itself. Figure 4.3 illustrates the store before and after the assignment $x := x$.

```
% $\Gamma_0 = \ldots$
  let $x$ = new(true) in
% $\Gamma_1 = \ldots$
  let _ = $x := x$ in
% $\Gamma_{rest} = \ldots$
  $e_{rest}$
```

What should Γ_0, Γ_1, and Γ_{rest} be? From the code above, it looks like Γ_1 should map x to $\mathsf{ref\ bool}$; but in that case, the assignment $x := x$ will not type check since it assigns a value of type $\mathsf{ref\ bool}$ to a reference cell that should contain a value of type bool. To show that the above program safely creates a cyclic data structure, one *either* need recursive types *or* impredicative quantified types. I will first describe a solution that makes use of recursive types (which I introduced in Section 4.2.1) and then one that uses impredicative existential types.

[3] The approximation indices used in each of the invariants in Figure 4.2(a) indicate the number of steps left to execute at each point. The last invariant uses approximation index $k - 3$ rather than $k - 2$ because the preceding instruction uses up one step to dereference y and another step to assign z to the result of the dereference.



(a) Before: $S :_k \Psi$

(b) After: $S' :_{k-1} \Psi'$

Figure 4.3: Example 2: Creating a Cycle in the Store

Cycles using Recursive Types Suppose that the compiler provides us with a typing derivation such that Γ_0 is the empty type environment and $\Gamma_1 = \Gamma_{rest} = \{\, x : \mathsf{ref}(\mu F)\,\}$, where F denotes the type function $\lambda\alpha.\,(\mathsf{bool} \cup \mathsf{ref}\ \alpha)$. (Notice that F is well founded by Lemma 3.11, intuitively, since the only occurrence of the type variable α is composed with the well-founded type constructor ref.) I show parts of the typing derivation below. The following subtree of the derivation establishes that that $\texttt{new}(\texttt{true})$ has type $\mathsf{ref}(\mu F)$, where F denotes $\lambda\alpha.\,(\mathsf{bool} \cup \mathsf{ref}\ \alpha)$. Hence, we pick $\Gamma_1 = \{\, \mathsf{ref}(\mu F)\,\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset \vDash_M \texttt{true} : \mathsf{bool}}\ (\texttt{M-true})}{\emptyset \vDash_M \texttt{true} : (\mathsf{bool} \cup \mathsf{ref}\ (\mu F)) \;=\; F(\mu F)}\ (\texttt{M-}\cup\texttt{i1})}{\emptyset \vDash_M \texttt{true} : \mu F}\ (\texttt{M-fold})}{\emptyset \vDash_M \texttt{new}(\texttt{true}) : \mathsf{ref}(\mu F)}\ (\texttt{M-new})$$

The following sub-derivation shows that $x := x$ is well-typed with respect to type environment $\Gamma_1 = \{\, x : \mathsf{ref}(\mu F)\,\}$. Note that since $F = \lambda\alpha.\,(\mathsf{bool} \cup \mathsf{ref}\ \alpha)$, we have $F(\mu F) = \mathsf{bool} \cup \mathsf{ref}(\mu F)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_1 \vDash_M x : \mathsf{ref}(\mu F)}\ (\texttt{M-var})}{\Gamma_1 \vDash_M x : \mathsf{ref}(F(\mu F))}\ (\mu F = F(\mu F))}{\Gamma_1 \vDash_M x : \mathsf{ref}(\mathsf{bool} \cup \mathsf{ref}(\mu F))} \qquad \dfrac{\dfrac{}{\Gamma_1 \vDash_M x : \mathsf{ref}(\mu F)}\ (\texttt{M-var})}{\Gamma_1 \vDash_M x : \mathsf{bool} \cup \mathsf{ref}(\mu F)}\ (\texttt{M-}\cup\texttt{i2})}{\Gamma_1 \vDash_M x := x : \mathsf{unit}}\ (\texttt{M-assign})$$

Given the typing derivation described above, to prove that the program is safe amounts to proving each of the Hoare triples in the program fragment given below. The store S_0 is the initial store, while S and S' denote the stores obtained by evaluating, respectively, $x = \texttt{new}(\texttt{true})$ in store S_0 and $x := x$ in store S. Stores S

and S' are shown in Figure 4.3.

```
{ $\exists\Psi_0.\ S_0 :_k \Psi_0$ }
let $x$ = new(true) in
{ $\exists\Psi.\ x :_{k-1,\Psi} \mathsf{ref}(\mu F)\ \wedge\ (k, \Psi_0) \sqsubseteq (k-1, \Psi)\ \wedge\ S :_{k-1} \Psi$ }
let _ = $x := x$ in
{ $\exists\Psi'.\ x :_{k-2,\Psi'} \mathsf{ref}(\mu F)\ \wedge\ (k-1, \Psi) \sqsubseteq (k-2, \Psi')\ \wedge\ S' :_{k-2} \Psi'$ }
$e_{rest}$
```

To prove the above Hoare triples, assume S_0 is an arbitrary initial store and that the address of the new cell is some location $\ell \notin \mathrm{dom}(S_0)$ — in Figure 4.3, $\ell = 100$. Then $S = S_0\,[\ell \mapsto \mathtt{true}]$ and $S' = S\,[\ell \mapsto \ell]$. Let Ψ_0 be the empty store typing. Pick Ψ such that $\lfloor\Psi\rfloor_{k-1} = \{\,\ell \mapsto \lfloor\mu F\rfloor_{k-1}\,\}$ and let $\Psi' = \lfloor\Psi\rfloor_{k-2}$.

Cycles using Impredicative Existential Types We can handle cycles in the store even without recursive types because λ^M supports impredicative quantified types. This explains why the semantic model of λ^M presented in Chapter 3 did not need to be altered or strengthened in any way in order to accommodate recursive types in Section 4.2.1.

In the absence of recursive types, the compiler generates a typing derivation such that Γ_0 is the empty type environment and $\Gamma_1 = \Gamma_{rest} = \{\,x : \exists F\,\}$, where the type function $\lambda\alpha.\,\mathsf{ref}(\mathsf{bool} \cup \alpha)$. (Notice that F is well founded by Lemma 3.11, informally, since the nonexpansive type function $\mathsf{bool} \cup \alpha$ is composed with the well-founded type constructor ref.) The compiler also inserts `pack` and `unpack` coercions into the original program to facilitate type checking. All coercions are erased prior to execution, that is, once we have proved the program safe. The modified program is as follows.

```
  let $x$ = pack (new(true)) in
% $\Gamma = \{\,x : \exists F\,\}$
  let _ = (unpack $x$ as $z$ in $z := x$) in
% $\Gamma_{rest} = \{\,x : \exists F\,\}$
  $e_{rest}$
```

Parts of the typing derivation are shown below. The following subtree of the derivation establishes that `pack (new(true))` is an existential package of type $\exists F$ and that the witness type is also $\exists F$ — that is, we are taking advantage of the *impredicativity* of our existential types. Note that since F denotes $\lambda\alpha.\,\mathsf{ref}(\mathsf{bool} \cup \alpha)$, we have $F(\exists F) = \mathsf{ref}(\mathsf{bool} \cup \exists F)$.

$$\dfrac{\text{type}(\exists F) \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset \models_M \texttt{true} : \mathsf{bool}}\;(\texttt{M-true})}{\emptyset \models_M \texttt{true} : \mathsf{bool} \cup \exists F}\;(\texttt{M-}\cup\texttt{i1})}{\emptyset \models_M \texttt{new}(\texttt{true}) : \mathsf{ref}(\mathsf{bool} \cup \exists F)}\;(\texttt{M-new})}{\emptyset \models_M \texttt{new}(\texttt{true}) : F(\exists F)}}{\emptyset \models_M \texttt{pack}\,(\texttt{new}(\texttt{true})) : \exists F}\;(\texttt{M-pack})$$

The following sub-derivation shows that $\texttt{unpack}\,x\,\texttt{as}\,z\,\texttt{in}\,z := x$ is well-typed with respect to type environment $\Gamma_1 = \{\, x : \exists F \,\}$. I've used Γ' as an abbreviation for $\Gamma_1\,[z \mapsto F(\exists F)]$.

$$\dfrac{\Gamma_1 \models_M x : \exists F \qquad \dfrac{\dfrac{}{\Gamma' \models_M z : \mathsf{ref}(\mathsf{bool} \cup \exists F)}\;(\texttt{M-var}) \qquad \dfrac{\dfrac{}{\Gamma' \models_M x : \exists F}\;(\texttt{M-var})}{\Gamma' \models_M z : \mathsf{bool} \cup \exists F}\;(\texttt{M-}\cup\texttt{i2})}{\Gamma_1\,[z \mapsto F(\exists F)] \models_M z := x : \mathsf{unit} \qquad \textit{where}\ \text{type}(\exists F)}\;(\texttt{M-assign})}{\Gamma_1 \models_M \texttt{unpack}\,x\,\texttt{as}\,z\,\texttt{in}\ z := x\ : \mathsf{unit}}\;(\texttt{M-unpack})$$

Given the program and typing derivation I've just described, proving the program safe amounts to proving each of the Hoare triples in the program fragment given below. The store S_0 is the initial store, while S and S' denote the stores obtained by evaluating, respectively, $x = \texttt{new}(\texttt{true})$ in store S_0 and $\texttt{unpack}\,x\,\texttt{as}\,z\,\texttt{in}\,z := x$ in store S. Stores S and S' are shown in Figure 4.3 as before.

$\{\, \exists \Psi_0.\ S_0 :_k \Psi_0 \,\}$
$\texttt{let}\ x\ \texttt{=}\ \texttt{pack}\,(\texttt{new}(\texttt{true}))\ \texttt{in}$
$\{\, \exists \Psi.\ x :_{k-1,\Psi} \exists F\ \wedge\ (k, \Psi_0) \sqsubseteq (k-1, \Psi)\ \wedge\ S :_{k-1} \Psi \,\}$
$\texttt{let}\ \_\ \texttt{=}\ (\texttt{unpack}\,x\,\texttt{as}\,z\,\texttt{in}\ z := x)\ \texttt{in}$
$\{\, \exists \Psi'.\ x :_{k-2,\Psi'} \exists F\ \wedge\ (k-1, \Psi) \sqsubseteq (k-2, \Psi')\ \wedge\ S' :_{k-2} \Psi' \,\}$
e_{rest}

To prove the above Hoare triples, assume S_0 is an arbitrary initial store and that the new cell is some $\ell \notin \mathrm{dom}(S_0)$; let $\ell = 100$ as we had before in Figure 4.3. Also, as before, $S = S_0\,[\ell \mapsto \texttt{true}]$ and $S' = S\,[\ell \mapsto \ell\,]$. Let Ψ_0 be the empty store typing. Finally, pick Ψ such that $\lfloor \Psi \rfloor_{k-1} = \{\, \ell \mapsto \lfloor \mathsf{bool} \cup \exists F \rfloor_{k-1} \,\}$ and let $\Psi' = \lfloor \Psi \rfloor_{k-2}$.

It is important to note that we choose Ψ such that $\lfloor \Psi \rfloor_{k-1}(\ell) = \lfloor \mathsf{bool} \cup \exists F \rfloor_{k-1}$, rather than, say, $\lfloor \mathsf{bool} \cup \alpha \rfloor_{k-1}$. The latter would be incorrect since mutable references should only store values of closed type. Hence, in this case, when we allocate the reference cell, we substitute the witness type for α. One crucial consequence of this is that our model does not permit the witness type of existentials of the form $\exists \alpha.\,(\ldots \mathsf{ref}\,(\ldots \alpha \ldots) \ldots)$ to be changed by future updates. Languages that fail to ensure that the witness types of such existentials never change usually run into

```
let val z = ref (fn x:'a => x) in
    (z := (fn x => x + 1);
     (!z) true)    (* error *)
```

Figure 4.4: Polymorphic References: SML Program

unsoundness in the presence of aliasing. Grossman [Gro02] encountered precisely this problem in Cyclone.

4.4 Discussion and Related Work

Mutable references are notorious for making life difficult for semanticists, type theorists, and language designers. In this section, I'll describe some of the problems that arise in the presence of mutable references, and discuss how these problems are dealt with in the indexed possible-worlds model, as well as in related work. I start, however, by pointing out a limitation of the semantic model that I've presented.

4.4.1 Limitation: Type Functions

The semantics that I have described models quantified types $\forall F$ and $\exists F$ (Section 3.3), as well as recursive types μF (Section 4.2.1), where F is a type function. A limitation of this semantics is that it does not allow us to express arbitrarily nested recursive and quantified types such as the following:

$$\exists(\lambda\alpha.\mathsf{ref}(\mu(\lambda\beta.(\mathsf{ref}\ \beta) \cup \alpha)))$$

In the above type, the recursive type cannot be written in the form μF since F is a single-argument type function while the type expression inside the μ refers to two variables: β, which is bound by μ itself, and α, which is bound by the existential. Swadi [Swa03] shows how best to handle type expressions (i.e., multi-argument type functions) and variables in our semantic approach. The solution is to use de Bruijn indices as we have done in the von Neumann model that we've built for our foundational PCC system (see Chapter 6).

4.4.2 Mixing Mutation and Quantified Types

Mutable references and quantified types often interact in subtle ways leading to unsoundness. A well-known example is the problem of polymorphic references in ML [Tof90] which is illustrated by the SML code shown in Figure 4.4. In ML, value

% $\Gamma_0 = \emptyset$

  let z = $\Lambda.\,\texttt{new}(\lambda x.x)$ in

% $\Gamma_1 = \{\, z : \forall F \,\}$ $(\textit{where } F = \lambda\alpha.\,\mathsf{ref}\ (\alpha \to \alpha))$

  let _ = $z[\,] := \lambda x.\, x + 1$ in

% $\Gamma_2 = \{\, z : \forall F \,\}$ $(\textit{where } F = \lambda\alpha.\,\mathsf{ref}\ (\alpha \to \alpha))$

  $(\,!\,(z[\,])\,)$ true

Figure 4.5: Polymorphic References: λ^M Program

bindings are considered the basic units of type inference for which all ambiguity must be resolved before type checking continues. Hence, for the code shown in Figure 4.4, ML type inference gives `z` the type $\forall\alpha.\,\mathsf{ref}\ (\alpha \to \alpha)$. With this type ascribed to `z`, the type checker decides that the body of the let expression is *well-typed*, when in fact, it clearly *violates type safety*: it updates `z` with a function of type $\mathsf{int} \to \mathsf{int}$ and then applies `z`'s contents to a value of type bool.

To ensure type safety in the presence of polymorphism and mutation, ML designers adopted the following restriction. Variables introduced by a `val` binding are allowed to be polymorphic only if the right-hand side is a value. This is called the *value restriction* on polymorphic declarations. The value restriction is a clever yet simple way to rule out types like $\forall\alpha.\,\mathsf{ref}\ (\alpha \to \alpha)$ by exploiting the fact that expressions of such types cannot be values in ML. Although it is at the expense of some expressiveness in the language, the value restriction achieves the desired goal which is to ensure that contents of mutable references always have *closed* types.

Type inference isn't needed in a target language for type-preserving compilation of ML since the compiler is expected to produce a target program with type annotations. Hence, in λ^M there is no need for constraints such as the value restriction. We simply require that all types in the codomain of the store typing Ψ be *closed* types (as specified in Section 3.2.2). The tricky part is deciding, for each location ℓ, what closed type to add to the store typing when the type of $\texttt{pack}\,\ell$ is of the form $\exists\alpha.\,\mathsf{ref}\,(\ldots\alpha\ldots)$ or the type of $\Lambda.\texttt{new}(e)$ is of the form $\forall\alpha.\,\mathsf{ref}\,(\ldots\alpha\ldots)$ (where $\texttt{new}(e)$ eventually allocates ℓ). I'll address this issue below with the help of some examples.

Polymorphic References Unlike ML expressions of types such as $\forall\alpha.\,\mathsf{ref}\ (\alpha \to \alpha)$, λ^M expressions of such types *can* be values. For example, the type abstraction $\Lambda.\,\texttt{new}(\lambda x.x)$ is a value and has type $\forall\alpha.\,\mathsf{ref}\ (\alpha \to \alpha)$. Let us consider the λ^M program shown in Figure 4.5 which resembles the SML program in Figure 4.4 in that it binds the variable z to a polymorphic reference of type $\forall\alpha.\,\mathsf{ref}\ (\alpha \to \alpha)$, then assigns the function $\lambda x.\, x+1$ to z, and then applies the contents of z to true. In the λ^M program, notice that the variable z *is* bound to a value, but that value is not a location (as the SML programmer expects), but rather a type abstraction. A program that assigns to or dereferences a type abstraction will not type check, but this λ^M program does type check because I have inserted type application coercions $z[\,]$ at appropriate points in the code to coerce z into a mutable reference that may be updated or dereferenced. The important thing to note is that the program in Figure 4.5 is safe. This is because evaluation of the program proceeds as follows.

- The expression $z[\,] := \lambda x.\, x+1$ first allocates a new cell ℓ_1 initialized to $\lambda x.x$ — the type of ℓ_1 is recorded in the store typing as (some approximation of) $\mathsf{int} \to \mathsf{int}$ — and then updates cell ℓ_1 with the function $\lambda x.\, x+1$.
- The expression $(\,!\,(z[\,])\,)\,\texttt{true}$ allocates a second new cell ℓ_2 initialized to $\lambda x.x$ — the type of ℓ_2 is recorded in the store typing as (some approximation of) $\mathsf{bool} \to \mathsf{bool}$ — and then dereferences ℓ_2 and applies the result to $\texttt{true}$.

Hence, unlike the SML program, the λ^M program allocates two separate locations ℓ_1 and ℓ_2. Since a λ^M store typing cannot map ℓ_1 and ℓ_2 to the open type $\alpha \to \alpha$, we must determine what types to instantiate α with. The typing derivation tells us that the first coercion $z[\,]$ effectively applies z to the type int while the second applies z to the type bool. Hence, at the first allocation point, we close the open type $\alpha \to \alpha$ by instantiating α with the closed type int and extend the store typing so that it maps ℓ_1 to the closed type $\mathsf{int} \to \mathsf{int}$. Similarly, at the second allocation point we close type $\alpha \to \alpha$ by instantiating α with bool and extend the store typing so that it maps ℓ_2 to the closed type $\mathsf{bool} \to \mathsf{bool}$.

Storing Universal Types in Mutable Cells Consider again the SML program shown in Figure 4.4 and this time assume that the variable `z` is given the type $\mathsf{ref}\ (\forall \alpha.\, \alpha \to \alpha)$. The SML type checker now comes to a different conclusion about well-typedness of the body of the let expression. Specifically, the assignment to `z` (which updates `z` with a function of type $\mathsf{int} \to \mathsf{int}$) does not type check and the program is (rightly) rejected.

A comparable program in λ^M might be as follows. Note that z has type $\mathsf{ref}\ (\forall \alpha.\, \alpha \to \alpha)$ and that this program is also ill-typed.

$$\begin{array}{l}\texttt{let}\ z = \texttt{new}(\Lambda.\, \lambda x.x)\ \texttt{in}\\ \texttt{let}\ \_ = z := \Lambda.\, (\lambda x.\, x+1)\ \texttt{in}\\ (!\, z)[\,]\,\texttt{true}\end{array}$$

The expression $\texttt{new}(\Lambda.\, \lambda x.x)$ allocates a new location ℓ and binds ℓ to z. The store typing is extended to map ℓ to (some approximation of) the type $\forall \alpha.\, \alpha \to \alpha$. Thus, the subsequent assignment to ℓ of a value of type $\forall\alpha.\,\mathsf{int} \to \mathsf{int}$ is not permitted by the model which ensures that only values of type $\forall\alpha.\,\alpha \to \alpha$ may be assigned to ℓ.

In a functional setting the only value that has type $\forall\alpha.\,(\alpha \to \alpha)$ is the polymorphic identity function [Wad89]. However, since mutation breaks parametricity, in a language with mutable references any number of functions with varying side-effects may have this type, as long as the function argument x is returned without ever being used in a context where safety is contingent upon x having a particular type.

Abstract References Consider the value $\mathsf{pack}\,\ell$ of type $\exists\alpha.\,\mathsf{ref}\,(\alpha \to \alpha)$ and let τ be the witness type for $\mathsf{pack}\,\ell$. The store typing cannot map ℓ to the open type $\alpha \to \alpha$, so we close the latter by instantiating α with the existential's witness type τ. Hence, upon allocation of cell ℓ, we extend the store typing to map ℓ to (some approximation of) the closed type $\tau \to \tau$.[4] This implies that cell ℓ may only be updated with functions of type $\tau \to \tau$ — that is, the witness type of the existential must remain unchanged.

For mutable reference types of the form $\mathsf{ref}\,(\ldots\alpha\ldots)$ when the existential that binds α occurs outside the ref, changing the witness type of the existential amounts to changing the type of the contents of the reference cell — i.e., it violates the type invariance principle. Changing the type of the contents of the reference cell may be unsafe because λ^M permits aliases to the unpacked mutable cell ℓ and these aliases have type $\mathsf{ref}\,(\ldots\tau\ldots)$ (where τ is the witness type). By ensuring that the witness types of such abstract references never change, the semantic model of λ^M ensures type invariance and ultimately type safety.

Grossman [Gro02] encountered the above problem in an earlier version of Cyclone. Cyclone's support for mutation of existentials permits changes in witness types, while its reference patterns allow the fields of existentials to be aliased. The combination of these features results in unsoundness. Nonetheless, each feature is safe on its own. In particular, mutation of existentials may violate type invariance, but such mutations are type safe as long as there are no aliases to the fields of the existentials.

Finally, note that even in a language that adheres to the type invariance principle, it is safe to change the witness type of an existential package that has a type like $\exists\alpha.\,(\mathsf{ref}\,\mathsf{int}) \times \alpha$. The change has no effect on the type of the reference cell inside the existential.

Storing Existential Types in Mutable Cells Thus far I have only discussed types where ref occurs inside an existential. I shall now consider types where an existential occurs inside ref. In each of the following examples it *is* safe to change an existential's witness type.

[4] Section 4.3.2 describes another example that allocates a cell of type $\exists\alpha.\,\mathsf{ref}\,(\ldots\alpha\ldots)$.

- Consider a location ℓ of type $\mathsf{ref}\,(\exists\alpha.\alpha \times \alpha)$. The store typing maps ℓ to (some approximation of) the closed type $\exists\alpha.\,\alpha \times \alpha$. Therefore, the model permits ℓ to contain a value $v = \mathtt{pack}\ \langle 3, 5\rangle$ with witness type int and subsequently, also allows ℓ to be updated with the value $v' = \mathtt{pack}\ \langle \mathtt{true}, \mathtt{false}\rangle$ whose witness type is bool.

- Suppose that a location ℓ has type $\mathsf{ref}\,(\exists\alpha.\,\alpha \to (\mathsf{ref\ bool}))$. The store typing maps ℓ to $\exists\alpha.\,\alpha \to (\mathsf{ref\ bool})$ so one can safely update ℓ with a function of type $\mathsf{bool} \to (\mathsf{ref\ bool})$ or with a function of type $(\mathsf{ref\ int}) \to (\mathsf{ref\ bool})$ and so on.

4.4.3 Shared References and Monotonicity

I've described how to model a language that deals with the problem of shared references (aliasing) by adopting the type invariance principle I specified in Chapter 1. By disallowing strong updates, this principle ensures that the types of all allocated locations are preserved, and type preservation ensures monotonicity. As for λ^I (see Section 2.2.5), in order to establish safety, we need to prove that our model of λ^M is Kripke monotone. Informally, the proof follows from the fact that monotonicity over the course of evaluation plays a central role at every layer of our model. Suppose that Ψ is the current store typing and Ψ' is some possible future store typing. The lowest layer of the model specifies that if ℓ maps to τ in state (k, Ψ) then ℓ maps to $\lfloor\tau\rfloor_j$ in state (j, Ψ'). This is captured by valid state extension ($\sqsubseteq$) which establishes the monotonicity of states.

The next layer of the model shows that if $\langle k, \Psi, v\rangle \in \tau$ then $\langle j, \Psi', v\rangle \in \tau$. For each type τ, we prove the monotonicity of sets of values of that type as we use up execution steps. To prove this property (specified as $\mathrm{type}(\tau)$) we must rely on the monotonicity of states.

The next layer establishes type preservation, that is, if e has type τ with respect to (k, Ψ) where $S :_k \Psi$, and we reach machine state (S', e') where $S' :_j \Psi'$, then e' has type τ with respect (j, Ψ'). This corresponds precisely to Kripke monotonicity. We prove the monotonicity (over the course of evaluation) of sets of expressions of each type. The proof relies on the monotonicity of sets of values of each type.

Hence, in λ^M we ultimately rely on approximately type-preserving updates and the absence of cell deallocation to establish monotonicity at each "layer" of the model. This monotonicity guarantee lies at the heart of why the mutation of shared references does not violate safety.

4.4.4 Coinduction and Non-wellfounded Sets

The circularity I described in Section 3.2.3 can be dealt with without constructing an *indexed* model — that is, one can construct a possible-worlds model (of general references) that is not stratified. To see how, let us consider the nature of possible-worlds. Peregrin's [Per93] analysis concludes that "a possible world in the intuitive sense can be explicated as a maximal consistent class of statements". Thus, to give the semantics of seemingly circular possible worlds one can use techniques like coinduction [Par69, MT91b, BM96] or non-wellfounded sets [Acz88].

It may be argued that using coinduction or non-wellfounded sets would be simpler than constructing a stratified model of types. I believe, however, that while the use of coinductive techniques may seem to simplify pencil-and-paper proofs, it is unlikely to lead to simpler machine-checked proofs, and in fact, it would increase the size of the trusted computing base. To see why, let us look at what the use of coinduction entails. One way to incorporate coinduction would be to add the Anti-Foundation Axiom (AFA) described by Aczel [Acz88] to the trusted computing base. Then we could use it to conclude that the circularity or lack of wellfounded sets does not imply inconsistency. Aczel's formulation of the AFA is particularly problematic[5] because it is based on graphs. To specify the AFA (i.e., to add it to the TCB) we would first need to be able to represent graphs; hence, we would essentially have to add graphs to the TCB. Since one of the goals of the Princeton Foundational PCC project is to minimize the trusted computing base, we have chosen not to use coinduction. In this thesis, I have shown how coinduction may be avoided in our semantics by showing how the possible worlds may be stratified.

4.4.5 Possible-Worlds Models

A common feature of a number of models of mutable state (also called *mutable store* in the literature) is that they specify *how* the state is allowed to vary over time — that is, they are possible-worlds models. Models for *Idealized Algol* developed by Reynolds and Oles [Rey81, Ole82, Ole85, Ole97] make use of functor categories; functors are important because they capture the fact that the size of the store, as well as its contents, may change over time. To specify how the state is allowed to change at any point in the program, they use functor categories indexed by possible worlds or *store shapes*. Note that these models do not handle general references. Stark [Sta94] (building on work done with Pitts on possible-worlds models of the nu-calculus [PS93]) describes a denotational semantics for *Reduced ML* that supports integer references.

More recently, Levy [Lev01, Lev02] described a possible-worlds model that supports general references, but not quantified types. There are interesting correspondences between Levy's model and mine. His *world-store* (w, s), where w is a world and s is a w-store, (i.e., each location in s is well-typed with respect to w) corresponds to a well-typed store $S :_k \Psi$ in my model. His accessibility relation between

[5] There are other formulations of AFA which I have not considered (see Barwise and Moss [BM96]).

worlds resembles state extension ($\sqsubseteq$) in my model. His model has the property: "if $w \leq w'$ then every w_τ-value is a w'_τ-value (where w_τ-value denotes a value of type τ in world w); this corresponds to type(τ) in my model. The definition (denotation to be more precise) of the type $\mathsf{ref}\ \tau$ in his model is: $[\![\mathsf{ref}\ \tau]\!]w = \$w_\tau$ (where $\$w_\tau$ denotes "the set of cells of type τ in w"). Notice that $[\![\mathsf{ref}\ \tau]\!]$ is defined in terms of the syntax τ rather than the semantics $[\![\tau]\!]$. Levy is faced with the same kind of circularity that I described in Section 3.2.3. He solves it by observing that recursive equations on domains have solutions. I solve it by showing that the hierarchy of type approximations has a limit.

An important difference between Levy's work and mine is that while his semantics makes use of complicated mathematics (functor categories, for instance), mine is based on the operational semantics of the language and requires only sets and induction.

Benefit of Possible-Worlds Semantics I modeled types in λ^I as predicates on stores S and values v. Let us compare this to the traditional Strachey approach where types are modeled as sets of values. In such a model, a reference type $\mathsf{box}\ \tau$ is defined simply as *Loc*, the set of all locations ℓ. This means that the semantics of type $\mathsf{box}\ \tau$ does not guarantee that a location $\ell \in \mathsf{box}\ \tau$ is an allocated location. As a result, the λ^I typing rule for dereferencing cannot be proved sound. Similarly, using a semantics where λ^M types are modeled as sets of values, one cannot validate the λ^M typing rules for dereferencing and assignment. Intuitively, this is due to the fact that these rules rely on the intuition that memory access works *locally*. Suppose, for instance, that we have a program e_{prog} that allocates a number of cells over the course of execution; let $!\,e_1$ be a subexpression of e_{prog}. Regardless of the number of cells allocated by the program, if one knows that e_1 has type $\mathsf{ref}\ \tau$ then one should be able to dereference e_1 and get a value of type τ — we've seen that the typing rule for dereferencing relies on this intuition.

More generally, program components often work with circumscribed collections of resources. Possible-worlds semantics allows us to identify the resources of interest in the worlds (stores or store typings) and to prove the soundness of typing rules that would not otherwise be sound. This is because these rules work locally, in that they do not explicitly keep track of all resources *not* altered by a program component.

Without possible-worlds semantics, we could still prove the soundness of more global typing rules that keep track of the whole store — for instance, the typing judgments could be augmented with store typings that track the set of allocated locations and their types. But the local typing rules that we've used for λ^I and λ^M are much simpler, and reasoning with them leads to type derivations (and hence, proofs) that are more manageable in terms of both size and complexity.

Notice that Hoare logic rules, rely on the same intuition (i.e., the intuition that memory access works locally) as our λ^I and λ^M typing rules. Hoare logic rules describe *only* those resources that *are* altered by a program component — it is *assumed* that any resource not mentioned by the Hoare logic rule is unchanged. Possible-worlds semantics is then the key to proving the soundness of such rules.

Others have used possible-worlds semantics to model languages with mutable state but, for the most part, they haven't proved any technical results showing what one gains from using possible-worlds semantics. Reynolds and Oles [Rey81, Ole82, Ole85, Ole97] showed (intuitively) that their possible-worlds models of Idealized Algol were more faithful to the stack discipline of Algol than traditional Strachey semantics. Levy [Lev02] argued that Strachey semantics did not effectively capture the semantics of references in a language with dynamic allocation. He did not, however, demonstrate any concrete benefit of his model, whereas I have proved the soundness of typing rules that cannot otherwise be validated.

4.4.6 Game Semantics

Hintikka [Hin75] advocated the use of game-theoretical semantics to model possible worlds. Game semantics is especially useful in removing from consideration all *impossible* possible worlds. Abramsky, Honda and McCusker[AHM98] describe a game semantics of general references that they show to be fully abstract. In this model, reference types are modeled by their behavior, or more specifically as a product of a "read method" and a "write method" in the style of Reynolds [Rey81]. This representation of references is quite different from that in location-based models such as the one presented in this thesis. It would be interesting to see if such a model could be incorporated into a foundational PCC system. While Abramsky *et al.*'s model does not support quantified types, Hughes [Hug97] has presented a games model of System F. It would be useful to investigate whether Hughes' model of System F can be incorporated into Abramsky *et al.*'s model of mutable references.

4.4.7 Other Related Work

The use of a store typing that maps locations to types is not new. Tofte [Tof90] uses this approach in his type soundness proof for polymorphic references. Tofte, however, makes use of coinduction to handle cycles in the memory graph. Harper [Har96] has shown how a progress-and-preservation proof can be arranged so that there is no need for coinduction. The model I've presented can handle cycles in memory by virtue of the index k. For a reference to a memory cycle to be well-typed, one only needs to know that it is well-typed to approximation k. With each memory dereference the k decreases. Hence, there is no need for coinduction.

Logical relations based on an operational semantics (or denotational-operational models, as I have called them) have been used by others for reasoning about equivalence of terms. Wand and Sullivan [WS97] describe a denotational semantics based on an operational term model and use the approach for proving the correctness of program transformations in a Scheme compiler. Pitts [Pit02] has made use of *operationally-based logical relations* to reason about the equivalence of programs written in a fragment of ML. Recently, Pitts has also shown how to handle existential types [Pit98] and parametric polymorphism [Pit00], but always in the absence of *general* references and recursive datatypes. I shall have more to say about this line of work in Section 8.1.

Chapter 5

Machine-Checked Proofs

A foundational proof-carrying code system demands that all proofs of safety be *machine-checkable*. This chapter looks at how foundational proofs of safety may be mechanized. For the prototype FPCC system we are building at Princeton, we have constructed machine-checked safety proofs based on models of types for von Neumann machines. In this chapter, to keep the presentation simple, I shall describe how to mechanize safety proofs based on models of types for the λ-calculus. Specifically, I shall focus on machine-checked proofs for λ^M (the polymorphic λ-calculus with references and existentials from Chapter 3) with recursive types (Section 4.2.1).

One may choose to represent the proof in any one of a variety of logics. In Section 5.1 I'll explain how to encode the semantics of λ^M in the Calculus of Inductive Constructions (CiC) [CH88, PM93] as that is the logic most suited to modeling the λ^M type hierarchy. In Section 5.2 I'll explain how to formulate a solution in higher-order logic. For our prototype FPCC system, we have built machine-checked proofs in higher-order logic represented in LF [HHP93] and type-checked by Twelf [PS99]. Note: The CiC encoding I present in Section 5.1 is due to Roberto Virga. The representation function I describe in Section 5.2.3 was defined and implemented by Roberto Virga; definition and proofs of the representation function were automated by Roberto Virga and Xinming Ou.

5.1 Representation in CiC

As I explained in Chapter 3 (Section 3.3.2), all definitions in the semantics of λ^M are carefully constructed to obey the *stratification invariant* which says that when considering a tuple $\langle k, \Psi, v\rangle$ the store typing Ψ is not required to be defined beyond $\lfloor\Psi\rfloor_k$. Hence, types in λ^M form a set that can be constructed using the recursively

```
Fixpoint itype [k : nat] : Type :=
  Cases k
    of O => UnitT
     | (S k’) => (prodT (itype k’)
                   ((location -> (itype k’))
                     -> exp -> Prop))
  end.

Definition istoretype [k : nat] : Type :=
  location -> (itype k).

Definition type: Type := (k: nat) (itype k).
```

Figure 5.1: CiC: Definition of Type Hierarchy

defined sets Type_k and $\mathit{StoreType}_k$ as follows.

$$\begin{array}{l}
\tau \in \mathit{Type}_0 \text{ iff } \tau = \{\} \\
\tau \in \mathit{Type}_{k+1} \text{ iff } \forall \langle j, \Psi, v\rangle \in \tau.\ j \leq k \ \wedge\ \Psi \in \mathit{StoreType}_j \\
\Psi \in \mathit{StoreType}_k \text{ iff } \forall \ell \in \mathrm{dom}(\Psi).\ \Psi(\ell) \in \mathit{Type}_k \\
\tau \in \mathit{Type} \text{ iff } \forall k.\ \lfloor \tau \rfloor_k \in \mathit{Type}_k.
\end{array}$$

The Calculus of Inductive Constructions (CiC) [CH88, PM93], upon which the Coq system [BBC+98] is based, can represent the above definitions quite directly. The CiC encoding is shown in Figure 5.1. Each set Type_k is modeled using a product type `(itype k)` in CiC. More specifically, Type_0 is modeled by the unit type `UnitT`, while Type_{k+1} is given by the product of the representation of Type_k and the set of membership functions for pairs $\langle \Psi, v\rangle$, where Ψ has type `(istoretype k)`.

It is straightforward to obtain the k-th approximation of a type ($\lfloor \tau \rfloor_k$): given an object `tau` of type `type`, its k-th approximation will correspond to the application `(tau k)`. To check that a triple $\langle k, \Psi, v\rangle$ is in `tau`, one needs to apply `tau` to $k+1$, and take the second component as follows.

```
(sndT (tau (S k)) Psi v) : Prop
```

The definition of the types and type constructors presented in Figure 3.6 can be given by recursion on the natural numbers. At each step, one has to construct an object of type `(itype k)`. The case $k = 0$ is trivial, since `IT` (which has type `UnitT`) is the only object belonging to `(itype 0)`. For the case $k+1$, one only has to provide a formula that decides which tuples $\langle k, \Psi, v\rangle$ belong to the type.

```
Definition refTy [tau : type] : type :=
  (nat_rect itype IT
    ([k : nat][tauk : (itype k)]
     (pairT tauk
      ([psi: (istoretype k)][v: exp]
       (Ex [l : location]
           (v = (loc l))
           /\
           (eqT (itype k) (psi l) (tau k))))))).
```

Figure 5.2: CiC: Definition of ref

Figure 5.2 illustrates the representation of the type constructor for mutable references. (The reader may want to compare this to the definition of ref in Figure 3.6.)

The predicate `istore_welltyped` models the relation $S :_k \Psi$ and is defined in Figure 5.3. The definition relies on the function

```
istoretype_aprx : (k,j:nat)
                  (lt j k) -> (istoretype k)
                           -> (istoretype j)
```

which is used to “lower” an approximation to the correct index. It is straightforward to define `istoretype_aprx` in Coq.

The λ^M operational semantics and the remaining semantic definitions can be similarly encoded in CiC.

Proof Checker and TCB I mentioned in Chapter 1 that one of the goals of the Foundational PCC project at Princeton is to minimize the size of the trusted computing base. Hence, in choosing a logic in which to encode our proofs, we must take into account the size of the proof checker (which is a trusted component). One drawback of representing the proofs in CiC is that existing CiC checkers are very large and complex programs. In the foundational PCC project, we encode proofs in higher-order logic represented in the LF metalogic [HHP93]. The minimal LF checker that we have developed[1] is at least an order of magnitude smaller than existing CiC checkers.

[1] We use the Twelf checker during proof development, but it is hardly a minimal checker. Since minimizing the size of the TCB is a central goal of our project, we have developed our own minimal LF checker [AMSV02, WAS03].

```
Definition istore_welltyped [k : nat; psi : (istoretype k);
                             s : store] : Prop :=
  (All [j : nat]
  (All [l : location]
  (All [v : exp]
    (h : (lt j k))
    ((s l) = (Some exp v)
      -> (ExT [psi' : (istoretype j)]
           (eqT (istoretype j) (istoretype_aprx k j h psi) psi')
           /\
           ((sndT (psi l)) psi' v)))))).
```

Figure 5.3: CiC: Definition of $S :_k \Psi$

5.2 Representation in Higher-Order Logic

In the foundational PCC system we are building at Princeton, proofs are encoded in higher-order logic. Unfortunately, higher-order logic does not provide as convenient a mechanism as CiC for making stratified metalogical types. As I shall explain, in effect, we must construct the stratification ourselves.

Consider the recursively defined sets Type_k and $\mathit{StoreType}_k$ from the previous section (page 5.1). The metalogical types of these sets may be written as follows.

$$\begin{array}{lcl}
\mathit{Type}_0 & = & \mathit{Unit} \\
\mathit{StoreType}_k & = & \mathit{Loc} \overset{\text{fin}}{\to} \mathit{Type}_k \\
\mathit{Type}_{k+1} & = & \mathit{Nat} \times \mathit{StoreType}_k \times \mathit{Val} \to o \\
\mathit{Type} & = & \mathit{Type}_i \quad \mathit{where}\ i \geq 0
\end{array}$$

The stratified metalogical types above cannot be described using higher-order logic. We need a single type of *Type*, not an infinite number of them.

5.2.1 A Hierarchy of Gödel Numberings

To achieve a single type of *Type*, I present a solution that replaces the (semantic) type (Type_k) in the store typing with its syntax, that is, with a type expression t (of type *TyExp*). A type expression t should uniquely identify a type τ — that is, a type expression may be thought of as the Gödel number of a type. Instead of encoding these Gödel numbers using integers, I will use finite trees of natural numbers (though a proof implementor is free to choose any other way of representing Gödel numbers).

Replacing types with type syntax in the store typing flattens out our type hierarchy, but now we need a way to map the type expressions t to the semantic types (of metalogical type Type_k) that they represent. One could try (unsuccessfully) to accomplish this by constructing a Gödel-numbering function $\mathcal{G}$ that is a mapping from type expressions t to all of the types τ that can be constructed using the λ^M type constructors. Unfortunately, the definitions of some of these type constructors will need to refer to $\mathcal{G}$, leading to a circularity — for instance, the definition of $\mathsf{ref}\ \tau$ (see Figure 3.6) will need to refer to $\mathcal{G}$ in order to relate the type expression $t = \Psi(\ell)$ to some type τ' (i.e., $\mathcal{G}(t) = \tau'$) so that it can then check if $\lfloor \tau' \rfloor_k = \lfloor \tau \rfloor_k$. I resolve this circularity by constructing a hierarchy of Gödel numbering functions $\mathcal{G}_k$; the semantics of types at one level of the hierarchy (e.g., types in the codomain of $\mathcal{G}_k$) can make use of lower levels of the Gödel numbering function (i.e., any $\mathcal{G}_j$ such that $j < k$). There are a number of subtleties that arise, as we shall see below.

The metalogical types (in higher-order logic) of what I will construct are given below. *Val* v is a λ^M value; *Loc* ℓ is an addressable store location; *Store* S is a store; *TyExp* t is a Gödel number (or type expression, or type syntax) which I encode as a tree of natural numbers; *StoreType* Ψ is a store typing, but in this model it maps locations to type expressions instead of types; *Type* τ is a set of triples, as before; *Rep* η is a *representation function* (or Gödel numbering function) that maps Gödel numbers to types.

$$\begin{array}{lcl}
\mathit{Val} & = & \textit{the type of } \lambda^M \textit{ values} \\
\mathit{Loc} & = & \mathit{Nat} \\
\mathit{Store} & = & \mathit{Loc} \overset{\text{fin}}{\to} \mathit{Exp} \\
\mathit{TyExp} & = & \mathit{Tree}(\mathit{Nat}) \\
\mathit{StoreType} & = & \mathit{Loc} \overset{\text{fin}}{\to} \mathit{TyExp} \\
\mathit{Type} & = & \mathit{Nat} \times \mathit{StoreType} \times \mathit{Val} \to o \\
\mathit{Rep} & = & \mathit{TyExp} \to \mathit{Type}
\end{array}$$

I use the terms "representation function" and "Gödel-numbering function" interchangeably. Some of the notation (related to representation functions) that I will use is as follows: $\mathcal{G}$ denotes the stratified Gödel numbering function that I will construct; $\mathcal{G}_k$ (where $k \geq 0$) denotes the k-th level of the stratified Gödel numbering function $\mathcal{G}$; the variable η denotes an arbitrary representation function. Their types will always be as shown below.

$$\begin{array}{lcl}
\mathcal{G} & : & \mathit{Nat} \to \mathit{Rep} \\
\mathcal{G}_k & : & \mathit{Rep} \\
\eta & : & \mathit{Rep}
\end{array}$$

Definition 5.1 (Approx)
We define $\lfloor\tau\rfloor_k$ as before. However, since now store typings map locations into type expressions, we need to parametrize the approximation of a store typing with respect to a representation function:

$$\lfloor\Psi\rfloor_{\eta,k} \stackrel{\text{def}}{=} \{\, \ell \mapsto \lfloor\eta(t)\rfloor_k \mid \Psi(\ell) = t \,\}$$

Note that the metalogical type of $\lfloor\Psi\rfloor_{\eta,k}$ is *Loc* → *Type*, which is different than the metalogical type of Ψ.

All the definitions given in Section 3.3 follow through, but need to be parametrized by η as well.

Definition 5.2 (State Extension)
A valid state extension *is defined as follows:*

$$(k,\Psi) \sqsubseteq_\eta (j,\Psi') \stackrel{\text{def}}{=} j \leq k \ \wedge\ (\forall \ell \in \mathrm{dom}(\Psi).\ \lfloor\Psi'\rfloor_{\eta,j}(\ell) = \lfloor\Psi\rfloor_{\eta,j}(\ell))$$

Definition 5.3 (Type)
A type *is a set τ of tuples of the form $\langle k,\Psi,v\rangle$ where $k \geq 0$ and where the set τ is closed under state extension; that is,*

$$\begin{aligned}\mathrm{type}_\eta(\tau) \stackrel{\text{def}}{=}\ & \forall k,j,\Psi,\Psi',v.\\ & (\langle k,\Psi,v\rangle \in \tau \ \wedge\ (k,\Psi) \sqsubseteq_\eta (j,\Psi')) \implies \langle j,\Psi',v\rangle \in \tau\end{aligned}$$

Definition 5.4 (Well-typed Store)
A store S is defined as well-typed to approximation k with respect to store typing Ψ as follows:

$$\begin{aligned}S :_{\eta,k} \Psi \stackrel{\text{def}}{=}\ & \mathrm{dom}(\Psi) \subseteq \mathrm{dom}(S) \ \wedge\\ & \forall j < k.\, \forall \ell \in \mathrm{dom}(\Psi).\ \langle j,\Psi,S(\ell)\rangle \in \lfloor\Psi\rfloor_{\eta,k}(\ell)\end{aligned}$$

Definition 5.5 (Expr : Type)
For any closed expression e and type τ,

$$\begin{aligned}e :_{\eta,k,\Psi} \tau \stackrel{\text{def}}{=}\ & \forall j,S,S',e'.\ (0 \leq j < k \ \wedge\ S :_{\eta,k} \Psi\\ & \qquad \wedge\ (S,e) \longmapsto^j_M (S',e') \ \wedge\ \mathrm{irred}(S',e'))\\ & \implies \exists\Psi'.\ (k,\Psi) \sqsubseteq_\eta (k-j,\Psi')\\ & \qquad \wedge\ S' :_{\eta,k-j} \Psi' \ \wedge\ \langle k-j,\Psi',e'\rangle \in \tau\end{aligned}$$

$$\begin{aligned}
\overline{\bot}_\eta &\stackrel{\text{def}}{=} \{\} \\
\overline{\mathsf{unit}}_\eta &\stackrel{\text{def}}{=} \{\langle k, \Psi, \texttt{unit}\rangle\} \\
\tau_1 \overline{\to}_\eta \tau_2 &\stackrel{\text{def}}{=} \{\langle k, \Psi, \lambda x.e\rangle \mid \forall v, \Psi', j < k.\ ((k, \Psi) \sqsubseteq_\eta (j, \Psi') \ \wedge\ \langle j, \Psi', v\rangle \in \tau_1) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \Longrightarrow\ e[v/x] :_{\eta,j,\Psi'} \tau_2\} \\
\overline{\mathsf{ref}}_\eta \tau &\stackrel{\text{def}}{=} \{\langle k, \Psi, \ell\,\rangle \mid \lfloor \Psi \rfloor_{\eta,k}(\ell) = \lfloor \tau \rfloor_k\} \\
\overline{\forall}_\eta F &\stackrel{\text{def}}{=} \{\langle k, \Psi, \Lambda.e\rangle \mid \text{wfupto}(k, F)\ \wedge \\
&\qquad\qquad\qquad \forall j, \Psi', t.\ (k, \Psi) \sqsubseteq_\eta (j, \Psi')\ \Longrightarrow\ e :_{\eta,j,\Psi'} F(\eta(t))\} \\
\overline{\exists}_\eta F &\stackrel{\text{def}}{=} \{\langle k, \Psi, \texttt{pack}\, v\rangle \mid \text{wfupto}(k, F)\ \wedge\ \exists t.\ \langle k, \Psi, v\rangle \in F(\eta(t))\} \\
\overline{\mu}_\eta F &\stackrel{\text{def}}{=} \{\langle k, \Psi, v\rangle \mid\ \langle k, \Psi, v\rangle \in F^{k+1}(\overline{\bot}_\eta)\}
\end{aligned}$$

Figure 5.4: Higher-Order Logic: Pretype Definitions (parameterized by η)

5.2.2 Pretypes

I explained in Section 5.2.1 that the definitions of some of the type constructors need to refer to the representation function. But this leads to a problem: which do we define first, the type constructors or the representation function? I shall solve this problem by first defining *pretype* constructors. Each pretype constructor (written with an $\overline{\mathsf{overbar}}$) needs a η parameter to stand in for some $\mathcal{G}_i$ (which has not been defined yet). The intent is to (inductively) define the stratified representation function $\mathcal{G}$ next, and then to define type constructors c from the corresponding pretype constructors $\bar{c}$ simply by instantiating the η with an appropriate level of $\mathcal{G}$ (i.e., with some $\mathcal{G}_i$). Hence, informally, the distinction between pretype constructors and type constructors is that the latter are not parametrized by a representation function η. Definitions of the pretype constructors are given in Figure 5.4.

The definitions of $\overline{\forall}_\eta F$ and $\overline{\exists}_\eta F$ deserve some comment. First, note that these definitions correspond to the definitions of quantified types presented in Section 4.1, that is, they assume that type application and unpack are virtual instructions. Second, notice that in the definition of $\overline{\forall}_\eta F$ (likewise $\overline{\exists}_\eta F$) the quantification is over type expressions t rather than types τ. This is necessary in order to correctly handle polymorphic and abstract references, that is, types of the form $\forall\alpha.\,(\ldots \mathsf{ref}\,(\ldots\alpha\ldots)\ldots)$ and $\exists\alpha.\,(\ldots \mathsf{ref}\,(\ldots\alpha\ldots)\ldots)$ (which I discussed at length in Section 4.4.2). Recall that upon allocation of a cell ℓ of a type such as $\mathsf{ref}\,(\alpha \to \alpha)$ (where α is bound by a universal or existential type constructor that occurs outside the ref), we initialize ℓ

with a value of some closed type of the form $\tau \to \tau$ and extend the store typing Ψ so that it maps ℓ to the type $\tau \to \tau$. In the higher-order logic encoding, at the point where ℓ's contents are initialized, there must be some type syntax (i.e., a Gödel number t) associated with the witness type τ with which we instantiate the bound type α. We need this type expression t in order to construct the type expression t' associated with the type $\tau \to \tau$ so that we can extend the store typing Ψ with a mapping from ℓ to t'. To guarantee that there exists a type expression t for the witness type τ, in the definitions of the pretypes $\overline{\forall}_\eta F$ and $\overline{\exists}_\eta F$, we require that the underlying higher-order logic quantifiers bind type expressions t instead of types τ.

Henceforth, I will use the metavariables τ and ς to range over types and pretypes, respectively. Also, I will use the metavariable c for type constructors and $\overline{c}$ for the corresponding pretype constructor.

Properties of Pretypes The pretype constructors in Figure 5.4 must satisfy certain properties so that types eventually defined using these pretype constructors can be proved valid. In particular, pretype constructors must be (1) nonexpansive, (2) approximately congruent, and (3) closed under state extension. I will describe each of these properties below, usually preceded by some auxiliary definitions. I start by defining equality and approximate equality for constructs of various metalogical types in our model.

Definition 5.6 (Equality and Approximate Equality)
Equality ($=$) and approximate equality ($\approx_k$) for types τ and τ', type functions F and F', and representation functions η and η' are defined as follows:

$$\begin{aligned}
\tau_1 = \tau_2 &\overset{\text{def}}{=} \forall k, \Psi, v.\ \langle k, \Psi, v\rangle \in \tau_1 \iff \langle k, \Psi, v\rangle \in \tau_2 \\
F_1 = F_2 &\overset{\text{def}}{=} \forall \tau.\ F_1(\tau) = F_2(\tau) \\
\eta = \eta' &\overset{\text{def}}{=} \mathrm{dom}(\eta) = \mathrm{dom}(\eta') \ \wedge\ (\forall t.\ \eta(t) = \eta'(t)) \\[1em]
\tau_1 \approx_k \tau_2 &\overset{\text{def}}{=} \lfloor \tau_1 \rfloor_k = \lfloor \tau_2 \rfloor_k \\
F_1 \approx_k F_2 &\overset{\text{def}}{=} \forall \tau.\ \lfloor F_1(\tau) \rfloor_k = \lfloor F_2(\tau) \rfloor_k \\
\eta \approx_k \eta' &\overset{\text{def}}{=} \mathrm{dom}(\eta) = \mathrm{dom}(\eta') \ \wedge\ (\forall t.\ \eta(t) \approx_k \eta'(t))
\end{aligned}$$

(1) Nonexpansive I define a notion of approximate nonexpansiveness, which is analogous to the notion of approximate wellfoundedness (Definition 4.1).

Definition 5.7 (Nonexpansive Up To)
A type functional F is defined as nonexpansive up to k *if*

$$\mathrm{nxupto}(k, F) \overset{\text{def}}{=} \forall j, \tau.\ 0 \le j \le k \implies F(\tau) \approx_j F(\lfloor \tau \rfloor_j)$$

A type functional F is nonexpansive (Definition 3.9) if $\mathrm{nxupto}(k, F)$ for all $k \geq 0$. Notice that nonexpansiveness vacuously holds for types $\overline{\bot}_\eta$ and $\overline{\mathsf{unit}}_\eta$ which may be thought of as zero-argument type functions.

Lemma 5.8 (Pretypes Nonexpansive)
For all representation functions η and for all $k \geq 0$,

- *For all j such that $0 \leq j \leq k$, and for all types τ, $\overline{\mathsf{ref}}_\eta(\tau) \approx_j \overline{\mathsf{ref}}_\eta(\lfloor \tau \rfloor_j)$.*
- *For all j such that $0 \leq j \leq k$, and for all types τ_1, τ_2, $(\tau_1 \overline{\rightarrow}_\eta \tau_2) \approx_j (\lfloor \tau_1 \rfloor_j \overline{\rightarrow}_\eta \lfloor \tau_2 \rfloor_j)$.*
- *If $\overline{c} \in \{\, \overline{\mu}, \overline{\forall}, \overline{\exists} \,\}$ and F is a type functional such that $\mathrm{nxupto}(k, F)$, then for all j such that $0 \leq j \leq k$, and for all types τ, $\overline{c}_\eta F(\tau) \approx_j \overline{c}_\eta F(\lfloor \tau \rfloor_j)$.*

Informally, nonexpansiveness says that if a constructor c is applied to some type τ, an index k, store typing Ψ, and value v, it should not inspect τ in greater detail than k — that is, if $\tau \approx_k \tau'$ then $c(\tau) \approx_k c(\tau')$ (see Definition 3.9 in Section 3.3.9). Now, if $\tau \approx_k \tau'$ for all $k \geq 0$ (equivalently, $\tau = \tau'$), then $c(\tau) \approx_k c(\tau')$ for all $k \geq 0$ (equivalently, $c(\tau) = c(\tau')$) — that is, nonexpansiveness implies *extensionality*. (A constructor is *extensional* if, when applied to equal arguments, it yields equal results.)

Corollary 5.9 (Pretypes Extensional)
For all pretype constructors $\overline{c}$ defined in Figure 5.4 and for all representation functions η and η' such that $\eta = \eta'$,

- *If $\overline{c} \in \{\, \overline{\bot}, \overline{\mathsf{unit}} \,\}$ then $\overline{c}_\eta = \overline{c}_{\eta'}$.*
- *For all τ, τ', if $\tau = \tau'$ then $\overline{\mathsf{ref}}_\eta(\tau) = \overline{\mathsf{ref}}_{\eta'}(\tau')$.*
- *For all $\tau_1, \tau_1', \tau_2, \tau_2'$, if $\tau_1 = \tau_1'$ and $\tau_2 = \tau_2'$ then $\tau_1 \overline{\rightarrow}_\eta \tau_2 = \tau_1' \overline{\rightarrow}_{\eta'} \tau_2'$.*
- *For all F, F', if $\overline{c} \in \{\, \overline{\mu}, \overline{\forall}, \overline{\exists} \,\}$ and F and F' are nonexpansive type functionals such that $F = F'$, then $\overline{c}_\eta F = \overline{c}_{\eta'} F'$.*

(2) Approximately Congruent A constructor is *approximately congruent* if, when applied to approximately equal arguments, it yields approximately equal results.

Lemma 5.10 (Pretypes Approximately Congruent)
For all pretype constructors $\overline{c}$ defined in Figure 5.4 and for all $k \geq 0$ and representation functions η, η' such that $\eta \approx_k \eta'$,

- *If $\overline{c} \in \{\, \overline{\bot}, \overline{\mathsf{unit}} \,\}$ then $\overline{c}_\eta \approx_k \overline{c}_{\eta'}$.*
- *For all τ, τ', if $\tau \approx_k \tau'$ then $\overline{\mathsf{ref}}_\eta(\tau) \approx_k \overline{\mathsf{ref}}_{\eta'}(\tau')$.*

- *For all* $\tau_1, \tau_1', \tau_2, \tau_2'$, *if* $\tau_1 \approx_k \tau_1'$ *and* $\tau_2 \approx_k \tau_2'$ *then* $\tau_1 \overline{\Rightarrow}_\eta \tau_2 \approx_k \tau_1' \overline{\Rightarrow}_{\eta'} \tau_2'$.
- *For all* F, F', *if* $\overline{c} \in \{\overline{\mu}, \overline{\forall}, \overline{\exists}\}$ *and* F *and* F' *are type functionals such that* $\mathrm{nxupto}(k, F)$, $\mathrm{nxupto}(k, F')$, *and* $F \approx_k F'$, *then* $\overline{c}_\eta F \approx_k \overline{c}_{\eta'} F'$.

(3) Closed Under State Extension Informally, let us say that a representation function is "well-behaved" if it maps type expressions to valid types. For every pretype constructor $\overline{c}$, if we are given a "well-behaved" representation function η, we want to be able to prove that $\overline{c}_\eta$ is a valid type, that is, $\mathrm{type}_\eta(\overline{c}_\eta)$. This property, however, cannot be proved directly; we will have to prove the approximate validity of types (upto level k) for all $k \geq 0$. Intuitively, this is because the representation function is stratified and defined inductively (see Section 5.2.3). Informally, if η is well-behaved at level k then we can show that for all pretype constructors $\overline{c}$, $\overline{c}_\eta$ is a valid type at level $k+1$ (since $\overline{c}$ relies on η up to level k only). Once we have proved the validity of all $\overline{c}_\eta$ at level $k+1$, we can show that η is well-behaved at level $k+1$, and so on.

Next, I define an approximate notion of type validity that requires only that a set τ be closed under state extension *upto* some $k \geq 0$ (rather than for all k).

Definition 5.11 (Type Up To)
A set τ *is a* type up to k *(where* $k \geq 0$*) with respect to a representation function* η *if it consists of tuples of the form* $\langle k', \Psi, v\rangle$ *where* $k' \geq 0$, *and if it is closed under state extension* ($\sqsubseteq_\eta$) *up to level* k:

$$\mathrm{typeupto}_\eta(k, \tau) \overset{\text{def}}{=} \forall j < k.\, \forall i, \Psi, \Psi', v.\\ (\langle j, \Psi, v\rangle \in \tau \ \wedge\ (j, \Psi) \sqsubseteq_\eta (i, \Psi')) \implies \langle i, \Psi', v\rangle \in \tau$$

We also need an approximate notion of well-behavedness for a representation function η which says that each τ in the codomain of η should a type up to k.

Definition 5.12 (Representation Function Well-Behaved Up To)
A representation function η *is* well-behaved up to k *iff for all type expressions* t, $\eta(t)$ *is a type up to* k *with respect to* η*; that is:*

$$\mathrm{repupto}(k, \eta) \overset{\text{def}}{=} \forall t.\ \mathrm{typeupto}_\eta(k, \eta(t))$$

Lemma 5.13 (Pretypes Closed Under State Extension)
For all pretype constructors $\overline{c}$ *defined in Figure 5.4 and for all* $k \geq 0$ *and representation functions* η, η' *such that* $\mathrm{repupto}(k, \eta)$ *and* $\eta \approx_k \eta'$,

- *If* $\overline{c} \in \{\overline{\bot}, \overline{\mathsf{unit}}\}$ *then* $\mathrm{typeupto}_\eta(k, \overline{c}_{\eta'})$.
- *For all* τ, *if* $\mathrm{typeupto}_\eta(k, \tau)$, *then* $\mathrm{typeupto}_\eta(k, \overline{\mathsf{ref}}_{\eta'} \tau)$.

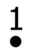
- *For all* τ_1, τ_2, *if* $\mathrm{typeupto}_\eta(k, \tau_1)$, *and* $\mathrm{typeupto}_\eta(k, \tau_2)$, *then* $\mathrm{typeupto}_\eta(k, \tau_1 \overline{\Rightarrow}_{\eta'} \tau_2)$.
- *Let* F *be a type functional such that:*

  1. $\mathrm{nxupto}(k+1, F)$*; and*
  2. *for all* τ, *if* $\mathrm{typeupto}_\eta(k, \tau)$ *then* $\mathrm{typeupto}_\eta(k, F(\tau))$.

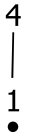
  If $\overline{c} \in \{\, \overline{\mu}, \overline{\forall}, \overline{\exists} \,\}$, *then* $\mathrm{typeupto}_\eta(k, \overline{c}_{\eta'} F)$.

5.2.3 Defining the Representation Function

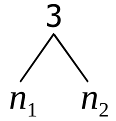
Having defined the pretype constructors, we are free to Gödelize them. But first, I must introduce some notation for representing trees of natural numbers. A tree constructor $\mathrm{tree}_i(c_0, t_1, \ldots, t_i)$ returns a tree with integer c_0 at the root and i subtrees $t_1, \ldots, t_i$, for example:



$\mathrm{tree}_0(1)$



$\mathrm{tree}_1(4, \mathrm{tree}_0(1))$



$\mathrm{tree}_2(3, n_1, n_2)$

For instance, suppose that the type constructors unit and ref are represented by tree nodes labeled with "2" and "4" respectively. Then, the type unit (which takes no arguments) may be represented by the tree $\mathrm{tree}_0(2)$, while the type $\mathsf{ref}\,\tau$ may be represented by the tree $\mathrm{tree}_1(4, t)$ where the tree t represents the type τ.

Before I present the definition of $\mathcal{G}$, I should note that for our FPCC implementation we have automated the generation of the Gödel-numbering function $\mathcal{G}$. We have a logic program written in Twelf that, given a template that supplies a one-to-one mapping from natural numbers to the set of pretype constructors, outputs the definition of $\mathcal{G}$. For the definitions and examples that follow, suppose that we have the following template.

`constructor`	1	$\overline{\bot}$
`constructor`	2	$\overline{\mathsf{unit}}$
`constructor`	3	$\overline{\Rightarrow}$
`constructor`	4	$\overline{\mathsf{ref}}$
`constructor`	5	$\overline{\forall}$
...		

The Gödel-numbering function $\mathcal{G}$ is defined inductively as follows.

- Level 0 of the hierarchy is a relation that maps every Gödel number to the bottom type, that is,
$$\mathcal{G}_0(t) \quad = \quad \{\}$$

- Level $i+1$ of the hierarchy is defined as follows:

$$\begin{array}{lcl}
\mathcal{G}_{i+1}(\mathrm{tree}_0(1)) & = & \overline{\bot}_{\mathcal{G}_i} \\
\mathcal{G}_{i+1}(\mathrm{tree}_0(2)) & = & \overline{\mathsf{unit}}_{\mathcal{G}_i} \\
\mathcal{G}_{i+1}(\mathrm{tree}_2(3, t_1, t_2)) & = & \mathcal{G}_i(t_1) \mathrel{\overline{\to}_{\mathcal{G}_i}} \mathcal{G}_i(t_2) \\
\mathcal{G}_{i+1}(\mathrm{tree}_1(4, t)) & = & \overline{\mathsf{ref}}_{\mathcal{G}_i}(\mathcal{G}_i(t)) \\
\ldots & &
\end{array}$$

I haven't shown the representation of $\overline{\mu} F$, $\overline{\forall} F$ and $\overline{\exists} F$ because this would require a Gödelization of type-functions as well as (closed) types. The way we handle this in the proof of a full-scale type system is to Gödelize types with free de Bruijn variables (which I refered to as "type expressions" in Section 4.4.1) instead of (closed) types.[2]

Some important properties of the representation function are as follows; I shall formally state the corresponding lemmas later.

- All representation levels are defined on the *same set* of type expressions, which constitutes the set of *valid* type expressions.

- Intuitively, each valid type expression t corresponds bijectively with a type τ freely built using the constructors of Figure 3.6, and increasing representation levels offer us increasingly better approximations of that type τ.

Automating Proofs for $\mathcal{G}$ I mentioned above that we have automated the definition of the Gödel-numbering function $\mathcal{G}$. Much more important, however, is the fact that we've also automated the proofs of various properties of $\mathcal{G}$. Generally, we manually prove the core clauses of each proof and then write a set of tactics and a logic program that glues together these proofs as appropriate, constructing a proof customized to the set of pretype constructors listed in the template (and hence, customized to the function $\mathcal{G}$ built for that set of constructors). In our proof implementation, all of the lemmas given in the rest of this section have been proved in such a fashion.

Lemma 5.14

- *For all $i \geq 0$, all type expressions in* $\mathrm{dom}(\mathcal{G}_i)$ *are valid type expressions.*

- *For all i, j such that $i \geq 0$ and $j \geq 0$,* $\mathrm{dom}(\mathcal{G}_i) = \mathrm{dom}(\mathcal{G}_j)$.

[2] In our FPCC system, since we Gödelize *open* types (that is, types with free de Bruijn variables), we then have to ensure that only *closed* types are stored in references.

- *The definition of $\mathcal{G}_{i+1}$ is extensional. Specifically, let η and η' be representation functions such that $\eta = \eta'$. If we apply the definition of $\mathcal{G}_{i+1}$ to η (i.e., replace all $\mathcal{G}_i$ in the definition with η), we get the same result as if we apply it to η'.*

Many of the properties that we wish to prove about the representation function may be proved by induction on the level of $\mathcal{G}$. For instance, suppose that one wants to prove some property P about the elements in the domain or the range (or both) of all $\mathcal{G}_i$. First, one defines P as a predicate of type $(\mathit{Nat} \times \mathit{TyExp} \times \mathit{Type}) \to o$. (Recall that the metalogical type of $\mathcal{G}$ is $\mathit{Nat} \to \mathit{TyExp} \to \mathit{Type}$.) Then, one proves that P holds for the base case — that is, if $\mathcal{G}_0$ maps some t to τ, then $P_0(t, \tau)$ holds. Next, one proves that P holds for the inductive step — that is, assuming that for all t and τ, $\mathcal{G}_i(t) = \tau$ implies $P_i(t, \tau)$, one shows for each of the cases in the definition of $\mathcal{G}_{i+1}$ that for all t', τ', if $\mathcal{G}_{i+1}$ maps t' to τ', then $P_{i+1}(t', \tau')$ holds. (For example, for the $\overline{\mathsf{ref}}$ case in the definition of $\mathcal{G}_{i+1}$, one would have to show that if $P_i(t, \mathcal{G}_i(t))$ holds, then $P_{i+1}(\mathrm{tree}_1(4, t), \overline{\mathsf{ref}}_{\mathcal{G}_i}(\mathcal{G}_i(t)))$ also holds.) Finally, one uses the following generic induction principle for $\mathcal{G}$ to construct the proof that the property P holds for all $\mathcal{G}_i$.

Lemma 5.15 ($\mathcal{G}$ Induction)
Let P be a predicate of type $(\mathit{Nat} \times \mathit{TyExp} \times \mathit{Type}) \to o$. Suppose that we have the following:

1. *For all t, τ, if $\mathcal{G}_0(t) = \tau$ then $P_0(t, \tau)$.*
2. *For all $n \geq 0$, if: for all t, τ, $\mathcal{G}_n(t) = \tau$ implies $P_n(t, \tau)$, then: for all t, τ, $\mathcal{G}_{n+1}(t) = \tau$ implies $P_{n+1}(t, \tau)$.*

Then, for all $k \geq 0$ and all t, τ, if $\mathcal{G}_k(t) = \tau$ then $P_k(t, \tau)$ holds.

Using the generic induction lemma above, we've automated proofs of various lemmas, some of which I state next.

Lemma 5.16 ($\mathcal{G}$ Partial Function)
For all $k \geq 0$ and for all type expressions t_1, t_2, if $t_1 = t_2$ then $\mathcal{G}_k(t_1) = \mathcal{G}_k(t_2)$.

Lemma 5.17 (Representable Implies Type Up To)
For all $k \geq 0$ and for all t, τ such that $\mathcal{G}_{k+1}(t) = \tau$, $\mathrm{typeupto}_{\mathcal{G}_k}(k, \tau)$.

Proof (sketch): For the base case, we start by proving that $\mathrm{repupto}(0, \mathcal{G}_0)$. □

Corollary 5.18 ($\mathcal{G}$ Well-Behaved Up To)
As a corollary of Lemma 5.17 we have: If $j \leq k$ then $\mathrm{repupto}(j, \mathcal{G}_k)$.

$$\begin{array}{lcl}
\bot & \stackrel{\text{def}}{=} & \bigcup_k \lfloor \overline{\bot}_{\mathcal{G}_k} \rfloor_k \\
\mathsf{unit} & \stackrel{\text{def}}{=} & \bigcup_k \lfloor \overline{\mathsf{unit}}_{\mathcal{G}_k} \rfloor_k \\
\tau_1 \rightarrow \tau_2 & \stackrel{\text{def}}{=} & \bigcup_k \lfloor \tau_1 \overline{\rightarrow}_{\mathcal{G}_k} \tau_2 \rfloor_k \\
\mathsf{ref}\ \tau & \stackrel{\text{def}}{=} & \bigcup_k \lfloor \overline{\mathsf{ref}}_{\mathcal{G}_k} \tau \rfloor_k \\
\forall F & \stackrel{\text{def}}{=} & \bigcup_k \lfloor \overline{\forall}_{\mathcal{G}_k} F \rfloor_k \\
\exists F & \stackrel{\text{def}}{=} & \bigcup_k \lfloor \overline{\exists}_{\mathcal{G}_k} F \rfloor_k \\
\mu F & \stackrel{\text{def}}{=} & \bigcup_k \lfloor \overline{\mu}_{\mathcal{G}_k} F \rfloor_k
\end{array}$$

Figure 5.5: Higher-Order Logic: Type Definitions

As we can see from the sampling of lemmas in this section, proofs of properties of $\mathcal{G}$ are too tedious and too long to be modified every time we want to add a type to our language. Automating the definition of $\mathcal{G}$ and the proofs of the lemmas about $\mathcal{G}$ has made it possible for us to add new types to our language simply by adding the corresponding pretypes to the template.

5.2.4 Types

Each type τ may now be defined as the union (over all $k \geq 0$) of the k-approximations of the corresponding pretype ς (where ς has access to the representation function $\mathcal{G}_k$). The type definitions appear in Figure 5.5.

Properties of Types We have to prove that the types defined in Figure 5.5 are valid types, that is, that they are closed under state extension. The definition of valid type (Definition 5.3) given in Section 5.2.1 is parametrized by a representation function η. Now that we have defined the Gödel-numbering function we can give a definition of type not parametrized by η. To do so we first combine the hierarchy of representation functions $\mathcal{G}_i$ into a *total* representation function $\widehat{\mathcal{G}}$ (of type *Rep*) as follows.

$$\widehat{\mathcal{G}}(t) \stackrel{\text{def}}{=} \bigcup_k \lfloor \mathcal{G}_k(t) \rfloor_k$$

The following properties of $\widehat{\mathcal{G}}$ follow from Lemma 5.16 and Corollary 5.18.

Lemma 5.19
($\widehat{\mathcal{G}}$ Partial Function) *For all type expressions t_1, t_2, if $t_1 = t_2$ then $\widehat{\mathcal{G}}(t_1) = \widehat{\mathcal{G}}(t_2)$.*

($\widehat{\mathcal{G}}$ Well-Behaved up to) *For all $k \geq 0$, the total representation function $\widehat{\mathcal{G}}$ is well-behaved up to k, i.e., $\forall k \geq 0.\ \mathrm{repupto}(k, \widehat{\mathcal{G}})$.*

Now we say that a set τ is a type iff $\mathrm{type}_{\widehat{\mathcal{G}}}(\tau)$. To prove this property, we can use Lemma 5.17 to show that for all $k \geq 0$, $\mathrm{typeupto}_{\widehat{\mathcal{G}}}(k, \tau)$, but only if we can first show that τ is *representable*.

Definition 5.20 (Representable)
A type τ is representable iff for each k, there exists some type expression t such that $\lfloor \tau \rfloor_k = \lfloor \widehat{\mathcal{G}}(t) \rfloor_k$; that is,

$$\mathrm{reppable}(\tau) \quad \stackrel{\mathrm{def}}{=} \quad \forall k \geq 0.\ (\exists t.\ \tau \approx_k \widehat{\mathcal{G}}(t))$$

Note that in our current semantics, when we allocate a reference cell of type τ, we extend the store typing Ψ with a new location ℓ mapped to a type expression t such that t represents τ. If a type τ is not representable, then it cannot be stored in a reference cell.

Theorem 5.21 (Types Representable)
Every type that can be constructed using the type constructors in Figure 5.5 is representable.

Theorem 5.22 (Types Valid)
Every type τ that can be constructed using the type constructors in Figure 5.5 is a valid type.

5.2.5 Judgments, Typing Rules, Safety

The semantics of judgments can now be defined using the total representation function $\widehat{\mathcal{G}}$ as follows.

Definition 5.23 (Semantics of Judgment)
For any type environment Γ and value environment σ I write $\sigma :_{\eta,k,\Psi} \Gamma$ ("σ approximately obeys Γ") if for all variables $x \in \mathrm{dom}(\Gamma)$ we have $\sigma(x) :_{\eta,k,\Psi} \Gamma(x)$; that is,

$$\sigma :_{\eta,k,\Psi} \Gamma \quad \stackrel{\mathrm{def}}{=} \quad \forall x \in \mathrm{dom}(\Gamma).\ \sigma(x) :_{\eta,k,\Psi} \Gamma(x)$$

I write $\Gamma \models^k_M e : \tau$ iff $FV(e) \subseteq \mathrm{dom}(\Gamma)$ and

$$\forall \sigma, \Psi.\ \sigma :_{\widehat{\mathcal{G}},k,\Psi} \Gamma \implies \sigma(e) :_{\widehat{\mathcal{G}},k,\Psi} \tau$$

Theorem 5.24 (Validity of Typing Rules + Safety)
Using these definitions as the interpretation of the typing operators, all the typing rules in Figure 3.7 hold, as does the statement of Theorem 3.8 (typability implies safety).

Proof: The proof corresponds closely to the proof shown in Sections 3.4 and 3.5. We are implementing a machine-checked version of this higher-order-logic proof in the Twelf system. That proof is for von Neumann machines instead of for lambda-calculus, since our application is in proof-carrying code for a real machine. □

Chapter 6

Foundational Proof-Carrying Code

I have shown how to construct proofs of safety for untyped λ-calculus programs. For a foundational proof-carrying code system, however, we need proofs of safety for real machine language programs. In this chapter, I shall give an overview of the Foundational Proof-Carrying Code (FPCC) project at Princeton and explain some of the differences between the semantic model of λ^M and the von Neumann model that we've built for our FPCC system. I'll conclude with a brief look at foundational proof-carrying code systems that establish type soundness using a syntactic rather than semantic approach.

6.1 Overview of the FPCC System

The goal of the FPCC project at Princeton is to build machine-checkable safety proofs for machine language programs from the minimal set of axioms. Figure 6.1 illustrates the architecture of our FPCC system; the trusted components are shaded. The FPCC system consists of a type-preserving compiler that compiles core ML into Sparc machine code and simultaneously produces a typed assembly language program. This typed assembly language is called LTAL (for Low-level Typed Assembly Language) [CWAF03, Che04]. LTAL is quite complex since it must be able to express the constructs of a real source language (core ML) compiled by a real compiler (a variant of SML/NJ) to a real target machine (the Sparc). Furthermore, it is specialized to the target machine architecture as well as the set of assumptions and conventions built into the compiler that produces it.

LTAL serves as an interface between the compiler and the FPCC checker. Chen *et al.* [CWAF03, Che04] have designed LTAL so that it is gives as much flexibility as possible to an optimizing compiler, and yet makes it possible to generate safety proofs for machine code. To achieve this goal, LTAL must be able to describe the machine state even, for instance, part-way through a sequence of instructions that allocates on the heap or a sequence of instructions that does data-type tag

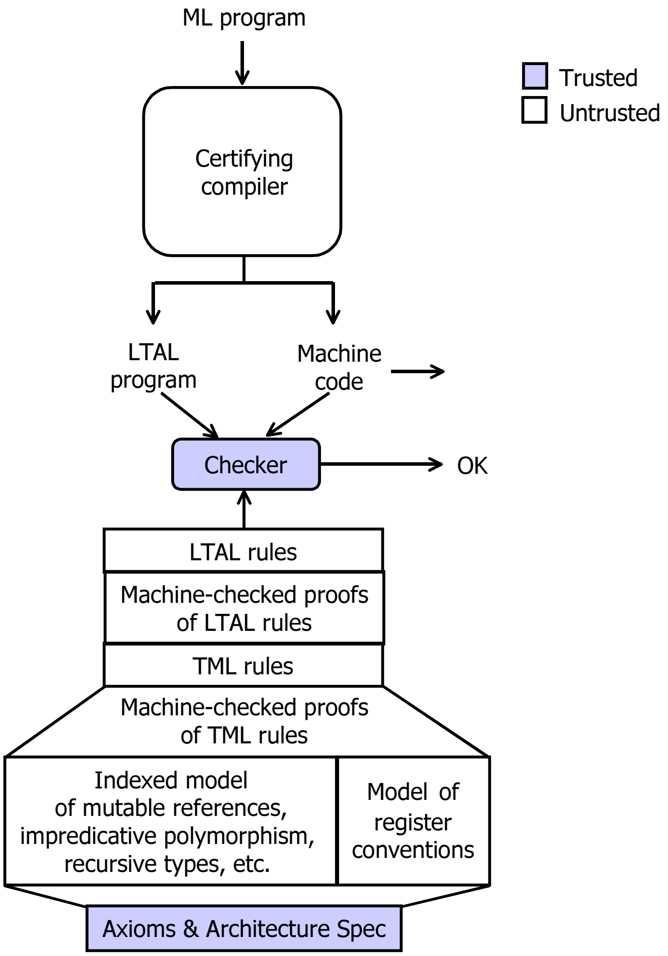

ML program
Trusted
Untrusted
Certifying compiler
LTAL program
Machine code
Checker
OK
LTAL rules
Machine-checked proofs of LTAL rules
TML rules
Machine-checked proofs of TML rules
Indexed model of mutable references, impredicative polymorphism, recursive types, etc.
Model of register conventions
Axioms & Architecture Spec

Figure 6.1: Foundational PCC System

discrimination. Hence, LTAL has no "macro" instructions; each LTAL instruction corresponds to one Sparc instruction (or is a coercion with no runtime effect). Also, LTAL gives the compiler the flexibility of choosing data representations (e.g., for tagged disjoint sums) by providing low-level type constructors that support various data representations and extraction and checking of tags. (Chen *et al.* [CWAF03] provide a detailed comparison of various typed assembly languages.)

Type-checking in LTAL is syntax directed — that is, Chen *et al.* use explicit coercions to guide the type-checking instead of relying on subtyping which would require a search. The LTAL typing rules are encoded as 4,000 lines of deterministic Prolog that reads the machine code and the LTAL and typechecks it. Since the

LTAL rules are syntax directed, the type checker (running as a logic program) does not need to backtrack.

For our FPCC system, we built a machine-checkable foundational proof of soundness for LTAL. We wanted to do construct this proof in a modular fashion, so we designed Typed Machine Language (TML) [Swa03, TASW04, TA04] to serve as an abstraction layer that hides the complex semantic models of types that one needs in order to construct a foundational proof. TML provides a rich set of constructors for types and instructions, and a powerful, orthogonal set of type primitives such as immutable and mutable references, recursive types, first-order continuations, impredicative existential and universal types, union and intersection types, and many other constructors that are useful in real typed assembly languages. Note that TML is not syntax directed; the presence of impredicative quantified types, equi-recursive types, unions and intersections, etc. makes type checking undecidable. However, TML is very useful for building semantic models of specialized, expressive, syntax-directed typed assembly languages. We build a semantic model (and soundness proof) for LTAL by combining together the TML primitives.

The FPCC system has axioms that specify the operational semantics of the Sparc [MA00] (1,600 lines of Twelf), as well as axioms for higher-order logic (135 lines of Twelf) and arithmetic (160 lines of Twelf). The specification of safety (see Section 6.2) is also part of the TCB (105 lines of Twelf). Based on these axioms, we have a (nearly complete) soundness proof of TML and LTAL that is about 124,000 lines of Twelf. The semantic model and machine-checked proofs of mutable references, impredicative polymorphism, etc. that I've described in this thesis — all suitably adapted to a von Neumann setting — are part of the TML soundness proof. In Section 6.3 I shall briefly describe the semantics of TML types and instructions.

Finally, the FPCC system has a small LF proof-checker and interpreter that is about 1,200 lines of C [WAS03]. This loads the Prolog rules for LTAL and checks them for soundness by loading and checking the Twelf proof of LTAL and TML soundness. It then interprets these rules to type-check an LTAL program from the ML compiler. This component is based on the work of Wu, Appel, and Stump [WAS03] who show how to build a trustworthy proof checker with small witnesses in two steps. First, one defines a language for *proof-schemes*. These represent the part of the foundational proof that only needs to be checked once; in FPCC, the proof schemes are the LTAL typing rules. One also provides a way to represent and check soundness theorems for the proof schemes. Next, one implements an interpreter to execute the proof scheme on the theorem and witness. The *proof witness* is that part of the foundational proof that is different for each machine language program; in FPCC, the witness is the LTAL program.

The size of the trusted computing base for the FPCC system is about 3000 lines. The size of the TCB in Necula's PCC [CLN$^+$00, NR01] is about 26,000 lines. The reader should refer to Appel and Wang [AW02] for a breakdown of the sizes of various trusted components, and Wu *et al.* [WAS03] for the sizes of proof schemes and proof witnesses, for both FPCC and Necula's PCC [CLN+00, NR01, NL98a, NL98b].

6.2 Von Neumann Model and Safety

A real machine language program runs on a real machine. The program's proof of type-soundness will therefore be relative to that machine's operational semantics. We'll assume that a real machine is a von Neumann machine such as the Sparc or the Pentium. The first step, therefore, is to build a model of a von Neumann machine. In this model, a *real machine state* comprises a register bank r and a memory m, each of which is a function from integers (register numbers or memory addresses) to integers (contents). We let the register $r(\textsc{pc})$ represent the program counter. Program execution begins in some initial state (r_0, m_0) such that the program (a sequence of machine instructions) is loaded at address ℓ_0 and $r_0(\textsc{pc}) = \ell_0$.

A machine instruction ι is modeled as a relation between machine states (r, m) and (r', m'): executing instruction ι in machine state (r, m) results in machine state (r', m'), provided that the execution does not violate the *safety policy.* For example, suppose that the safety policy requires that "only addresses > 1000 may be updated." This is specified (as part of the safety policy) using the predicate $\text{writable} \stackrel{\text{def}}{=} \lambda x.\, x > 1000$. Then, the store instruction is defined as follows.

$$\begin{array}{lcl}\texttt{store}(i,j,c)((r,m),(r',m')) & \stackrel{\text{def}}{=} & \text{writable}(r(j)+c) \quad \wedge \quad m'(r(j)+c) = r(i) \quad \wedge \\ & & (\forall x \neq (r(j)+c).\, m'(x) = m(x)) \quad \wedge \quad r' = r\end{array}$$

Next, we specify the step relation $(r, m) \longmapsto (r', m')$ which formally describes a single instruction execution. It says that if the word in memory m at location $r(\textsc{pc})$ decodes to a valid instruction ι and if incrementing the program counter of register bank r gives us the register bank r'', then the instruction ι maps state (r'', m) to state (r', m'). The decode relation in the definition below maps machine words to machine instructions (such as $\texttt{store}(i, j, c)$ above).

$$\begin{array}{lcl}(r,m) \longmapsto (r',m') & \stackrel{\text{def}}{=} & \exists r'', \iota.\ \ \text{decode}(m(r(\textsc{pc})), \iota) \quad \wedge \\ & & \qquad r'' = r\,[\textsc{pc} := r(\textsc{pc}) + 1] \quad \wedge \quad \iota((r'',m),(r',m'))\end{array}$$

Therefore, the specification of a real machine, such as the Sparc, consists of two parts, a "syntactic" part that specifies the encoding of machine instructions (the decode relation) and a semantic part that specifies machine instruction semantics (the definitions of `store`, `load`, etc.). The Sparc specification is the work of Michael and Appel [MA00], who explain it in more detail.

Notice that the step relation $\longmapsto$ omits any transition that violates the safety policy, even if the real machine would be capable of executing it. Hence, from any state (r, m), such that the next instruction violates the safety policy, there will be no successor in the $\longmapsto$ relation.

A state (r, m) in which some program is loaded is *safe for k steps* if

$$\begin{aligned}\mathrm{safen}(k, r, m) \quad &\overset{\mathrm{def}}{=} \quad \forall r', m'.\ (r, m) \longmapsto^{k-1} (r', m') \\ &\qquad\qquad \Longrightarrow \exists r'', m''.\ (r', m') \longmapsto (r'', m'')\end{aligned}$$

A state is *safe*, written $\mathrm{safe}(r, m)$, if $\forall k \geq 0.\mathrm{safen}(k, r, m)$.

To prove the safety of machine language programs, we design a type system in which we can say that if the registers collectively satisfy some type $(r{:}\tau)$, and the program counter contains the address of a continuation that accepts that type $(r(\textsc{pc}) : \mathsf{codeptr}\,(\tau))$, then that state is safe. The soundness theorem is that typability implies safety.

6.3 Typed Machine Language

This section gives an overview of Typed Machine Language (TML) and its semantics. In particular, I shall focus on how the von Neumann model of TML differs from the semantic model of λ^M that I presented in previous chapters.

States An *abstract machine state* $s = (\Psi, r, m)$ consists of a real part (r, m) and a virtual part Ψ, called the *alloc-map*, which corresponds to the store typings that we saw in Chapter 5. The alloc-map has no run-time manifestation; it is a mapping from allocated locations to type expressions (i.e., the Gödel numbers of types).

Values Recall that in the model of λ^M, values were irreducible syntactic terms of the lambda-calculus. For the von Neumann model, we let values be vectors v, where v is a sequence of integers, or equivalently, a function from natural numbers to integers. I write $v(0)$ to denote the first slot of the vector. Consider the following examples of values on a von Neumann machine.

- Suppose that v represents a boxed pair value in state (Ψ, r, m). This means that $v(0)$ is the address of two words in memory such that $m(v(0))$ is the first element and $m(v(0) + 1)$ is the second.

- If v is a numeric value in state (Ψ, r, m), then $v(0)$ is equal to the integer to be represented.

- If v is a continuation value in state (Ψ, r, m), then $v(0)$ must be the address ℓ of some machine code within memory m.

The reason for using a vector instead of a scalar is to represent the entire set of arguments to a multi-argument function as a single value v. For example, in state (Ψ, r, m), if the set of arguments is the contents of the register bank then one would choose $v = r$. I shall write $\overrightarrow{n}$ to denote the constant vector whose every slot contains the value n.

Closed and Open Types In lieu of the type functions supported by λ^M (and used to construct types such as $\forall F$, $\exists F$, and μF), TML supports types with free de Bruijn variables. As a result, in TML one may express arbitrarily nested recursive and quantified types (see the discussion in Section 4.4.1). Hence, we need a notion of *closed types* φ as well as *open types* τ. A closed type is a set of tuples of the form $\langle k, s, v\rangle$ where k is an approximation index (i.e., a natural number), s is a state, and v is a value (i.e., a vector). An open type may have free type variables — we use de Bruijn indices starting at 0 — so it must be interpreted in an *environment* ρ that maps the variables. An environment ρ is a sequence of closed types. I write $\rho_\emptyset$ for the empty environment; I use $\cdot$ to add a closed type φ to the head of an environment ρ as follows.

$$\begin{array}{rcl}(\varphi \cdot \rho)(i) & \overset{\text{def}}{=} & \{\, \langle k, s, v\rangle \mid (i = 0 \ \wedge\ \langle k, s, v\rangle \in \varphi) \quad \vee \\ & & \qquad\qquad\quad (i > 0 \ \wedge\ \langle k, s, v\rangle \in \rho(i-1))\,\}\end{array}$$

An open type is a function from environments to closed types. Consider, for example, the de Bruijn variable $\underline{3}$; we represent this open type using the function $\lambda\rho.\, \rho(3)$, which picks the third closed type from the environment.

Representation Functions and Open Types Since an alloc-map Ψ is a function from locations ℓ to type expressions t, most of the definitions that follow will have to be parametrized by a representation function η that maps type expressions t to (open) types τ (as discussed in Chapter 5, Section 5.2.1).

Alloc-maps and Closed Types Though representation functions η map type expressions t to open types τ, the alloc-map Ψ may only map locations to *closed* types (as discussed in Section 4.4.2). For this reason, the definition of the k-approximation of an alloc-map (shown below), uses the empty environment when interpreting the open type $\eta(\Psi(\ell))$.

$$\lfloor \Psi \rfloor_{\eta,k} \overset{\text{def}}{=} \{\, \ell \mapsto \lfloor (\eta(t))\rho_\emptyset \rfloor_k \mid \Psi(\ell) = t \,\}$$

Well-Typed States A real machine state (r, m) is well-typed to approximation k with respect to an alloc-map Ψ (written $(r, m) :_{\eta,k} \Psi$)if every allocated location ℓ contains a value of the right type — i.e., the type that Ψ says it should have —

to approximation k. Note that since values in the von Neumann model are vectors, we have to fetch the contents of cell ℓ in memory m and construct a vector $\overrightarrow{m(\ell)}$ before we can check that this value belongs to the type specified by Ψ.

$$\begin{array}{rcl}(r,m) :_{\eta,k} \Psi & \stackrel{\text{def}}{=} & \text{dom}(\Psi) \subseteq \text{dom}(m) \ \wedge \\ & & \forall j < k.\, \forall \ell \in \text{dom}(\Psi). \langle j, (\Psi, r, m), \overrightarrow{m(\ell)} \rangle \in \lfloor \Psi(\ell) \rfloor_{\eta,k}\end{array}$$

Conventions The programs we are judging come from compilers that manage their registers, stack frame, allocation heap, and so on using certain low-level conventions. There are *convention invariants* that each computation state must satisfy (written $\text{stateconvention}(r, m)$), and other convention invariants that must be satisfied as a program steps from state to state (written $\text{extendconvention}((r, m), (r', m'))$). For instance, the *spill area* holds local variables that don't fit in registers and the *heap area* holds values of mutable and immutable reference types. Each compiler temporary (or "local variable") is represented either as a register or as a temporary value in the spill area. The predicate $\text{stateconvention}(r, m)$ requires that in machine state (r, m) the spill area is disjoint from the heap area; that both the heap and spill area are readable and writable according to the safety policy; that the temporary values in the spill area are laid out correctly in memory (e.g., memory locations that store the contents of different local variables should be disjoint); and so on. Meanwhile, $\text{extendconvention}((r, m), (r', m'))$ requires, for instance, that certain registers remain unchanged as we step from state (r, m) to state (r', m'). To indicate the preservation of low-level convention invariants between machine states, we write $\text{convention}((r, m), (r', m'))$.

$$\begin{array}{rcl}\text{convention}((r,m),(r',m')) & \stackrel{\text{def}}{=} & \text{stateconvention}(r,m) \ \wedge \ \text{stateconvention}(r',m') \\ & & \wedge \ \text{extendconvention}((r,m),(r',m'))\end{array}$$

Valid State Extension A safe program steps from state to state such as to preserve certain invariants facilitating the proof of safety. The extend-state relation ($\sqsubseteq$) specifies how a state may change during zero or more computation steps. Specifically, we can step from a state s that is well-typed to approximation k to a state s' by modifying the registers and memory in any way that preserves the conventions and leaves state s' well-typed to approximation $j \leq k$; the alloc-map may be extended, but it must leave the types of previously allocated locations unchanged to approximation j.

$$\begin{array}{l}(k, (\Psi, r, m)) \sqsubseteq_\eta (j, (\Psi', r', m')) \ \stackrel{\text{def}}{=} \\ \qquad j \leq k \ \ \wedge \ \ \text{convention}((r,m),(r',m')) \ \ \wedge \\ \qquad (\forall \ell \in \text{dom}(\Psi).\ \lfloor \Psi' \rfloor_{\eta,j}(\ell) = \lfloor \Psi \rfloor_{\eta,j}(\ell)) \ \ \wedge \\ \qquad (r,m) :_{\eta,k} \Psi \ \ \wedge \ \ (r',m') :_{\eta,j} \Psi'\end{array}$$

$$
\begin{array}{rl}
n ::= & 0 \mid 1 \mid 2 \mid \cdots \\
\tau ::= & \textsf{top} \mid \textsf{bottom} \mid \textsf{int} \\
& \mid \textsf{const}\ n \mid \textsf{var}\ n \\
& \mid \textsf{readable} \mid \textsf{writable} \mid \textsf{executable} \\
& \mid \textsf{amplify}\ \tau \\
& \mid \textsf{int}_{<}\ \tau \mid \textsf{int}_{=}\ \tau \mid \textsf{int}_{>}\ \tau \\
& \mid \textsf{plus}\ \tau_1\, \tau_2 \mid \textsf{times}\ \tau_1\, \tau_2 \mid \textsf{modulo}\ \tau_1\, \tau_2 \\
& \mid \textsf{intersect}\ \tau_1\, \tau_2 \mid \textsf{union}\ \tau_1\, \tau_2 \\
& \mid \textsf{singleton}\ \tau_1\, \tau_2 \mid \textsf{apply}\ \tau_1\, \tau_2 \\
& \mid \textsf{box}\ \tau \mid \textsf{ref}\ \tau \mid \textsf{codeptr}\ \tau \\
& \mid \textsf{rec}\ \tau \mid \textsf{forall}\ \tau \mid \textsf{exists}\ \tau
\end{array}
$$

Figure 6.2: TML Type Constructors

The development of the rest of the semantic model of TML closely resembles the presentation in Chapter 5. As in Chapter 5 we proceed by defining TML pretypes, then the Gödel-numbering relation $\mathcal{G}$, and then we construct TML types by supplying the pretype constructors with appropriate representation functions $\mathcal{G}_i$. There are minor differences in most of the definitions since we must account for closed versus open types, type environments, and the fact that types are now sets of tuples of the form $\langle k, (\Psi, r, m), v\rangle$ rather than tuples $\langle k, \Psi, v\rangle$ as in previous chapters. We say a closed TML type φ is *valid* if it is closed under state extension. We say an open TML type τ is *valid* if it is both nonexpansive *and* closed under state extension. We prove these properties by defining approximate notions of each property (e.g., nxupto and typeupto) and then showing that the property holds for all $k \geq 0$. The reader should refer to Appel *et al.* [ARSA04] for a presentation of the semantic model of TML that more closely resembles our FPCC implementation than what I have described in this section.

TML Type Constructors The full suite of TML type constructors is shown in Figure 6.2; these constructors are described in detail by Swadi [Swa03]. Next, I'll briefly describe some of the TML pretype constructors that we haven't seen before.

The TML pretypes defined below each take a representation function η and an environment ρ and return a closed TML type. As in Chapter 5, to distinguish pretype constructors from type constructors, I mark the former with an $\overline{\mathsf{overbar}}$.

$$\begin{array}{lcl}
\overline{\mathsf{const}}_\eta n & \stackrel{\text{def}}{=} & \lambda\rho.\ \{\, \langle k, s, v\rangle \ \mid\ v(0) = n \,\} \\
\overline{\mathsf{var}}_\eta n & \stackrel{\text{def}}{=} & \lambda\rho.\ \{\, \langle k, s, v\rangle \ \mid\ \langle k, s, v\rangle \in \rho(n) \,\} \\
\overline{\mathsf{writable}}_\eta & \stackrel{\text{def}}{=} & \lambda\rho.\ \{\, \langle k, s, v\rangle \ \mid\ \exists n.\ \langle k, s, v\rangle \in (\overline{\mathsf{const}}_\eta n)(\rho) \ \wedge\ \mathrm{writable}(n) \,\} \\
\overline{\mathsf{int}}_{<\eta}\ \tau & \stackrel{\text{def}}{=} & \lambda\rho.\ \{\, \langle k, s, v\rangle \ \mid\ \exists n_1, n_2.\ \langle k, s, \overrightarrow{n_1}\rangle \in \tau(\rho) \ \wedge \\
 & & \qquad\qquad\qquad\qquad\qquad \langle k, s, v\rangle \in (\overline{\mathsf{const}}_\eta n_2)(\rho) \ \wedge\ n_2 < n_1 \,\}
\end{array}$$

The type $\mathsf{const}\ n$ is the singleton integer type containing the number n; value v has type $\mathsf{const}\ n$ if the first slot of the vector contains n. The type $\mathsf{var}\ n$ is the nth de Bruijn index; value v has this type if it has the nth closed type in the environment ρ. A value v has type $\mathsf{writable}$ iff the safety policy permits updates at address $v(0)$. The type $\mathsf{int}_<\ \tau$ is the type of all numeric values v such that $v(0)$ is strictly less than the number n_1 of type τ.

As before, $\mathsf{ref}\ \tau$ is the type ascribed to mutable reference cells that contain a value of type τ. A value v has type $\mathsf{ref}\ \tau$ if the first slot of the vector v contains a location ℓ such that (1) the store typing Ψ says that ℓ must contain a value of type τ for at least $k-1$ steps, and (2) the contents of memory at location ℓ belong to type τ to approximation j for all $j < k$. Note that requirement (2) is missing from the definition of the λ^M type $\mathsf{ref}\ \tau$ in Chapter 3 (see Figure 3.6) because types in that model do not take a store (or memory) as an argument — that is, they are predicates on $\langle k, \Psi, v\rangle$ rather than $\langle k, \Psi, S, v\rangle$.

$$\begin{array}{lcl}
\overline{\mathsf{ref}}_\eta\tau & \stackrel{\text{def}}{=} & \lambda\rho.\ \{\, \langle k, (\Psi, r, m), v\rangle \ \mid\ \lfloor\Psi\rfloor_{\eta,k}(v(0)) = \lfloor\tau(\rho)\rfloor_k \ \wedge \\
 & & \qquad\qquad\qquad\qquad\quad \forall j < k.\ \langle j, (\Psi, r, m), \overrightarrow{m(v(0))}\rangle \in \tau(\rho) \,\}
\end{array}$$

The type $\mathsf{codeptr}\ \tau$ is the type of a first-order continuation, that is, an address that is safe to jump to at any point in the future as long the register bank r' at that future point satisfies the precondition τ.

$$\begin{array}{lcl}
\overline{\mathsf{codeptr}}_\eta\tau & \stackrel{\text{def}}{=} & \lambda\rho.\ \{\, \langle k, s, v\rangle \ \mid\ \forall\, \Psi', r', m', j \leq k. \\
 & & \qquad\qquad\quad (\ (k, s) \sqsubseteq_\eta (j, (\Psi', r', m')) \ \wedge \\
 & & \qquad\qquad\qquad r'(\textsc{pc}) = v(0) \ \wedge \\
 & & \qquad\qquad\qquad \langle j, (\Psi', r', m'), r'\rangle \in \tau(\rho)\ ) \\
 & & \qquad\qquad\quad \Longrightarrow \mathrm{safen}(j, r', m') \,\}
\end{array}$$

Vector Typings Given a value v, we would like to construct *vector typings*, written ϕ, that constrain each element in a vector v with a type. We define the TML type $\mathsf{singleton}$ which is a vector typing that constrains exactly one element of a vector, leaving the type of every other element of the vector unconstrained. Singleton vector pretypes are defined below; the type τ_1 is meant to represent the index of the element that must have the type τ_2.

$$\overline{\mathsf{singleton}}_\eta \tau_1 \tau_2 \quad \stackrel{\text{def}}{=} \quad \lambda\rho.\ \{\ \langle k, s, v\rangle \ \mid\ \exists i.\ \langle k, s, \overrightarrow{i}\ \rangle \in \tau_1(\rho)\ \ \wedge\ \ \langle k, s, \overrightarrow{v(i)}\rangle \in \tau_2(\rho)\ \}$$

We can construct vector typings by composing two or more $\mathsf{singleton}$ vector typings using the intersection type constructor ($\mathsf{intersect}$). For example, if we have a vector v such that $v(0) : \tau_1$ and $v(1) : \tau_2$, we describe the type of v using the following vector typing:

$$\mathsf{intersect}\,(\mathsf{singleton}\,(\mathsf{const}\ 0)\,\tau_1)\,(\mathsf{singleton}\,(\mathsf{const}\ 1)\,\tau_2)$$

I abbreviate the above vector typing to $\{0 : \tau_1, 1 : \tau_2\}$.

Instruction Typings Tan *et al.* [TASW04] show how to model instructions in TML. Machine instructions can be viewed at many levels of abstraction. At the lowest level, an instruction is just an integer n. At the next level, it is a relation ι on real machine states (r, m) and (r', m') — I'll call these TML instructions. In Section 6.2 we saw that we can relate integers n that encode machine instructions to TML instructions ι using the decode relation. In TML, to express the fact that a location ℓ contains an integer n that decodes to a TML instruction ι, we define the TML type $\mathsf{instr}(\iota)$. The semantics of the corresponding pretype constructor $\overline{\mathsf{instr}}$ is as follows.

$$\overline{\mathsf{instr}}_\eta\ \iota \quad \stackrel{\text{def}}{=} \quad \lambda\rho.\ \{\ \langle k, s, v\rangle \ \mid\ \exists n.\ \langle k, s, v\rangle \in (\overline{\mathsf{box}}(\overline{\mathsf{int}_=}(\overline{\mathsf{const}}\ n)))\rho \quad \wedge \quad \mathrm{decode}(n, \iota)\ \}$$

In the above definition, we require that code locations be of immutable reference type in order to prohibit self-modifying code.

At higher levels of abstraction, machine instructions may be viewed as relations on Hoare-logic style preconditions and postconditions, expressed in terms of types. Informally, a TML instruction may be described by two vector typings ϕ_1 and ϕ_2, where ϕ_1 describes some sufficient precondition (in terms of types) necessary for the safe execution of the instruction, and ϕ_2 describes some guaranteed postcondition that results from the execution of the instruction. For example, the precondition ϕ_1 and postcondition ϕ_2 for the TML instruction $\mathsf{add}\ s_1\ s_2\ d$ are given by

$$\begin{array}{rcl} \phi_1 & = & \{s_1 : \mathsf{int},\ s_2 : \mathsf{int}\} \cap \phi \\ \phi_2 & = & \{s_1 : \mathsf{int},\ s_2 : \mathsf{int}\} \cap \phi\,[d \mapsto \mathsf{int}]), \end{array}$$

where ϕ is a vector typing that does not specify a type for d — I write $d \notin \mathrm{dom}(\phi)$ to denote this last condition. We require that s_1 and s_2 have integer types in the precondition, and in the postcondition we extend ϕ by assigning d an integer type. (For readability, I've used the infix notation $\cap$ instead of $\mathsf{intersect}$, the TML intersection type constructor.)

In order to model control-flow instructions, we additionally need to know the safety condition necessary for the jump target. This is specified by Γ which maps program code locations ℓ to the precondition ϕ that must be satisfied to guarantee the safety of execution from ℓ. Therefore, to describe a control-flow instruction we need a map Γ (whose domain contains all possible jump targets), as well as the vector typings ϕ_1 and ϕ_2.

We formalize this last view of instructions using judgments of the form $\Gamma; \ell \vdash \{\phi_1\}\, \iota\, \{\phi_2\}$. The TML instruction judgment $\Gamma; \ell \vdash \{\phi_1\}\, \iota\, \{\phi_2\}$ says that instruction ι at address ℓ is well-formed with respect to precondition ϕ_1, postcondition ϕ_2, and the invariant map Γ. We include the instruction location ℓ in the instruction judgment in order to be able to compute the destination address for pc-relative jump instructions. The typing rule for the add instruction may be specified as follows.

$$\frac{d \notin \mathrm{dom}(\phi)}{\Gamma; \ell \vdash \{\phi \cap \{s_1 : \mathsf{int},\ s_2 : \mathsf{int}\}\}\ \mathsf{add}\ s_1\ s_2\ d\ \{\phi \cap \{d : \mathsf{int}\}\}}\ \mathsf{TML\text{-}add}$$

Notice that the postcondition in the above rule does not specify the type of s_1 and s_2. The above formulation suffices because we can simply pick a ϕ that specifies that s_1 and s_2 have type int without violating the precondition. Furthermore, in the event that the destination register is the same as one of the source registers, say $d = s_1$ (as is the case in the instruction $\mathtt{r_3} \leftarrow \mathtt{r_3} + \mathtt{1}$), we choose a ϕ that specifies the type of s_2 but not of s_1. This gives us the desired pre- and postconditions without violating the requirement that ϕ should not specify a type for d (denoted $d \notin \mathrm{dom}(\phi)$).

We have typing rules for other arithmetic, memory access, and control flow instructions, but I will not describe them here. Each of these instruction typing rules must be proved as a lemma. To prove the rules as lemmas, we define the semantics of instruction judgments. Informally, the semantics must connect the lower-level view of instructions as relations on machine states (r, m) and (r', m'), to the more abstract view of instructions as relations on Γ, ϕ_1, and ϕ_2. Tan and Appel [TA04] describe the model of TML instruction judgments and show how to prove various instruction typing rules as lemmas. They also define a typed calculus of instructions with the goal of making it easy (in terms of proof reuse) to add new instruction typing rules to the system. Their core calculus has rules for reasoning about a sequence of instructions. One can extend the core calculus with any instruction typing rule that has been proved as a lemma. As an example, we may want to add the following typing rule for add which says that if s_1 and s_2 are integers with values m and n then in the postcondition, d must be an integer equal to $m + n$.

$$\frac{d \notin \mathrm{dom}(\phi)}{\begin{array}{l}\Gamma; \ell \vdash \{\phi \cap \{s_1 : \mathsf{int}_{=}(\mathsf{const}\, m),\ s_2 : \mathsf{int}_{=}(\mathsf{const}\, n)\}\} \\ \qquad\qquad \mathsf{add}\ s_1\ s_2\ d \\ \qquad\qquad \{\phi \cap \{d : \mathsf{plus}(\mathsf{const}\, m)(\mathsf{const}\, n)\}\}\end{array}}\ \mathsf{TML\text{-}add'}$$

Proving Type Safety Informally, the safety theorem we must prove is as follows.
If (1) program p is loaded at ℓ_0 with $r_0(\textsc{pc}) = \ell_0$,
(2) the initial state (r_0, m_0) satisfies the program precondition ϕ_0, and
(3) the machine instructions in program p are well-typed,
then safe(r_0, m_0).

Machine instructions in program p are well-typed if all program code locations are safe with respect to their preconditions. These preconditions are produced by the certifying compiler which generates an LTAL program with typing annotations for each instruction. The preconditions can be expressed using vector-typings Γ that map each program code location ℓ_i to the precondition for its safe execution given by the type $\mathsf{codeptr}\, \phi_i$.

$$\Gamma \quad = \quad \{\, \ell_0 : \mathsf{codeptr}\, \phi_0,\ \ell_1 : \mathsf{codeptr}\, \phi_1, \ldots, \ell_n : \mathsf{codeptr}\, \phi_n \,\}$$

Since a program is a list of machine words, our proof obligation is now to show that each of these words decodes to an instruction that satisfies the preconditions specified by Γ. We use the decode prover of Michael and Appel to construct the following vector-typing for the list of words p loaded at ℓ_0 in memory m.

$$\Delta(p) \quad = \quad \{\, \ell_0 : \mathsf{instr}\, \iota_0,\ \ell_1 : \mathsf{instr}\, \iota_1,\ \ldots,\ \ell_n : \mathsf{instr}\, \iota_n \,\}$$

We define the TML subtype (or sub-vector-typing) relation $\subset$ as follows.

$$\begin{array}{lcl} \tau_1 \subset_k \tau_2 & \overset{\text{def}}{=} & \forall \rho, s, v.\ \langle k, s, v\rangle \in \tau_1(\rho) \implies \langle k, s, v\rangle \in \tau_2(\rho) \\ \tau_1 \subset \tau_2 & \overset{\text{def}}{=} & \forall k.\, \tau_1 \subset_k \tau_2 \end{array}$$

Now the proof obligation that $\Delta(p)$ respects the invariants in Γ can be expressed as $\Delta(p) \subset \Gamma$. The proof of $\Delta(p) \subset \Gamma$ proceeds by induction on the approximation index k. See Tan *et al.* [TASW04] for a detailed description of the proof technique.

6.4 Related Work: Syntactic Approach to FPCC

Unlike the FPCC project at Princeton which takes a *semantic* approach to proving program safety, more recent foundational PCC systems are based on the *syntactic*

$$\begin{tikzcd}
\vdash P \arrow[r, Rightarrow, "\textit{translate}"] \arrow[d, "(\textit{evaluate})"] & S \arrow[d, "(\textit{step})"] \\
\vdash P' \arrow[r, Rightarrow, dashed, "\textit{translate?}"] & S'
\end{tikzcd}$$

Figure 6.3: Syntactic FPCC: Connecting TAL Evaluation to Real Machine Steps

approach. Hamid *et al.* [HST$^+$02] were the first to demonstrate the syntactic approach to foundational PCC where one constructs proofs of program safety in two steps. First, one proves the soundness of the typed assembly language as a syntactic metatheorem [WF94]. This theorem says that a program in a well-typed state can always take another step (*progress*), and that the execution of such a step results in a well-typed state (*preservation*).

Next, one relates steps in the typed assembly language to steps of the real machine. Specifically, one must define a translation from the TAL machine states P to real machine states S, and then prove the following lemma: if a well-typed TAL state P translates to a real machine state S, and P steps to P' according to the TAL operational semantics, and S steps to S' on the real machine, then P' translates to S'. Figure 6.3 due to Hamid *et al.* [HST$^+$02] illustrates this lemma.

The typed assembly language mentioned above is designed with essentially the same design goals as LTAL. For instance, it should be expressive enough to support many different source language constructs and compiler optimizations. Also, each TAL instruction should correspond to one real machine instruction so one can easily prove that each TAL step corresponds to a real machine step. Hamid *et al.*'s syntactic FPCC system compiles source language programs to Featherweight Typed Assembly Language (FTAL). They split the conventional "malloc" instruction into two FTAL instructions that each map to a single real machine instruction. This, in turn, requires that their type system be expressive enough to describe the machine state even part way through the sequence of instructions that allocates on the heap. FTAL is a simple typed assembly language that supports memory allocation, mutable references, and recursive types, but not quantified types. Their system targets a toy subset of a real machine architecture. Proofs are encoded in the Calculus of Inductive Constructions (CiC) [CH88, PM93] and checked using Coq [BBC$^+$98].

Crary [Cra03] has recently extended Hamid *et al.*'s approach to a realistic architecture. Crary's system uses an assembly language called TAL Two (TALT), a descendant of TALx86 [MCG^{+}99, MWCG99]. While TALT is expressive enough to serve as a target language for a type-preserving ML compiler, it must still use macro instructions like "malloc." Consequently, there is a gap between TALT instructions and real machine instructions. Crary's FPCC system targets the Intel IA32 architecture. Crary uses LF [HHP93] to encode proofs and the Twelf metatheorem checker [PS99, Sch00b] to check them. The latter is a large and complex piece of software, and using it to validate safety proofs increases the size of the TCB. The pure LF checker used in our (semantic) FPCC system is simpler and smaller than existing CiC checkers and the Twelf metatheorem checker.

Chapter 7

Semantics of a Region Calculus

All practical languages provide some mechanism for the reuse of memory, but the model of mutable state that I've described in this thesis does not allow memory to be reclaimed. Deallocation or recycling of memory implicitly allows memory to be reused later, possibly at a *different* type. This is clearly at odds with the type invariance principle I described in Chapter 1 which ensures that the types of allocated locations are preserved. Type preservation is important because it allows us to prove that the logical relations we've defined are Kripke monotone (see Section 4.4.3). The problem is that in order to support memory reuse we must define logical relations that allow memory cells to be reused *at a different type* and yet, paradoxically, are Kripke monotone, that is, they ensure that *types are preserved*.

In this chapter, I'll show how one may go about constructing such a model. In particular, I'll describe how to model a low-level lambda-calculus with primitives for region-based memory management [TT94, TT97]. The central ideas underlying the model described in this chapter were developed during discussions with David Walker.

The main idea is as follows. First, in any given computation state s, I allow each location ℓ to have more than one type — for instance, ℓ may simultaneously have the types $\mathsf{ref}\,\tau_1\,\mathsf{at}\,\nu_1$ and $\mathsf{ref}\,\tau_2\,\mathsf{at}\,\nu_2$, where ν_1 and ν_2 are the names of distinct regions in which the location was allocated at some point during the computation prior to reaching state s — as long as, if X is the set of regions in which ℓ has been allocated thus far, there is at most one region ν_i in X that is "live" in the current state s.[1] (Ahmed, Jia, and Walker [AJW03] also use *live* and *dead* predicates and have a similar invariant on regions.) The point is to ensure that if ℓ has type $\mathsf{ref}\,\tau_1\,\mathsf{at}\,\nu_1$ in the current state, it'll continue to have that type in all future states, even after the region ν_1 has been deallocated. Second, I use a capability-based type system [CWM99, WCM00, Wal01] (which keeps track of the set of live regions) to statically check that a region is live when it's accessed. For instance, the typing rule

[1] A region is *live* if it's been allocated but not yet freed. Once a live region is freed, it's *dead*.

$$
\begin{array}{llcl}
\textit{Values} & v & ::= & x \mid \ell \mid h \mid \texttt{unit} \mid \lambda(x_1,\ldots,x_n).e \\[1em]
\textit{Expressions} & e & ::= & \texttt{let}\ x = \texttt{new}(v_1)\ \texttt{at}\ v_2\ \texttt{in}\ e \mid \\
 & & & \texttt{let}\ x = !\,v\ \texttt{in}\ e \mid \texttt{let}\ v_1 := v_2\ \texttt{in}\ e \mid \\
 & & & \texttt{let}\ \texttt{newrgn}\ x\ \texttt{in}\ e \mid \texttt{let}\ \texttt{freergn}\ v\ \texttt{in}\ e \mid \\
 & & & v(v_1,\ldots,v_n) \mid \texttt{halt}
\end{array}
$$

Figure 7.1: Regions (λ^R): Syntax

for deferencing says that $!\,v$ has type τ if and only if the value v has type $\mathsf{ref}\ \tau\ \mathsf{at}\ \nu$ *and* the region ν is *live*.

7.1 Syntax

The language I consider is based on the Calculus of Capabilities [CWM99, WCM00, Wal01], a continuation-passing-style language motivated by the goal of using region-based memory management to certify the memory safety of object code. I'll discuss only a simplified version of that language here. In particular, I won't show how to model region polymorphism and quantified types; these are crucial features, but they can be added to the model I present without too much difficulty.

The syntax of the language, which I call λ^R, is given in Figure 7.1. As before, I use the metavariables x and ℓ to range over the countably infinite sets *Var* (of variables) and *Loc* (of locations), respectively. Furthermore, I use the metavariable h to range over a countably infinite set *RegionHandle* of region handles. Later I will use the metavariable ν to range over a countably infinite set of region names *RegionName*. Informally, the crucial distinction between region handles h and region names ν is that region handles may be reused once a region has been deallocated, while region names are never reused. Note that the syntax for λ^R does not make use of region names. I'll discuss the use of region handles versus region names when I describe the operational semantics below.

A term v is a value if it is a variable x, a location ℓ, a region handle h, the constant `unit`, or a function $\lambda(x_1,\ldots,x_n).e$. The language includes terms for allocating (and initializing) a new location in a specified region, dereferencing, update, creating a new region, deallocating a region, and function application. I omit all types from the syntax of λ^R (just as I did for λ^M).

7.2 Operational Semantics

The small-step operational semantics for λ^R is given by the relation $(S, e) \longmapsto_R (S', e')$ between abstract machine states. An abstract machine state (S, e) is a pair of a store S and a closed term e, but stores now have a more complicated structure. A *store* S in λ^R is a finite map from region handles to regions. A *region* is a finite map from locations to closed values.

$$\begin{array}{lcl} \mathit{Region} & = & \mathit{Loc} \overset{\text{fin}}{\to} \mathit{Val} \\ \mathit{Store} & = & \mathit{RegionHandle} \overset{\text{fin}}{\to} \mathit{Region} \end{array}$$

I shall assume that all stores are well-formed, that is, that there is no location ℓ that simultaneously belongs to more than one region in the store.

The reader may wonder why I've chosen to represent stores as a finite map from *region handles* to regions rather than as a finite map from *region names* to regions (which is the representation used by Walker *et al.* [WCM00, Wal01]). The reason is that I want to keep the dynamic semantics completely free of region names. The requirement that region names never be reused is a critical component (as I'll explain in detail below) of the semantic model I wish to construct. However, on a von Neumann machine there is no infinite supply of region names. Since my ultimate goal is to construct a von Neumann model based on the ideas in this chapter, I shall show that it is possible to make use of only (recyclable) region handles in the dynamic semantics and restrict the use of (nonrecyclable) region names to the static semantics.

The operational semantics for λ^R is given in Figure 7.2. When convenient, I abbreviate $S(h)(\ell)$ to $S(h.\ell)$ and $S\,[h \mapsto S(h)\,[\ell \mapsto v]]$ to $S\,[h.\ell \mapsto v]$. I write $S\backslash h$ to denote a store that does not map h (that is, $h \notin \mathrm{dom}(S\backslash h)$), but is otherwise equivalent to the store S.

In a language with region-based memory management, one must specify the region in which to allocate a new cell. The declaration $\texttt{new}(v)\,\texttt{at}\,h$ allocates a reference cell in the region with handle h and initializes the cell's contents to the value v. The corresponding rule (`RO-newat`) checks that store S contains the specified region handle h. It then extends that region in the store with a previously unallocated location ℓ initialized to v. Note that the rule requires that the new location ℓ does not occur in *any* existing region of the store; this guarantees the well-formedness of the extended store. The rules for dereferencing and updating a location ℓ (see `RO-deref` and `RO-assign`) check that there exists a region in the store that contains the location. The rule for creating a new region (`RO-newrgn`) extends the store with a region handle h that is not in current use, mapped to an empty region. The declaration $\texttt{freergn}\,h$ deallocates the region with handle h. The corresponding rule (see `RO-freergn`) checks that the store contains the region handle h mapped to some

$$\frac{h \in \mathrm{dom}(S) \qquad \forall h'.\, \ell \notin \mathrm{dom}(S(h'))}{(S,\, \texttt{let}\ x = \texttt{new}(v)\ \texttt{at}\ h\ \texttt{in}\ e) \longmapsto_R (S\,[h.\ell \mapsto v],\, e[\ell/x])}\ \texttt{(RO-newat)}$$

$$\frac{\exists h.\, (h \in \mathrm{dom}(S)\ \wedge\ \ell \in \mathrm{dom}(S(h)))}{(S,\, \texttt{let}\ x = \texttt{!}\,\ell\ \texttt{in}\ e) \longmapsto_R (S,\, e[S(h.\ell)/x])}\ \texttt{(RO-deref)}$$

$$\frac{\exists h.\, (h \in \mathrm{dom}(S)\ \wedge\ \ell \in \mathrm{dom}(S(h)))}{(S,\, \texttt{let}\ \ell := v\ \texttt{in}\ e) \longmapsto_R (S\,[h.\ell \mapsto v],\, e)}\ \texttt{(RO-assign)}$$

$$\frac{h \notin \mathrm{dom}(S)}{(S,\, \texttt{let}\ \texttt{newrgn}\, x\ \texttt{in}\ e) \longmapsto_R (S\,[h \mapsto \{\}],\, e[h/x])}\ \texttt{(RO-newrgn)}$$

$$\frac{h \in \mathrm{dom}(S)}{(S,\, \texttt{let}\ \texttt{freergn}\, h\ \texttt{in}\ e) \longmapsto_R (S\backslash h,\, e)}\ \texttt{(RO-freergn)}$$

$$\frac{}{(S,\, (\lambda(x_1,\ldots,x_n).e)v_1,\ldots,v_n) \longmapsto_R (S,\, e[v_1,\ldots,v_n/x_1,\ldots,x_n])}\ \texttt{(RO-app)}$$

$$\frac{}{(S,\, \texttt{halt}) \longmapsto_R (S,\, \texttt{unit})}\ \texttt{(RO-halt)}$$

Figure 7.2: Regions (λ^R): Operational Semantics

region and removes the region and h from the store.

The specification of safety is exactly as for λ^M; refer to Definition 3.1 (in Section 3.3.1) and replace all occurrences of $\longmapsto_M$ with $\longmapsto_R$.

7.3 Semantic Model of Types

Types in λ^M were modeled as sets of tuples of the form $\langle k, \Psi, v\rangle$ where k is an approximation index, Ψ is a store typing, and v is a value. But stores in λ^R are more complicated than those in λ^M, and accordingly, the model of store typings is also more complicated. There are a number of things that the semantic model must keep track of as I shall explain next.

Regions are uniquely identified by region names ν. A store typing Ψ must keep track of information about each region, live or dead. Hence, I shall model a store typing Ψ as a finite map from region names ν to the appropriate information for that region. Notice that since a store typing must keep track of information about both live and dead regions, the domain of a store typing can never shrink.

There are three things we must keep track of for each region. First, we must keep track of whether the region is *live* or *dead*. The second piece of information to track is a region's handle. Specifically, when a new a region is allocated, the static semantics (as we shall see) gives the region a fresh region name (i.e., a name that has never been used before). Meanwhile, the dynamic semantics assigns the new region a region handle h. We must keep track of the region handle h assigned to each region ν (so that we can, for instance, define the semantics of region handle types $\mathsf{handle}\,(\nu)$ which I'll describe below). Third, for each location allocated in the region, we must keep track of the (approximate) permissible update type of the reference cell, just as we did for λ^M.

A *region typing* Υ is a finite map from locations to types. A store typing Ψ is a finite map from region names to tuples of the form (ω, h, Υ), where the metavariable ω ranges over the set $\{\, \mathit{live}, \mathit{dead} \,\}$, h is a region handle, and Υ is a region typing. The metalogical types of region typings, store typings, and types are as follows.

$$\begin{array}{lcl}
\mathit{Type}_0 & = & \mathit{Unit} \\
\mathit{RegionType}_k & = & \mathit{Loc} \overset{\text{fin}}{\to} \mathit{Type}_k \\
\mathit{StoreType}_k & = & \mathit{RegionName} \overset{\text{fin}}{\to} (\{\, \mathit{live}, \mathit{dead} \,\} \times \mathit{RegionHandle} \times \mathit{RegionType}_k) \\
\mathit{Type}_{k+1} & = & \mathit{StoreType}_k \times \mathit{Val} \to o
\end{array}$$

If $\Psi(\nu) = (\omega, h, \Upsilon)$ then $\mathrm{status}(\Psi(\nu))$ denotes ω; $\mathrm{live}(\Psi(\nu))$ is true if and only if $\omega = \mathit{live}$; $\mathrm{dead}(\Psi(\nu))$ is true if and only if $\omega = \mathit{dead}$; $\mathrm{hndl}(\Psi(\nu))$ denotes h; and $\mathrm{rgnty}(\Psi(\nu))$ denotes Υ.

There are certain well-formedness constraints that have to be imposed on store typings. The constraints may be better understood if I first explain how I intend to model cell allocation as well as the allocation and deallocation of regions in the semantics of λ^R. Consider a scenario where we have a current store typing Ψ and a location ℓ that has never been allocated in any region. It follows that ℓ does not occur in any region in the current store typing Ψ. Now, suppose that we execute an instruction that allocates a new cell — say we pick ℓ — to contain values of type τ_1 in region ν_1. It must be the case that $\mathrm{live}(\Psi(\nu_1))$ — that is, we cannot allocate in a region that is *dead* or nonexistent. In the semantics, we update the store typing to Ψ' which is identical to Ψ, except that $\mathrm{rgnty}(\Psi'(\nu))$ also maps ℓ to the appropriate approximation of type τ_1.

Next, suppose that ν_1 is freed. In the semantics, Ψ' is updated to Ψ'' which is identical to Ψ' except for the fact that $\text{dead}(\Psi''(\nu_1))$. Note that region ν_1 in the current store typing Ψ'' contains the location ℓ mapped (approximately) to type τ_1. Nonetheless, since $\text{dead}(\Psi''(\nu_1))$, one may now *reallocate* ℓ at a different type τ_2 in some other region ν_2 as long as $\text{live}(\Psi''(\nu_2))$. In the future, once ν_2 has been freed (and marked *dead*), ℓ may be reallocated again in a different region, possibly at a different type, as long as every region in which ℓ was allocated up till that point is *dead*. The recurring requirement in the above scenario is that each location ℓ may belong to at most one *live* region in the store typing. Similary, we require that each region handle h may be assigned to at most one *live* region in Ψ. A store typing that satisfies these requirements is said to be *well-formed*.

Definition 7.1 (Well-formed Store Type)
A store typing Ψ is well-formed *if and only if there exists no location ℓ or region handle h that belongs to multiple* live *regions in Ψ; that is,*

$$\begin{array}{lcl}\text{wellformed}(\Psi) & \overset{\text{def}}{=} & \forall \ell, \nu_1, \nu_2.\ (\ (\ (\ell \in \text{dom}(\Psi(\nu_1)) \ \wedge\ \ell \in \text{dom}(\Psi(\nu_2))) \\ & & \qquad\qquad\quad \vee\ \text{hndl}(\Psi(\nu_1)) = \text{hndl}(\Psi(\nu_2))\ ) \\ & & \qquad\qquad \wedge\ \nu_1 \neq \nu_2\ ) \\ & & \qquad \Longrightarrow\ (\text{dead}(\Psi(\nu_1))\ \vee\ \text{dead}(\Psi(\nu_2)))\end{array}$$

We've seen that a location may be added to a store typing multiple times — intuitively, every time the location is (re)allocated — as long as the well-formedness of the store typing in preserved. To model λ^R we must require, in fact, that locations never be removed from the store typing. This allows us to maintain the illusion that reference types are (approximately) preserved, even when their regions are freed. The requirement that locations never be removed from the store typing is enforced by the extend-state relation ($\sqsubseteq$) relation. Before I can specify the extend-state relation, however, I must define the notion of the k-approximation of a type.

Definition 7.2 (Approx)
The k-approximation of a set is the subset of its elements whose index is less than k; I also extend this notion pointwise to region typings and store typings:

$$\begin{array}{lcl}\lfloor \tau \rfloor_k & \overset{\text{def}}{=} & \{\, \langle j, \Psi, v \rangle \mid j < k \ \wedge\ \langle j, \Psi, v \rangle \in \tau \,\} \\ \lfloor \Upsilon \rfloor_k & \overset{\text{def}}{=} & \{\, (\ell \mapsto \lfloor \tau \rfloor_k) \mid \Upsilon(\ell) = \tau \,\} \\ \lfloor \Psi \rfloor_k & \overset{\text{def}}{=} & \{\, (\nu \mapsto (\omega, h, \lfloor \Upsilon \rfloor_k)) \mid \Psi(\nu) = (\omega, h, \Upsilon) \,\}\end{array}$$

The extend-state relation $(k, \Psi) \sqsubseteq (j, \Psi')$ ensures that store typings only *grow* (in terms of both region names and the locations in those regions); that once a region is *dead*, it cannot become *live* again; that the handle assigned to a region never changes; and that the types of locations in all regions, live and dead, are approximately preserved.

Definition 7.3 (State Extension)
A valid state extension *is defined as follows:*

$$\begin{array}{rcl}(k,\Psi) \sqsubseteq (j,\Psi') & \overset{\text{def}}{=} & j \leq k \ \wedge \\ & & (\forall \nu \in \mathrm{dom}(\Psi). \quad \nu \in \mathrm{dom}(\Psi') \\ & & \quad \wedge \ \mathrm{dead}(\Psi(\nu)) \implies \mathrm{dead}(\Psi'(\nu)) \\ & & \quad \wedge \ \mathrm{hndl}(\Psi(\nu)) = \mathrm{hndl}(\Psi'(\nu)) \\ & & \quad \wedge \ (\forall \ell \in \mathrm{dom}(\mathrm{rgnty}(\Psi(\nu))). \\ & & \qquad \lfloor \mathrm{rgnty}(\Psi(\nu)) \rfloor_j(\ell) = \lfloor \mathrm{rgnty}(\Psi'(\nu)) \rfloor_j(\ell)))\end{array}$$

One can easily prove that above extend-state relation is both reflexive and transitive.

In Chapter 3, to model mutable references, we had to ensure that λ^M types were closed under state extension. The same is required of types in λ^R. A type τ in λ^R is a set of tuples of the form $\langle k, \Psi, v\rangle$ that is closed under state extension (see Definition 3.4 in Section 3.3.3).

7.4 Reference, Region Handle, and Function Types

I shall now specify the semantics of types in λ^R. Note that the semantics of each type have to be such that we can prove that the type is closed under valid state extension.

A location ℓ has type $\mathsf{ref}\ \tau\ \mathsf{at}\ \nu$ for k steps with respect to store typing Ψ if the region name ν is in $\mathrm{dom}(\Psi)$ — that is, it doesn't matter whether ν is *live* or *dead*, but it must not be nonexistent — and if the region typing $\Psi(\nu)$ says that ℓ will contain a value of type τ for at least $k-1$ steps.

$$\mathsf{ref}\ \tau\ \mathsf{at}\ \nu \quad \overset{\text{def}}{=} \quad \{\langle k, \Psi, \ell\,\rangle \mid \lfloor \mathrm{rgnty}(\Psi(\nu)) \rfloor_k(\ell) = \lfloor \tau \rfloor_k\}$$

Valid state extension ($\sqsubseteq$) guarantees that the types of all locations in all regions are approximately preserved, thus, allowing us to prove that $\mathsf{ref}\ \tau\ \mathsf{at}\ \nu$ is a valid type. Note that if the above definition required that ν be *live*, we would not be able to prove type($\mathsf{ref}\ \tau\ \mathsf{at}\ \nu$).

A region handle h has type $\mathsf{handle}\,(\nu)$ if ν is in $\mathrm{dom}(\Psi)$ — ν may be *live* or *dead* — and if, according to Ψ, ν's region handle is equal to h.

$$\mathsf{handle}\,(\nu) \quad \overset{\text{def}}{=} \quad \{\langle k, \Psi, h\,\rangle \mid \mathrm{hndl}(\Psi(\nu)) = h\}$$

Since state extension ensures that region handles are preserved, and since the above definition does not require that ν be *live*, we can easily show that $\mathsf{handle}\,(\nu)$ is a valid type.

A function of the form $\lambda(x_1, \ldots, x_n).e$ in λ^R is assigned a type of the form $(C, \tau_1, \ldots, \tau_n) \rightarrow \mathsf{unit}$, where C is a set of region names. The function (continuation, to be precise) expects n arguments of types τ_1 through τ_n and requires that each of the regions $\nu_i \in C$ be *live* when the function is applied. Before I can formally define the semantics of $(C, \tau_1, \ldots, \tau_n) \rightarrow \mathsf{unit}$, I need two auxiliary definitions.

First, I specify what it means for a store S to be well-typed to approximation k with respect to a store typing Ψ (written $S :_k \Psi$). Informally, $S :_k \Psi$ holds if and only if for every *live* region ν in the store typing Ψ: ν's region handle h is in $\mathrm{dom}(S)$; the store region $S(h)$ contains every location in ν's region typing Υ; and the contents of these locations have the type that Υ says they should have to approximation k.

Definition 7.4 (Well-typed Store)
A store S is defined to be well-typed to approximation k with respect to store typing Ψ as follows:

$$\begin{array}{rcl} S :_k \Psi & \overset{\text{def}}{=} & \forall \nu \in \mathrm{dom}(\Psi).\, \forall h, \Upsilon.\;\; \Psi(\nu) = (\mathit{live}, h, \Upsilon) \implies \\ & & \quad (\, h \in \mathrm{dom}(S) \;\wedge \\ & & \quad\;\; \mathrm{dom}(\Upsilon) \subseteq \mathrm{dom}(S(h)) \;\wedge \\ & & \quad\;\; (\forall j < k.\, \forall \ell \in \mathrm{dom}(\Upsilon).\; \langle j, \lfloor \Psi \rfloor_j, S(h.\ell) \rangle \in \lfloor \Upsilon \rfloor_k(\ell))\,) \end{array}$$

Next, I define what it means for a closed expression e to be well-typed for k steps with respect to store typing Ψ. I write $e :_{k,\Psi} \mathsf{unit}$ to denote the latter (since every λ^R expression e has type unit). Intuitively, $e :_{k,\Psi} \mathsf{unit}$ says that in a state (S, e) such that $S :_k \Psi$, term e behaves like an element of unit for k steps of computation. If (S, e) reaches an irreducible state (S', e') in less than k steps, then it must be the case that e' is the value $\mathtt{unit}$, and S' must be a store that is well-typed with respect to a well-formed store typing Ψ' that is a valid extension of Ψ. Otherwise, (S, e) can take k steps without getting stuck — that is, there exist S' and e' such that $(S, e) \longmapsto_R^k (S', e')$.

Definition 7.5 (Expr : Type)
For any closed expression e, I define $e :_{k,\Psi} \mathsf{unit}$ as follows:

$$\begin{array}{rcl} e :_{k,\Psi} \mathsf{unit} & \overset{\text{def}}{=} & \forall j, S, S', e'.\; (0 \leq j < k \;\wedge\; S :_k \Psi \\ & & \qquad\qquad\qquad \wedge\; (S, e) \longmapsto_R^j (S', e') \;\wedge\; \mathrm{irred}(S', e')) \\ & & \quad \implies \exists \Psi'.\; (k, \Psi) \sqsubseteq (k - j, \Psi') \;\wedge\; \mathrm{wellformed}(\Psi') \\ & & \qquad\qquad \wedge\; S' :_{k-j} \Psi' \;\wedge\; \langle k - j, \Psi', e' \rangle \in \mathsf{unit} \end{array}$$

The semantics of function types can now be defined as follows.

$$\begin{array}{lcl} (C, \tau_1, \ldots, \tau_n) \to \mathsf{unit} & \stackrel{\text{def}}{=} & \{\langle k, \Psi, \lambda(x_1, \ldots, x_n).e\rangle \mid \\ & & \quad \forall v_1, \ldots, v_n, \Psi', j < k. \\ & & \qquad ((k, \Psi) \sqsubseteq (j, \Psi') \ \wedge \ \text{wellformed}(\Psi') \\ & & \qquad\quad \wedge \ (\forall \nu \in C.\ \text{live}(\Psi'(\nu))) \\ & & \qquad\quad \wedge \ \langle j, \Psi', v_1\rangle \in \tau_1) \ \wedge \ldots \wedge \ \langle j, \Psi', v_n\rangle \in \tau_n) \\ & & \qquad \Longrightarrow \ e[v_1, \ldots, v_n/x_1, \ldots, x_n] :_{j,\Psi'} \mathsf{unit}\ \} \end{array}$$

7.5 Judgments and Typing Rules

Thus far I have only dealt with closed terms as these are the ones that "step" at run time. Now I turn to terms with free variables on which the static type-checking rules operate. There are two kinds of type judgments in λ^R. The main typing judgment has the form $\Delta; \Gamma \vDash_R e$ where Δ is a *region map* that maps *used* region names to either *live* or *dead*, Γ is a type environment that maps term variables to types, and e is an expression. There is also a typing judgment for values that has the form $\Delta; \Gamma \vDash_R v : \tau$. A value environment σ is a mapping from term variables to values.

Definition 7.6 (Semantics of Judgment)
For any region map Δ and store typing Ψ, I define $\Delta : \Psi$ (read "Δ agrees with Ψ") as follows:

$$\Delta : \Psi \ \stackrel{\text{def}}{=} \ \text{dom}(\Delta) = \text{dom}(\Psi) \ \wedge \ (\forall \nu \in \text{dom}(\Delta).\ \Delta(\nu) = \text{status}(\Psi(\nu)))$$

For any type environment Γ and value environment σ, $\sigma :_{k,\Psi} \Gamma$ (read "σ approximately obeys Γ") is defined as follows:

$$\sigma :_{k,\Psi} \Gamma \ \stackrel{\text{def}}{=} \ \forall x \in \text{dom}(\Gamma).\ \sigma(x) :_{k,\Psi} \Gamma(x)$$

The semantics of $\Delta; \Gamma \vDash_R^k v : \tau$ and $\Delta; \Gamma \vDash_R^k e$ (where $RN(\Gamma)$[2] $\subseteq \text{dom}(\Delta)$, $FV(v) \subseteq \text{dom}(\Gamma)$, and $FV(e) \subseteq \text{dom}(\Gamma)$) are defined as follows:

$$\begin{array}{lcl} \Delta; \Gamma \vDash_R^k v : \tau & \stackrel{\text{def}}{=} & \forall \sigma, \Psi.\ \text{wellformed}(\Psi) \ \wedge \ \Delta : \Psi \ \wedge \ \sigma :_{k,\Psi} \Gamma \ \Longrightarrow \ \langle k, \Psi, \sigma(v)\rangle \in \tau \\ \Delta; \Gamma \vDash_R^k e & \stackrel{\text{def}}{=} & \forall \sigma, \Psi.\ \text{wellformed}(\Psi) \ \wedge \ \Delta : \Psi \ \wedge \ \sigma :_{k,\Psi} \Gamma \ \Longrightarrow \ \sigma(e) :_{k,\Psi} \mathsf{unit} \end{array}$$

I write $\Delta; \Gamma \vDash_R e$ if for all $k \geq 0$ we have $\Delta; \Gamma \vDash_R^k e$. Finally, I write $\vDash_R e$ to mean $\Delta_0; \Gamma_0 \vDash_M e$ for the empty region map Δ_0 and the empty context Γ_0. The meanings of $\Delta; \Gamma \vDash_R v : \tau$ and $\vDash_R v : \tau$ are analogous.

[2] $RN(\Gamma)$ denotes the set of region names that appear in the codomain of Γ

$$\boxed{\Delta; \Gamma \vDash_R v : \tau}$$

$$\frac{}{\Delta; \Gamma \vDash_R x : \Gamma(x)}\ (\texttt{RV-var}) \qquad \frac{}{\Delta; \Gamma \vDash_R \mathsf{unit} : \mathsf{unit}}\ (\texttt{RV-unit})$$

$$\frac{\Delta \leq \Delta' \qquad \forall \nu' \in C.\, \Delta'(\nu') = \mathit{live} \qquad \Delta'; \Gamma\,[x_1 \mapsto \tau_1] \ldots [x_n \mapsto \tau_n] \vDash_R e}{\Delta; \Gamma \vDash_R \lambda(x_1, \ldots, x_n).e : (C, \tau_1, \ldots, \tau_n) \rightarrow \mathsf{unit}}\ (\texttt{RV-abs})$$

$$\boxed{\Delta; \Gamma \vDash_R e}$$

$$\frac{\begin{array}{cc} \Delta; \Gamma \vDash_R v_1 : \tau & \Delta; \Gamma \vDash_R v_2 : \mathsf{handle}\,(\nu) \\ \Delta(\nu) = \mathit{live} & \Delta; \Gamma\,[x \mapsto \mathsf{ref}\ \tau\ \mathsf{at}\ \nu] \vDash_R e \end{array}}{\Delta; \Gamma \vDash_R \texttt{let}\ x = \texttt{new}(v_1)\,\mathsf{at}\,v_2\ \texttt{in}\ e}\ (\texttt{R-newat})$$

$$\frac{\Delta; \Gamma \vDash_R v : \mathsf{ref}\ \tau\ \mathsf{at}\ \nu \qquad \Delta(\nu) = \mathit{live} \qquad \Delta; \Gamma\,[x \mapsto \tau] \vDash_R e}{\Delta; \Gamma \vDash_R \texttt{let}\ x = \texttt{!}\,v\ \texttt{in}\ e}\ (\texttt{R-deref})$$

$$\frac{\Delta; \Gamma \vDash_R v_1 : \mathsf{ref}\ \tau\ \mathsf{at}\ \nu \qquad \Delta; \Gamma \vDash_R v_2 : \tau \qquad \Delta(\nu) = \mathit{live} \qquad \Delta; \Gamma \vDash_R e}{\Delta; \Gamma \vDash_R \texttt{let}\ v_1 := v_2\ \texttt{in}\ e}\ (\texttt{R-assign})$$

$$\frac{\Delta\,[\nu \mapsto \mathit{live}]; \Gamma\,[x \mapsto \mathsf{handle}\,(\nu)] \vDash_R e \qquad \nu \notin \mathrm{dom}(\Delta)}{\Delta; \Gamma \vDash_R \texttt{let}\ \texttt{newrgn}\,x\ \texttt{in}\ e}\ (\texttt{R-newrgn})$$

$$\frac{\Delta; \Gamma \vDash_R v : \mathsf{handle}\,(\nu) \qquad \Delta(\nu) = \mathit{live} \qquad \Delta\,[\nu \mapsto \mathit{dead}]; \Gamma \vDash_R e}{\nu; \Gamma \vDash_R \texttt{let}\ \texttt{freergn}\,v\ \texttt{in}\ e}\ (\texttt{R-freergn})$$

$$\frac{\Delta; \Gamma \vDash_R v : (C, \tau_1, \ldots, \tau_n) \rightarrow \mathsf{unit} \qquad \Delta; \Gamma \vDash_R v_i : \tau_i \qquad \forall \nu' \in C.\, \Delta(\nu') = \mathit{live}}{\Delta; \Gamma \vDash_R v\,(v_1, \ldots, v_n)}\ (\texttt{R-app})$$

$$\frac{}{\Delta; \Gamma \vDash_R \texttt{halt}}\ (\texttt{R-halt})$$

Figure 7.3: Regions (λ^R): Type-checking Lemmas

The typing rules for λ^R are given in Figure 7.3. I write $\Gamma\,[x \mapsto \tau]$ to denote the type environment that extends Γ by mapping x to τ where $x \notin \mathrm{dom}(\Gamma)$. I write $\Delta\,[\nu \mapsto \omega]$ to denote the type environment that either updates Δ (if $\nu \in \mathrm{dom}(\Delta)$) or extends Δ (if $\nu \notin \mathrm{dom}(\Delta)$) by mapping ν to ω. Notice that the typing rule for abstraction (`RV-abs`) must ensure that the body of the function type checks at some point in the future with respect to a future region map Δ'. This future region map must be a valid extension of the current region map Δ, written $\Delta \leq \Delta'$. The relation $\Delta \leq \Delta'$ ensures that the region map only grows over time and that once a region is *dead* it cannot become *live* again.

Definition 7.7 (Region Map Extension)
A valid region-map extension *is defined as follows:*

$$\begin{aligned}\Delta \leq \Delta' \quad \overset{\text{def}}{=} \quad & \forall \nu \in \mathrm{dom}(\Delta).\ \nu \in \mathrm{dom}(\Delta') \\ & \qquad\qquad\qquad \wedge\ (\Delta(\nu) = \mathit{dead} \implies \Delta'(\nu) = \mathit{dead})\end{aligned}$$

Each of the λ^R typing rules can be proved as a lemma.

A program in λ^R is a closed expression that contains neither location symbols ℓ nor region handles h. Since we are not doing subject-reduction, we do not need to type-check executing programs, which means that we do not need to type-check locations and region handles.

Based on the above definitions, we can prove the desired safety theorem which says that if $\vDash_R e$, then given an arbitrary store S, machine state (S, e) is safe (see Theorem 3.8).

7.6 Discussion

In this chapter, I've described how to construct an indexed possible-worlds model that supports region deallocation and yet is (at least in a technical sense) type preserving. The central trick becomes apparent if we consider the semantics of mutable reference types. The semantics of $\mathsf{ref}\ \tau\ \mathsf{at}\ \nu$ does not say anything about whether or not one can safely read or write the reference cell — it simply says that if we were to read from (or write to) the cell, we would get (or have to put in) a value of type τ. The task of checking whether a cell can be accessed without violating memory safety is relegated to the typing rules for dereferencing and assignment. These rules check if the appropriate capability is held, that is, if ν is *live* according to Δ.

It is important to extend λ^R with type and region polymorphism which I have not discussed in this chapter. Following the model of λ^R (extended with type and region polymorphism), I conjecture that it will be possible to construct a von

Neumann model of a realistic low-level TAL with region-based memory management primitives for use in a foundational PCC system.

Chapter 8

Conclusions and Future Work

A proof-carrying code infrastructure should not be specific to the target type system of a particular compiler or source language. Furthermore, its safety guarantee should not be contingent upon the assumption that such a complex, low-level type system is sound. For these reasons, machine-checkable proofs of type soundness for real machine languages are essential. Real languages have features for mutating state that make it harder to prove safety. I have shown how to prove type soundness for a language with mutable references and impredicative polymorphism by constructing indexed Kripke logical relations based on the operational semantics of the language. I have also shown how to apply this technique to a language with primitives for region-based memory management.

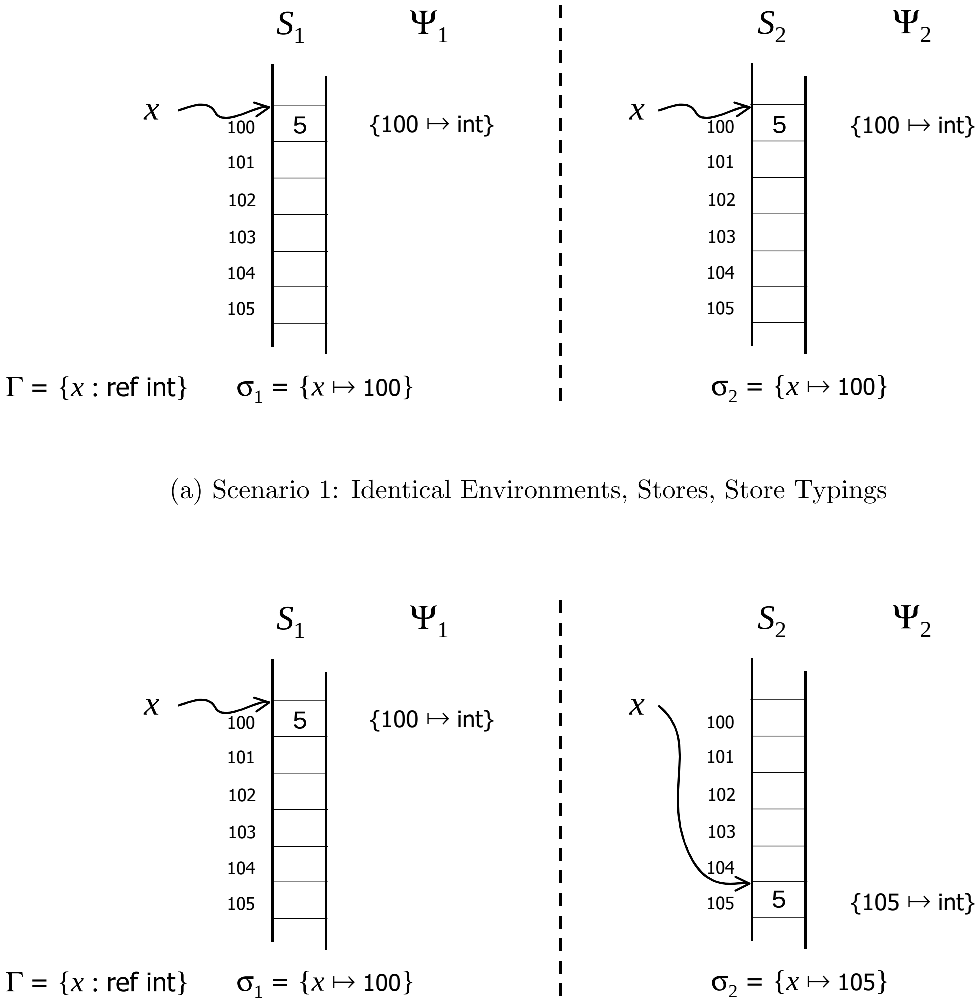
There exist only two other models of languages with general references and *neither* of these allow (impredicative) quantified types to be stored in mutable cells; they also do not permit memory deallocation and reuse. These are Abramsky *et al.*'s game semantics [AHM98] and Levy's possible-worlds model which employs category-theoretic machinery [Lev02]. I've presented an indexed possible-worlds model of types that is guided by the notion of approximations inherent in domain theory. But the particular advantage of the indexed model is that it permits relatively simple and direct proofs of type safety for a wide range of language features — features that are usually hard to deal with because they lead to various forms of circularity — without the need to "import" large mathematical theories, such as domain theory or category theory. These proofs are short enough to permit implementation as machine-checked proofs in a simple higher-order logic. I have described how we construct an indexed possible-worlds model for a von Neumann machine, which allows us to prove type safety for a low-level TAL that can express the constructs of a real source language (core ML) compiled by a real compiler (derived from SML/NJ) to a real target machine (the Sparc). In this final chapter I'll discuss an important direction for future research.

8.1 A PER Model

Informally, two expressions are *operationally equivalent* (also called *observational equivalence* or *contextual equivalence* in the literature) if no program context can distinguish them. Much work has been done to develop techniques for reasoning about program equivalence [Mil77, MT89, MT91a, MT92, JM91, How89, How96, PS93, RP95, Pit98, Pit00, Pit02]. As a result we have useful techniques for reasoning about operational equivalence in functional languages (e.g., Milner's context lemma for typed λ-calculus [Mil77]) as well as languages with side-effects such as dynamic allocation and mutation of first-order references. In particular, Pitts [Pit96] has demonstrated the use of *operationally-based logical relations* to prove a context lemma for a higher-order language with assignable variables that store only first-order values. In more recent work, he has extended the idea to languages with first-order references, recursive functions, existential types and parametric polymorphism [Pit98, Pit00, Pit02]. However, there haven't been any operationally-based techniques developed to date that permit reasoning about equivalence in the presence of general references, that is, references that can store other references, functions, quantified types, and so on. Hence, a useful direction for future research is to extend the logical relations model described in this dissertation to permit reasoning about observational equivalence as well as safety.

Appel and McAllester [AM01] showed how to extend their indexed model for the simply typed λ-calculus with recursive types (and no mutable state) to a partial equivalence relation (PER) model. A PER model allows one to reason about *operational approximation*, a weaker notion than operational equivalence. A term e_1 operationally approximates a term e_2 if and only if for any two observationally equivalent program contexts $\mathcal{C}_1$ and $\mathcal{C}_2$ (written $\mathcal{C}_1 \approx_{\text{obs}} \mathcal{C}_2$), if $\mathcal{C}_1[e_1]$ reaches an irreducible state s_1, then $\mathcal{C}_2[e_2]$ reaches an irreducible state s_2 that is observationally equivalent to s_1. Thus, if e_1 operationally approximates e_2 *and vice versa*, then e_1 and e_2 are operationally equivalent.

Building a PER model for a language with mutable state (especially higher-order references) is not an easy task. In the rest of this section, I'll describe some of the challenges. First, consider how one should model an evaluation context $\mathcal{C}$. In a functional language it suffices to model $\mathcal{C}$ as a value environment σ that maps variables to values — this is the "context" in which an open expression will be evaluated. But in a language with mutable state, the context must also describe the state. A review of the semantics of judgments $\Gamma \vDash_M e : \tau$ in Chapter 3 shows us how to pick an appropriate context in which to evaluate an open expression e. The context consists of a value environment σ, a store typing Ψ, and a store S such that σ satisfies Γ with respect to Ψ and S is well-typed with respect to Ψ. For the PER model we'll have judgments of the form $\Gamma \vDash e_1 \leq e_2 : \tau$ (read "e_1 approximates e_2 in τ"). To define the semantics of such judgments, we need to pick an appropriate



(a) Scenario 1: Identical Environments, Stores, Store Typings

(b) Scenario 2: Related Environments, Stores, Store Typings

Figure 8.1: Observationally Equivalent Contexts

context $\mathcal{C}_1 = (\sigma_1, \Psi_1, S_1)$ in which to evaluate e_1, and a context $\mathcal{C}_2 = (\sigma_2, \Psi_2, S_2)$ in which to evaluate e_2. Furthermore, we'll have to make sure that $\mathcal{C}_1$ and $\mathcal{C}_2$ are observationally equivalent (denoted $\mathcal{C}_1 \approx_{\text{obs}} \mathcal{C}_2$).

This brings us to the second challenge which is the definition of $\mathcal{C}_1 \approx_{\text{obs}} \mathcal{C}_2$. I won't define $\approx_{\text{obs}}$ here. Instead, I'll characterize the desired definition by presenting examples of pairs of observationally equivalent and inequivalent contexts. For each of the examples that follow, assume that we wish to reason about the operational equivalence of the expressions e_1 and e_2, where:

- e_1 *denotes* `let _ =` $x :=\ (!\,x+1)$ `in` $x :=\ (!\,x+1)$
- e_2 *denotes* $x :=\ (!\,x+2)$

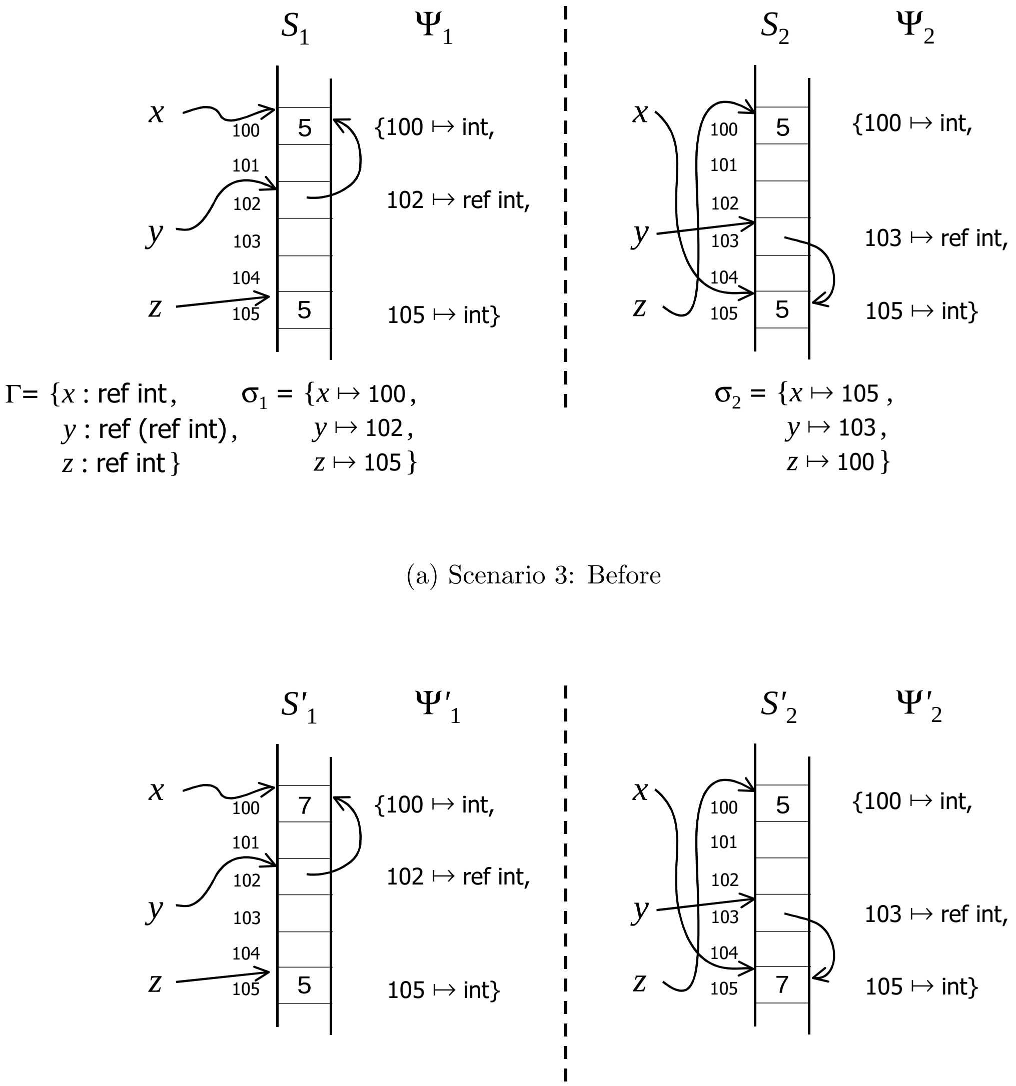
Also, in each example, let $\mathcal{C}_1 = (\sigma_1, \Psi_1, S_1)$ denote the context in which we evaluate e_1 and let $\mathcal{C}_2 = (\sigma_2, \Psi_2, S_2)$ denote the context in which we evaluate e_2.

Scenario 1 Figure 8.1(a) illustrates contexts $\mathcal{C}_1 = (\sigma_1, \Psi_1, S_1)$ and $\mathcal{C}_2 = (\sigma_2, \Psi_2, S_2)$. We wish to construct a model where $\approx_{\text{obs}}$ is defined so that $\mathcal{C}_1 \approx_{\text{obs}} \mathcal{C}_2$ holds. This is easy to do since $\mathcal{C}_1$ and $\mathcal{C}_2$ are identical. Furthermore, we must specify the semantics of $\Gamma \vDash e_1 \leq e_2 : \tau$ so that it allows us to conclude that if $\mathcal{C}_1 \approx_{\text{obs}} \mathcal{C}_2$, then the evaluation of $\mathcal{C}_1[e_1]$ and $\mathcal{C}_2[e_2]$ results in observationally equivalent contexts: $(\sigma_1, \Psi_1, S_1\,[100 \mapsto 7]) \approx_{\text{obs}} (\sigma_2, \Psi_2, S_2\,[100 \mapsto 7])$.

Scenario 2 Consider the contexts $\mathcal{C}_1$ and $\mathcal{C}_2$ depicted in Figure 8.1(b). These contexts are not identical but given a language that does not support equality testing for references it is impossible to write a well-typed program that can distinguish between them. Therefore, in a semantic model for such a language, $\mathcal{C}_1 \approx_{\text{obs}} \mathcal{C}_2$ holds. Such a model would require some way of keeping track of the fact that location 100 in $\mathcal{C}_1$ is *related* to location 105 in $\mathcal{C}_2$. For this particular example, given the relation $R = \{\,(100, 105)\,\}$, we can conclude that $\mathcal{C}_1 \approx_{\text{obs}} \mathcal{C}_2$ since $\Psi_1(100) = \Psi_2(R(100))$ and $S_1(100) = S_2(R(100))$.

Scenario 3 For the contexts $\mathcal{C}_1$ and $\mathcal{C}_2$ in Figure 8.2(a) reasoning about observational equivalence is harder due to aliasing. Nonetheless, the contexts are observationally equivalent: in both cases, variables x and z point to cells that contain equivalent values, while variable y points to a cell whose contents point to the same cell as variable x. Evaluating e_1 and e_2 in their respective contexts results in the (observationally equivalent) contexts depicted in Figure 8.2(b). In a model that can somehow keep track of how locations in $\mathcal{C}_1$ are *related* to locations in $\mathcal{C}_2$ (perhaps using a relation R as in the previous example), one can conclude that $\mathcal{C}_1 \approx_{\text{obs}} \mathcal{C}_2$.

Scenario 4 The contexts depicted in Figure 8.3(a) are identical to those in Figure 8.2(a) except for the fact that in context $\mathcal{C}_2$ of Figure 8.3(a), the variable y points to a cell whose contents point to the location aliased by z instead of x. Evaluating e_1 and e_2 in their respective contexts results in the contexts depicted in Figure 8.3(b) — I'll refer to these as $\mathcal{C}_1'$ and $\mathcal{C}_2'$. Clearly the latter are not observationally equivalent; for instance, the program $!\,(!\,y)$ can distinguish between them. Hence, our



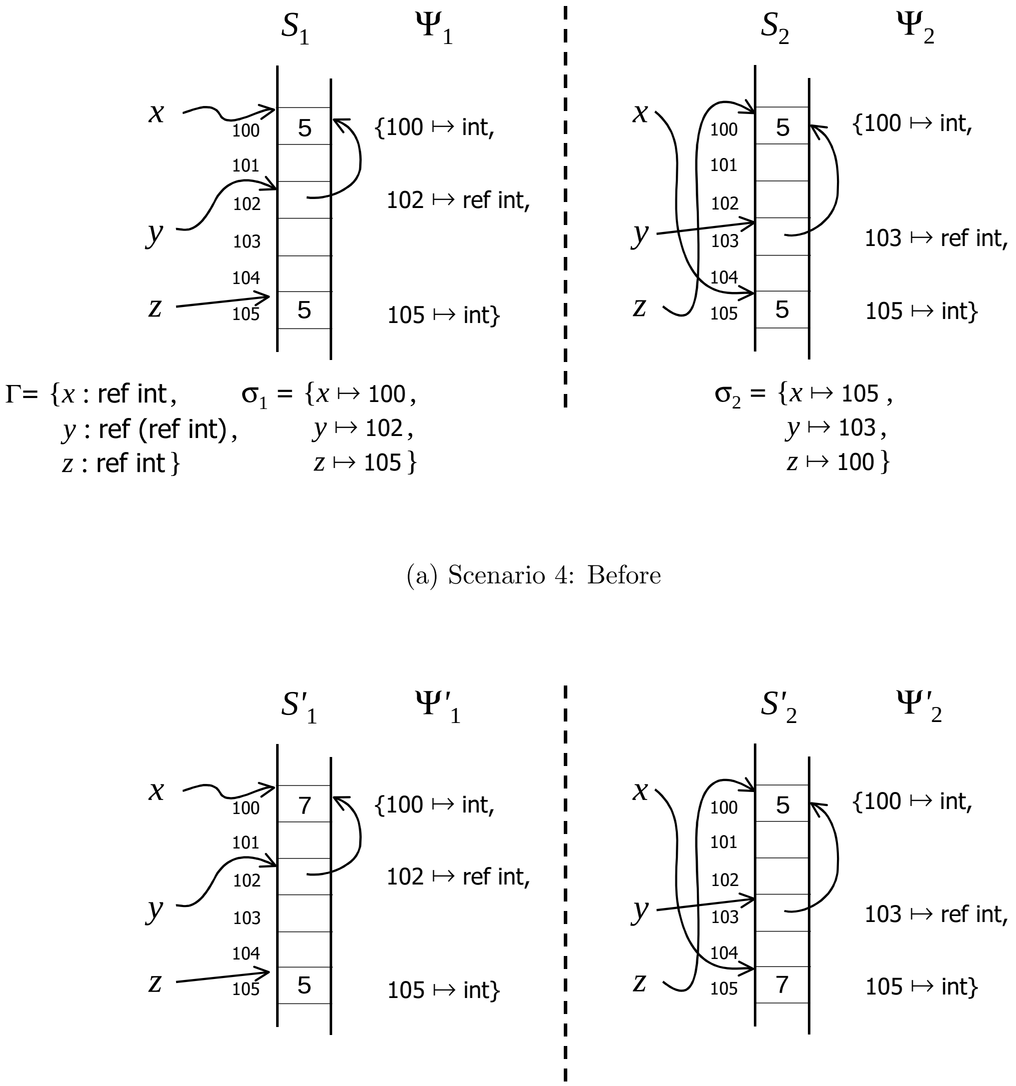
(a) Scenario 3: Before

(b) Scenario 3: After

Figure 8.2: Observational Equivalence and Aliasing

model should conclude that *neither* of the two pairs of contexts in Figure 8.3 are observationally equivalent.

This last conclusion is not entirely satisfactory, however. Imagine a scenario where we know more about the program context in which e_1 and e_2 will be placed.



(a) Scenario 4: Before

(b) Scenario 4: After — States Distinguishable Via y

Figure 8.3: Observationally Equivalent?

For instance, suppose that an optimizing compiler that wants to replace a program fragment e_1 with an observationally equivalent fragment e_2 can guarantee that the code to be executed after e_1 (alternatively e_2) will not dereference y. In such a situation, it is safe to consider the contexts $\mathcal{C}_1'$ and $\mathcal{C}_2'$ observationally equivalent.

Alternatively, if we know that the code to be executed after e_1 dereferences y, but that it can only run safely for at most one step after that dereference, then it would again be safe to consider $\mathcal{C}_1'$ and $\mathcal{C}_2'$ observationally equivalent.

Building a flexible model that can track such information is nontrivial. On the one hand, we need a richer model of evaluation contexts than what I've informally proposed above. On the other hand, more complicated contexts make it harder to define both observational equivalence and the accessibility (or extend-state) relation on worlds.

Bibliography

[AAV02] Amal Ahmed, Andrew W. Appel, and Roberto Virga. A stratified semantics of general references embeddable in higher-order logic. In *IEEE Symposium on Logic in Computer Science (LICS), Copenhagen, Denmark*, pages 75–86, July 2002. 38, 46

[AAV03] Amal Ahmed, Andrew W. Appel, and Roberto Virga. An indexed model of impredicative polymorphism and mutable references. Available at `http:// www.cs.princeton.edu/~appel/papers/impred.pdf`, January 2003. 38

[Acz88] Peter Aczel. *Non-Well-Founded Sets*. Center for the Study of Language and Information, Stanford University, 1988. 101

[AF00] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *ACM Symposium on Principles of Programming Languages (POPL), Boston, Massachusetts*, pages 243–253, January 2000. 7, 9, 15

[AHM98] Samson Abramsky, Kohei Honda, and Guy McCusker. A fully abstract game semantics for general references. In *IEEE Symposium on Logic in Computer Science (LICS), Indianapolis, Indiana*, pages 334–344, June 1998. 11, 38, 103, 147

[AJW03] Amal Ahmed, Limin Jia, and David Walker. Reasoning about hierarchical storage. In *IEEE Symposium on Logic in Computer Science (LICS), Ottawa, Canada*, pages 33–44, June 2003. 135

[AM01] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems*, 23(5):657–683, September 2001. 9, 13, 46, 63, 85, 148

[AMSV02] Andrew W. Appel, Neophytos Michael, Aaron Stump, and Roberto Virga. A trustworthy proof checker. In Iliano Cervesato, editor, *Workshop on the Foundations of Computer Security*, pages 37–48. DIKU, July 2002. diku.dk/publikationer/tekniske.rapporter/2002/02-12.pdf. 107

[And72] J. P. Anderson. Computer security technology planning study vols. i and iii. Technical Report ESD-TR-73-51, HQ Electronic Systems Division: Hanscom AFB, MA, Fort Washington, Pennsylvania, October 1972. 3

[App01] Andrew W. Appel. Foundational proof-carrying code. In *IEEE Symposium on Logic in Computer Science (LICS), Boston, Massachusetts*, pages 247–258. IEEE, June 2001. 7

[ARSA04] Andrew W. Appel, Christopher D. Richards, Kedar N. Swadi, and Amal Ahmed. A machine-checkable soundness proof for typed machine language. Submitted for publication, April 2004. 128

[AW02] Andrew W. Appel and Daniel C. Wang. JVM TCB: Measurements of the trusted computing base of Java virtual machines. Technical Report CS-TR-647-02, Princeton University, April 2002. 7, 123

[BBC$^+$98] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Yann Coscoy, David Delahaye, Daniel de Rauglaudre, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, , Henri Laulhère, César Muñoz, Chetan Murthy, Catherine Parent-Vigouroux, Patrick Loiseleur, Christine Paulin-Mohring, Amokrane Saïbi, and Benjamin Werner. The Coq Proof Assistant reference manual. Technical report, INRIA, 1998. 106, 133

[BCP99] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 155(1–2):108–133, November–December 1999. 36

[BM96] Jon Barwise and Lawrence Moss. *Vicious Circles: On the Mathematics of Non-wellfounded Phenomena*. Cambridge University Press, 1996. 101

[CAB$^+$86] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, Englewood Cliffs, New Jersey, 1986. 4, 15

[Car97] Luca Cardelli. Type systems. In A. B. Tucker, editor, *The Computer Science and Engineering Handbook*. CRC Press, Boca Raton, Florida, 1997. 4

[CGW89] Thierry Coquand, Carl A. Gunter, and Glynn Winskel. Domain theoretic models of polymorphism. *Information and Computation*, 81(2):123–167, 1989. 37

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988. 105, 106, 133

[Che04] Juan Chen. *A Low-Level Typed Assembly Language with a Machine-Checkable Soundness Proof.* PhD thesis, Princeton University, June 2004. 121

[CLN$^+$00] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Ken Cline, and Mark Plesko. A certifying compiler for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, British Columbia, Canada*. ACM Press, June 2000. 8, 123, 124

[Cra03] Karl Crary. Toward a foundational typed assembly language. In *ACM Symposium on Principles of Programming Languages (POPL), New Orleans, Louisiana*, pages 198–212, January 2003. 133

[CWAF03] Juan Chen, Dinghao Wu, Andrew W. Appel, and Hai Fang. A provably sound TAL for back-end optimization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), San Diego, California*, pages 208–219. ACM Press, June 2003. 121, 122

[CWM99] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *ACM Symposium on Principles of Programming Languages (POPL), San Antonio, Texas*, pages 262–275, January 1999. 135, 136

[DG71] P. Deutsch and C. A. Grant. A flexible measurement tool for software systems. In *Information Processing (Proceedings of the IFIP Congress)*, pages 320–326, 1971. 3

[DMTW97] Allyn Dimock, Robert Muller, Franklyn Turbak, and J. B. Wells. Strongly typed flow-directed representation transformations. In *International Conference on Functional Programming (ICFP), Amsterdam, The Netherlands*, pages 85–98, June 1997. 5

[ES00a] Úlfar Erlingsson and Fred B. Schneider. IRM enforcement of java stack inspection. In *IEEE Symposium on Security and Privacy*, pages 246–255, Oakland, California, May 2000. 3

[ES00b] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *WNSP: New Security Paradigms Workshop*. ACM Press, 2000. 3

[ET99] David Evans and Andrew Twyman. Flexible policy-directed code safety. In *IEEE Security and Privacy*, pages 32–45, Oakland, CA, May 1999. 3

[FJM$^+$96] M. P. Fiore, A. Jung, E. Moggi, P. O'Hearn, J. Riecke, G. Rosolini, and I. Stark. Domains and denotational semantics: History, accomplishments and open problems. Technical Report CSR-96-2, School of Computer Science, The University of Birmingham, 1996. 30pp., available from `http://www.cs.bham.ac.uk/`. 37

[FKT01] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *International Journal of Supercomputer Applications*, 15(3):200–222, 2001. 1

[Gro02] Dan Grossman. Existential types for imperative languages. In *European Symp. on Programming (ESOP)*, pages 21–33, Grenoble, France, April 2002. 96, 99

[Gun92] Carl A. Gunter. *Semantics of Programming Languages.* MIT Press, Cambridge, Massachusetts, 1992. 20

[Har96] Robert Harper. A note on: "A simplified account of polymorphic references" [Inform. Process. Lett. **51** (1994), no. 4, 201–206; MR 95f:68142]. *Information Processing Letters*, 57(1):15–16, 1996. 103

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993. 8, 105, 107, 134

[Hin75] K. Jaakko Hintikka. Impossible possible worlds vindicated. *Journal of Philosophical Logic*, 4:475–484, 1975. 103

[HMS03] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Computability classes for enforcement mechanisms. Technical Report TR 2003-1908, Cornell University, August 2003. 3

[How89] Douglas J. Howe. Equality in lazy computation systems. In *Proceedings of the Fourth Annual Symposium of Logic in Computer Science*, pages 198–203, Asilomar Conference Center, Pacific Grove, California, June 1989. IEEE Computer Society Press. 148

[How96] Douglas J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, 1996. 148

[HPW91] Paul Hudak, Simon Peyton Jones, and Philip Wadler. Report on the programming language Haskell: Version 1.1. Technical report, Yale University and Glasgow University, August 1991. 36

[HR98] Nevin C. Heintze and Jon G. Riecke. The SLam Calculus: Programming with secrecy and integrity. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, January 1998. 4

[HR00] Michael R. A. Huth and Mark D. Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, Cambridge, England, 2000. 26

[HST$^+$02] Nadeem Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. A syntactic approach to foundational proof-carrying code. In *IEEE Symposium on Logic in Computer Science (LICS), Copenhagen, Denmark*, pages 89–100, July 2002. 133

[Hug97] Dominic J. D. Hughes. Games and definability for System F. In *IEEE Symposium on Logic in Computer Science (LICS), Warsaw, Poland*, pages 76–86, June 1997. 103

[JM91] Trevor Jim and Albert R. Meyer. Full absraction and the context lemma. In Takayasu Ito and Albert R. Meyer, editors, *Theoretical Aspects of Computer Software (TACS)*, volume 526 of *Lecture Notes in Computer Science*, pages 131–151. Springer, 1991. 148

[Kri63] Saul A. Kripke. Semantical considerations on modal logic. In *Proceedings of a Colloquium: Modal and Many Valued Logics*, volume 16, pages 83–94, 1963. 25

[Lam71] Butler Lampson. Protection. In *Proceedings of the 5th Symposium on Information Sciences and Systems*, pages 437–443, Princeton, New Jersey, 1971. 3

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3:125–143, March 1977. 3

[Lea02] Christopher Adam League. *A Type-Preserving Compiler Infrastructure*. PhD thesis, Yale University, May 2002. 36

[Lev01] Paul Blain Levy. *Call-by-Push-Value.* Ph. D. dissertation, Queen Mary, University of London, London, UK, March 2001. 101

[Lev02] Paul Blain Levy. Possible world semantics for general storage in call-by-value. In *Computer Science Logic, 16th International Workshop, CSL 2002 Proceedings*, volume 2471 of *Lecture Notes in Computer Science*, pages 232–246, Edinburgh, Scotland, UK, September 2002. Springer. 11, 38, 101, 103, 147

[LST03] Christopher League, Zhong Shao, and Valery Trifonov. Precision in practice: A type-preserving Java compiler. In *12th International Conference on Compiler Construction (CC'03)*, April 2003. 8

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, second edition, 1999. 3, 5

[MA00] Neophytos G. Michael and Andrew W. Appel. Machine instruction syntax and semantics in higher-order logic. In *17th International Conference on Automated Deduction*, June 2000. 8, 123, 124

[MCG$^+$99] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *ACM Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, GA, May 1999. 5, 134

[Mil77] Robin Milner. Fully abstract models of typed lambda calculi. *Theoretical Computer Science*, 4(1), 1977. 148

[Mit96] John C. Mitchell. *Foundations for Programming Languages.* MIT Press, Cambridge, Massachusetts, 1996. 20, 33

[MMH96] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, pages 271–283, January 1996. 36, 37

[MP88] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988. 36

[MPS86] David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymophic types. *Information and Computation*, 71(1/2):95–130, 1986. 63

[MT89] Ian A. Mason and Carolyn L. Talcott. Axiomatizing operational equivalence in the presence of side effects. In *Proceedings of the Fourth Annual Symposium of Logic in Computer Science*, pages 284–293, Asilomar Conference Center, Pacific Grove, California, June 1989. IEEE Computer Society Press. Extended version appeared as "Inferring the equivalence of functional programs that mutate data," *Theoretical Computer Science*, 105(2):167–215, 9 November 1992. 148

[MT91a] Ian A. Mason and Carolyn L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1(3):287–327, 1991. 148

[MT91b] Robin Milner and Mads Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87(1):209–220, 1991. 101

[MT92] Ian A. Mason and Carolyn L. Talcott. References, local variables and operational reasoning. In *IEEE Symposium on Logic in Computer Science (LICS), Santa Cruz, California*, pages 186–197, June 1992. 148

[MWCG98] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, pages 85–97, January 1998. 5

[MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999. 3, 5, 36, 37, 134

[Mye99] Andrew C. Myers. Practical mostly-static information flow control. In *ACM Symposium on Principles of Programming Languages (POPL), San Antonio, Texas*, pages 228–241. ACM Press, January 1999. 3

[Nac97] Carey Nachenberg. Computer virus-antivirus coevolution. *Communications of the ACM*, 40(1):46–51, January 1997. 3

[Nec97] George Necula. Proof-carrying code. In *ACM Symposium on Principles of Programming Languages (POPL), Paris, France*, pages 106–119. ACM Press, January 1997. 6

[NL98a] George Necula and Peter Lee. The design and implementation of a certifying compiler. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada*, pages 333 – 344, June 1998. 6, 7, 124

[NL98b] George C. Necula and Peter Lee. Efficient representation and validation of proofs. In *IEEE Symposium on Logic in Computer Science (LICS), Indianapolis, Indiana*, June 1998. 124

[NR01] George C. Necula and S. P. Rahul. Oracle-based checking of untrusted software. In *ACM Symposium on Principles of Programming Languages (POPL), London, England*, pages 142–154. ACM Press, January 2001. 123, 124

[Ole82] Frank Joseph Oles. *A Category-Theoretic Approach to the Semantics of Programming Languages*. Ph. D. dissertation, Syracuse University, Syracuse, New York, August 1982. 101, 103

[Ole85] Frank Joseph Oles. Type algebras, functor categories, and block structure. In Maurice Nivat and John C. Reynolds, editors, *Algebraic Methods in Semantics*, pages 543–573. Cambridge University Press, Cambridge, England, 1985. 101, 103

[Ole97] Frank Joseph Oles. Functor categories and store shapes. In Peter W. O'Hearn and Robert D. Tennent, editors, *ALGOL-like Languages, Volume 2*, pages 3–12. Birkhäuser, Boston, Massachusetts, 1997. 101, 103

[Par69] D. Park. Fixpoint induction and proofs of program properties. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 5, pages 59–78, Edinburgh, 1969. Edinburgh University Press. 101

[Par02] Manish Parashar, editor. *Grid Computing - GRID 2002, Third International Workshop, Baltimore, Maryland, USA, November 18, 2002, Proceedings*, volume 2536 of *Lecture Notes in Computer Science*. Springer, 2002. 1

[Per93] Jaroslav Peregrin. Possible worlds: A critical analysis. *The Prague Bulletin of Mathematical Linguistics*, 59-60:9–21, 1993. 101

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. 20, 36

[Pit96] Andrew M. Pitts. Relational properties of domains. *Information and Computation*, 127(2):66–90, 1996. 148

[Pit98] Andrew M. Pitts. Existential types: Logical relations and operational equivalence. *Lecture Notes in Computer Science*, 1443:309–326, 1998. 104, 148

[Pit00] Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10:321–359, 2000. 104, 148

[Pit02] Andrew M. Pitts. Operational semantics and program equivalence. In G. Barthe, P. Dybjer, L. Pinto, and J. Saraiva, editors, *Applied Semantics: Advanced Lectures*, volume 2395 of *Lecture Notes in Computer Science*, pages 378–412. Springer-Verlag, 2002. 104, 148

[PJHH$^+$93] Simon L. Peyton Jones, Cordelia V. Hall, Kevin Hammond, Will Partain, and Philip Wadler. The Glasgow Haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, July 1993. 5

[PM93] Christine Paulin-Mohring. Inductive definitions in the system Coq; rules and properties. In M. Bezem and J. F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer-Verlag, 1993. 105, 106, 133

[PS93] Andrew M. Pitts and Ian D. B. Stark. Observable properties of higher order functions that dynamically create local names, or: What's *new*? In Andrzej M. Borzyszkowski and Stefan Sokołowski, editors, *Mathematical Foundations of Computer Science*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141, Berlin, 1993. Springer-Verlag. 101, 148

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *The 16th International Conference on Automated Deduction*. Springer-Verlag, July 1999. 8, 105, 134

[RC91] Jonathan Rees and William Clinger. Revised^4 report on the algorithmic language Scheme. *Lisp Pointers*, 4(3):1–55, July–September 1991. 3

[Rey81] John C. Reynolds. The essence of Algol. In Jaco W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372, Amsterdam, 1981. North-Holland. 101, 103

[Rey83] John C. Reynolds. Types, abstraction, and parametric polymorphism. *Information Processing*, pages 513–523, 1983. 4

[RP95] Eike Ritter and Andrew M. Pitts. A fully abstract translation between a lambda-calculus with reference types and Standard ML. In Mariangiola Dezani-Ciancaglini and Gordon D. Plotkin, editors, *Typed Lambda*

Calculi and Applications (TLCA), Edinburgh, UK, volume 902 of *Lecture Notes in Computer Science*, pages 397–413. Springer, April 1995. 148

[SA95] Zhong Shao and Andrew W. Appel. A type-based compiler for Standard ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), La Jolla, California*, pages 116–129. ACM Press, 1995. 5

[Sch00a] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and Systems Security*, 3(1):30–50, February 2000. 3

[Sch00b] Carsten Schürmann. *Automating the Meta-Theory of Deductive Systems*. Ph. D. thesis, Carnegie Mellon University, Pittsburgh, PA, 2000. 134

[Sma97] Christopher Small. MiSFIT: A tool for constructing safe extensible C++ systems. In *Proceedings of the Third USENIX Conference on Object-Oriented Technologies*, Portland, OR, June 1997. 3

[SP01] Eijiro Sumii and Benjamin C. Pierce. Logical relations for encryption. In *Computer Security Foundations Workshop*, June 2001. To appear in *Journal of Computer Security*. 4

[SS75] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975. 7

[Sta94] Ian D. B. Stark. *Names and Higher-Order Functions*. Ph. D. dissertation, University of Cambridge, Cambridge, England, December 1994. 101

[Swa03] Kedar N. Swadi. *Typed Machine Language*. PhD thesis, Princeton University, November 2003. 96, 123, 128

[SWM00] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *European Symp. on Programming (ESOP)*, pages 366–381, Berlin, March 2000. 9

[TA04] Gang Tan and Andrew W. Appel. A typed calculus for machine instructions and its semantics in higher-order logic. Submitted for publication, February 2004. 123, 131

[TASW04] Gang Tan, Andrew W. Appel, Kedar N. Swadi, and Dinghao Wu. Construction of a semantic model for a typed assembly language. In Bernhard Steffen and Giorgio Levi, editors, *Fifth International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '04)*, volume 2937 of *LNCS*, pages 30–43. Springer Verlag Lecture Notes in Computer Science, January 2004. 123, 130, 132

[TG00] Robert D. Tennent and Dan R. Ghica. Abstract models of storage. *Higher-Order and Symbolic Computation*, 13(1/2):119–129, 2000. 37

[TMC$^+$96] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Philadephia, Pennsylvania*, pages 181–192, May 1996. 5

[Tof90] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, November 1990. 96, 103

[TT94] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, pages 188–201, January 1994. 135

[TT97] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997. 135

[Vis00] Mahesh Viswanathan. *Foundations for the Run-time Analysis of Software Systems*. Ph. D. thesis, University of Pennsylvania, 2000. 3

[Wad89] Philip Wadler. Theorems for free! In *ACM Symposium on Functional Programming Languages and Computer Architecture (FPCA)*, London, September 1989. 4, 99

[Wal01] David Walker. *Typed Memory Management*. Ph. D. thesis, Cornell University, 2001. 9, 135, 136, 137

[WAS03] Dinghao Wu, Andrew W. Appel, and Aaron Stump. Foundational proof checkers with small witnesses. In *5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, August 2003. 107, 123, 124

[WCM00] David Walker, Karl Crary, and Greg Morrisett. Typed memory management in a calculus of capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4):701–771, May 2000. 135, 136, 137

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 15 November 1994. 9, 133

[WLAG93] Robert Wahbe, Steven Lucco, Thomas Anderson, and Susan Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*, pages 203–216, December 1993. 3

[WM00] David Walker and Greg Morrisett. Alias types for recursive data structures. In *Workshop on Types in Compilation*, Montreal, Canada, September 2000. 9

[WS97] Mitchell Wand and Gregory T. Sullivan. Denotational semantics using an operationally-based term model. In *ACM Symposium on Principles of Programming Languages (POPL), Paris, France*, pages 386–399, January 1997. 104

[XP98] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada*, pages 249–257, June 1998. 4

[XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *ACM Symposium on Principles of Programming Languages (POPL), San Antonio, Texas*, pages 214–227, January 1999. 4