THE APPROXIMATION MODALITY
IN MODELS OF HIGHER-ORDER TYPES

Christopher D. Richards

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

Advisor: Andrew W. Appel

June 2010
In this thesis we advance the state of the art in intensional type systems for low-level code. The need for powerful, carefully designed type-systems for low-level code is well-documented. We argue for an intensional, or semantic, such system, one which is distinguished by the use of a modality of type approximation to solve the problems of self-reference that arise in the models of recursive types, impredicative type quantification, and unrestricted mutable references.
ACKNOWLEDGEMENTS

To everyone whose friendship, expertise, encouragement, and humor I so greatly relied on throughout my time at Princeton, thank you. I would like to acknowledge the following people in particular.

First, I thank my advisor, Andrew Appel, for his guidance, his insight, and his open door. I admire and owe much to his unerring sense of what is possible and what is necessary.

I’m grateful to the members of my committee: to readers David Walker and Paul-André Melliès for their valuable comments and suggestions, which benefitted this work significantly; and to David August and Sanjeev Arora for giving generously of their time and support.

My co-author Jérôme Vouillon deserves special mention for having implemented – perhaps conjured is the better word – the machine-checked proofs relating to the results of this thesis. There is none more ninja than he.

I thank my fellow minions of the Foundational Proof-Carrying Code project, with whom I did it “the particular strange way they do it at Princeton”: Amal Ahmed, Lujo Bauer, Juan Chen, Eun-Young Lee, Neophytos Michael, Xinming Ou, Kedar Swadi, Gang Tan, Roberto Virga, Daniel Wang, and Dinghao Wu.

Thanks also to my PL comrades Daniel Dantas, Limin Jia, Frances Perry, Yitzhak Mandelbaum, and Jay Ligatti; and to Carl Kingsford, Manoj Prabhakaran, Ding Liu, and Wen Xu, who made Room 223 such an enjoyable place in which to work and share ideas, and with whom I shared the proud distinction of being Most Proximate to the Cookies.

I am deeply indebted to Aquinas Hobor, who volunteered to read several drafts of this work in their entirety, and whose criticism was insightful and essential. Aquinas is a rare friend.

I would be remiss not to acknowledge the tireless work of Melissa Lawson, our graduate program coordinator, who makes things happen.

I am ever grateful for the support of my family, especially my mother, the late Karen Richards; my father, David Richards; my brother, Benjamin Richards; my sister-in-law, Kara Pangburn; and my mother- and father-in-law, Anne and Tom Williams.

Above all I thank my wife, Catherine Williams, whose patience, love, and wit sustained and inspired me.

This work was funded in part by NSF grants CCF-0540914 and CNS-0627650.
In loving memory
of Karen F. Richards
## CONTENTS

Abstract iii

1 Introduction 1
   1.1 Foundational Proof Carrying Code 1
   1.2 Methodology 3
   1.3 Contributions 4
   1.4 Related Work 4
   1.5 Organization 6
   1.6 Conventions 7
   1.7 Notice of Prior Publication 7

2 Types for End-to-End Safety 8
   2.1 Von Neumann Small-Step Operational Semantics 8
   2.2 Desired Types for TAL 12

3 A Typed Assembly Language 17
   3.1 Preliminaries 17
   3.2 Terms 18
   3.3 Operational Semantics 18
   3.4 Types 19
   3.5 Roadmap 20

4 Logical Aspects of the System 22
   4.1 The Universe 22
   4.2 Typing Judgments 23
   4.3 Basic Types 23
   4.4 Kripke Models 26
   4.5 A Logic of Approximation 26
   4.6 Quantified Types 31
   4.7 Types for Propositions 34
   4.8 Related Work 36

5 A Kripke Model of Typed Machine State 38
   5.1 The Sets of Worlds and Types 38
   5.2 Approximated Types 40
   5.3 The Accessibility Relation 40
5.4 Relating Types and their Approximations 41
5.5 Recursive Types 42

6 Types for Data and Control 45
6.1 Nonexpansive Types 45
6.2 Semantics of General References 49
6.3 Control Types 53
6.4 Example: Dataflow Analysis 55
6.5 Conclusion 55

7 Instruction Typings 56
7.1 Hoare logic 56
7.2 Sketch I: Relation to Hoare Logic 56
7.3 Sketch II: Adding Details 59
7.4 Instruction-Typing Rules 60

8 Whole-Program Safety 64
8.1 Running Example 64
8.2 Encoding Von Neumann Program Syntax with Types 64
8.3 Von Neumann Program Static Semantics 65
8.4 Safety Expressed as Subtyping 66
8.5 Soundness theorem 67

9 Foundational Proof Carrying Code 70
9.1 Related TALs with Syntactic Proofs 70
9.2 Princeton FPCC 71
9.3 Further Contributions 74

10 Conclusion 75
10.1 Shortcomings 75
10.2 On the Unseemly Effectiveness... 75

Bibliography 77
**LIST OF FIGURES**

1.1 Diagram of a typical Foundational PCC system 2

2.1 Operational semantics for STANLEY 11
2.2 An example Standard ML (SML) function. 13
2.3 Figure 2.2 after CPS-conversion. 14
2.4 Figure 2.2 after CPS- and closure-conversion. 14

3.1 Terms for STANLEY 18
3.2 Operational semantics for STANLEY (reprise) 19
3.3 Types for STANLEY 19

4.1 Subtyping Rules: Gödel-Löb logic 32
4.2 Subtyping Rules: Quantification 33
4.3 Subtyping Rules: value-irrelevant judgements 35

5.1 A Coq definition of Type_n. 39

6.1 Subtyping Rules: Just and slot 47

7.1 Jumping to a code pointer that expects a list 58
7.2 A k-node list, but for a garbage pointer at the end 58
7.3 Schematic diagram of the i-LOAD rule 60
7.4 Instruction-typing rules for STANLEY 63

8.1 Example safety derivation 68
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Types exported in the Typed Assembly Language (TAL) interface</td>
<td>21</td>
</tr>
<tr>
<td>3.2</td>
<td>Types internal to the soundness proof and definitions of other types</td>
<td>21</td>
</tr>
</tbody>
</table>
1 INTRODUCTION

This thesis explores mathematical structures useful in supporting certain kinds of computer security and program verification. These are extremely broad areas of inquiry. Here we restrict our attention to the provision of static guarantees about the absence of specific classes of undesirable behavior at run-time – that is, safety properties. The question motivating our project and others like it is this: for some reasonable definition of safety, how can we guarantee the safety of the actual, low-level program that one is about to run? By low-level, we mean code that is generally the end product of a compiler and intended for execution on a machine.

Thus, our broad aim is to facilitate the transformation of safe source level languages into safe machine level languages, and in particular by creating provably safe machine level languages with a rich enough type system to support such interesting and important source languages as Java [27] and Standard ML [43].

1.1 FOUNDATIONAL PROOF CARRYING CODE

Our main point of departure is Foundational Proof Carrying Code (FPCC) [7], first proposed by Appel and implemented in various forms by Appel et al. [4, 6, 8, 9, 11, 22, 57, 58, 65], Shao et al. [28, 51, 66], Crary et al. [24, 25], and Necula et al. [19, 20]. The goal of FPCC is to design a system of software delivery with the smallest possible trusted computing base (TCB), the set of software components a code consumer must trust before running a foreign piece of code. The typical non-Proof Carrying Code (PCC) TCB includes the entire delivered program and a substantial runtime system. At a minimum the PCC and FPCC techniques remove the delivered program from the TCB, facilitating the safe use of mobile code.

We sketch the components of a typical FPCC system (see Figure 1.1 on the following page). In such a system, there are two principals, the code producer and the code consumer. On the code producer side, there is a type-preserving compiler and a type checker designed to emit a type-checking witness. The type preserving compiler translates a strongly typed, high-level source program into strongly typed, low-level typed assembly language. The resulting low-level program is then fed into the type checker. The checker then verifies that the typed assembly language program is well-typed and will therefore not go wrong, and emits a proof witness to that effect. Significantly, none of the software components on the code producer side are trusted.
The type-checked object code and its proof of type-checking are then delivered to the code consumer. There are three components on the code consumer side, of which only one is executable: the proof checker. Recall that the checker on the code producer side is the type checker and type-checking witness producer. By contrast, the proof checker on the code consumer side emits nothing; its sole purpose is to verify the proof it receives from the code producer, and upon success, to permit the runtime system to execute the supplied code.

To verify the safety of the supplied code, the proof checker requires four inputs. The program itself and its proof of safety we have already addressed. In addition to those, the proof checker also requires a correctness specification, or safety policy, and a machine checkable soundness proof of the type system used in the object code. The latter is static and independent of any given program, and can be understood as the library of proofs with respect to which each given program’s proof of safety is derived.

But of the components on the code consumer side – the proof checker, its four inputs, and the runtime system – only the proof checker, the safety policy, and the runtime system must be trusted. As for the other components, the consumer checks, not trusts, the static soundness proof and the type-checking witness; and the soundness proof proves just that type-checking implies safety, and hence that...
the program is safe. Hence our goal of avoiding having to trust the program, or a promise of safety from the person who produced, it is achieved.

But who checks the checkers? The answer here is, the code consumer, or a party to whom she delegates this task. Because the code consumer must manually verify the safety of the proof checker, we say that the proof checker is trusted. Likewise, the correctness specification is trusted. For what would count as a proof of correctness of the correctness specification? The term safety policy speaks to this issue. What constitutes safe behavior for a particular application is simply a matter of policy, as the chain of mechanical verification must eventually come to an end.

I.2 METHODOLOGY

In an intensional (or semantically characterized) type system, we present types by giving them definitions as particular mathematical objects, usually sets with some kind of structure. Milner [42] achieved the first proofs of safety, or soundness, using the semantic technique, and this technique remained in current use as statically typed languages with polymorphism, recursion, etc. were given rigorous soundness proofs [1, 17].

There were several problems with this approach, among them that to model typed calculi (or even untyped calculi) required intricate mathematics. These difficulties motivated Wright and Felleisen [64] to propose an alternate, syntactic technique based on proof-theory.\(^1\) However, the present work returns to the semantic method, using the mixed operational-denotational approach proposed by Appel and Felty [8].

Advantages. The advantages of the semantic method over the proof-theoretic method lie along the axes of modularity and scalability. Take modularity. Adding a component to the system, whether a new type operator or a new type rule, does not necessitate revalidating the system as a whole. You are still obligated to prove that you can do something useful with your new type operator if you want to throw it in with the mix, but the change is incremental.

Bell et al. observe some further facts about the modularity of semantic proof systems [13]. An API-safe program requires a commitment to a minimum set of types, but not a maximum. Hence, if your program found it convenient to use a custom type that the code (and proof) producer never imagined, you could still go ahead and use it as long as you proved it sound – you just couldn’t use it in the

\(^1\) This technique, called subject reduction, was borrowed from combinatory logic; see e.g. Hindley [30].
interface. More importantly, the size of the trusted base scales with the number of type operators in the interface, not the number of type rules.

Disadvantages. The design of a semantic system has few constraints, in that you can add whatever lemmas (read rules) you like as long as you can prove them. Since not every true proposition is a useful one, such a system’s interface is susceptible to growing warts. In this thesis we give a clean interface whose elements are well-motivated.

1.3 Contributions

Our principal technical contribution is the first semantics to account for all three of recursive types, impredicative polymorphism and unrestricted mutable references at the same time, with the same ease as a syntactic system. Our proofs are machine-checked.

A second contribution is methodological, in that we help demonstrate the continuing viability of the semantic method in types research, exploiting the advantages discussed above.

In addition, we tie together two strands of research that heretofore ran in large part parallel to each other, namely, those of the semantics of types and the semantics of modal logics related to K4W, the Gödel-Löb logic of provability [37]. Nakano [47, 48] connected the latter to models of recursive types, and we extend that connection to types for impredicative polymorphism and first-class mutable references.

1.4 Related Work

This section divides the related work into the broad categories of ancestor work, sibling work, and historical work.

1.4.1 Ancestor work

The following work can be considered ancestral to this one, in the sense that we extend or borrow from it more or less directly. Our principal sources of this kind are the Princeton FPCC project [4, 6–9, 11, 22, 57–59, 65], the mid-2000s papers by Vouillon and Melliès [63] and Melliès and Vouillon [40], and Nakano’s work of 2000-2001 [47, 48]. Going back further, we have PCC [49], type preserving compilation [46], type semantics from before the syntactic progress-preservation
hegemony, theorem-proving, and so on. We compare the present results to those of the FPCC project in detail in Chapter 9.

1.4.2 Sibling work

The important, related, but not directly influential work that follows includes other authors’ (fairly) recent solutions to problems similar to those we address in this thesis. Among these are Shao et al.’s CAP [66] and XCAP [51]; and game-theoretic results from Abramsky et al. [2]. Continuing on, there is Ahmed’s step-indexed logical relations [5]; the relational model for restricted references by Benton and Leperchey [14]; and Sumii and Pierce’s bisimulations [56]. Many of these are primarily concerned with techniques for proving contextual equivalence in the presence of recursion, polymorphism, and so on. Not all of them have machine-checkability as a goal, or are concerned with low-level matters.

Hobor [32] and Hobor et al. [33, 34] use the “very modal model” of Appel et al. [12] to model not mutable references, but another species of indirection pertaining to resource invariants in concurrent separation logic. (A resource invariant may govern access to other resource invariants, and this leads to difficulties in specifying the metalogical type of such entities.)

Going a bit farther afield, Castagna et al. [18] shows a stratified semantics of types for the $\pi$-calculus. Given that stratification has been so effective in models of mutable reference types, Castagna et al.’s semantics comports well with the intuitive analogy between mutable references and the communications channels of the $\pi$-calculus.

We discuss this category of related work further in Chapter 9.

1.4.3 Historical

Last, there are results whose influence is a bit too far removed to merit substantial exposition, but nevertheless deserve mention as important milestones. In the semantics of modal and intuitionistic logic, Kripke [35] introduced frame models (often called “possible-worlds” models), and Mitchell and Moggi [45] showed how to employ Kripke models in the denotational semantics of programming languages. Löb [37], citing Gödel, described the modal logic of provability, whose modality Nakano (cited previously) discovered could be recast as a way to add recursion to a logic while maintaining soundness. As for models of types generally, the models of the FPCC project, and of Mellies and Vouillon, each owe a debt to the ideal model of MacQueen et al. [38].
1 INTRODUCTION

1.5 ORGANIZATION

In Chapter 2 we motivate our choice of types by considering our type system in the context of an end-to-end system, where the needs of programmers using high-level type-safe languages filter down the literal and conceptual toolchain and ultimately determine the space of plausible low-level type-system designs. To aid intuition, we then present the operational semantics for a toy von Neumann machine. Chapter 3 then outlines the TAL it will be our task to build.

Having established the set of desired types, we proceed in Chapter 4 to give definitions for, and prove the desired properties of, the subset of our types having natural analogues in logic. To get there, we first give some basic logical machinery, such as the entailment relation $\vdash$. We then introduce a few simple types, for instance union and intersection, that compose with each other in straightforward ways.

In this context, we present the approximation modality – originally due to Nakano [47, 48], citing the provability logic of Gödel and Löb [37] – which directly underlies the contributions of this thesis, in particular the definitions and lemmas for what we are calling the higher-order types (though historically ornery types (HOTs) would be nearly as accurate). These types, especially general (that is, both covariant and contravariant) recursive types, first-class mutable references, and impredicative polymorphism, are all amenable to models constructed using the approximation modality. The above types all have in common an element of self-reference – recursive types, obviously so – and the key insight of approximation types is that you can avoid infinite regress by observing that in many instances, definitions involving a type $\tau$ that seem to require a circular reference to $\tau$ can get by instead with an approximated $\mathbb{A}\tau$, which is “good enough” in a particular technical sense.

In Chapter 5, we return to the notion of approximation and ground it in a Kripke model [45] of our modal system, which lets us move from the straightforward but tedious technique of stratification to something more modular and abstract. We then exploit the Kripke model to define recursive types.

Chapter 6 lays out the remaining types for data, including mutable reference types and continuations (here called code pointers in English and codeptr in concrete syntax). To the subtyping rules for data, Chapter 7 adds instruction-typing rules and explains their use by analogy to Hoare logic. Then, having constructed all the components of a low-level type system, we show in Chapter 8 how to use them to prove the safety of whole programs.
1 INTRODUCTION

In Chapter 9 we discuss in detail the particular merits of the modal approach to approximation types relative to its conceptual ancestry in the Princeton FPCC project, and put it in the context of competing FPCC projects. Finally, in Chapter 10 we conclude.

1.6 CONVENTIONS

Typography. We write type constructors in sans-serif and machine instructions in SMALL-CAPS.

Definitions. One of our concerns is to demonstrate the conceptual parsimony of our set of base types, in the sense that a small number wisely-chosen base types may be combined to realize a suite of types standard among higher-order, typed (HOT) languages. To this end we define, using Coq [23, 52] as the underlying logical framework, a set of primitive types; and then use these to define synthetic types, written using only previously-introduced types and type operators. Primitive definitions are set off by the symbol \( \text{\texttt{prim}} \), and synthetic, the symbol \( \text{\texttt{syn}} \). This is something of a hack, since proper use of a module system would better show off this conceptual boundary.

Terminology. In this thesis, the use of “approximation” is always oriented towards the loss of information. The approximation modality \( \triangleright \) is a forgetful operator. Just as \( \triangleright \tau \) makes fewer distinctions (is less picky, has more elements) than \( \tau \), \( \triangleright \triangleright \tau \) makes fewer distinctions than \( \triangleright \tau \). This is in direct contrast with another use of “approximation,” in which the typical process is to start with a rough likeness of some desired object and improve the resemblance using iterated refinements. In that usage, one speaks of \( k \)th approximations such that the amount of detail inhering in the candidate object increases with \( k \).

1.7 NOTICE OF PRIOR PUBLICATION

An extended abstract of this work was published as Appel et al. [12].
2 TYPES FOR END-TO-END SAFETY

The principal constraints on the design of our system come from the two endpoints of the toolchain it inhabits. From above, we have a source program written in high-level, strongly-typed languages; and from below, a machine architecture targeted for deployment. The set of high-level language features we want to accommodate will determine the constituents of our low-level type menagerie and their associated rules, while the machine architecture(s) we target will influence how we model those constituent types denotationally. We address the type menagerie problem in Section 2.2 on page 12. Here we discuss the machine architecture that our imagined toolchain targets. Since the soundness of a type system is always relative to the operational semantics of the language to which it’s applied, our discussion concerns the operational semantics of a toy machine.¹

2.1 von Neumann small-step operational semantics

The von Neumann architecture [61] describes a model of computing where programs and data occupy the same address space, and computation proceeds using a fetch-decode-execute (FDE) loop. Instances of the architecture are parameterized by the amount of memory available; the number of general- and special-purpose registers; the instruction set; the smallest addressable unit; and a host of other design decisions we couldn’t hope to anticipate. Therefore instead of formalizing the von Neumann architecture, we formalize a particular toy example of it.

2.1.1 States

At any instant the state of the machine is uniquely determined by the contents of the memory \( m \) and registers \( v \). We write \( s = (m, v) \) to denote a machine state. We model \( m \) and \( v \) as finite maps from \( \mathbb{N} \) to \( \mathbb{Z} \). For \( m(i) \) we write \( m[i] \), and for \( v(i) \) we write \( v_i \). We distinguish one register \( v_{pc} \) as the program counter, such that \( m[v_{pc}] \) is the machine instruction currently being executed.

¹ We restrict our attention to Curry-style systems of type-assignment, where it makes sense consider a type system independently of the language in its entirety. Contrast Church-style systems, where types are an essential part of the syntax of the term language.
2.1.2 Instructions

One cycle of the fetch-decode-execute loop is a step, and it takes the machine from one state \((m, v)\) to another state \((m', v')\), written \((m, v) \rightarrow (m', v')\).\(^2\) We interpret machine instructions as two-place relations on states. For example, we render a pseudo-statement \(v_3 := m[v_2 + 9]\) as \(\text{LOAD 3, 2, 9}\), such that

\[
\text{LOAD 3, 2, 9}(s, s')
\]

if and only if \(s = (m, v)\), and \(s' = (m', v')\) is the state obtained by storing into \(v_3\) the value at address \(v_2 + 9\) of \(m\). (Observe that for the present example, we will have \(m = m'\).) For legibility we sometimes use an alternate notation, for example writing expression (2.1) above as

\[
(m, v) \xrightarrow{\text{LOAD 3, 2, 9}} (m', v').
\]

An instructional is an function of zero or more arguments yielding an instruction when fully applied. Thus \(\text{LOAD 3, 2, 9}\) is an instruction, but \(\text{LOAD}\) an instructional. If \(F\) is an \(n\)-ary instructional with domain \(D_1 \times \cdots \times D_n\), the family of instructions in its range is denoted \(F D_1, \ldots, D_n\).

Our machine’s instruction set \(I\) with elements denoted \(\iota\) is given by

\[
I = \text{LOAD N, N, Z} \cup \text{STORE N, Z, N} \cup \text{ADDIMM N, N, Z} \cup \text{JUMP N}
\]

(2.2)

Without loss of generality we treat unconditional jumps only. Though the resulting instruction set lacks Turing-completeness, it is already sufficient to motivate the use of HOTs in proofs of type-soundness. Observe that the ability to reason about first-class mutable references and closures does not depend on the availability of conditional branching.

2.1.3 Instruction syntax

We now consider how to map machine instruction syntax, that is, the internal structure of machine instructions represented as strings of bits, to a semantic representation.

\(^2\) The register set here is analogous to the syntactic category value in presentations of the \(\lambda\)-calculus, hence the notation. The analogy is rough: while a \(\lambda\)-calculus value has no successor in the step relation, in general a register set does. Both are in their own respects irreducible, but our machine does not operate by reduction.
On a von Neumann machine, instructions are represented, or *encoded*, by integers \( d \in \mathbb{Z} \). If \( d \) encodes an instruction \( \iota \), we write \( d \equiv \iota \). We will leave the encode relation abstract, and mention that the construction is just a matter of arithmetic, as per Michael and Appel [41]. The technique, in brief, is to completely characterize the relation by a set of axioms similar in form to the following one for a fictitious instructional \( F \).

**Axiom.** If \( n = 7 \), and \( d = 2^{12}n + 2^8x + 2^4y + z \) for some \( x, y, z \in \mathbb{Z}_{2^4} \), then \( d \equiv F x, y, z \).

The example assumes a machine with 16-bit words and an instructional \( F \) with opcode \( n = 7 \) and three 4-bit operands \( x, y, z \).

There is some subtlety to how we handle the interplay between instruction encodings and instruction semantics as discussed in the following section. As a running example, suppose \( d \equiv \text{JUMP} 17 \). Now, it could be that two distinct encodings \( d \) and \( d' \) map to the same effect on machine state. For instance, we might have \( d' \neq d \) but \( d' \equiv \text{JUMP} 17 \); though of course, the architecture manual might not call the instruction associated with \( d' \) by the name “\text{JUMP} 17”.

Thus, if we required the definition of each instruction to specify a unique encoding, as follows,

**Definition (JUMP).** Let \( s = (m, v) \) and \( s' = (m', v') \) be machine states, and let \( a \in \mathbb{N} \). Then \( \text{JUMP} a (s, s') \) if and only if \( m[v_{pc}] = d \) and ... our system would lose a measure of modularity. Therefore we find it convenient to separate our treatments of encodings and semantics, and define instruction semantics without an encoding baked in. We are left with the problem of how to ensure our model is faithful, since a faithful model of (for instance) \( \text{JUMP} \) must not validate \( s \xrightarrow{\text{JUMP} 17} s' \) if the program counter in \( s \) fails to point to an encoding of \( \text{JUMP} 17 \).

We must have a way to tie encodings to their corresponding semantics without losing modularity. We make this important connection in the rules for the operational semantics, where we restrict each rule’s applicability by adding an encode-relation fact as a side condition. The denotational and operational semantics of the machine are the subject of the next section.

2.1.4 Instruction semantics

We define the instructionals generating the instruction set \( I \), and enforce the machine’s denotational semantics, as follows.
The resulting rules are displayed in Figure 2.1.

**Definition 2.1 (LOAD).** Let \( s = (m, v) \) and \( s' = (m', v') \) be machine states, and let \( a, b \in \mathbb{N}, c \in \mathbb{Z} \). Then LOAD \( a, b, c \langle s, s' \rangle \) if and only if \( v \) and \( v' \) agree everywhere save at \( \text{pc} \) and \( a \), with \( v'_\text{pc} = v_\text{pc} + 1 \) and \( v'_a = m[v_b + c] \); and the memory \( m' = m \) is unchanged.

**Definition 2.2 (STORE).** Let \( s = (m, v) \) and \( s' = (m', v') \) be machine states, and let \( a, c \in \mathbb{N}, b \in \mathbb{Z} \). Then STORE \( a, b, c \langle s, s' \rangle \) if and only if \( v' \) and \( v \) agree everywhere save at \( \text{pc} \), with \( v'_\text{pc} = v_\text{pc} + 1 \); and memories \( m' \) and \( m \) agree everywhere save at \( v_a + b \), with \( m[v_a + b] = c \).

**Definition 2.3 (ADDIMM).** Let \( s = (m, v) \) and \( s' = (m', v') \) be machine states, and let \( a, b \in \mathbb{N}, c \in \mathbb{Z} \). Then ADDIMM \( a, b, c \langle s, s' \rangle \) if and only if \( v' \) and \( v \) agree everywhere save at \( \text{pc} \) and \( a \), with \( v'_\text{pc} = v_\text{pc} + 1 \) and \( v'_a = v_b + c \); and the memory \( m' = m \) is unchanged.

**Definition 2.4 (JUMP).** Let \( s = (m, v) \) and \( s' = (m', v') \) be machine states, and let \( a \in \mathbb{N} \). Then JUMP \( a \langle s, s' \rangle \) if and only if \( v' \) and \( v \) agree everywhere save at \( \text{pc} \) with \( v'_\text{pc} = v_a \), and the memory \( m' = m \) is unchanged.

The small-step relation is just the union of each of the instructions; in symbols, \( \mapsto = \cup_{i \in I} i \). Finally, we produce a faithful operational semantics by rendering the four cases of the small step relation as rules, each guarded by a precondition guaranteeing that the program counter points to the appropriate integer encoding. The resulting rules are displayed in Figure 2.1.
2.2 Desired Types for Tal

The typing rules we want are determined by the features of the source languages we’d like to model. Let ML and Java [27] be representative examples. We aim to show that, in order to support the type systems of these source languages, a type-preserving compilation scheme will require its lower-level type systems to have impredicative polymorphic types, recursive types, and types for general mutable references.

Recall that a universal or existential quantifier is impredicative when its domain of quantification includes not only type variables, ground types, and other monotypes, but also second- and higher-order types with no predicativity restriction that requires a quantified type at level \( N \) to quantify over types only of level \(< N\) (for instance, the very quantifier under consideration). By general mutable references we refer to references permitted to store and fetch items of any valid type, including, significantly, higher-order functions, impredicatively quantified packages, and other general mutable references.

Remark. Concerning types, we use polymorphic and quantified interchangeably. In a further abuse of terminology, in the service of brevity we use polymorphism as equivalent to polymorphic types, and references as equivalent to reference types. There are untyped – not to mention unsafe – languages with references, polymorphism, or both, but we exclude these from consideration unless noted otherwise.

2.2.1 ML

ML’s type system supports higher-order functions, records and tuples, discriminated unions, mutable references, recursive data structures, and parametric let-polymorphism. Of these, the higher-order functions, references, and recursive data structures are particularly relevant. (We will use the concrete syntax of SML [43] in the examples that follow.)

Impredicative polymorphism. We begin with higher-order functions, which instigate the use of impredicatively polymorphic types at (at least) the assembly and machine level. To illustrate how the type structure of a higher-order function decomposes at those lower levels, we refer to the snippet of SML code in Figure 2.2 on the next page.

Impredicative existential quantifiers lurk in the types of \( g \) and of the anonymous function in the body of \( f \). In lower levels of compilation, a function with free
variables is represented as a closure, that is, a pair consisting of one, a function with no free variables, and two, its environment, a record of the variables in scope of the function's definition.

Let $C[]$ be any context accepting a function $h : A \to B$. If it accepts one such function it must accept any of them, without regard to their implementations. In particular $C[]$ has no way to know in advance what particular set of free variables a given $h$ depends on, and therefore no way to know the number or types of those free variables, or in other words, the type of the environment record. Since no single simple type is adequate to describe the environment of a closure-converted $h$, we abstract over the type of the environment, giving the closure an existential type

$$\exists \rho. (A \times \rho \to B) \times \rho,$$

where types $A$ and $B$ correspond to the original function's argument and return types, respectively. In translation, both $g$ and the anonymous function are represented by closures with this type-scheme.

Assuming an ML-like source language, these existential quantifications are necessarily impredicative [44], even though the source-level type system is expressly limited to predicative prenex-form quantification, in which any type quantifiers must appear at the outermost level of a type expression. In an expression such as

```ml
local
    fun add1 x = x + 1
  in  map add1 [0, 1, 2]
end
```

the function map has the universally quantified type $(\alpha \to \beta) \to \alpha \text{ list} \to \beta \text{ list}$; function add1, the monotype $\text{int} \to \text{int}$; and the implicit type application at map add permits the instantiation of the former with the latter. In
fun f (g, k) = 
    let fun h ((), k') = 
        g (57, k') 
    in k h 
    end 

Figure 2.3: Figure 2.2 after CPS-conversion.

fun f (G, env as {}, k) = 
    let fun h ((), env' as {G'}, k') = 
        let val (g', g'env) = G' 
        in g' (57, g'env, k') 
        end 
    in k (h, {G}) 
    end 

Figure 2.4: Figure 2.2 after CPS- and closure-conversion.

a closure-based representation, however, the image of add1 has a existentially 
quantified type, rendering the quantification regime impredicative.

Before moving on to recursive types, we note that the model of any language 
having an ML-style module system with functors will require impredicativity, since 
the standard way to model functors makes critical use of it.

Recursive types. We must have recursive types corresponding to ML’s datatype 
declarations. Indeed, since such a declaration allows the bound type variable to 
occur on the left-hand side of an arrow type, our recursive types have to handle 
contravariant as well as covariant instances.

General mutable references. ML’s type system allows references to store other 
references and recursively-typed data. It also allows references to store higher-
order functions, such that after typed closure-conversion, our system’s reference 
types must permit the storage of impredicatively existential packages.

2.2.2 Java

Java’s type system supports objects with private, mutable fields, methods with 
access to those fields, and subtyping.
As a simultaneous result of Java's popularity and its reliance on standard object-oriented type concepts, the Java example, if anything, motivates our design goals better than the ML one. One requires, under the hood, all three of mutable references, recursive types, and impredicative quantifiers just to define a trivial class. Taking the `Num` class below as an example, we follow the analysis of League et al. [36].

```java
class Num {
    int val;
    public:
        // ...
        int valOf () {
            return val;
        }
}
```

**Recursive types.** The methods of the `Num` class can access its private fields by virtue of the implicitly defined `this` reference. An intermediate representation will model methods implicitly referring to `this` by using explicit self-application, for example, rewriting the method `valOf` as the function

```java
int valOf (self this) {
    return this.val;
}
```

Of course, `self` is the type of the class we are in the midst of defining. Thus, at a minimum the elaborated type of a `Num` object must have the form

\[
\mu \text{self}.\{\text{valOf : self -> int, ...} \times \text{int},
\]

where \(\mu\) is the recursive type constructor.

**Impredicative polymorphism.** To make the private fields of `Num` inaccessible to methods defined outside the class, we quantify over the fields' concrete types with an existential type. Thus the only way to access those fields is through methods defined within the scope of the quantifier. Our revised encoding of `Num` is now

\[
\exists \text{priv} \cdot \mu \text{self}.\{\text{valOf : self -> int, ...} \times \text{priv},
\]

But observe that this also stands for the type of any class extending `Num` with a new field. For example, consider a `Num2` class that adds to `Num` a `link` field pointing
to another Num2 instance. Because the existential type variable priv may hide instances of the very type in whose definition it appears – as it would in the case of Num2 and link – the quantification must be impredicative.

For completeness’ sake, we note that a proper encoding of Num would add to the existential and recursive type constructors still one more recursive type constructor. We refer the reader to League et al. for further details.

General mutable references. Evidently the constructor in our Num2 example must perform a destructive update of the link field. By the arguments above, the type of link is simultaneously recursive and impredicative, evidencing the need for general references.
3 A TYPED ASSEMBLY LANGUAGE

In this chapter we present a simple TAL called STANLEY. This TAL is much simpler than stock hardware would require, but nonetheless complex enough to demand the sophisticated techniques customarily demanded by assembly and machine code, where at last the language can no longer delegate hairy implementation issues to the “next level down,” because there is none. Here we introduce our TAL by giving the terms of the language, its small-step operational semantics, and the types exposed in its interface – that is, the set of types exported to clients of the TAL, such as a type-preserving compiler.

3.1 preliminaries

A standard presentation of a TAL would give the syntax of types inductively. In the presentation that follows, however, we use the notation

\[ \tau ::= \ldots | \text{codeptr} \tau | \ldots \]

purely as a shorthand for a non-inductive set of declarations

\[
\begin{align*}
\text{tp} & : \text{type.} \\
\vdots \\
\text{codeptr} & : \text{tp} \to \text{tp.} \\
\vdots
\end{align*}
\]

to which we later give denotations in the form of mathematical objects – typically sets, functions, and relations with some particular structure. We treat the operational semantics rules the same way, writing

\[
\begin{align*}
P & \\
\hline
Q
\end{align*}
\]

not to represent a member of an inductive set, but as a theorem we later prove. Thus, instead of a syntactic category of types, we have a set of types that share an underlying structure; instead of inductive typing rules, we have typing rules whose semantics are given compositionally by the semantics of their constituents.

Here we present a suite of useful types that is complete enough to achieve our design goals, but we don't restrict the set of types in advance. There's no need, since in subsequent chapters we will prove the soundness of each type's typing
3 A TYPED ASSEMBLY LANGUAGE

| Memory          | $m : \mathbb{N} \rightarrow \mathbb{Z}$ | Instructions $I ::= \text{LOAD } i, j, k$ |
| Register file   | $v : \mathbb{N} \rightarrow \mathbb{Z}$ | $| \text{STORE } i, k, j$ |
| Register numbers| $i, j \in \mathbb{N}$                  | $| \text{ADDM} i, j, k$ |
| Constants       | $k \in \mathbb{Z}$                    | $| \text{JUMP } i$ |

Figure 3.1: Terms for STANLEY

rules independently of the soundness of any others. This open-endedness is in contrast to the requirements of an inductive, syntactic system, where the types and so forth must be the least sets closed under application of the relevant formation rules.

The typing rules for our TAL take the form of a set of subtyping rules paired with a set of instruction typing rules. Since our typing (resp. instruction-typing) rules follow from the semantic definitions of the types (resp. rules) they concern, we defer presenting any rules of either flavor until after we have begun to discuss definitions in Chapter 4.

3.2 TERMS

Our machine is as we described in Chapter 2, but here we alter the presentation to emphasize its features in common with other TALs.

Under the rubric term we include just those components of our system having to do with the operational aspects of computation: memories $m$, register banks $v$, constants $i, j, k$, and the four instructions that follow. They are the load instruction, written LOAD $i, j, k$; the store instruction, written STORE $i, k, j$; the add-immediate instruction, written ADDIM $i, j, k$; and the jump instruction, written JUMP $i$. We present these components in summary form in Figure 3.1. We gave detailed descriptions of the above terms in Sections 2.1.1 and 2.1.2 on page 9.

3.3 OPERATIONAL SEMANTICS

The operational semantics of our system is given by a small-step relation comprising four rules, one for each family of instructions. We write $m[x \rightarrow y, \ldots]$ for the memory $m$ updated so that $x$ maps to $y$ (and so on), and similarly for the register
3 A TYPED ASSEMBLY LANGUAGE 19

3.2 Operational semantics for STANLEY (reprise)

\[ \begin{align*}
\text{I-LOAD} & : m[v_{pc}] \triangleright LOAD i, j, k \\
& \quad (m, v) \mapsto (m, v[i \mapsto m[v_j + k], pc \mapsto pc + 1]) \\
\text{I-STORE} & : m[v_{pc}] \triangleright STORE i, k, j \\
& \quad (m, v) \mapsto (m[v_i + k \mapsto v_j], v[pc \mapsto pc + 1]) \\
\text{I-ADD-IMM} & : m[v_{pc}] \triangleright ADDIMM i, j, k \\
& \quad (m, v) \mapsto (m, v[i \mapsto v_j + k, pc \mapsto pc + 1]) \\
\text{I-JUMP} & : m[v_{pc}] \triangleright JUMP i \\
& \quad (m, v) \mapsto (m, v[pc \mapsto v_i])
\end{align*} \]

Figure 3.2: Operational semantics for STANLEY (reprise)

\[ n \in \mathbb{Z} \]

\[ \tau ::= \text{top} \mid \tau \land \sigma \mid \tau \lor \sigma \mid \tau \land \alpha : \tau \mid \alpha : \tau \mid \forall \alpha : \tau \mid \exists \alpha : \tau \mid \text{rec } \alpha \cdot \tau \mid \text{just } n \mid \{ v_n : \tau \} \mid \text{offset}(n, \tau) \mid \text{ref } \tau \mid \text{codeptr } \tau \]

Figure 3.3: Types for STANLEY

file \( v \). Recall that \( m[v_{pc}] \triangleright \text{INSTR} \) means that the program counter points to the instruction \text{INSTR} in memory. The rules of the operational semantics were first presented in Figure 2.1 on page 11. We show them again in Figure 3.2 for the sake of convenience.

Recall the denotational semantics for these instructions given in Section 2.1.4 on page 10. To review, the \text{LOAD} instruction fetches the contents of memory at offset \( k \) from the address in register \( v_j \); places the result in register \( v_i \); and advances the program counter. The \text{STORE} instruction stores the value of register \( v_j \) into the memory cell at offset \( k \) from the address in register \( v_i \); and advances the program counter. The \text{ADDIMM} instruction adds constant \( k \) to the value in register \( v_j \) and places the result in register \( v_i \); and advances the program counter. Finally, the \text{JUMP} instruction sets the program counter to the address in register \( v_i \).

3.4 TYPES

Types classify values, which in this case are the register files \( v \). The types of the TAL interface are as follows. Types \text{top}, \( \tau \land \sigma \), and \( \tau \lor \sigma \) are the top, intersection, and untagged union types, respectively. Any value can have type \text{top}. A value has
type $\tau \land \sigma$ provided it has both type $\tau$ and $\sigma$. A value has type $\tau \lor \sigma$ provided it has type $\tau$, or $\sigma$, or both.

The symbol $\alpha$ stands for a type variable. Types $\forall \alpha::. \tau$ and $\exists \alpha::. \tau$ are the universal and existential impredicative- and parametric-polymorphic types, respectively. Type $\text{rec } \alpha::. \tau$ is the recursive type. Type $\text{just } n$ is the singleton type classifying the integer $n$. Type $\{ v_n : \tau \}$ asserts that the register $v_n$ has type $\tau$. Type $\text{offset}(n, \tau)$ is an address-relative type – for example, if address 4 has type $\tau$, address 0 has type $\text{offset}(4, \tau)$. Type $\text{ref } \tau$ is the type of mutable references to type $\tau$. Type $\text{codeptr } \tau$ classifies continuations (entry points) with the precondition encoded by $\tau$.

The types of the TAL interface are summarized in Figure 3.3 on the previous page. In Chapter 4 and beyond we will give these types formal definitions. Often these definitions will be in terms of more primitive types that are of interest mainly to the implementer of the TAL, and which therefore are omitted from the interface, and do not appear in the figure.

3.5 ROADMAP

Ultimately, in Chapter 7 and Chapter 8, we will show how to tie together the operational semantics of Figure 3.2 and the types of Figure 3.3. Along the way we will define a complex type system that includes all the types of Figure 3.3 as well as a number of others not exposed in the interface but used internally in the soundness proof. As a cross-referencing aid, Table 3.1 on the following page lists each type exposed in the interface, the page and equation number of its definition, and brief description of its significance. Table 3.2 lists each type internal to the soundness proof in the same way. With this roadmap outlined, we are now ready to start defining our type system, starting with the parts which have a close connection to formal logic.
<table>
<thead>
<tr>
<th>Notation</th>
<th>Page</th>
<th>Equation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>top</td>
<td>23</td>
<td>(4.1)</td>
<td>Top type</td>
</tr>
<tr>
<td>(\tau \land \sigma)</td>
<td>23</td>
<td>(4.3)</td>
<td>Intersection type</td>
</tr>
<tr>
<td>(\tau \lor \sigma)</td>
<td>23</td>
<td>(4.4)</td>
<td>Untagged union type</td>
</tr>
<tr>
<td>(\forall x : A. \tau)</td>
<td>31</td>
<td>(4.8)</td>
<td>Universal type</td>
</tr>
<tr>
<td>(\exists x : A. \tau)</td>
<td>31</td>
<td>(4.9)</td>
<td>Existential type</td>
</tr>
<tr>
<td>rec (F)</td>
<td>43</td>
<td>(5.5)</td>
<td>Recursive type</td>
</tr>
<tr>
<td>just (u)</td>
<td>46</td>
<td>(6.1)</td>
<td>Singleton type</td>
</tr>
<tr>
<td>{v_j : \tau}</td>
<td>47</td>
<td>(6.3)</td>
<td>Von Neumann register type</td>
</tr>
<tr>
<td>offset((n, \tau))</td>
<td>48</td>
<td>(6.5)</td>
<td>Pointer arithmetic type</td>
</tr>
<tr>
<td>ref (\tau)</td>
<td>51</td>
<td>(6.8)</td>
<td>Mutable reference type</td>
</tr>
<tr>
<td>codeptr((\tau))</td>
<td>54</td>
<td>(6.12)</td>
<td>Code pointer (continuation) type</td>
</tr>
</tbody>
</table>

Table 3.1: Types exported in the TAL interface

<table>
<thead>
<tr>
<th>Notation</th>
<th>Page</th>
<th>Equation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bot</td>
<td>23</td>
<td>(4.2)</td>
<td>Bottom type</td>
</tr>
<tr>
<td>(\tau \Rightarrow \sigma)</td>
<td>23</td>
<td>(4.5)</td>
<td>Implication type</td>
</tr>
<tr>
<td>(\triangleright \tau)</td>
<td>28</td>
<td>(4.6)</td>
<td>Approximation modality</td>
</tr>
<tr>
<td>(\Box\tau)</td>
<td>29</td>
<td>(4.7)</td>
<td>Necessity modality</td>
</tr>
<tr>
<td>!(\tau)</td>
<td>34</td>
<td>(4.10)</td>
<td>Configs where all values have (\tau)</td>
</tr>
<tr>
<td>?(\tau)</td>
<td>34</td>
<td>(4.11)</td>
<td>Configs where some value has (\tau)</td>
</tr>
<tr>
<td>(\tau \iff \sigma)</td>
<td>35</td>
<td>(4.12)</td>
<td>Type of configs in (\tau) iff in (\sigma)</td>
</tr>
<tr>
<td>(\tau = \sigma)</td>
<td>36</td>
<td>(4.13)</td>
<td>Asserts (\tau) equal to (\sigma)</td>
</tr>
<tr>
<td>(u \hat{\tau})</td>
<td>46</td>
<td>(6.2)</td>
<td>Asserts (u) has type (\tau) (reflected)</td>
</tr>
<tr>
<td>plus((i, j, k))</td>
<td>48</td>
<td>(6.4)</td>
<td>Asserts (i + j = k)</td>
</tr>
<tr>
<td>(l \mapsto \tau)</td>
<td>49</td>
<td>(6.6)</td>
<td>Configs where mem. typing (\Psi(l) = \tau)</td>
</tr>
<tr>
<td>validmem((m))</td>
<td>50</td>
<td>(6.7)</td>
<td>Type of configs matching memory (m)</td>
</tr>
<tr>
<td>safemem((m))</td>
<td>53</td>
<td>(6.10)</td>
<td>Type of configs safe for memory (m)</td>
</tr>
<tr>
<td>safe</td>
<td>53</td>
<td>(6.11)</td>
<td>Type of safe configs</td>
</tr>
<tr>
<td>(\Delta_p)</td>
<td>64</td>
<td>(8.1)</td>
<td>Program code type</td>
</tr>
<tr>
<td>(\Gamma_p)</td>
<td>65</td>
<td>(8.2)</td>
<td>Program invariants type</td>
</tr>
</tbody>
</table>

Table 3.2: Types internal to the soundness proof and definitions of other types
The goals of this chapter are threefold. One: to understand the model of (the core of) our type system as a Kripke model of a modal logic of approximation. Two: to identify the subset of our types and subtyping rules corresponding naturally to that model, and to prove the rules sound. Third, and last: to lay the groundwork, in the form of ancillary definitions and properties, for the construction and analysis of historically troublesome types in the following two chapters.

Here we begin to build a model for the typed assembly language previously described. For each object or relation we give its definition, faithful to its implementation in Coq, and use the definition to prove salient properties.\(^1\) For a complete inventory of the types we define here and elsewhere, see Tables 3.1 and 3.2 on the previous page. We begin with the logical “scaffolding”, such as the universe of which types are modeled as subsets, the turnstile, and some simple types that have direct counterparts in logic.

To augment the last point, what we are exhibiting is presented for the most part as a type system. This is a matter of taste; it could just as well be presented as a logic for program analysis. This is not surprising, since it was designed as a type system for program analysis (in particular data flow analysis), and in any event, the former are just a special case of the latter.

### 4.1 The Universe

We model types as sets of indexed values. The set \(V\) of values is just the set of register files. For the index set, let \(W\) be a nonempty set with elements \(w\) called worlds, for reasons that will become clear. We leave this set abstract for now; there are further requirements on the index set but we postpone the details until Section 4.4. Thus we have the universe

\[
U = W \times V
\]

We call an element \(u \in U\) a configuration. We let \(\tau\) and \(\sigma\) range over types, which are subsets of \(U\).

\(^1\) Many lemmas are designated as having been “proved in Coq”. The proofs of these lemmas, and indeed the Coq counterparts to our entire system, are available at [http://www.pps.jussieu.fr/~vouillon/smot/][62].
4.2 Typing Judgments

We build the model of typing judgments in two stages. First, we would like to say what it means for a value to inhabit a type at a particular world.

Definition 4.1 (Forcing). We say that a value \( v \) inhabits type \( \tau \) at world \( w \), and write \( w \vdash v : \tau \), if and only if \( (w, v) \in \tau \). Alternatively, we say that \( w \) forces \( v : \tau \). We omit \( v \) and write \( w \vdash \tau \) when \( v \) is apparent from context.

We choose the \( \vdash \) notation and “forcing” terminology to be suggestive of modal logic. We will describe the connection precisely in Section 4.4.

Following Swadi \[57\], we find it convenient to express our typing judgments as subtyping rules. Though our typing rules, as a consequence, are not syntax-directed, they may be used to model a syntax-directed set of rules. In fact, Wu \[65\] gives a fully worked out system showing how to use the kind of semantic type constructors defined in this thesis to construct a syntax-directed TAL for real ML compiler targeted to a real target machine.

Definition 4.2 (Subtyping). We say that \( \sigma \) is a subtype of \( \tau \), and write \( \sigma \vdash \tau \), if and only if \( \sigma \subseteq \tau \).

Subtyping rules, when expressed as \( \frac{A}{B} \), are modeled by implication in the logical framework.

Some elementary facts follow directly from Definition 4.2.

Lemma 4.3 (Id). For any \( \tau \), \( \tau \vdash \tau \).

Lemma 4.4 (Cut). If \( \tau_1 \vdash \tau_2 \) and \( \tau_2 \vdash \tau_3 \), then \( \tau_1 \vdash \tau_3 \).

4.3 Basic Types

We define some useful types, beginning with types whose semantics coincide with those of elementary logical and set-theoretic operators. We give them first because they are among the simplest to implement.

\[
\begin{align*}
\text{top} & \overset{\text{prim}}{=} U \\
\text{bot} & \overset{\text{prim}}{=} \{\} \\
\tau \land \sigma & \overset{\text{prim}}{=} \tau \cap \sigma \\
\tau \lor \sigma & \overset{\text{prim}}{=} \tau \cup \sigma \\
\sigma \Rightarrow \tau & \overset{\text{prim}}{=} \{(w, v) \mid (w, v) \in \sigma \Rightarrow (w, v) \in \tau\}
\end{align*}
\]
The top and intersection types are familiar from System $F_{\leq}$ and systems of refinement types, respectively (among others).

Remark 4.5. We often abbreviate “$\top \vdash \tau$” as “$\vdash \tau$”. In addition, we will sometimes write “$\sigma \land \tau_1 \vdash \tau_2$” as “$\sigma, \tau_1 \vdash \tau_2$” to clarify $\sigma$’s role as an auxiliary hypothesis.

The union type is purely set-theoretic and untagged, making it appropriate for use in data flow applications and in constructing (for example) tagged-union types. Besides symmetry with the top type, the bottom type is required for the proper functioning of our Hoare logic, especially in conjunction with the code pointer type. For instance, $\text{codeptr}(\bot)$ intuitively types the initial entry point of a program at which control begins with no prior assumptions. The implication type constructor is the least usual of the lot. Its sole use is in the construction of more complex types. The types above entail the following subtyping rules. Rules with longer names are also given an abbreviated name.

Lemma 4.6 (Top-R). For all $\tau, \tau \vdash \top$.

Lemma 4.7 (Bot-L). For all $\tau, \bot \vdash \tau$.

Lemma 4.8 (Intersection-L1, Inter-L1). For any $\tau_1$ and $\tau_2$, $\tau_1 \land \tau_2 \vdash \tau_1$.

Lemma 4.9 (Intersection-L2, Inter-L2). For any $\tau_1$ and $\tau_2$, $\tau_1 \land \tau_2 \vdash \tau_2$.

Lemma 4.10 (Intersection-R, Inter-R). For any $\tau_1, \tau_2, \tau_3$, if $\tau_1 \vdash \tau_2$ and $\tau_1 \vdash \tau_2$, then $\tau_1 \vdash \tau_2 \land \tau_3$.

Lemma 4.11 (Union-R1). For any $\tau_1$ and $\tau_2$, $\tau_1 \vdash \tau_1 \lor \tau_2$.

Lemma 4.12 (Union-R2). For any $\tau_1$ and $\tau_2$, $\tau_2 \vdash \tau_1 \lor \tau_2$.

Lemma 4.13 (Union-L). For any $\tau_1, \tau_2, \tau_3$, if $\tau_1 \vdash \tau_3$ and $\tau_2 \vdash \tau_3$, then $\tau_1 \lor \tau_2 \vdash \tau_3$.

Lemma 4.14 (Implies-I, Imp-I). If $\sigma \land \tau_1 \vdash \tau_2$, then $\sigma \vdash \tau_1 \Rightarrow \tau_2$.

Lemma 4.15 (Implies-E, Imp-E). If $\sigma \vdash \tau_1 \Rightarrow \tau_2$, then $\sigma \land \tau_1 \vdash \tau_2$.

Proofs. Directly from the relevant definitions.

The following lemmas will be useful in proving derived subtyping rules later on.
Lemma 4.16 (Weakening). For any $\tau_1$, $\tau_2$, and $\sigma$, if $\tau_1 \vdash \sigma$, then $\tau_1 \land \tau_2 \vdash \sigma$. In symbols,

$$
\frac{\tau_1 \vdash \sigma}{\tau_1 \land \tau_2 \vdash \sigma}.
$$

Proof. By derivation.

$$
\begin{align*}
\frac{}{\text{Inter-L2}} & \\
\frac{\tau_2 \land \sigma \vdash \sigma}{\text{Imp-I}} & \quad \frac{\tau_1 \vdash \sigma \quad \sigma \vdash \tau_2 \Rightarrow \sigma}{\text{Cut}} \\
\frac{}{\text{Imp-E}} & \quad \frac{\tau_1 \vdash \tau_2 \Rightarrow \sigma}{\tau_1 \land \tau_2 \vdash \sigma}
\end{align*}
$$

$\Box$

Lemma 4.17 (Inter-Lift). For all types $P$, if $\tau \vdash \sigma$, then $P \land \tau \vdash P \land \sigma$.

Proof. By derivation.

$$
\begin{align*}
\frac{}{\text{Inter-L1}} & \\
\frac{P \land \tau \vdash P \land \sigma}{\text{Cut}} & \quad \frac{\tau \vdash \sigma}{\text{Imp-I}} \\
\frac{}{\text{Inter-R}} & \quad \frac{P \land \tau \vdash P \land \sigma}{P \land \tau \vdash P \land \sigma}
\end{align*}
$$

$\Box$

Lemma 4.18. For all types $P$, the expression $P \Rightarrow \cdot$ is monotone: If $\tau \vdash \sigma$, then $P \Rightarrow \tau \vdash P \Rightarrow \sigma$.

Proof. By derivation.

$$
\begin{align*}
\frac{P \Rightarrow \tau \vdash P \Rightarrow \tau}{\text{Id}} & \quad \frac{(P \Rightarrow \tau) \land P \vdash \tau \vdash \sigma}{\text{Cut}} \\
\frac{}{\text{Imp-I}} & \quad \frac{(P \Rightarrow \tau) \land P \vdash \sigma}{P \Rightarrow \tau \vdash P \Rightarrow \sigma}
\end{align*}
$$

$\Box$

Lemma 4.19. For all $P$, the expression $\cdot \Rightarrow P$ is anti-monotone: If $\tau \vdash \sigma$, then $\sigma \Rightarrow P \vdash \tau \Rightarrow P$.

Proof. By derivation.
4.4 Kripke Models

The standard way to prove a modal logic sound is to give it a possible-worlds, or Kripke model. Suppose we have a logic $L$, comprising variables, constants, connectives, and inference rules. Then a Kripke model for $L$ is a triple $(W, R, \models)$ interpreted as follows. The set $W$ is as before, a nonempty set of worlds, or contexts with respect to which the validity of a formula (or equivalently, typings) are relative. For example, in a type system with references we might take $W$ to be the set of store-typings, such that each $w \in W$ is map from locations to types.

The relation $R \subseteq W \times W$, called the accessibility relation, specifies the possible transitions between worlds. If $(w, w') \in R$, we write $w \ R \ w'$ and say that $w$ reaches $w'$. To continue the store-typing example, we will write $\text{dom}(w)$ to denote the set of locations to which $w$ assigns types. In this case, the accessibility relation might require that $\text{dom}(w) \subseteq \text{dom}(w')$, and that the images $w(\text{dom} w)$ and $w'(\text{dom} w)$ coincide. The properties required of the accessibility relation determine the axioms satisfied by the modalities of the logic, that is, they determine which modal logic it is.

In a typical modal logic setting, the symbol $\models$, standing for an entailment relation, is defined inductively on the structure of expressions to give meanings to the various constants and connectives of the logic. We do not define $\models$ inductively. Rather, we give explicit intensional definitions to each connective. Consequently $\models$ in our system is little more than syntactic sugar.

4.5 A Logic of Approximation

To the non-modal operators given in Equations (4.1) to (4.5) we add a modality called approximately, written $\triangleright$. Intuitively, $\triangleright \tau$ stands for a weakened version...
of the type \( \tau \), where “weakened” has a specific technical definition we will explain below. With the approximation modality our logic can express induction hypotheses in a uniform, concise way, and we can use those hypotheses the approximation modality's characteristic elimination rule. The rule, called the Löb rule after logician Martin Löb, looks like this:

\[
\begin{align*}
\vdash \tau & \vdash \tau \\
\vdash & \tau
\end{align*}
\]

It expresses the idea that, if assuming that some type \( \tau \) holds in weakened form lets us conclude \( \tau \), then \( \tau \) holds immediately. This rule is directly analogous to the principle of strong mathematical induction, where we may show a property \( P \) holds for every \( n \) by showing \( P(k') \) under the assumption that \( P(k) \) for all \( k < k' \).

Since the semantics of a modal operator critically depend on its related accessibility relation, we give the necessary conditions such that the relation gives rise to a modality that behaves in the desired way.

4.5.1 The required accessibility axioms

We will defer the formal definition of the worlds \( w \in W \) and the accessibility relation \( R \) until Chapter 5, but here we give the features we want our construction to satisfy.

Our logic is to be an instance of the modal logic K4W (also known as Gödel-Löb logic), and in any model thereof, \( R \) must be transitive and (converse-) well-founded. We have this requirement precisely because it permits the induction principle we are after.

\[
R \text{ well-founded} \overset{\text{def}}{=} \text{from any world } w \\
\text{there is no infinite path } w R w' R w'' R \ldots
\]

Hereafter, \( R \) always refers to a transitive and converse-wellfounded accessibility relation unless otherwise noted.

As a computation steps, it moves via \( R \) from world to world. Since there are no infinite paths through \( R \), the question then arises whether our system accounts for infinite computations. It does, and to see this, consider that worlds represent facts from the point of view of the attention-limited observer, and thus the existence of a terminal world implies only that observation, not computation, always ceases eventually.

In a more technical sense, consider also that the choice of a (converse-) well-founded \( R \), in particular one partially ordered according to the number of future
computation steps, is driven by the need to prove the standard safety theorem, where a safe computation is said to be safe if it cannot get stuck: that is, for any \( k \), if it can take \( k \) steps, then it can take \( k + 1 \) steps. (We discuss safety in detail in Section 6.3.1 on page 53.) That there are no infinite paths through \( R \) simply makes it possible to prove the safety theorem by induction, and does not serve to restrict the behavior of well-typed programs.

4.5.2 The modality: definition and proofs

We arrive at the model of the approximation modality, the featured element of our type system.

\[
\triangleright \tau \overset{\text{prim}}{=} \{ (w, v) \mid \forall w'. w \mathrel{R} w' \Rightarrow (w', v) \in \tau \} \tag{4.6}
\]

We gave the intuition for wanting this operator in Section 4.5 on page 26. More formally, its properties are as follows.

Lemma 4.20 (Approx-Lift). The \( \triangleright \) operator is monotone: if \( \sigma \vdash \tau \), then \( \triangleright \sigma \vdash \triangleright \tau \).

Lemma 4.21 (Approx-Inter). The \( \triangleright \) operator distributes over intersection:

\[
\triangleright \bigwedge \tau_i = \bigwedge \triangleright \tau_i.
\]


Theorem 4.22 (Löb Rule, a/k/a Approx-Fix). For every type \( \tau \),

\[
\frac{\triangleright \tau \vdash \tau}{\vdash \tau}.
\]

Proof. Since \( R \) is well-founded, it satisfies the following induction principle:

\[
\forall P. \left[ \forall w. \left[ \forall w'. w \mathrel{R} w' \Rightarrow P(w') \right] \Rightarrow P(w) \right] \Rightarrow \forall w. P(w).
\]

By taking \( P(w) = (w, v) \in \tau \), we get:

\[
\forall v. \left[ \forall w. \left[ \forall w'. w \mathrel{R} w' \Rightarrow (w', v) \in \tau \right] \Rightarrow (w, v) \in \tau \right] \Rightarrow \forall w. (w, v) \in \tau.
\]

Finally, with respect to the quantification over \( v \), by the distributive property of quantification over implication we have:

\[
\left[ \forall w, v. \left[ \forall w'. w \mathrel{R} w' \Rightarrow (w', v) \in \tau \right] \Rightarrow (w, v) \in \tau \right] \Rightarrow \forall w, v. (w, v) \in \tau.
\]
Then by the definition of $\triangleright$,

$$\forall w, v. (w, v) \in \triangleright \Rightarrow (w, v) \in \tau \Rightarrow \forall w, v. (w, v) \in \tau.$$  

But this is just $\triangleright \tau \vdash \tau \Rightarrow \vdash \tau$, as required.

**Lemma 4.23 (Approx-Rep).** For all $\tau$, $\triangleright \tau \vdash \triangleright \tau$.

**Lemma 4.24 (Approx-Top).** It is always the case that $\vdash \triangleright \text{top}$.

**Proofs of 4.23–4.24.** Proved in Coq [62].

### 4.5.3 Necessity

Closely related to $\triangleright$ is the modality $\Box$, spoken necessarily. The $\Box$ operator is an essential part of a generalization of Theorem 4.22 that lies at the heart of our technique for proving whole-program safety. Intuitively, $\Box \tau$ that asserts that $\tau$ holds in a given world, as well as in any world more approximate than it. In formal terms,

$$\Box \tau \overset{\text{syn}}{=} \tau \land \triangleright \tau.$$  

(4.7)

The operator $\Box$ has the following properties.

**Lemma 4.25 (Nec-Lift).** The $\Box$ operator is monotone: if $\sigma \vdash \tau$, then $\Box \sigma \vdash \Box \tau$.

**Lemma 4.26 (Nec-Inter).** The $\Box$ operator distributes over intersection:

$$\Box \bigwedge \tau_i = \bigwedge \Box \tau_i.$$  

**Lemma 4.27 (Nec-Left).** For any type $\tau$, $\Box \tau \vdash \tau$. That is, if $\tau$ holds of a world $w$ and all worlds reachable from $w$, it remains the case that $\tau$ holds of $w$.

**Lemma 4.28 (Nec-R1).** If $\Box \sigma \vdash \tau$, then $\Box \sigma \vdash \Box \tau$. That is, if $\Box \sigma$ is a subtype of $\tau$, it is so by virtue of being a subtype of the restriction of $\tau$ to $\Box \tau$.

**Lemma 4.29 (Nec-Top).** The top type has $\vdash \Box \text{top}$.

**Proofs of 4.25–4.29.** Proved in Coq [62].

**Lemma 4.30 (Nec-R2).** For any type $\tau$, $\vdash \tau \vdash \Box \tau$.

**Proof.** By derivation.
Lemma 4.31. Repeated applications of □ are idempotent; that is,

\[ \square \square \tau = \square \tau. \]

Proof. By derivation, and the fact that \( P \vdash Q \) and \( Q \vdash P \) together imply \( P = Q \).

4.5.4 Necessary types

We say that a type \( \tau \) is necessary whenever it entails its own approximation. Equivalently, a type is necessary if the fact that a type obtains in a present world implies that it will also obtain at all worlds reachable from it. With very few exceptions, most of the types defined in this thesis are necessary in this sense. This is by design, since necessary types are required to construct well-formed recursive types.

\[ \tau \text{ necessary} \overset{\text{def}}{=} \tau \vdash \triangleright \tau \]

Lemma 4.32. For every type \( \tau \), the types \( \square \tau \) and \( \triangleright \tau \) are necessary.

Lemma 4.33. A type \( \tau \) is necessary if and only if \( \tau = \square \tau \).

Lemma 4.34. The intersection of necessary types is necessary.

Proofs of 4.32–4.34. Proved in Coq [62].
Corollary 4.35 (Generalized Löb Rule).

\[ \sigma, \triangleright \tau \vdash \tau \quad \sigma \text{ necessary} \]
\[ \sigma \vdash \tau \]

Proof. By derivation.

\[
\begin{align*}
\sigma \Rightarrow \tau \vdash \sigma \Rightarrow \tau & \quad \text{Id} \\
\sigma \land (\sigma \Rightarrow \tau) \vdash \tau & \quad \text{Imp-E} \\
\triangleright (\sigma \land (\sigma \Rightarrow \tau)) \vdash \triangleright \tau & \quad \text{Approx-Lift} \\
\triangleright \sigma \land \triangleright (\sigma \Rightarrow \tau) \vdash \triangleright \tau & \quad \text{Approx-Inter} \\
\sigma, \triangleright \sigma \land \triangleright (\sigma \Rightarrow \tau) \vdash \sigma, \triangleright \tau \quad \text{Inter-Lift} \\
\sigma, \triangleright \sigma \land \triangleright (\sigma \Rightarrow \tau) \vdash \sigma, \triangleright \tau & \quad \text{Cut} \\
\sigma \land \triangleright (\sigma \Rightarrow \tau) \vdash \tau & \quad \text{Imp-I} \\
\triangleright (\sigma \Rightarrow \tau) \vdash \sigma \Rightarrow \tau & \quad \text{Theorem 4.22} \\
\vdash \sigma \Rightarrow \tau & \quad \text{Imp-E} \\
\sigma \vdash \tau & \quad \text{Theorem 4.22}
\end{align*}
\]

We have just described all the core elements of a first-order Gödel-Löb logic. The lemmas proved above, excepting those for the derived operator \(\square\), are summarized as the subtyping rules in Figure 4.1 on the following page.

4.6 QUANTIFIED TYPES

To the first-order types above we add constructors for universal and existential impredicatively-quantified (-polymorphic) types as follows.

\[
\forall x : A . \tau \quad \text{prin} = \bigcap_{a \in A} \tau[a/x] 
\]

(4.8)

\[
\exists x : A . \tau \quad \text{prin} = \bigcup_{a \in A} \tau[a/x] 
\]

(4.9)

Here \(A \in \{\text{Type}, \text{Mem}, \text{Loc}\}\), such that it ranges over not just types, but also memories and locations, respectively. Our semantic approach allows us to condense what would otherwise be six nearly identical operators into two.

We now give some elementary subtyping rules for the quantifiers. Each quantifier has both a left and a right subtyping rule.
Lemma 4.36 (Forall-R). If \( \tau \vdash \sigma \) and \( x \) not free in \( \tau \), then \( \tau \vdash \forall x : A. \sigma \).

Lemma 4.37 (Forall-L). For any \( a \in A \), we have \( \forall x : A. \tau \vdash \tau[a/x] \).

Lemma 4.38 (Exists-L). If \( \sigma \vdash \tau \) and \( x \) not free in \( \tau \), then \( \exists x : A. \sigma \vdash \tau \).

Lemma 4.39 (Exists-R). For any \( a \in A \), we have \( \tau[a/x] \vdash \exists x : A. \tau \).

Proofs of 4.36–4.39. Directly from Definitions (4.8) and (4.9).

The quantifiers also have the following commutative properties with respect to the \( \triangleright \) operator.

Lemma 4.40 (Approx-Forall). For any type \( \tau \), we have \( \forall x : A. \triangleright \tau \vdash \triangleright \forall x : A. \tau \).

Proof. Suppose \( u \in \forall x : A. \triangleright \tau \). We are to show \( u \in \triangleright \forall x : A. \tau \). So let \( u' \in U \) with \( u R u' \); we must show \( u' \in \forall x : A. \tau \). So let \( a \in A \); we must show \( u' \in \tau[a/x] \). By our premise, \( u \in \triangleright \tau[a/x] \). But \( u R u' \), so by definition of \( \triangleright \), \( u' \in \triangleright \tau[a/x] \). But this is what we were to show.

Lemma 4.41 (Forall-Approx). For any type \( \tau \), we have \( \triangleright \forall x : A. \tau \vdash \forall x : A. \triangleright \tau \).

Proof. Suppose \( u \in \triangleright \forall x : A. \tau \). We are to show \( u \in \forall x : A. \triangleright \tau \). So let \( \alpha \in A \), and suppose \( u' \in U \) with \( u R u' \). We must show \( u' \in \tau[\alpha/x] \). But by our premise and
the fact that \( u \stackrel{R}{\leftrightarrow} u' \), we have \( u' \in \forall x : A. \tau \), and so \( u' \in \tau[a/x] \). But this is what we were to show.

**Lemma 4.42 (Approx-Exists).** For any type \( \tau \), we have \( \exists x : A. \gg \bowtie \vdash \exists x : A. \tau \).

**Proof.** Suppose \( u \in \exists x : A. \gg \bowtie \). We are to show \( u \in \gg \bowtie \exists x : A. \tau \). So let \( u' \in U \) with \( u \stackrel{R}{\leftrightarrow} u' \); we must show \( u' \in \exists x : A. \tau \). By our premise, there is an \( \alpha \in A \) such that \( u \in \gg \bowtie \tau[a/x] \). It follows that \( u' \in \tau[a/x] \). But then \( \alpha \) is a witness to \( u' \in \exists x : A. \tau \), which is what we were to show.

The subtyping rules for the quantified types appear in Figure 4.2. The following derived subtyping rule will also prove useful.

**Lemma 4.43 (Exists-LR).** If \( \tau \vdash \sigma \), with \( x \) not free in \( \sigma \) and \( y \) not free in \( \tau \), then \( \exists x : A. \tau \vdash \exists y : A. \sigma \). Or, equivalently, \( \exists x : A. \tau \vdash \exists x : A. \sigma \), for given that \( y \) is not free in \( \tau \), we are at liberty to rename the bound variable on the right of the turnstile. The use of this fact in the proof is notated \( \alpha \)-renaming.

**Proof.** By derivation.

\[
\begin{align*}
\tau \vdash \sigma & \quad \text{Lemma 4.38} \\
\exists x : A. \tau \vdash \sigma & \quad \text{Lemma 4.39} \\
\sigma \vdash \exists y : A. \sigma & \quad \text{Cut} \\
\exists x : A. \tau \vdash \exists y : A. \sigma & \quad \alpha\text{-renaming} \\
\exists x : A. \tau \vdash \exists x : A. \sigma & \quad \square
\end{align*}
\]
4.7 TYPES FOR PROPOSITIONS

Intuitively, the type constructors discussed in this section let us construct types that are always equivalent either to top or to bottom. In other words, they allows us to remove the computational aspect of a type expression and turn it into a propositional expression. For instance, given two types, we can construct the type expression that is top if and only if the two types are co-extensive, and bottom otherwise. (We will carry out this example below.)

In more precise terms, we want to be able to create a type expression whose satisfaction depends solely on the world component of its argument. The constructors that give rise to these sorts of judgements do so by ignoring entirely the “value” component of candidate configurations. These constructors, called everywhere and somewhere, respectively, are given below.

\[
! \tau \overset{\text{prim}}{=} \{(w, v) \mid \forall v'. (w, v') \in \tau\} \tag{4.10}
\]

\[
? \tau \overset{\text{prim}}{=} \{(w, v) \mid \exists v'. (w, v') \in \tau\} \tag{4.11}
\]

We call the first constructor everywhere because at a given world \(w\), \(!\tau\) either holds for any value \(v\) (if \(\tau\) itself does also), or is empty (otherwise). Likewise, we call the second constructor somewhere because it produces a type that either holds for any \(v\) or is empty, depending on whether, at a particular \(w\), \(\tau\) holds for at least one \(v\).

The core properties of everywhere and somewhere are as follows.

**Lemma 4.44 (Everywhere-Lift).** The operator \(!\) is monotone: if \(\sigma \vdash \tau\), then \(!\sigma \vdash !\tau\).

**Lemma 4.45 (Everywhere-Rep).** For any \(\tau\), \(!\tau \vdash !!\tau\).

**Lemma 4.46 (Everywhere-L).** The operator \(!\) has a left rule (but not a right rule): \(!\tau \vdash \tau\).

**Lemma 4.47 (Everywhere-Intersection).** The operator \(!\) distributes over intersection: \(!\sigma \land !\tau \vdash !((\sigma \land \tau))\).

**Lemma 4.48 (Everywhere-Top).** It is always the case that \(\top \vdash !\top\).

**Lemma 4.49 (Approx-Everywhere).** The operator \(!\) commutes with \(\triangleright\) thus: \(\triangleright\tau \vdash \triangleright !\tau\).

**Lemma 4.50 (Everywhere-Approx).** The operator \(\triangleright\) commutes with \(!\) thus: \(\triangleright !\tau \vdash !\triangleright \tau\).
### Subtyping Rules: value-irrelevant judgements

#### Lemma 4.51 (Everywhere-Implies)
The operator ! commutes with implication:

\[ !((\sigma \Rightarrow \tau) \vdash ?\sigma \Rightarrow ?\tau) \]

#### Lemma 4.52 (Somewhere-L)
The operator ? has a left (but not a right) rule: If \( \sigma \vdash !\tau \), then \( ?\sigma \vdash !\tau \).

#### Lemma 4.53 (Everywhere-Somewhere)
If a type holds at some world, it is everywhere true that it holds at some world: \( ?\tau \vdash !?\tau \).

#### Lemma 4.54 (Approx-Somewhere)
The operator ? commutes with \( \hat{\cdot} \) thus: \( ?\hat{\tau} \vdash !?\tau \).

**Proofs of 4.44–4.54.** Proved in Coq [62].

We summarize the above lemmas in rule form in Figure 4.3.

**Example.** To illustrate the use of everywhere we will define a constructor for type equality, such that given two types it yields the type expression that is top if and only if the two types are co-extensive, and bottom otherwise.

First we define a notion of value-relative equivalence.

\[ \tau \leftrightarrow \sigma \overset{\text{syn}}{=} \tau \Rightarrow \sigma \wedge \sigma \Rightarrow \tau \quad (4.12) \]

We don’t use this much except as a building block for full-blown type equality, so our explanation will be brief. Let \( \tau \) be a nontrivial type with \( v : \tau \) and \( v' \not\in \tau \). Consider the types \( \tau \) and top. The value \( v \) has type \( \tau \), and it also has type top. So
$v$ has type (is a witness to) $\tau \iff \text{top}$. By contrast, the same judgment cannot be made of $v'$, since, taking the biconditional right to left, the antecedent $v' : \text{top}$ is true but the consequent, $v' : \tau$, is false.

We then define extensional type equality in terms of equivalence as follows.

$$\tau = \sigma \overset{\text{syn}}{=} \downarrow (\tau \iff \sigma) \quad (4.13)$$

By wrapping the body of Definition (4.12) with the everywhere operator, the particular value at which we evaluate the typing judgment is made irrelevant. Instead, satisfaction of the typing judgment is relative only to an arbitrary value; hence the judgment is satisfied exactly when the two argument types agree at every value.

### 4.8 Related Work

We conclude this chapter with a discussion of some important related work. In Chapter 6 and Chapter 7, we build on our self-contained modal system so it can do something interesting, such as verify safety properties of machine language programs.

#### 4.8.1 Origin of the approximation modality

The approximation modality is due to Nakano, who developed the idea in two papers. The first paper [47] introduces a type system with recursive types and an approximation modality $\bullet$ (to which our $\triangleright$ is directly analogous), which for example allows the derivation

$$\vdash Y : (\bullet X \rightarrow X) \rightarrow X,$$

where $Y$ is the fixed-point combinator. The approximation modality allows, even in the presence of recursive types, a strongly normalizing calculus that may be sensibly used as a logic. The second paper [48] generalizes the first, and makes precise the relationship to the intuitionistic version of the Gödel-Löb logic of provability.

Our work builds on and departs from Nakano's in two principal ways. First, we use the approximation modality in calculi that are by design capable of nontermination. Second, our main contribution is to show how the approximation modality is not just a modality for recursion, but also a modality for mutable references, and in general a modality for realizing inductive properties of small-step programs.
4.8.2 A previous modal interpretation

To enrich the type system beyond just covariant recursive types and immutable references, while still retaining impredicative quantification, previous work relativized satisfaction of the typing relation to certain indices. First, relative to indices $k$ representing computation steps before the observer terminates [9]; then relative to store-typings $\Psi$ [6]. The general approach then was, in the semantic definitions of type constructors that described structured or executable data (references, code pointers, etc.), to quantify over these bounds to guarantee that the type constructor, considered as a predicate, held monotonically for as many steps as one cared for.

Ahmed [4] noticed that what the models were doing was in essence inlining the definitions of an S4-type modal operator of necessity. She showed that taken together, the index $k$ and store-typing $\Psi$ constituted a world, or context, in which to evaluate typings. Write the pair $(k, \Psi)$ as $w$, and the intersection of their individual relations as $R$, and we are able to talk in modal terms of all worlds reachable from a given world.
5 A Kripke Model of Typed Machine State

In completing the Kripke model underlying our type system, it remains to construct a suitable set of worlds $W$ and an accessibility relation $R$ on $W$ satisfying the Gödel-Löb axioms.

5.1 The Sets of Worlds and Types

The point of relativizing typing judgments to worlds is to make precise the contexts in which those judgments hold. In a language with mutable references, a typing is only valid insofar as it is consistent with the machine state, in particular the heap. We therefore introduce the notion of a store-typing $\Psi : \text{Loc} \rightarrow \text{Type}$ as a component of each world.

5.1.1 First Attempt

A naive solution suggests itself, namely that we let $W$ be $\text{Loc} \rightarrow \text{Type}$, and let Type consist of subsets of the universe $U = W \times V$, as follows.

$$
W = \text{Loc} \rightarrow \text{Type} \\
\text{Type} = \mathcal{P} (W \times V). 
$$

(5.1)

However there are no pairs $(W, \text{Type})$ satisfying Equation (5.1). To see this, identify the set Type with its characteristic proposition-valued membership function, and expand the definition of $W$; thus

$$
\text{Type} = ((\text{Loc} \rightarrow \text{Type}) \times V) \rightarrow o,
$$

where the contravariant occurrence of Type entails that it has an inconsistent cardinality.

5.1.2 Stratified worlds

In order to untangle the mutual dependence in (5.1), we follow Ahmed et al. [6] and build stratified families of sets $W_n$ and $\text{Type}_n$, as follows.

$$
W_n \overset{\text{def}}{=} \{ n \} \times (\text{Loc} \rightarrow \text{Type}_n) \\
\text{Type}_n \overset{\text{def}}{=} \mathcal{P} \left( \bigcup_{k<n} W_k \times V \right) 
$$

(5.2)
Fixpoint stype (n : nat) : Type :=
  match n with
  0 => unit
  | S n =>
      prodT (stype n)
      (prodT value
       (location -> optionT (stype n)) -> Prop)
  end.

Figure 5.1: A Coq definition of Typeₙ.

Observe that Equation (5.2) entails the base case Type₀ = \{\emptyset\}. In general, for each
n, the set of worlds Wₙ is the cartesian product of the singleton set n – representing
the number of steps before the observer’s attention span expires – and the set
of partial maps from locations to stratified types of degree n. A stratified type of
degree n is then a subset of the union over the space of configurations with worlds
of index strictly less than n. In this way the mutually recursive definition is made
well-founded.

The set of worlds W is just the union of all the Wₙ, thus

\[ W \overset{\text{def}}{=} \bigcup_{n \in \mathbb{N}} Wₙ; \]

and a type drawn from Type is a subset of configurations, thus

\[ \text{Type} \overset{\text{def}}{=} \mathcal{P}(W \times V), \]

but observe that Type is not the union of the Typeₙ. By construction, each stratified
type of a given degree is a subset of each stratified type of any greater degree,
and every stratified type is in turn a subset of Type, hence we have the (strict)
inclusion

\[ \text{Type}_0 \subset \ldots \subset \text{Type}_n \subset \ldots \subset \text{Type}. \]

As shown in Figure 5.1, the construction of the stratified sets Wₙ and Typeₙ
embeds very cleanly in the Calculus of Inductive Constructions, the dependently
typed logical framework used in Coq. This is an important property, given that we
wish to check our proofs mechanically.
5.2 APPROXIMATED TYPES

From any fully realized type we can obtain its approximation of degree \( n \). To show this we introduce the following function,

\[
\lfloor \cdot \rfloor_n : \text{Type} \rightarrow \text{Type}_n,
\]

which we define as follows.

\[
\lfloor \tau \rfloor_n \overset{\text{def}}{=} \{ ((k, \Psi), v) \in \tau \mid k < n \}.
\] (5.3)

In words, the approximation of degree \( n \) of a type \( \tau \) is the subset of its elements with index strictly less than \( n \). Informally, we think of this as the set of objects that an observer cannot distinguish from a \( \tau \) as long as the observer observes it for fewer than \( n \) steps.

We extend the notion of approximation on types to approximation on store typings in a pointwise fashion, as follows.

\[
\lfloor \Psi \rfloor_n \overset{\text{def}}{=} \begin{cases} 
\text{dom} \lfloor \Psi \rfloor_n = \text{dom} \Psi, \\
\lfloor \Psi \rfloor_n(l) = \lfloor \Psi(l) \rfloor_{n'}.
\end{cases}
\]

5.3 THE ACCESSIBILITY RELATION

We can now define \( R \), the accessibility relation on worlds:

\[
(n, \Psi) R (n', \Psi') \overset{\text{def}}{=} n > n' \land \forall l \in \text{dom} \Psi. \Psi'(l) = \lfloor \Psi(l) \rfloor_{n'}.
\] (5.4)

The relation \( R \) has two conjuncts. First, the index \( n' \) of the world on the right must be strictly less than that on the left. Second, for every location in the domain of the store typing on the left, the store typing on the right must agree to the \( n' \)th approximation.

Lemma 5.1. The relation \( R \) satisfies the axioms of 4.5.1 on page 27, namely, it is converse-wellfounded and transitive.

Proof. First to show converse well-foundedness. The relation \( R \) is converse well-founded because it is the intersection of a converse well-founded relation, \( > \), with another relation.

To show transitivity, let \((n, \Psi) R (n', \Psi') \) and \((n', \Psi') R (n'', \Psi'') \). We must show \((n, \Psi) R (n'', \Psi'') \). First to show \( n > n'' \). But by hypothesis \( n > n' \) and \( n' > n'' \),
so by transitivity of $>$ we are done. Now to show $\forall l \in \text{dom } \Psi. \Psi''(l) = \lfloor \Psi(l) \rfloor_{n''}$. Let $l \in \text{dom } \Psi$. By hypothesis, then, $\Psi'(l) = \lfloor \Psi(l) \rfloor_{n'}$ and $\Psi''(l) = \lfloor \Psi'(l) \rfloor_{n''}$. Substituting, we have $\Psi''(l) = \lfloor \lfloor \Psi(l) \rfloor_{n'} \rfloor_{n''}$. But since $n' > n''$, the right-hand side collapses to $\lfloor \Psi(l) \rfloor_{n''}$ and we are done.

If $w R w'$, we say that world $w'$ approximates world $w$.

### 5.4 Relating Types and Their Approximations

A key use of the approximation modality is to relate a type and its approximation. Recall Equation (4.6) on page 28 where we defined the approximation $\triangleright \tau$ of $\tau$ as the set of configurations $(w, v)$ such that for any $w'$, if $w R w'$ then $(w', v) \in \tau$. Recall also, as per Equation (5.3) on the preceding page, that $\lfloor \tau \rfloor_k$, the approximation of $\tau$ to degree $k$, is the subset of $\tau$ whose elements $(j, \Psi, v)$ have $j < k$.

**Lemma 5.2 (Approximation and Equality I).** A type and its $k$-approximation are related as follows:

$$(k, \Psi) \vDash \triangleright (\tau = \lfloor \tau \rfloor_k)$$

**Proof.** Let $v$ be a value, let $w = (k, \Psi)$, and let $w'$ such that $w R w'$. We must show $(w', v) \in \tau$ if and only if $(w', v) \in \lfloor \tau \rfloor_k$.

First to show the implication from left to right. Suppose $(w', v) \in \tau$. We must show $w' = (k', \Psi')$ such that $k > k'$; but $k > k'$ by our assumption that $w R w'$. Now to show the implication from right to left. Suppose $(w', v) \in \lfloor \tau \rfloor_k$. But $\lfloor \tau \rfloor_k$ is a subset of $\tau$ by construction, so $(w', v) \in \tau$, and we are done. 

Parallel to the notion of a $k$-approximation of a type, where $k$ is an integer, we move up a level of abstraction and introduce $w$-approximations, where $w$ is a world.

**Definition 5.3.** Let $\tau$ be a type and let $w$ be a world. The $w$-approximation of $\tau$, written $\lfloor \tau \rfloor_w$, is given by $\{(w', v) \in \tau \mid w R w'\}$.

**Lemma 5.4 (Approximation and Equality II).** A type and its $w$-approximation are related as follows:

$w \vDash \triangleright (\tau = \lfloor \tau \rfloor_w)$

**Proof.** Let $v$ be a value, and let $w$ and $w'$ be worlds with $w R w'$. We must show $(w', v) \in \tau$ if and only if $(w', v) \in \lfloor \tau \rfloor_w$. 

First to show the implication from left to right. Suppose \((w', v) \in \tau\). We must show \(w R w'\); but this is immediate from our assumption. Now to show the implication from right to left. Suppose \((w', v) \in [\tau]_w\). But \([\tau]_w\) is a subset of \(\tau\) by construction, so \((w', v) \in \tau\), and we are done. \(\square\)

5.5 Recursive Types

We now turn to the recursive type constructor \(\text{rec}\) and related concepts. We treat \(\text{rec}\) in this chapter because it unusual among our types for using the approximation operator \([\cdot]_w\) directly, instead of the higher-level \(\triangleright\) abstraction.

5.5.1 Contractive and nonexpansive type functions

Our model of recursive types is that of Appel and McAllester [9], except our \(\triangleright\) and \(\Box\) modalities admit a more perspicuous notation for the notions contractive and nonexpansive, as follows.

Definition 5.5. Let \(F\) be a function from types to types, and let \(\tau\) and \(\sigma\) be types. We say that \(F\) is contractive whenever

\[
\triangleright (\tau = \sigma) \vdash F \tau = F \sigma.
\]

Intuitively, if \(F\) is contractive, it adds to the concrete structure implied by its argument type. The definition captures this idea by picking out \(F\) such that if \(\tau\) and \(\sigma\) are approximately equal as types, then \(F(\tau)\) and \(F(\sigma)\) are equal. The point is that a contractive \(F\) adds a layer of indirection (as with a reference cell or a code pointer), ensuring that an additional step must be taken to resolve the indirection (as with a fetch or a jump, respectively). If a program wishes to discern an \(F(\tau)\) from an \(F(\sigma)\) – for instance, by getting stuck when expecting the former and getting the latter – it must first eliminate \(F\) by taking a step; but having done so, it cannot then discern the underlying \(\sigma\) from a \(\tau\), by our assumption that they are approximately equal.

Definition 5.6. Let \(F\) be a function from types to types, and let \(\tau\) and \(\sigma\) be types. We say that \(F\) is nonexpansive whenever

\[
\Box (\tau = \sigma) \vdash F \tau = F \sigma.
\]

Compared to a contractive type function, a nonexpansive type function does nothing to reduce the concrete structure implied by its argument type.
**Theorem 5.7.** If a type function $F$ is contractive and it has a fixpoint, the fixpoint is unique:

\[
\vdash \sigma = F(\sigma) \quad \vdash \tau = F(\tau) \\
\vdash \sigma = \tau
\]

**Proof.** By derivation.

\[
\vdash \tau = F\tau \\
\vdash \sigma = F\sigma \\
\vdash (\sigma = \tau) \vdash \sigma = F\sigma \\
\vdash (\sigma = \tau) \vdash \tau = F\tau \\
\vdash (\sigma = \tau) \vdash (\sigma = F\sigma) \land (\tau = F\tau) \\
\vdash (\sigma = \tau) \vdash (\sigma = F\sigma) \land (\tau = F\tau) \land (\tau = \tau) \\
\vdash (\sigma = \tau) \vdash (\sigma = \tau) \\
\vdash (\sigma = \tau) \vdash (\sigma = \tau) \\
\vdash (\sigma = \tau) \vdash (\sigma = \tau)
\]

Note the Approx-Fix rule used in the final step, which completely encapsulates the use of an induction principle.

**Lemma 5.8.** For any type $\tau$ and contractive type function $F$, $w \models F[\tau]_w$ if and only if $w \models F(\tau)$.

**Proof.** Directly from Definition 5.5 as applied to Lemma 5.4.

\[
\Box
\]

### 5.5.2 The recursive type constructor and its subtyping rules

The recursive type constructor $\text{rec}$ from type functions to types is given by the equation

\[
(w, v) \in \text{rec} F \iff (w, v) \in F[\text{rec} F]_w.
\]  \hspace{1cm} (5.5)

Equation (5.5) is well-defined, because the use of $[\cdot]_w$ on the right-hand side ensures that whether $(w, v) \in \text{rec} F$ depends solely on the membership of those $(w', v)$ with $w R w'$, and there are guaranteed to be finite such $w'$ by the (converse) well-foundedness of $R$.

**Theorem 5.9 (Rec-Fixpoint).** For any contractive type function $F$, the type $\text{rec} F$ is a fixpoint of $F$; that is,

\[
\vdash \text{rec} F = F(\text{rec} F).
\]

By Theorem 5.7 the fixpoint is unique.
Proof. Let \((w, v)\) be a configuration. It is sufficient to show

\[ w \models \text{rec } F \iff w \models F(\text{rec } F). \]

But

\[ w \models \text{rec } F \iff w \models F|\text{rec } F|_w, \]
\[ \iff w \models F(\text{rec } F), \]

by Definition (5.5); and

by Lemma 5.8;

and we are done.

We have completed a description of the types in our system that are agnostic of data and control. The next chapter introduces the types for data and control, allowing us to type local fragments of whole programs.
The various types that remain, while comprising those for data and control, fall into two basic categories that don’t quite coincide with “data” and “control.” Rather, we make a distinction between those types the introduction and elimination of which does not – or does – require a concrete machine instruction to be executed. For example, a register datum can be accessed directly, since in a von Neumann machine registers are always immediately at hand; but the contents of a reference need to be fetched first and cannot be accessed directly. These categories correspond, respectively, to the formal notions of nonexpansive and contractive defined in Section 5.5 on page 42. We devote a section to each in turn, first the nonexpansive types and then the contractive, the latter further broken down between references and code pointers.

6.1 NONEXPANSIVE TYPES

Nonexpansive types have in common that their models are world-agnostic, that is, though such a type comprises elements \((w, v) \in U\), it makes no reference to the world \(w\). In particular, their models make no use either of the accessibility relation \(R\) or the modality \(\Box\). Among the types we will define in this chapter, the following:

\[
\text{just } n, \quad \{v_n : \tau\}, \quad \text{and } \text{offset}(n, \tau),
\]
pertaining respectively to singleton integers, register contents, and address arithmetic, are all of this variety.

6.1.1 Singleton integers

The values \(v\) model register files, which can be considered vectors. We want to support a notion of scalar values, too, in particular to be able to describe the contents of a particular register. The consequent design decision is how to represent values in general. Obviously, in a syntactic presentation, we would add a new syntactic category with two constructors, one for scalars and one for vectors. Here we take a different tack, using the same (vector) model for both scalar and vector values, with scalars satisfying the restriction that each register in the register bank contains the same integer.

We write \(\vec{n}\) to denote the vector that maps every element of its domain to the integer \(n\). Now the type of singletons \(n\) is modeled as just that subset of
configurations \((w, v)\) with \(v = \overrightarrow{n}\), or in notation,
\[
\text{just } n \overset{\text{prim}}{=} \{(w, v) \mid v = \overrightarrow{n}\}.
\] (6.1)

**Example: general integers.** Singleton integer types are especially useful when combined with quantification. One basic application is the construction of a general type of integers
\[
\exists n: \mathbb{N}. \text{just } n.
\]
Singleton integer types can also be used in a more subtle way to support dataflow analysis, which we will discuss in Section 6.4. The subtyping rules for just \(n\) follow.

**Lemma 6.1 (Just-Unique).** For any type \(\tau\), if \(\alpha \in \text{Loc}\) is free in \(\tau\), then
\[
\text{just } l \land \text{just } l' \vdash \tau[l/\alpha] \Rightarrow \tau[l'/\alpha].
\]
This justifies our calling just \(l\) a singleton type.

**Lemma 6.2 (Approx-Just).** Singleton types are oblivious to worlds, and are therefore world-invariant. For any location \(l\), we have \(\text{just } l \vdash \Rightarrow \text{just } l\).

**Lemma 6.3 (Approx-Just-2).** When just \(l\) is on the left-hand side of an implication, \(\Rightarrow\) commutes with implication as follows:
\[
\text{just } l \Rightarrow \Rightarrow \tau \vdash (\text{just } l \Rightarrow \tau).
\]

**Lemma 6.4 (Somewhere-Just).** For any location \(l\), we have \(\text{top} \vdash ? \text{just } l\). This reflects that there is always a value with type just \(l\), namely \(\overrightarrow{l}\).

**Proof.** Proved in Coq [62].

The above lemmas are summarized in Figure 6.1 on the following page.

### 6.1.2 Internalized typing predicates

Next we derive a synthetic propositional type. It’s equivalent to top (true) if and only if in every world the singleton type just \(u\) is a subset of \(\tau\). Intuitively, this is the case exactly when \(u\) has type \(\tau\).
\[
u \in \tau \overset{\text{syn}}{=} \llbracket \text{just } u \Rightarrow \tau \rrbracket
\] (6.2)

More verbosely, \((w, v) \in u \in \tau\) exactly when for any \(v'\), if \(v' = \overrightarrow{u}\), \((w, v') \in \tau\). This expression is very similar to the typing judgment \(w \vdash v : \tau\), except the judgment \(w \vdash v : (u \in \tau)\) doesn’t depend on \(v\) and is limited to typing scalars. The corresponding subtyping rule follows.
6.1.3 Register contents

The next type constructor allows us to talk about the individual elements, or “slots”, of a vector.

\[
\{v_j : \tau\} \overset{\text{prim}}{=} \{(w, v) \mid w \triangleright v(j) \in \mathbb{Z} + \tau\}
\] (6.3)

The intuition is that if \(v\) has type \(\{v_j : \tau\}\), the scalar quantity \(v(j)\) has type \(\tau\).

**Lemma 6.6** (Slot-Lift). The slot constructor is monotone: if \(\sigma \vdash \tau\), then \(\{v_j : \sigma\} \vdash \{v_j : \tau\}\).

**Lemma 6.7** (Approx-Slot). The slot constructor commutes with \(\triangleright\): for any \(\tau\), we have \(\{v_j : \triangleright \tau\} \vdash \triangleright \{v_j : \tau\}\).

**Proof.** Proved in Coq [62].

The above lemmas are summarized in Figure 6.1.
6 TYPES FOR DATA AND CONTROL

6.1.4 Address arithmetic

To account for types representing address arithmetic and integer comparisons, we introduce a primitive, world-agnostic type that incorporates a summation relation.

\[
\text{plus}(i, j, k) = \{ (w, v) \mid i + j = k \} \tag{6.4}
\]

Now we can compose existing types to make \text{offset} as follows.

\[
\text{offset}(n, \tau) \equiv \exists \ell: \text{Loc.} \quad \text{just} \ l \land \\
\exists \ell': \text{Loc.} \quad \text{plus}(l, n, l') \land l' \approx \tau \tag{6.5}
\]

To say \( w \vdash v : \text{offset}(n, \tau) \) means that \( v \) represents a root pointer containing some address \( l \), and that the value at address \( l + n \) has type \( \tau \).

The subtyping rule for \text{offset} is as follows.

**Lemma 6.8 (Offset-Sub).** The \text{offset} constructor is monotone: If \( \tau \vdash \sigma \), then \( \text{offset}(n, \tau) \vdash \text{offset}(n, \sigma) \). In symbols,

\[
\begin{align*}
\tau & \vdash \sigma \\
\text{offset}(n, \tau) & \vdash \text{offset}(n, \sigma)
\end{align*}
\]

**Proof.** By derivation. To conserve space, we use \( P \) to stand for \( \text{plus}(\ell, n, \ell') \), and we omit the domains of quantification.

\[
\begin{align*}
\tau & \vdash \sigma \\
\ell' \approx \tau & \vdash \ell' \approx \sigma \\
\text{plus}(\ell, n, \ell') & \land \ell' \approx \tau \vdash \text{plus}(\ell, n, \ell') \land \ell' \approx \sigma \\
\exists \ell'. \text{plus}(\ell, n, \ell') & \land \ell' \approx \tau \vdash \exists \ell'. \text{plus}(\ell, n, \ell') \land \ell' \approx \sigma \\
\text{just} \ell & \land \exists \ell'. \text{plus}(\ell, n, \ell') \land \ell' \approx \tau & \vdash \text{just} \ell \land \exists \ell'. \text{plus}(\ell, n, \ell') \land \ell' \approx \sigma \\
\text{offset}(n, \tau) & \vdash \text{offset}(n, \sigma)
\end{align*}
\]

**Example: lists.** To illustrate a practical use of the constructors defined thus far, in particular \( \exists, \text{rec}, \text{just}, \) and \( \text{offset}(), \) we show how a standard list type is defined in our system. As a preliminary step we construct the type of pairs,

\[
\text{pair}(\sigma, \tau) \equiv \text{offset}(0, \text{ref} \ \sigma) \land \text{offset}(1, \text{ref} \ \tau),
\]
and then do the standard construction, abstracting over the disjoint union of a $\textit{Nil}$ element and a $\textit{Cons}$ element,

$$\begin{align*}
\text{list } \tau & \overset{\text{syn}}{=} \text{rec } \alpha. \text{ just } 0 \lor \\
& \exists l : \text{Loc}. \text{ just } l \land \text{greater}(l, 0) \\
& \land \text{pair}(\tau, \alpha).
\end{align*}$$

As the example shows, we don’t take disjoint union as primitive. Instead, we use a simple union and add conditions to the disjuncts to guarantee disjointness.

This concludes our discussion of nonexpansive types. Next we tackle contractive types: first references in Section 6.2, and then code pointers in Section 6.3 on page 53.

### 6.2 Semantics of General References

The aim of this section is to give a denotational model of $\text{ref}(\tau)$, the type of general mutable references to $\tau$. To that end we first introduce a few simpler types that will serve as building blocks.

#### 6.2.1 Location Typings

We define a new primitive type constructor, written $l \rightarrow \tau$. Informally, any value inhabits this type so long as the location $l$ has type $\tau$ in the corresponding world. In symbols,

$$l \rightarrow \tau \overset{\text{prim}}{=} \{ (n, \Psi), v \mid \Psi(l) = \tau \}. \quad (6.6)$$

**Lemma 6.9 (Store Typing Preservation).**

$$
\begin{align*}
\models l \rightarrow \tau & \vdash \exists \sigma : \text{Type}. l \rightarrow \sigma \\
\models l \rightarrow \tau & \vdash \forall \sigma : \text{Type}. l \rightarrow \sigma \Rightarrow \sigma = \tau
\end{align*}
$$

**Proof.** We assume $(k, \Psi) \models l \rightarrow \tau$, that is, $\Psi(l) = \tau$. Let $(k', \Psi')$ be a world such that $(k, \Psi) R (k', \Psi')$. By Definition (5.4), we have $\text{dom } \Psi \subset \text{dom } \Psi'$, and therefore, for some type $\sigma$, we have $\Psi'(l) = \sigma$, that is, $(k', \Psi') \models l \rightarrow \sigma$. This proves the first assertion.

Now, let $\sigma$ be a type such that $(k', \Psi') \models l \rightarrow \sigma$, that is, $\Psi'(l) = \sigma$. By Definition 5.4, $\sigma = [\tau]_{k'}$. Hence, by Lemma 5.2, $(k', \Psi') \models \sigma = \sigma$ as required to prove the second assertion. 

\[\square\]
In particular, we have the following result,

\[ l \mapsto \tau \land \triangleright (u : \tau) \vdash \triangleright \left[ \forall \sigma : \text{Type. } l \mapsto \sigma \Rightarrow \triangleright (u : \sigma) \right], \]

that if a storable value \( u \) can be stored at location \( l \) now, it can still be stored at this location in the future.

6.2.2 Valid memory-typings

Next, given a memory \( m \) mapping locations (that is, nonnegative integers) to integers, we can construct the type of configurations that are well-formed (valid) with respect to the contents of \( m \).

\[ \text{validmem}(m) \overset{\text{syn}}{=} \forall l : \text{Loc. } \forall \tau : \text{Type. } l \mapsto \tau \Rightarrow \triangleright (m(l) : \tau) \quad (6.7) \]

Here we are concerned with the relation between the contents at each location in memory and the type recorded for the same location in the configuration’s \( \Psi \) component. Intuitively we want to admit just those configurations whose \( \Psi \) is a well-typing for \( m \). More precisely, though, we require that each memory cell has approximately the type assigned to it in \( \Psi \). An approximation is sufficient because from a given point in the program’s execution, any use of the contents of memory would first require a fetch instruction; after which execution the typing would no longer be approximate.

**Lemma 6.10 (Unchanged memory).** If we have

\[ (k + 1, \Psi) \vdash \text{validmem}(m) \]

then there exists a store typing \( \Psi' \) such that

\[ (k + 1, \Psi) R (k, \Psi') \]

\[ (k, \Psi') \vdash \text{validmem}(m) \]

\[ \text{dom } \Psi' = \text{dom } \Psi \]

**Proof.** We take \( \Psi' = [\Psi]_k \). We clearly have assertions 6.10 and 6.10. We now prove assertion 6.10 using Definition 6.7. Let \( l \) be a location and \( \tau \) be a type such that \( (k, \Psi') \vdash l \mapsto \tau \). By Definition 6.6, as \( \text{dom } \Psi' = \text{dom } \Psi \), there exists \( \tau' \) such that \( (k + 1, \Psi) \vdash l \mapsto \tau' \). Furthermore, by assertion 6.10, \( (k + 1, \Psi) \vdash \triangleright (m(l) : \tau') \). Hence, by assertion 6.2.1, \( (k, \Psi') \vdash \triangleright (m(l) : \tau) \) as wanted. \( \square \)
6.2.3 Reference types

We present the type constructor for mutable references. Significantly, its definition is synthesized solely from other existing constructors.

\[ \text{ref } \tau \equiv \exists l : \text{Loc. just } l \land \exists \tau' : \text{Type. } (l \mapsto \tau' \land \triangleright \tau = \tau') \]  

(6.8)

A mutable reference type (to a particular type \( \tau \)) contains just those configurations whose value component is a location such that the configuration's store typing maps that location to a type approximately equal to \( \tau \). We don't require strict equality, because any use of the data stored in the location of interest must follow a fetch instruction, using up one tick of the observer's clock it could have otherwise used to possibly distinguish the two types.

We now show that the expected store operations are permitted. For instance, if the location \( l \) has type \( \text{ref } \tau \), the storable-value \( u \) has type \( \tau \) and the current memory \( m \) is valid, we should be able to store \( u \) at location \( l \); that is, the memory \( m[l := u] \) should be valid in some world one step after the current world.

**Lemma 6.11 (Memory update).** If the type \( \tau \) is necessary and

\[
(k + 1, \Psi) \vdash \text{validmem}(m) \\
(k + 1, \Psi) \vdash (l \mapsto \text{ref } \tau) \\
(k + 1, \Psi) \vdash (u \mapsto \tau)
\]

then there exists a store typing \( \Psi' \) such that

\[
(k + 1, \Psi) R (k, \Psi') \\
(k, \Psi') \vdash \text{validmem}(m[l := u]) \\
\text{dom } \Psi' = \text{dom } \Psi
\]

**Proof.** We take \( \Psi' = [\Psi]_k \) as for the previous Lemma 6.10. This ensures that \((k + 1, \Psi) R (k, \Psi')\) and \(\text{dom } \Psi' = \text{dom } \Psi\). We now prove assertion 6.11 using Definition 6.7. Let \( l' \) be a location and \( \tau' \) be a type such that \((k, \Psi') \vdash (l' \mapsto \tau')\). If \( l' \neq l \), we can conclude as in Lemma 6.10. So, we suppose that \( l' = l \). From assumption 6.11 and by Definition 6.8, there exists a type \( \tau'' \) such that \((k + 1, \Psi) \vdash \sigma \mapsto l \mapsto \tau'' \) and \((k + 1, \Psi) \vdash \triangleright \sigma = \tau'' \). From assumption 6.11, using the fact that \( \tau \) is necessary, we therefore get \((k + 1, \Psi) \vdash \triangleright (u \mapsto \tau'')\). Hence, by assertion 6.2.1, \((k, \Psi') \vdash \triangleright (u \mapsto \tau')\) as wanted. \( \square \)
Lemma 6.12 (Memory allocation). If the type $\tau$ is necessary, and

$$(k + 1, \Psi) \vdash \text{validmem}(m)$$
$$(k + 1, \Psi) \vdash (u \in \tau)$$
$$l \not\in \text{dom} \Psi$$

then there exists a store typing $\Psi'$ such that

$$(k + 1, \Psi) R (k, \Psi')$$
$$(k, \Psi') \vdash \text{validmem}(m[l := u])$$
$$(k, \Psi') \vdash (l \in \text{ref } \tau)$$
$$\text{dom} \Psi' = \text{dom} \Psi \cup \{l\}$$

Proof. We take $\Psi' = [\Psi]_k \cup \{l \mapsto [\tau]_k\}$. This is well-defined as $l \not\in \text{dom} \Psi'$. Clearly, we have $(k + 1, \Psi) R (k, \Psi')$ and $\text{dom} \Psi' = \text{dom} \Psi \cup \{l\}$. We now prove assertion 6.12 using Definition 6.7. Let $l'$ be a location and $\tau'$ be a type such that $(k, \Psi') \vdash l' \mapsto \tau'$. If $l' \neq l$, we can conclude as in Lemma 6.10. So, we suppose that $l' = l$. From assumption 6.12, using the fact that $\tau$ is necessary, we get $(k, \Psi') \vdash \tau = [\tau]_k$. Thus, using the fact that $\tau' = \Psi'(l) = [\tau]_k$, we have $(k, \Psi') \vdash (u \in \text{ref } \tau)$ as wanted. Finally, by Lemma 5.2, $(k, \Psi') \vdash \tau = [\tau]_k$, and by definition of $\Psi'$, $(k, \Psi') \vdash l \mapsto [\tau]_k$. Therefore, we have assertion 6.12 by Definition 6.8.

Lemma 6.13 (Memory access). If

$$w \vdash (l \in \text{ref } \tau)$$
$$w \vdash \text{validmem}(m)$$
$$w R w'$$

then

$$w' \vdash (m(l) \in \tau)$$

Proof. From assumption 6.13 there exists a type $\tau'$ such that $w \vdash l \mapsto \tau'$ and $w \vdash \tau = \tau'$. Using assumption 6.13 we get $w \vdash (m(l) \in \tau')$ and then $w \vdash \tau = (m(l) \in \tau)$. Finally, by definition of $\vdash$, $w' \vdash (m(l) \in \tau)$, as wanted.

Having thus constructed reference types and verified them to have the correct properties, we now turn to code pointers and their supporting cast.
6.3 CONTROL TYPES

Our system exports just one type for reasoning about control, namely codeptr. But codeptr is not a primitive type. Like ref, and consistent with our goals of modularity, it is defined entirely by the composition of existing types. We accomplish this by taking the notion of safety as primitive, as discussed below.

6.3.1 Safety

We inductively define safety (for some number of steps) for machine states $s$ as follows.

$$
safe_0(s) \overset{\text{def}}{=} \text{True}
$$

$$
safe_{k+1}(s) \overset{\text{def}}{=} (\exists s'. s \rightarrow s') \land \forall s'. s \rightarrow s' \Rightarrow safe_k s'
$$

Any machine state is automatically safe for zero steps. A machine state is safe for $k + 1$ steps if it can take at least one step, and if any state it steps to is, in turn, safe for $k$ steps. Our definition of safety requires safe programs to loop forever, but this is only to simplify the presentation and does not lose generality.

6.3.2 Safe memories and configurations

Using Definition (6.9), we can define a type containing just those configurations with memories safe for at least as many steps as required by their index components. The point of this is to devise small but coherent types out of which we can build a code pointer type.

Since a code pointer type is one which, if certain preconditions are satisfied, it is safe to execute, we now devise the type of safe configurations. At the core of this definition will be the notion of configurations safe with respect to a memory.

$$
safemem(m) \overset{\text{prim}}{=} \{(n, \Psi), v) \mid safe_n(m, v)\}
$$

With safemem($m$) we capture the concept of configurations that can be pushed (for an appropriate number of steps) through the operational semantics without getting stuck, when $m$ is the memory. To be precise, it classifies configurations $((n, \Psi), v)$ such that the machine state $(m, v)$ can take $n$ steps without getting stuck.

Now recall that validmem($m$) is the type of configurations valid with respect to $m$. Using validmem($m$) and safemem($m$) we construct the type of safe configurations.

$$
safe \overset{\text{syn}}{=} \forall m: \text{Mem. validmem}(m) \Rightarrow safemem(m)
$$

(6.11)
A configuration is safe if, whenever it is valid with respect to a memory \( m \), it is also safe with respect to \( m \).

### 6.3.3 Code pointers

We now have a sufficient stock of simple types from which to construct types for code pointers. In the following definition we admit precisely those configurations picking out an address \( l \) such that, if that program counter points to \( l \), the configuration is safe.

\[
\text{codeptr}(\tau) \syndef \exists l:\text{Loc. } just\ l \land \mathcal{A}(\{v_{pc} : just\ l\} \land \tau \Rightarrow \text{safe})
\] (6.12)

The subtyping rule for \( \text{codeptr} \) is as follows.

**Lemma 6.14 (Codeptr-Sub).** The \( \text{codeptr} \) constructor is anti-monotone: If \( \tau \vdash \sigma \), then \( \text{codeptr}(\sigma) \vdash \text{codeptr}(\tau) \). In symbols,

\[
\tau \vdash \sigma \quad \text{codeptr}(\sigma) \vdash \text{codeptr}(\tau).
\]

**Proof.** By derivation. We use \( P \) to abbreviate \( \{v_{pc} : just\ l\} \).

\[
\begin{array}{c}
\tau \vdash \sigma \\
\hline
\text{Lemma 4.17}
\end{array}
\]

\[
\begin{array}{c}
P \land \tau \vdash P \land \sigma \\
\hline
\text{Lemma 4.19}
\end{array}
\]

\[
\begin{array}{c}
P \land \sigma \Rightarrow \text{safe} \vdash P \land \tau \Rightarrow \text{safe} \\
\hline
\text{E'where-, Approx-Lift}
\end{array}
\]

\[
\begin{array}{c}
\mathcal{A}(P \land \sigma \Rightarrow \text{safe}) \vdash \mathcal{A}(P \land \tau \Rightarrow \text{safe}) \\
\hline
\text{Lemma 4.17}
\end{array}
\]

\[
\begin{array}{c}
just\ l \land \mathcal{A}(P \land \sigma \Rightarrow \text{safe}) \vdash just\ l \land \mathcal{A}(P \land \tau \Rightarrow \text{safe}) \\
\hline
\text{Lemma 4.43, Def. codeptr}
\end{array}
\]

Whenever we want to want to reason about a code pointer but ignore its argument type, we use the following simple corollary.

**Corollary 6.15.** The type \( \text{codeptr}(\text{bot}) \) subsumes all other code pointer types. That is, for any type \( \sigma \),

\[
\text{codeptr}(\sigma) \vdash \text{codeptr}(\text{bot}).
\]

**Proof.** By derivation.
6 TYPES FOR DATA AND CONTROL

\[
\frac{\text{Bot-L}}{\text{bot} \vdash \sigma} \quad \frac{\text{Codeptr-Sub}}{\text{codeptr}(\sigma) \vdash \text{codeptr}(\text{bot})}
\]

This concludes our construction of types for control.

6.4 EXAMPLE: DATAFLOW ANALYSIS

As an example of the modularity and power of our types, we will describe how they can be used to support dataflow analysis. Existential quantification together with singleton types permit us to assert the identity of two quantities in different parts of the machine state, and therefore to use the type system for analyzing dataflow, as explicated in Juan Chen’s thesis [21]. For example, with the register types \( \{ v_j : \tau \} \) introduced below, we know that for any integers \( m \neq n \) and \( i \neq j \), if we have two successive instructions

\[
\begin{align*}
  l : & \text{codeptr}(... \{ v_i : \text{just } n \} \land \{ v_j : \text{just } m \} \ldots) \\
  l + 1 : & \text{codeptr}(... \{ v_i : \text{just } n \} \land \{ v_j : \text{just } n \} \ldots)
\end{align*}
\]

then we know that the instruction at location \( l \) transferred the value of \( v_i \) to \( v_j \) (assuming no other register held the value \( n \)).

6.5 CONCLUSION

We have now defined all our types for data and control and given the corresponding subtyping rules. What remains is to give our typing rules for instructions, and to show how to apply both our subtyping and instruction-typing rules to prove programs safe.
7 INSTRUCTION TYPINGS

An instruction doesn’t have a type of its own, but instead relates the types that hold prior to its execution to those that hold after. In this way our rules for instruction typing are analogous to the rules of a conventional Hoare logic [31], save that they use types to express predicates on the machine state; accommodate arbitrary control flow using techniques from Tan [58]; and feature the use of the novel type constructor $\triangleright$. In the sequel we examine how our instruction-typing rules relate to rules of a conventional Hoare logic. First we review Hoare logic. We then introduce the form of our instruction-typing rules with two sketches, the first sketch showing in general terms how to transpose a Hoare rule into an analogous instruction-typing rule, and the second sketch filling in details omitted in the first to simplify the presentation. Finally we describe the actual instruction-typing rules of our system.

7.1 HOARE LOGIC

In a Hoare logic, $\{P\} S \{Q\}$ is a valid triple if, whenever the machine state can be described by the formula $P$, the statement $S$ is safe to execute, and after execution, the state of the machine can be described by formula $Q$. If the truth of a Hoare triple $\{P\} S \{Q\}$ depends on the truth of a set of premises $\{A_1, A_2, \ldots, A_n\}$, we write the corresponding rule as follows.

\[
\begin{array}{c}
A_1 \quad A_2 \quad \ldots \quad A_n \\
\{P\} S \{Q\}
\end{array}
\]  

(7.1)

In words, if $A_1$ through $A_n$ hold, then whenever $P$ describes the machine state and $S$ is safe to execute, $Q$ describes the machine state after executing $S$. Conditions $P$ and $Q$ are expressed in a suitable logic.

7.2 SKETCH I: RELATION TO HOARE LOGIC

We consider how an instruction-typing rule analogous to example (7.1) would appear in our system. Instead of formulas in an unspecified logic, $P$ and $Q$ will now be encoded as types, and the premises $A_i$ will represent subtyping judgments and side conditions. The form of the rule will be as follows, with the comma notation
below representing intersection in lieu of $\land$.

$$\begin{array}{c}
n \not\in S \\
A_1 & A_2 & \ldots & A_n \\
\end{array}$$

\[\text{ref(\text{just \(n\))}, \triangleright \text{codeptr \(Q\)} \vdash \text{codeptr \(P\)}}\] (7.2)

In words, suppose \(n\) encodes the instruction \(S\); that \(A_1\) through \(A_n\) hold; and that we are considering a location \(l\) pointing to the encoding of instruction \(S\). Then, if \textit{after executing \(S\) it is safe to jump from \(l\) to a continuation expecting the machine state to satisfy \(Q\)}, we conclude that it is safe to jump to \(l\) prior to executing \(S\) if the machine state satisfies \(P\).

Compared to example 7.1, there are four novel elements to consider in example 7.2:

1. The premise \(n \not\in S\), which lets us reason about the integer representation of the instruction within the rule, and which identifies the rule as pertaining to \(S\) and not some other instruction.
2. The expression \text{ref(\text{just \(n\))} to the left of the turnstile, which restricts the rule’s applicability to instances where the program counter points to the integer encoding \(S\). The equivalent in Hoare logic is to do pattern-matching on the syntax of \(S\) in the conclusion.
3. The use of the turnstile to separate pre- and postconditions; and the placement of the postcondition \(Q\), to the left of the precondition \(P\), reversing the usual order.
4. The use of the type constructor \(\triangleright\) to modify the postcondition \(Q\).

The last two points merit further discussion, which we take up in the following two sections.

7.2.1 \textit{The order of \(P\) and \(Q\)}

The model of computation in a standard Hoare logic is that of structured control flow without gotos or jumps, and thus its statements \(S\) all have exactly one entry point and one exit point. Thus a triple \(\{P\}S\{Q\}\) does not have to contemplate whether it is \textit{valid} to suppose the machine state satisfies \(P\). If it were invalid to assume \(P\), or equivalently, if the one entry point into \(S\) were invalid, there would be no useful way to interpret the triple.

In our system with jumps and therefore the possibility of multiple entry and exit points, we cannot simply speak of some condition \(P\) holding at this point or another in the program. Instead, we have to speak of whether it is valid to assume
start: jump walk
...
walk: v1 := m[v1 + 1] ; codeptr(⋯∧{v2:list})
      v1 := m[v1 + 1]
      ...

Figure 7.1: Jumping to a code pointer that expects a list

\[
\begin{array}{c}
\text{x} \quad \text{e} \quad \text{n} \quad \ldots \quad \text{u} \\
\end{array}
\]

\[
\begin{array}{c}
v1 \\
\end{array}
\]

\[
\quad k \text{ nodes}
\]

Figure 7.2: A k-node list, but for a garbage pointer at the end

P. Wrapping P with a codeptr and reasoning in terms of continuations achieves exactly the desired effect. Of course, from Lemma 6.14 we know that codeptr is contravariant with respect to subtyping, so instead of reasoning forwards from precondition P to postcondition Q as in the world of ordinary Hoare logic, we follows Tan [58] in reasoning “backwards” from codeptr Q to codeptr P. Or rather from \(\triangleright\) codeptr Q, a subtlety we discuss presently.

7.2.2 Approximately codeptr Q

To explain the import of \(\triangleright\) codeptr Q we first review the notion of safety. As in Section 4.5.1 on page 27, we say that a program is safe if for any k, no observation lasting k steps (a k-observation) will find the program in a stuck state. Returning to the matter of \(\triangleright\) codeptr Q, suppose we are in the context of a k + 1-observation, where control is at the start label (that is, code pointer) of Figure 7.1, and is about to transfer to the code pointer walk, which expects a list in v1. As per Figure 7.2, register v1 points to what would be a list of k nodes, except that the tail of the last node points to garbage instead of containing a nil constant.

Despite this defect in the list structure, however, the computation is safe in the context of a k + 1-observation. After the jump to walk, one step having been executed, we will have a k-observation, and the program cannot traverse the list to the point of fetching the garbage pointer in k steps or fewer. We say that the list-like structure is “approximately” a list, and write its type as \(\triangleright\) list, meaning that, when considering the soundness this jump to walk, it’s as good as a list as
long as the program first takes at least one step. The jump is such a step, and so the structure is good enough – the program is safe.

Now when considering the general case of an instruction typing rule, we observe that the postcondition assumption $\text{codeptr}Q$ is too strong, in the sense that the well-typedness of the next instruction to be executed isn’t required until after the current instruction has completed. Therefore we assume not $\text{codeptr}Q$ but $\triangleright \text{codeptr}Q$, the one-step approximation of $\text{codeptr}Q$

Restating the list example in more general terms, $\triangleright \tau$ is the type such that an observer with finite attention-span cannot distinguish its elements from those of $\tau$, provided the observer first waits at least one step. Because the observer’s attention-span is finite, waiting a step may cause his attention to lapse; and critically, a lapse of attention validates any typing.

7.3 SKETCH II: ADDING DETAILS

Our first sketch glossed over some boilerplate details which we now fill in. Refining rule (7.2) yields the following.

$$
\frac{n \not\vdash S}{\nu_{pc} \not\in \text{dom } \phi} \quad \text{ref(\text{just } n), } \triangleright \text{offset}(1, \text{codeptr}(\phi \land Q)) \vdash \text{codeptr}(\phi \land P)
$$

The new symbol $\phi$ expresses as a type the register invariants preserved by the instruction $S$. The added premise stipulates that the program counter is permitted to vary. Last, the (one-step approximate) postcondition continuation type is attached to the adjacent address, reflecting that the postcondition of the current instruction is derived from the type of the following instruction.

To “permit” the program counter to vary, and in general to give the same “permission” to the parts of the state that must change as a result of executing an instruction, we stipulate that they are not in the domain of $\phi$, a relation defined as follows.

$$
i \not\in \text{dom } \phi \overset{\text{def}}{=} \forall w, v, u. (w, v) \in \phi \Rightarrow (w, v[i := u]) \in \phi
$$

We say a slot $i$ is not in the domain of type $\phi$ if, for any configuration already in $\phi$, arbitrarily varying the value at slot $i$ of its vector component results in a configuration still in $\phi$. In other words, $\phi$ “doesn’t care” about slot $i$. 
7 INSTRUCTION TYPINGS

Figure 7.3: Schematic diagram of the \texttt{I-LOAD} rule. Expressions in dashed boxes are approximated (in the sense of $\triangleright$).

7.4 INSTRUCTION-TYING RULES

We now describe the instruction-typing rules actually present in our system, one each for \texttt{LOAD}, \texttt{STORE}, \texttt{ADDMEM}, and \texttt{JUMP}. These rules take the form of refinements to the sketch in Section 7.3. We discuss the \texttt{I-LOAD} rule in detail, and give concise explanations of the remaining rules.

\textit{Lemma 7.1 (I-LOAD).} The instruction-typing rule for \texttt{LOAD} is as follows.

\[
   \begin{array}{c}
   n \not\in v_i := m[v_j + k] \\
   i \not\in \text{pc} \\
   v_i \not\in \text{dom } \phi \\
   v_{\text{pc}} \not\in \text{dom } \phi \\
   \text{ref(just } n) \triangleright \text{offset}(1, \text{codeptr}(\phi \land \{v_i : \tau\})) \\
   \vdash \text{codeptr}(\phi \land \{v_j : \text{offset}(k, \text{ref } \tau)\})
   \end{array}
\]

\textit{Proof.} Proved in Coq [62].

In words, suppose integer \( n \) encodes the instruction to load into register \( v_i \) the contents of the memory cell at offset \( k \) from location \( j \), and that \( v_i \) is not the program counter – after all, this is not the jump rule. Since the point of the load is to modify \( v_i \), we further assume that \( v_i \) is allowed to vary, and similarly for \( v_{\text{pc}} \), since the program counter will increment as a result of executing the load instruction.

From these premises it follows that given a location \( l \), if \( l \) points to the encoded \texttt{LOAD} instruction, and it is \textit{approximately} the case that the instruction at location
$l + 1$ is a code pointer expecting a \( \tau \) in \( v_i \), we have immediately that \( l \) is a code pointer expecting a location \( p \) in \( v_j \) such that location \( p + k \) points to a reference cell containing a \( \tau \). These facts are displayed schematically in Figure 7.3 on the preceding page.

The key point is that it suffices to assume a weaker, approximate typing for the next instruction because we will have taken a step before reaching it. With that step used up, the instruction’s typing will be indistinguishable from a non-approximate one. Note also the inclusion of \( \phi \) as a conjunct in the type of the instruction’s input and output types, appropriately guarded by clauses excluding from its domain the explicitly-modified \( v_i \) and the implicitly-modified \( v_{pc} \). Adding \( \phi \) in this way expresses that the instruction leaves every other slot unchanged.

**Lemma 7.2 (i-store).** The instruction-typing rule for \texttt{store} is as follows.

$$
\frac{n \vdash m[v_i + k] := v_j \quad v_{pc} \not\in \text{dom } \phi}{
\text{ref}(\text{just } n), \triangleright \text{offset}(1, \text{codeptr } \phi) \quad \vdash \text{codeptr}(\phi \land \{v_i : \text{offset}(k, \text{ref } \tau)\} \land \{v_j : \tau\})}
$$

*Proof.* Proved in Coq [62].

The intuitive meaning of the rule is similar to that of \texttt{i-load}, with the following difference. The \texttt{load} instruction modifies a register, and since the type of a register may vary during execution, the \texttt{i-load} rule has to account for the possibility that the type of the assigned register might change as a result of execution. Since the \texttt{store} instruction modifies the memory instead of a register, and the type of a memory cell may not vary during execution, the postcondition in \texttt{i-store} is simply the invariant \( \phi \).

**Lemma 7.3 (i-addimm).** The instruction-typing rule for \texttt{addimm} is as follows.

$$
\frac{n \vdash v_i := v_j + k \quad i \neq \text{pc}}{
\text{ref}(\text{just } n), \triangleright \text{offset}(1, \text{codeptr } (\phi \land \{v_i : \text{int}\})) \quad \vdash \text{codeptr}(\phi \land \{v_j : \text{int}\})}
$$

*Proof.* Proved in Coq [62].
The intuitive meaning of the rule is similar to that of i-load, the only difference being that the precondition for issuing an add-immediate instruction requires the source register \( v_j \) to hold an integer instead of a pointer to a reference cell.

**Lemma 7.4 (i-jump).** The instruction-typing rule for \texttt{jump} is as follows.

\[
\begin{array}{c}
\text{n \ $?\$ } \text{jump } v_i \\
\text{i \neq pc} \quad \text{vpc \not\in \text{dom } \phi} \\
\text{ref(just n), \triangleright offset(1, codeptr(bot))} \\
\text{\vdash codeptr(\phi \land \{v_i : \text{codeptr } \phi\})}
\end{array}
\]

**Proof.** Proved in Coq [62]. \qed

The rule for \texttt{jump} expresses that it is safe to arrive at a jump instruction with argument \( v_i \) only if we know \( v_i \) to be a code pointer compatible with the invariant \( \phi \). Since control does not necessarily pass to the instruction adjacent to and immediately following the jump, its type doesn't usefully serve as a postcondition. Therefore we suppose it has, after one step, the type \text{codeptr}(\text{bot})\), which by Corollary 6.15 types any instruction.

The instruction-typing rules for \texttt{STANLEY} are summarized in Figure 7.4 on the next page. We have now presented all of the rules of our \texttt{TAL}, both its subtyping rules and its instruction-typing rules. In the next chapter we show how to use the rules to verify the safety of whole programs.
Figure 7.4: Instruction-typing rules for STANLEY
This chapter explains the technique for using the TAL rules to prove whole programs safe. The next section establishes a running example to make the discussion concrete. Subsequent sections discuss how to represent the syntax of whole von Neumann programs; how to represent the static semantics for such programs; how to define whole-program safety; and how to prove the soundness theorem.

8.1 Running Example

To help illustrate how to reason about the safety of programs in our system, we give a program fragment annotated with pseudocode.

\[
\begin{array}{ccc}
\text{Machine inst.} & i & \text{Pseudocode} \\
7320 & \text{LOAD } 3, 2, 0 & r_3 := m(r_2 + 0) \\
4231 & \text{ADDEMM } 2, 3, 1 & r_2 := r_3 + 1 \\
6007 & \text{JUMP } 7 & \text{pc := } r_7 \\
\end{array}
\]

(8.1)

This fragment fetches the value in memory pointed to by \( r_2 \) into \( r_3 \), moves that same value incremented by one into \( r_2 \), and then jumps to the address in \( r_7 \). Consistent with this example would be part of a subroutine that performs a calculation on its argument, leaves the result in \( r_2 \), and then jumps to a return address.

8.2 Encoding Von Neumann Program Syntax with Types

Let us represent a von Neumann program as a finite map \( p : \mathbb{N} \rightarrow \mathbb{N} \) taking \( i \) to the \( i \)th machine instruction in the program. When loaded into memory, the machine instructions \( p(i) \) are laid out in an array at some offset to the program’s base address. Without loss of generality, we henceforth assume a base address of zero.

Definition 8.1 (\( \Delta \)-Notation). If \( i \in \text{dom} \ p \), \( i \) is a reference to a singleton integer \( p(i) \) located \( i \) words from the beginning of the program. We call the type that captures this state of affairs \( \Delta_i \), defined as

\[
\Delta_i \equiv \text{offset}(i, \text{ref}(\text{just} \ p(i))) .
\]

The entire program in \( \Delta \)-notation is thus given by

\[
\Delta_p \equiv \bigwedge_{i \in \text{dom} \ p} \Delta_i .
\]
Since each program $p$, and therefore each $\text{dom}_p$, is finite, this definition always expands to a type expression composed solely of types in the exported TAL interface, namely $\land$, offset, ref, and just.

The instructions in the example fragment are encoded as the integers of 7320, 4231, and 6007, in order. Thus, by setting

$$
\begin{align*}
\Delta_0 &= \text{offset}(0, \text{ref}(\text{just} 7320)) , \\
\Delta_1 &= \text{offset}(1, \text{ref}(\text{just} 4231)) , \\
\Delta_2 &= \text{offset}(2, \text{ref}(\text{just} 6007)) , \\
\Delta_p &= \Delta_0 \land \Delta_1 \land \Delta_2 ,
\end{align*}
$$

we obtain the $\Delta$-type for example 8.1.

### 8.3 Von Neumann Program Static Semantics

The $\Delta$-notation characterizes the instructions of a program $p$ in terms of their layout in memory as machine words. We can also characterize the instructions of $p$ in terms of the preconditions for their safety when they are interpreted semantically.

**Definition 8.2 (Γ-Notation).** If $p(i) \ni \iota_i$, and the precondition for the safety of $\iota_i$ is given by $\phi_i$, we call the induced type $\Gamma_i$, which is defined as

$$
\Gamma_i \xrightarrow{\text{syn}} \text{offset}(i, \text{codeptr} \phi_i).
$$

The entire program in $\Gamma$-notation is thus given by

$$
\Gamma_p \xrightarrow{\text{syn}} \bigwedge_{i \in \text{dom}_p} \Gamma_i .
$$

Since each program, and therefore the domain of $p$, is finite, this definition always expands to a type expression composed solely of types in the exported TAL interface, namely $\land$, offset, and codeptr.

In our example, the first instruction, a load, requires that its source register $v_2$ be a reference to an integer; the second, an add-immediate, that its source register $v_3$ be an integer; and the third, a jump, that the return address in $v_7$ be a code
pointer expecting an integer \( v_2 \). Therefore we set
\[
\begin{align*}
\Gamma_0 &= \text{offset}(0, \text{codeptr}(\sigma \land \{ v_2 : \text{ref(int)} \})) , \\
\Gamma_1 &= \text{offset}(1, \text{codeptr}(\sigma \land \{ v_3 : \text{int} \})) , \\
\Gamma_2 &= \text{offset}(2, \text{codeptr}(\sigma \land \{ v_2 : \text{int} \})) , \\
\Gamma_p &= \Gamma_0 \land \Gamma_1 \land \Gamma_2 , \\
\end{align*}
\]
where \( \sigma = \{ v_7 : \text{codeptr}(\{ v_2 : \text{int} \}) \} , \)

and so obtain the \( \Gamma \)-type for example 8.1.

In practice, a type-preserving compiler is responsible for translating a well-typed source program in a sound type system into a machine program with the corresponding \( \Gamma_i \) types. For more on the production of \( \Gamma_p \) in the context of a type-preserving compiler, see Morrisett et al. [46] and Chen [21].

### 8.4 Safety expressed as subtyping

Once we have both the \( \Delta_p \) and \( \Gamma_p \) types established, we want to show that \( \Delta_p \) entails \( \Gamma_p \) — meaning that the type of the loaded program, considered as an array of machine words, is a subtype of programs having the aggregate code pointer type \( \Gamma_p \).

**Definition 8.3 (Safe Programs).** Given a program \( p \), we say that \( p \) is safe whenever
\[
\Delta_p \vdash \Gamma_p .
\]

As a first step in showing \( \Delta_p \vdash \Gamma_p \), we consider how to show \( \Delta_i \vdash \Gamma_i \) for each \( i \in \text{dom} \ p \). Observe that at each \( i \), in order to satisfy \( \Gamma_i \) we get to assume the satisfaction not only of \( \Delta_i \) but also the approximation \( \Gamma_i \vdash \Gamma_{i+1} \), because the program will have stepped at least once before reaching the instruction typed by \( \Gamma_{i+1} \). Hence our example fragment gives us the following proof obligations. Again we use a comma to stand for intersection.
\[
\begin{align*}
\Delta_0 \vdash \Gamma_1 \vdash \Gamma_0 \\
\Delta_1 \vdash \Gamma_2 \vdash \Gamma_1 \\
\Delta_2 \vdash \Gamma_2
\end{align*}
\]

Once these are proved, we can add conjuncts constituting the remainder of \( \Delta_p \) to the left-hand side of each turnstile by weakening, and thus account for the entire loaded binary image. Likewise, assuming conservatively that the \( j \)th
instruction could be the target of a jump for any $j$, we can add conjuncts of the form $\triangleright \Gamma_j$ to the left-hand side. For each $i \in \text{dom } p$, we now have
\[
\left[ \bigwedge_{j \in \text{dom } p} \Delta_j \right], \left[ \bigwedge_{j \in \text{dom } p} \triangleright \Gamma_j \right] \vdash \Gamma_i,
\]
which simplifies to
\[
\Delta_p, \left[ \bigwedge_{j \in \text{dom } p} \triangleright \Gamma_j \right] \vdash \Gamma_i.
\]
But now by Lemma 4.21, the distributive property of $\triangleright$ over intersection, we can move the $\triangleright$ operator to the outside of the intersection, thus
\[
\Delta_p, \triangleright \left[ \bigwedge_{j \in \text{dom } p} \Gamma_j \right] \vdash \Gamma_i,
\]
which in turn simplifies to
\[
\Delta_p, \triangleright \Gamma_p \vdash \Gamma_i.
\]
Since we have (8.2) for each $i \in \text{dom } p$, repeated applications of the Inter-R rule yield
\[
\Delta_p, \triangleright \Gamma_p \vdash \bigwedge_{i \in \text{dom } p} \Gamma_i, \quad \text{or equivalently,} \quad \Delta_p, \triangleright \Gamma_p \vdash \Gamma_p.
\]
But then by Lemma 4.35, the generalized Löb rule, we have
\[
\Delta_p \vdash \Gamma_p,
\]
which is what we wanted to show.

To conclude this section, we present the derivation of the safety proof for example 8.1 in Figure 8.1 on the following page.

### 8.5 Soundness Theorem

Finally, having shown a program safe, we want to know that once it is initially loaded, execution can begin without assuming anything about the state of the machine.
Lemma 8.4. Given a program \( p \), if \( p \) is safe, then

\[
\Delta_p \vdash \text{codeptr}(\bot).
\]

Proof. By derivation.

\[
\Delta_p \vdash \Gamma_p \quad \Gamma_p \vdash \text{codeptr}(\bot) \quad \text{Corollary 6.15}
\]

\[
\Delta_p \vdash \text{codeptr}(\bot) \quad \text{Cut}
\]

We will show that Lemma 8.4 implies the safety of executing the program under minimal assumptions. To begin this process, Lemma 8.5 establishes that for any program loaded into memory, we can construct a suitable world to start from.

Lemma 8.5. If program \( p \) is loaded at \( l \) in machine-state \((m, v)\), then for any natural number \( k \) one can construct a world \((k, \Psi)\) such that,

\[
(k, \Psi) \models \text{validmem}(m) \quad (8.3)
\]

\[
(k, \Psi) \models v : \{v_{pc} : \Delta_p\} \quad (8.4)
\]

Proof. The type constructors used in \( \Delta_p \) are very simple: just intersection, \( \text{ref} \), and \( \text{just} \). Therefore we can use a simple construction of \( \Psi \):

\[
\Psi = \{(l + i) \mapsto \text{just} p(i) \}_k \mid i \in \text{dom} p\}
\]

Figure 8.1: Deriving the safety of example (8.1) on page 64. We omit the manipulation of offsets and the subproof that \( \Gamma_1 \) is a subtype of the I-Load postcondition.
We first check Equation (8.3), that is, by Equation (6.7), if $(k, \Psi) \models l \rightarrow \tau$, then $(k, \Psi) \models \triangleright(m(l') \varepsilon \tau)$. By Definition (6.6), when the hypothesis holds, there exists $i \in \text{dom } p$ such that $l' = l + i$ and $\tau = [\mathit{just}(p(i))]_k$. The conclusion then becomes $(k, \Psi) \models (m(l + i) \varepsilon [\mathit{just}(p(i))]_k)$, that is, $(k, \Psi) \models \triangleright(\mathit{just}(p(i)) \Rightarrow [\mathit{just}(p(i))]_k)$. This is a consequence of Lemma 5.2.

We now consider Equation (8.4). Let $i \in \text{dom } p$ and $v'$ be a value. We have

$$(k, \Psi) \models l + i \rightarrow [\mathit{just}(p(i))]_k$$

$$(k, \Psi) \models \triangleright \mathit{just}(p(i)) = [\mathit{just}(p(i))]_k.$$  

by Definition 6.6 and Lemma 5.2. Therefore, by Equation (6.8), $(k, \Psi) \models l + i \varepsilon \mathit{ref}(\mathit{just}(p(i)))$. Hence, by Equation (6.5), $(k, \Psi) \models l \varepsilon \mathit{offset}(i, \mathit{ref}(\mathit{just}(p(i))))$ for all $i \in \text{dom } p$. By intersection, we get $(k, \Psi) \models l \varepsilon \Delta_p$. We conclude by Equation (6.3).

\[\square\]

**Theorem 8.6 (Soundness).** If program $p$ is loaded at $l$ in machine-state $(m, v)$, and $\Delta_p \vdash \mathit{codeptr}(\text{bot})$, then state $(m, v)$ is safe (for any number of steps $k$).

**Proof.** By Lemma 8.5, construct the world $w = (k + 1, \Psi)$. By Lemma 6.10, there exists a world $w' = (k', \Psi')$ such that $w R w'$ and $w' \models \mathit{validmem}(m)$. We have $w \models v : \{v_{\mathit{pc}} : \Delta_p\}$, and thus $w \models v : \{v_{\mathit{pc}} : \mathit{codeptr}(\text{bot})\}$. As the program is loaded at $l$, we also have $w \models v : \{v_{\mathit{pc}} : \mathit{just}l\}$. By Equation (6.12) we get $w' \models v : \mathit{safe}$, so by Equation (6.11) we have $w' \models v : \mathit{safemem}(m)$.

This concludes the presentation of our TAL.
9 FOUNDATIONAL PROOF CARRYING CODE

9.1 RELATED TALS WITH SYNTACTIC PROOFS

We just showed a semantic proof of soundness. Now to compare this with the following systems, with similar goals, that employ the standard syntactic method.

9.1.1 Original TAL

The TAL of Morrisett et al. [46] is not actually foundational, in two respects. First, because the type system’s safety proof is of the pen and paper, rigorous-argument variety – in other words, a standard mathematical proof – and is not machine-checked, the type system is incorporated into the trusted base. Second, the lowest level of the calculus is not directly executable machine code, as it contains macro instructions, for instance malloc to allocate memory.

9.1.2 Crary’s TALT

The TALT system [24, 25] comprises three layers. At the top is XTALT, a typed assembly language that enjoys decidable type-checking. The XTALT language is then proved sound with respect to TALT proper, which is more theoretically elegant, admits simpler proofs, but is less well suited to serve as the output of a compiler. The typing rules of TALT are proved sound with respect to the operational semantics of an abstract machine, which is in turn shown to be a faithful simulation of a safe subset of the IA-32 operational semantics. This system is foundational in the sense that the soundness of the typing rules is not trusted but rather verified using the Twelf [54] metatheory.

At this writing Twelf does not produce a witness for its metatheoretic proofs (though it could in principle). The lack of a witness means the proof can not be mechanically verified independently, hence Twelf in its entirety is incorporated into the trusted base. An independent verifier could be much smaller [11], since it would not have to perform term reconstruction; implement a Prolog-like operational semantics with backtracking; or indeed, construct the proof to be verified from a raw set of rules, mode annotations, and hints about the structure of typing contexts.

That said, Crary’s metatheoretic approach has the advantage of promoting rapid development. His system required roughly two person-years to complete; by contrast, the Princeton FPCC project required at least 20.
9.1.3 **Shao’s XCAP**

Shao et al.’s Certified Assembly Programming [51, 66] is based directly on the Calculus of Inductive Constructions. Like our system, CAP/XCAP supports separate verification of code modules, permits impredicative quantification and (with extensions) mutable references, and is expressive enough to specify invariants for assembly code. But impredicativity is achieved by means of a syntactic specification of validity rules for impredicative propositions. This method does not permit elimination rules for the quantifiers, but they show how to work around the lack of elimination rules when existentials appear at top level in Hoare rules, which is good enough for a TAL.

### 9.2 Princeton FPCC

In this section we set this thesis in the context of the Princeton FPCC project and its various results in the form of previous theses. We divide our improvements over previous work into two categories, those pertaining to practical advances and those pertaining to conceptual refinement.

#### 9.2.1 Practical advances

Intuitively, codeptr(A) can be thought of as an abbreviation for the fictional type \( A \rightarrow \text{Answer} \). With this in mind, and recalling the rules for quantifier exchange in first-order predicate logic, we would expect the following type isomorphism in a system with continuation types and polymorphism:

\[
\text{codeptr}(\exists \alpha. F(\alpha)) \cong \forall \alpha. \text{codeptr}(F(\alpha)).
\]

Now consider in particular a special case of the above,

\[
\text{codeptr}(\exists \alpha: \text{Type. } \alpha \times A \times \text{codeptr}(B))
\cong \forall \alpha: \text{Type. } \text{codeptr}(\alpha \times A \times \text{codeptr}(B)) \tag{9.1}
\]

where the left and right-hand sides are De Morgan dual ways of writing the CPS- and closure-converted type of a function \( f : A \rightarrow B \). The previous system [3] failed to validate this equivalence, a side effect of having embedded Ahmed’s semantics into an insufficiently convenient logic, a formulation of Church’s higher-order logic (HOL). Since we model a type as a indexed family of sets, our model is easily expressed in a logic with dependent types, but has defied formalization in HOL.
without recourse to a complicated work-around, such as the Gödel-numbering scheme used in the Princeton FPCC project.

In our system, by contrast, both isomorphisms 9.2.1 and 9.1 hold. Furthermore, none of the operations on values of quantified type – existential pack and unpack, type abstraction and application – require taking a step in the operational semantics. (As types are not manifest at runtime in a statically checked system, such steps would be effective no-ops, and therefore a nuisance.) To have both of these properties simultaneously is a technical advance over previous work \[4\], which presented two alternative models of quantifiers, each of which possessed one property but not the other.

We believe Ahmed’s set-theoretic model can be adjusted (by simplifying the model of quantified types) to eliminate this nuisance. However, it is not obvious how to directly formalize such a repaired model – or equally, the mathematics of Chapter 5 – in a logic without dependent types, in particular HOL. To our knowledge, no previous attempt to express Equation (5.2) in HOL could do better than the solutionless 5.1. To avoid circularity in the HOL representation, Ahmed chooses the range of (HOL-encoded) \(\Psi\) to be a set of reified syntactic type expressions. This causes the failure of Equation (9.1) – in particular, the left side \((\text{codeptr}(\exists \alpha \ldots))\) has no useful semantics, while the right side \((\forall \alpha.\text{codeptr}(\ldots))\) has the expected semantics. The Princeton FPCC compiler \[22\] works around this problem by carefully using only the \(\forall\text{codeptr}\) formulation of function-closures.

### 9.2.2 Conceptual parsimony

Previous work in the FPCC project has also attempted to abstract away from the explicit stratification first introduced in the indexed model of Appel and McAllester \[9\], with the results being somewhat less orthogonal than those of the present work.

**Swadi’s subtype induction.** Swadi \[57\] is concerned with proving a rule directly analogous to our generalized Löb rule. He frames it as an explicit induction, with the base case \(\Gamma_0\) and the induction hypothesis expressed as a specialized form of subtyping, called “subtype-plus”, defined as follows.

\[
\Gamma_0 \equiv \forall \rho, k, v. (\rho, k, v) \in \Gamma
\]

\[
\Gamma_1 \subseteq \Gamma_2 \equiv \forall \rho, k, v. (\rho, k + 1, v) \in \Gamma
\]
The subtype-plus relation combines the notions of approximation and ordinary subtyping. Thus, Swadi has the rule

$$\Gamma \vdash \Delta \rightarrow \Delta \subseteq \Gamma$$

whereas we have

$$\Delta \rightarrow \Gamma \vdash \Delta \subseteq \Gamma$$

which cleanly separates those two notions, and requires just one form of subtyping (here expressed as entailment).

Ahmed’s modal presentation. In our terms, Ahmed [4] takes necessity as primitive. There is no analog to our “approximately” operator, $\triangleright$, in terms of which “necessarily”, $\Box$, can be defined. Since the accessibility relation $R_A$ corresponding to Ahmed’s modality is reflexive, there exist infinite non-increasing chains

$$w R_A w' R_A w'' R_A \ldots$$

and therefore that relation is unsuited to provide an induction principle. By taking $\triangleright$ as primitive we can express both the $\Box$ operator and the induction principle without leaving the type system.

Tan’s turnstile. In his logic for control flow [58], Tan defines a Hoare-style judgment

$$F; \Psi' \vdash \Psi,$$

where $F$ represents the syntax of the program of interest, analogous to our $\Delta$; and, continuing left to right, $\Psi'$ and $\Psi$ are the postcondition and precondition, respectively. Here, as in our own system, the judgment actually pertains to the one-step approximation of the postcondition. But in contrast to our system, where we would write

$$F; \triangleright \Psi' \vdash \Psi,$$

the turnstile is defined such that the notion of approximation is inseparable from that of entailment.

Because his aim is not to treat the semantics of types per se, and has no other occasion to use approximation, it doesn’t subtract from his system’s modularity that the notion of approximation is built into the semantics of the turnstile. However, making the approximation explicit, as we do, increases clarity by making it more apparent to the reader that the judgment is not actually circular.

The proof technique described in Section 8.3 on page 65 is adapted from that of Tan et al. [60], where it used the subtype-plus operator and subtype-induction described on the preceding page.
9.3 FURTHER CONTRIBUTIONS

The author made significant contributions to the Princeton FPCC project, which constitute a significant part of his Ph.D. work. These include work on the semantics of Typed Machine Language [57] generally, and on its kinding system in particular [10]; and on the implementation of the system as a whole [3].
10 CONCLUSION

We have demonstrated a semantic model of types sufficient to prove the soundness of a full-featured TAL, one targetable by a certifying Java or ML compiler. To that end, the model simultaneously includes type constructors for recursive data structures, impredicative polymorphism, and general references. Our model is the first to do so without awkward encodings or side conditions, and it makes an important connection in identifying the approximation modality of Gödel-Löb and Nakano as an elegant way to abstract over the bookkeeping details endemic to step-indexed reasoning.

10.1 shortcomings

We do not show how to deallocate memory. Ahmed [4] shows how to encode a region calculus, and recently Hawblitzel and Petrank [29] demonstrated a technique for verifying industrial-strength garbage collectors automatically and mechanically. However, they use Z3 [26] as a verifier, and it's unclear how to integrate the fact of a collector's verification into a semantic proof written in Coq (for example). McCreight et al. [39] show a garbage collector verified in Coq, but their collector is too limited for practical use.

The well-formedness of recursive types depends on a contractiveness side condition. As a remedy, we could insert a contractiveness assertion into the definition of the recursive type constructor. In that scenario, \( \text{rec } F \) with \( F \) not contractive would have the well-defined semantics of \( \text{bot} \).

We show type- and memory-safety only; it would be nontrivial to extend the system to permit relational parametricity results, or in general any result that typically follows from a system of logical relations.

Some of these shortcomings are addressed in recent work by Ni [50]; Petersen et al. [53]; Birkedal et al. [16] and Benton and Tabareau [15].

10.2 ON THE UNSEEMLY EFFECTIVENESS OF APPROXIMATION IN TYPE SEMANTICS

Evidently the approximation modality is sufficient to solve a number of hard problems that have plagued advanced type theories. The reader may not unreasonably wonder whether it is more powerful than required – whether lighter-weight solutions exist. We leave the resolution of this question to future work. However we remark that the modality recommends itself as a single, reusable component that
performs effectively without introducing any adverse conditions on the theory or the maintenance of its machine-checked proofs.


