Semantic Foundations for Typed Assembly Languages

AMAL AHMED, ANDREW W. APPEL, CHRISTOPHER D. RICHARDS,
KEDAR N. SWADI, GANG TAN, and DANIEL C. WANG
Princeton University

Typed Assembly Languages (TALs) are used to validate the safety of machine-language programs. The Foundational Proof-Carrying Code project seeks to verify the soundness of TALs using the smallest possible set of axioms—the axioms of a suitably expressive logic plus a specification of machine semantics. This paper proposes general semantic foundations that permit modular proofs of the soundness of TALs. These semantic foundations include Typed Machine Language (TML), a type theory for specifying properties of low-level data with powerful and orthogonal type constructors, and $\mathcal{L}_c$, a compositional logic for specifying properties of machine instructions with simplified reasoning about unstructured control flow. Both of these components, whose semantics we specify using higher-order logic, are useful for proving the soundness of TALs. We demonstrate this by using TML and $\mathcal{L}_c$ to verify the soundness of a low-level, typed assembly language, LTAL, which is the target of our core-ML-to-SPARC compiler.

To prove the soundness of the TML type system we have successfully applied a new approach, that of *step-indexed logical relations.* This approach provides the first semantic model for a type system with updatable references to values of impredicative quantified types. Both impredicative polymorphism and mutable references are essential when representing function closures in compilers with typed closure conversion, or when compiling objects to simpler typed primitives.

Categories and Subject Descriptors: F.3.2 [**Logics and meanings of programs**]: Semantics of Programming Languages; D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms: Languages, Verification

Additional Key Words and Phrases: Typed Assembly Languages, Proof-Carrying Code, Semantic Models, Logical Relations, Control Flow

1. INTRODUCTION

Typed Assembly Languages (TALs) [Morrisett et al. 1999; Morrisett et al. 2002; Morrisett et al. 1999; Hamid et al. 2002; Crary 2003; Chen et al. 2003] are used to prove the safety of machine-language programs. The *soundness* property for a typed assembly language is that if a TAL program type checks, then the corresponding machine-language program obeys a predetermined safety policy. Unfortunately, to express the constructs of a real source language as compiled by a real compiler to a real target machine, a TAL must be quite feature-laden and complex. Therefore, its soundness proof is huge, and too tedious for anyone to check by hand. Furthermore, a real TAL is an industrial software artifact just like a compiler, and is modified regularly just like any software, which means that the soundness proof never stands still. For both of these reasons, it is essential that the proof of soundness be machine checked, and that this machine-checked proof be modular and maintainable.

Authors' current addresses: A. Ahmed, Toyota Technological Institute at Chicago; A. W. Appel and C. D. Richards, Princeton University; K. N. Swadi, Persistent Systems (Pune, India); G. Tan, Lehigh University; D. C. Wang, Microsoft Corporation.

The central goal of the Foundational Proof-Carrying Code (FPCC) project [Appel 2001] at Princeton is to build a machine-checked and modular soundness proof for a realistic TAL. The guiding principle of the FPCC project has been as follows: *Everything, including the soundness of TALs, should be verified with respect to the smallest-possible trusted computing base (TCB).* This poses the intellectual challenge of understanding the minimal set of assumptions one must make in constructing a protection mechanism, and yields a system that is fundamentally more secure.

To achieve a minimal TCB, we follow the semantic approach [Appel and Felty 2000] to FPCC. We start with an architecture specification (i.e., machine semantics), defined using some simple, yet expressive, logic. This logic should consist of a small set of axioms and definitional principles from which it should be possible to build up most of modern mathematics. For the FPCC project, we use higher-order logic extended with a few axioms of arithmetic. The architecture specification, as well as a predetermined safety policy, can easily be defined using axioms of higher-order logic and arithmetic. Based on the foundation of logic and machine semantics, we define the denotational semantics of the syntactic types and typing judgments of TALs. Intuitively, the semantics of each type characterizes the safe operational use of values of that type. We can then prove TAL typing rules as individual lemmas, instead of having to trust them as axioms. Finally, we prove the soundness theorem, which says that if a program type checks, then it is safe according to the safety policy. Using this approach, since the typing rules are proved as lemmas, a TAL typing derivation is converted into a safety proof that relies on the smallest possible set of axioms—i.e., axioms of logic plus machine semantics (Figure 1).

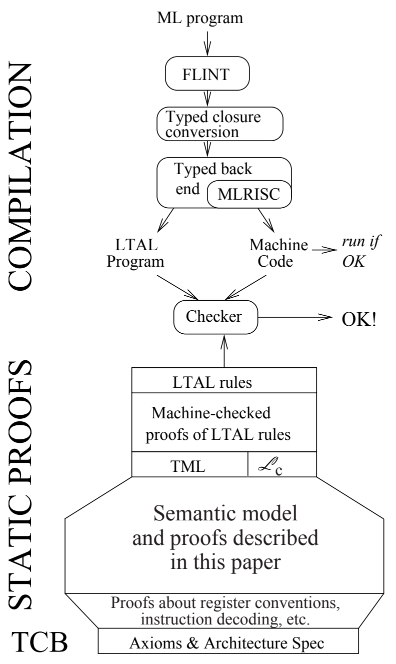

ML program
FLINT
Typed closure conversion
Typed back end
MLRISC
LTAL Program
Machine Code
run if OK
Checker
OK!
COMPILATION
STATIC PROOFS
LTAL rules
Machine-checked proofs of LTAL rules
TML
$\mathscr{L}_c$
Semantic model and proofs described in this paper
Proofs about register conventions, instruction decoding, etc.
TCB
Axioms & Architecture Spec

Fig. 1. Foundational Proof-Carrying Code. The Compiler translates an ML program into an LTAL [Chen et al. 2003] program and the corresponding machine-language program. MLRISC is the SML/NJ compiler's untyped back end. Chen adapted it to type-preserving compilation without modifying its internals [Chen et al. 2003]. The Checker first reads the trusted base (Axioms and Architecture Spec.), then reads the LTAL rules, then reads and checks a proof of soundness of the LTAL w.r.t. the trusted base. Then it applies the (syntax-directed) LTAL rules to the LTAL program and the machine-language program, which verifies both that the LTAL is type-safe and that it correctly assembles to the given machine-language program. If all of this succeeds, then we know that the program is safe. The only trusted components are the Axioms, Architecture Specification, and the Checker; the LTAL need not be trusted because it is proved sound in a way that can be checked by the Checker.

In this paper, we explain how our FPCC system modularly organizes the soundness proofs of Typed Assembly Languages. Modular soundness proofs present a challenge since the design goals of a typical TAL—including syntax-directed type checking, accommodating low-level optimizations, and dealing with particulars in the target machine—complicate the soundness proof. Instead of directly proving the soundness of a TAL with respect to machine semantics, our solution is to design and implement an intermediate layer between the machine semantics and the TAL. This layer has orthogonal and primitive features that TALs can use to justify their soundness. Given this design, we first prove the soundness of our intermediate layer with respect to the machine semantics, and then prove the soundness of TALs based on the interface provided by the intermediate layer (see Figure 1). All of our proofs are machine checked in higher-order logic using the Twelf system [Pfenning and Schürmann 1999]. The TAL of FPCC is the Low-level Typed Assembly Language (LTAL) of Chen et al. [2003], but LTAL has many engineering details that we will not describe in this paper; instead, we will show how to prove soundness of a much simpler example TAL.

Our intermediate layer (for FPCC and in this paper) consists of two parts: a Typed Machine Language (TML) plus a control logic $\mathcal{L}_c$. When verifying properties of assembly-language programs, we need the ability to specify properties of both machine states (data) and properties of assembly instructions (code). TML is designed as an expressive type theory that can specify rich properties of machine states, while $\mathcal{L}_c$ supports specification of properties of assembly instructions in the style of Hoare Logic. This separation is by design, since the logic $\mathcal{L}_c$ is parameterized by a specification language for machine states. In FPCC and this paper, we use TML as the specification language for machine states.

TML is an expressive type theory with orthogonal type primitives, including intersection types, union types, recursive types, mutable references, polymorphism, existential types, address-arithmetic types, and so on. Type checking in TML is not syntax directed; however, the type constructors of a TAL (with syntax-directed type checking) can be expressed as combinations of TML operators. These operators can express a wide variety of TALs for a wide variety of source languages, but TML is not infinitely general: for example, languages with "weak updates" (i.e., type-invariant mutable references) can be modeled, but not those with "strong updates" (i.e., mutable references whose type may be changed dynamically).

Because TML has such a rich set of operators, constructing a semantic model for TML based on machine semantics is a nontrivial problem. We require a model that can handle the various circularities that arise in the presence of features such as recursive functions, recursive types, impredicative quantified types, and mutable references. In particular, we wish to model general references—that is, updatable references that can store values of *any* type, including functions, as well as references and values with recursive types or impredicative existential types (i.e., we need a model that can handle cycles in the store). A naive semantic model cannot be well-founded in the presence of such features. We have addressed this challenge using the technique of *step-indexed logical relations*. While most logical relations in the literature are defined simply by induction on the type structure, step-indexed logical relations are indexed not just by types, but also by the number of steps available for future evaluation. This stratification is required for ensuring that the definition of the logical relation is well-founded, even in the presence of (semantically) challenging features such as recursive types, mutable references, and impredicative polymorphism.

Our control logic $\mathcal{L}_c$ is a generalized Hoare logic of *multiple-entry and multiple-exit program fragments* that permits reasoning about control flow in a very modular way. The control-flow constructs of TALs—including individual instructions, basic blocks, conditional branches, indirect jumps, functions, labels, continuations, and loops—can all be expressed using a few simple and general primitives of $\mathcal{L}_c$.

Having the intermediate layer provides a separation of concerns. For instance, the primary design goal for the layer is to be expressive, not to support decidable type checking or type inference. Therefore, it is not well suited as the language for a compiler to communicate proof witnesses to a checker. That role is fulfilled in the FPCC system by LTAL, whose types and rules are specialized for the FPCC-ML compiler and the SPARC: they are less orthogonal and less general, but they permit syntax-directed type checking. Furthermore, since TML and $\mathcal{L}_c$ are largely language-independent and machine-independent, they can provide semantic foundations for different TALs—in particular, they provide expressive and orthogonal primitives with which more complicated features of TALs can be encoded.

The major contributions of this paper are as follows.

—We have designed and implemented a general and powerful intermediate layer, consisting of Typed Machine Language (TML) and a control logic $\mathcal{L}_c$, that serves as a semantic foundation for Typed Assembly Languages, and permits modular and maintainable soundness proofs for TALs.

—To define semantics of TML types, we have successfully applied a new approach, that of *step-indexed logical relations*. As far as we know, this is the first semantic model of a type system with mutable references, (contravariant) recursive types, impredicative polymorphism, and code pointers. These advanced types are essential for compiling ML to machine code—for instance, the representation of ML function closures demands mutable references to impredicatively quantified types. We specify the semantics of TML by induction over both the structure of types, as well as the number of future evaluation steps.

—The semantic approach described in this paper scales to real systems: it has been demonstrated in a complete system that includes a type-preserving optimizing compiler from core ML to a typed assembly language for the SPARC called LTAL [Chen et al. 2003], a type-checker for LTAL expressed as syntax-directed Horn clauses [Wu et al. 2003; Wu 2005], a machine-checked proof of the soundness of the Horn clauses, and a tiny trustworthy LF proof-checker that can both check the soundness proof [Appel et al. 2003] and interpret the Horn clauses [Wu et al. 2003].

In the rest of the paper, we first specify the untyped operational semantics for von Neumann machines and our safety policy (Section 2). We then present the syntax of TML and demonstrate its ability to encode complicated types (Section 3). We show how to build a semantic model for TML type constructors—including mutable and immutable references, recursive types, and impredicatively quantified types—using step-indexed logical relations (Section 4). We then introduce the second component of our intermediate layer, a compositional logic for control flow, or $\mathcal{L}_c$ (Section 5).

To illustrate the power of our intermediate layer, we present a simple TAL and demonstrate how the TAL's type soundness is established on top of TML and $\mathcal{L}_c$ (Section 6). We wrap up with a historical overview of the FPCC project (Section 7) and a discussion of related work (Section 8).

2. MACHINE MODEL AND SAFETY POLICY

We specify a small-step operational semantics that characterizes both the "raw" machine semantics and the safety policy. This specification makes no reference to any particular type system—only to the untyped syntax and semantics of the target machine. For the FPCC project's target machine we formally specified the SPARC instruction-set architecture [Michael and Appel 2000; Appel 2001]. For simplicity in this paper, however, we use an imaginary machine with a simple instruction set. The imaginary machine we axiomatize for this paper machine operates on infinite-length integers, though the FPCC project's SPARC specification performs modular arithmetic on 32-bit integers.

A machine state S is given by a pair $\langle R,\ M\rangle$ of a register bank R and a memory M, both of which are modeled as finite maps from addresses (numbers) to contents (also numbers). Every register in the machine is assigned an index in the register bank. On the SPARC, for example, we have 0–31 as general-purpose registers. Special registers, including the program counter and the condition-code register, are assigned indexes that are greater than 31. For readability, we use $\mathrm{r}_0, \mathrm{r}_1, \ldots, \mathrm{r}_{31}$ for the indexes of general-purpose registers. We use pc for the index of the program counter. We use the metavariable r to range over the concrete registers r_0, r_1, etc., of the machine, and the metavariable l to range over locations (addresses) in memory.

The machine instructions supported by our operational semantics are as follows. Each instruction is assumed to be of size one.

$$\begin{array}{lcl} (\textit{machine instructions})\ i & ::= & \texttt{add}\ r_d, r_s, n \mid \texttt{ld}\ r_d, r_s[n] \mid \texttt{st}\ r_d[n], r_s \\ & \mid & \texttt{goto}\ l \mid \texttt{bz}\ r_s, l \mid \texttt{bnz}\ r_s, l \mid \texttt{jmp}\ r_d \end{array}$$

The instruction "$\texttt{add}\ r_d, r_s, n$" adds n to the contents of register r_s, and puts the result in register r_d. The instruction "$\texttt{ld}\ r_d, r_s[n]$" loads the memory contents at address $r_s + n$ into register r_d. The instruction "$\texttt{st}\ r_d[n], r_s$" puts the contents of r_s into the memory slot at address $r_d + n$. The branch-always instruction "$\texttt{goto}\ l$" unconditionally jumps to the absolute address l. The branch-if-zero instruction "$\texttt{bz}\ r_s, l$" jumps to l if the value in r_s is zero, and falls through to the next instruction otherwise. The branch-if-not-zero instruction "$\texttt{bnz}\ r_s, l$" behaves analogously if the value in r_s is not zero. The indirect-jump instruction "$\texttt{jmp}\ r_d$" jumps to the address in register r_d. Note that the machine does not have a "halt" instruction. Instead, we axiomatize a "return address" that a program can jump to in order to exit safely (see discussion of return_addr(l) later in this section, page 7).

Modeling Machine Instructions. A machine instruction i is modeled as a relation between two machine states. For example, the semantics of the load instruction, "$\texttt{ld}\ r_d, r_s[n]$", is specified as follows, where $\langle R,\ M\rangle$ represents the state in which the load instruction is executed and $\langle R',\ M'\rangle$ represents the state after executing the load.

$$\begin{aligned}&(\texttt{ld}\ r_d, r_s[n])(\langle R,\ M\rangle, \langle R',\ M'\rangle)\ \triangleq\\ &\qquad \big(R'(\mathsf{pc}) = R(\mathsf{pc}) + 1\big)\ \wedge\ \big(R'(r_d) = M(R(r_s) + n)\big)\\ &\qquad \wedge\ \big(\forall x \notin \{\mathsf{pc}, r_d\}.\ R'(x) = R(x)\big)\ \wedge\ \big(\forall x.\ M'(x) = M(x)\big)\\ &\qquad \wedge\ \text{readable\_loc}(R(r_s) + n, \langle R,\ M\rangle)\end{aligned}$$

The semantics specifies that the load instruction increments the program counter and loads the value at the memory address $R(r_s)+n$ into the register r_d, while the contents of all other registers, as well as the contents of memory, remain unchanged. The semantics of other instructions are defined in a similar fashion.

One important property of our machine semantics is that it is deliberately partial: unsafe operations are omitted from the semantics. Our FPCC system uses *memory safety* as the safety policy. Unsafe operations, therefore, are those that read from unreadable addresses, and those that write to unwritable addresses.

In the FPCC system, we axiomatize the set of readable and writable locations using the following predicates:

$$\begin{aligned}\text{readable\_loc}(l, \langle R,\ M\rangle)\ &\triangleq\ \ldots\\ \text{writable\_loc}(l, \langle R,\ M\rangle)\ &\triangleq\ \ldots\end{aligned}$$

The readable_loc predicate specifies whether a memory address l is readable, while writable_loc specifies whether a memory address l is writable. Note that both predicates take the machine state $\langle R,\ M\rangle$ as an argument. This is necessary so that we may rely, for instance, on heap allocation information stored in the registers and memory when determining whether a given location is readable or writable. In particular, the convention of the SML/NJ compiler specifies a heap area and a register-spilling area using values of special registers. Then, we axiomatize that the heap area and register-spilling area are both readable and writable.

Let us return to the semantics of the load instruction above. The reader may have noticed that the definition of "$\texttt{ld}\ r_d, r_s[n]$" uses the readable_loc predicate to ensure that the address being read from is readable. To see the ramifications of this, suppose that in some state $\langle R,\ M\rangle$ the program counter points to a `ld` instruction that would, if executed, load from an address that is unreadable. Then, since the semantics of the `ld` instruction requires that the address must be readable, there does not exist a state $\langle R',\ M'\rangle$ such that $(\texttt{ld}\ r_d, r_s[n])(\langle R,\ M\rangle, \langle R',\ M'\rangle)$.

To model von Neumann's idea that integers can encode instructions, we define a decode relation $\mathsf{decode}(n, i)$ that decodes a machine integer n into a machine instruction i. The decode relation is straightforward to define using arithmetic in higher-order logic.

Step Relation and Safety. The machine operational semantics is modeled by a step relation, $\mapsto$, that steps from one state $\langle R,\ M\rangle$ to the next state $\langle R',\ M'\rangle$. The next state $\langle R',\ M'\rangle$ is the result of decoding the current machine instruction i (in state $\langle R,\ M\rangle$), and then executing i.

$$\langle R,\ M\rangle \mapsto \langle R',\ M'\rangle\ \triangleq\ \exists i\,.\ \mathsf{decode}(M(R(\mathsf{pc})), i)\ \wedge\ i\,(\langle R,\ M\rangle, \langle R',\ M'\rangle)$$

We write $S \mapsto^j S'$ to denote that there exists a chain of j steps of the form $S \mapsto S_1 \mapsto \ldots S_j$ where S_j is S'. The step relation is partial; some states have no successor states and we call them *stuck* states. One example of a stuck state is a state in which the program counter points to an integer that cannot be decoded into an instruction. Our modeling of machine instructions makes the step relation even more partial; for example, if the current instruction in a state S is a load instruction that would load from an unreadable address, then S is a stuck state.

Using this partial step relation, we can define safety. A state S is *safe for k steps* if it will not reach a stuck state within k steps. We say a state S is safe if it is safe for any number of steps.

$$\text{safe\_state}(S, k) \triangleq \forall S'.\ \forall j < k.\ \ S \mapsto^j S' \ \Rightarrow\ \exists S''.\ S' \mapsto S''$$

Safe Machine Code. Machine code C is a finite map from addresses to integers (the encodings of machine instructions). We define a predicate $\mathsf{loaded}(C, S)$ to specify that the code C is loaded into state S.

$$\mathsf{loaded}(C, \langle R,\ M\rangle) \ \triangleq\ \forall l \in \text{dom}(C).\ M(l) = C(l)$$

Next, we specify safety for machine code. Given a program start address *startLoc*, we say machine code C is safe if, for any machine state S, if C is loaded in memory in state S, the program counter in state S points to *startLoc*, and S satisfies some initial conditions (which we explain below), then S is a safe state.[1]

Definition 2.1 (Safe Machine Code).

$$\begin{aligned}&\text{safe\_code}(C, \mathit{startLoc}) \ \triangleq\\ &\quad \forall R, M.\ \mathsf{loaded}(C, \langle R,\ M\rangle)\ \wedge\ R(\mathsf{pc}) = \mathit{startLoc}\ \wedge\ \text{init\_cond}(\langle R,\ M\rangle)\\ &\qquad\qquad \Rightarrow\ \forall k.\ \text{safe\_state}(\langle R,\ M\rangle, k).\end{aligned}$$

The init_cond(S) predicate axiomatizes the initial state. One component of the initial conditions that S must satisfy specifies a return address such that machine code C can always jump to that address to reach a safe state.

$$\text{return\_addr}(l)\ \triangleq\ \forall R, M.\ R(\mathsf{pc}) = l\ \Rightarrow\ \forall k.\ \text{safe\_state}(\langle R,\ M\rangle, k)$$

In our implementation, we designate register 15 to store the return address, which means that return_addr($R(15)$) is part of the specification init_cond($\langle R, M\rangle$). Thus, according to the definition of safe_code(C, *startLoc*), a terminating program is safe as long as it jumps to the return address at the end. There are other conditions specified by init_cond(S) that constrain how registers and memory are organized in the initial state; we omit further details.[2]

A Foundation for Semantic Models. The small-step relation $\mapsto$ specifies how machine instructions execute and also, by omitting certain steps (or more specifically, by omitting the ability to load from and store to certain addresses), specifies a

[1]In our FPCC system, we handle position-independent code—that is, we prove a program safe no matter where it is loaded. However, to simplify the current presentation, we have assumed that the addresses where machine code is loaded are fixed, and that program execution begins at the address *startLoc*.

[2]In Section 4.6, we will introduce the notion of a *valid state* (specified by the valid_state predicate), which is an invariant on machine states. The init_cond predicate subsumes the conditions required by valid_state, thus ensuring that the initial state is also a valid state.

safety policy. The trusted computing base for our system is comprised solely of the the step relation, together with the axioms of higher-order logic — that is, nothing defined after this section is in the TCB. We are interested in proving the soundness property for the type-checker and assembler of a Typed Assembly Language, which says: *If a TAL program type-checks and assembles to a machine-language program C, then C will execute in conformance with the safety policy.* The remainder of this paper shows how to construct semantic models of TALs such that soundness can be proved. These semantic models will make use of the step relation $\mapsto$, the notion of machine states $\langle R, M\rangle$, the decode relation $\mathsf{decode}(n, i)$, the definition of safe machine code, as well as other predicates defined in this section.

One could imagine safety policies more sophisticated than memory safety. In particular it seems natural to achieve a type-safety policy: "This module will be safe when linked against such-and-such an interface specified in the type system of ML or Java." The semantic methods we describe in the rest of the paper should be capable of such policies. But we wanted to demonstrate that type systems are useful for guaranteeing even those safety properties that don't explicitly mention types at all. Therefore, we wanted to avoid mentioning types in the safety policy.

Furthermore, if the safety policy is type safety rather than memory safety, then the type system itself would have to be part of the trusted base—part of the specification of the theorem to be proved. In the FPCC project we wanted the simplest, smallest, and clearest specification of a nontrivial safety theorem to be proved about machine-language programs. That is, the type system is part of the proof, not part of the statement of the theorem to be proved. Just because the memory-safety policy is simple does not mean that proving adherence to it is trivial: it is, of course, undecidable in general and difficult in practice—unless one has the right technique. The right technique is provably sound type systems.

3. TYPED MACHINE LANGUAGE

We present a low-level type system called Typed Machine Language (TML) that is intended to serve as a semantic foundation for types in TALs. TML provides an expressive set of orthogonal type constructors that can be combined in a variety ways to specify properties of data and machine states.

3.1 TML Type Constructors

We show the primitive TML type constructors in Figure 2. They include the top type, the bottom (or empty) type, the integer type int, intersection and union types, and the singleton integer type $\mathsf{const}(n)$, which contains only the integer value n. These type constructors will be given semantic interpretations in Section 4; here we show how to combine them to synthesize other types.

Integer-comparison types include less-than, equal-to and greater-than types. For example, $\mathsf{int}_{<}(\mathsf{const}(n))$ contains all integers that are less than n. Other comparison

type t	::=	top	contains all values
	|	bottom	contains no values
	|	int	integers
	|	$t_1 \cap t_2 \mid t_1 \cup t_2$	intersection and union types
	|	$\mathsf{const}(n)$	singleton integer type
	|	$\mathsf{int}_<(t) \mid \mathsf{int}_=(t) \mid \mathsf{int}_>(t)$	integer-comparison types
	|	$\mathsf{plus}(t_1, t_2) \mid \mathsf{times}(t_1, t_2) \mid \mathsf{mod}(t_1, t_2)$	arithmetic types
	|	$\mathsf{readable} \mid \mathsf{writable}$	safety-property types
	|	$\mathsf{box}(t) \mid \mathsf{ref}(t)$	immutable and mutable reference types
	|	$\mathsf{codeptr}\,(t)$	code-pointer type
	|	$\mathsf{rec}\,(t)$	recursive type
	|	$\mathsf{forall}(t) \mid \mathsf{exists}(t)$	quantified types
	|	$\underline{n} \mid \mathsf{subst}(t_1, t_2)$	type variables and substitutions
	|	$\{t_1 : t_2\}$	single-slot vector type
	|	$\mathsf{kd\_scalar}(t) \mid \mathsf{kd\_numeric}(t)$	kind coercions

Fig. 2. TML type language.

types and a range type can be defined as follows.

$$\begin{aligned}
\mathsf{int}_{\leq}(t) &\triangleq \mathsf{int}_<(t) \cup \mathsf{int}_=(t)\\
\mathsf{int}_{\geq}(t) &\triangleq \mathsf{int}_>(t) \cup \mathsf{int}_=(t)\\
\mathsf{int}_{\neq}(t) &\triangleq \mathsf{int}_>(t) \cup \mathsf{int}_<(t)\\
\mathsf{range}(m, n) &\triangleq \mathsf{int}_{\geq}(\mathsf{const}(m)) \cap \mathsf{int}_<(\mathsf{const}(n))
\end{aligned}$$

Arithmetic types serve to capture arithmetic operations at the type level. They include plus, times, and modulo types. Other arithmetic types can be synthesized from these, for instance:

$$\mathsf{minus}(t_1, t_2) \triangleq \mathsf{plus}(t_1, \mathsf{times}(\mathsf{const}(-1), t_2)).$$

TML also has types that model safety properties of addresses. Specifically, the readable and writable types contain all addresses that are readable and writable (respectively) according to the safety policy. In addition, TML has types for immutable references ($\mathsf{box}\,(t)$) and type-invariant mutable references ($\mathsf{ref}\,(t)$)—that is, $\mathsf{ref}\,(t)$ is the type ascribed to references whose contents may be updated, but only with values of type t.

Note that we model mutability separately from the permission to read or write a memory location. Thus, we can model, for instance, a reference cell that cannot be written by the current program, but whose contents may change (due to the actions performed by another process). In programming languages such as ML, a mutable-reference type implicitly carries the capability to read from and write to the cell. An ML reference can therefore be modeled by the following combination of TML types:

$$\mathsf{ref}\,(t) \cap \mathsf{readable} \cap \mathsf{writable}.$$

An address belongs to type $\mathsf{codeptr}\,(t)$ if it is safe to jump to the address provided that the precondition t is met. TML also has recursive types $\mathsf{rec}\,(t)$, universally quantified types $\mathsf{forall}(t)$, and existentially quantified types $\mathsf{exists}(t)$.

Recursive types and quantified types introduce new type variables. We use de Bruijn indices to represent type variables. The notation $\underline{n}$ denotes the nth outermost bound type variable, with indices starting at $\underline{0}$. Therefore, in type $\mathsf{forall}(t)$, the index $\underline{0}$ in t refers to the quantified type variable. TML has no explicit type functions—just expressions with free de Bruijn variables that may be bound by quantifiers or other operators—and we do not need higher-order kinds. (A simple first-order kind system will be introduced later.) The type $\mathsf{subst}(t, t_1)$ in TML substitutes t_1 for de Bruijn index $\underline{0}$ in t.

Vector Types, Kinds, and Kind-Coercion Types. The types we have introduced so far are all scalar types, in the sense that they type a single value. For example, a single address belongs to type $\mathsf{codeptr}\,(t)$ if it is safe to jump to the address given the precondition t. In the process of designing TML, we found that it is sometimes necessary to type a *sequence* of values at once. For instance, on a von Neumann machine, the precondition t in a code-pointer type $\mathsf{codeptr}\,(t)$ usually describes the types of the entire register bank. For instance, the type t may say that register 1 is of type int, and register 2 is of type $\mathsf{ref}\,(\mathsf{int})$. In this case, we need to type a sequence (vector) of values in the register bank, not just a single value in a particular register.

TALs usually have different kinds of types for scalar values and vector values. For example, the Cornell TAL [Morrisett et al. 1999] has register-file types that describe register banks. This means that there are several different syntactic classes of variables. If we had N different classes of variables in TML, each with a different semantic meta-type, then we would need N different binding operators. Furthermore, since we are using de Bruijn indices, we would need N^2 inference rules.[3] To avoid such duplication, we made the design decision to fold different classes of values into a single class of vector values. Thus, all TML types characterize vector values. A scalar type happens to judge only slot zero of a vector value.

A single-slot vector type $\{t_1 : t_2\}$ constrains only one slot of a vector value. A vector value v belongs to type $\{\mathsf{const}(n) : t\}$ if and only if $v(n)$ (the contents of the n-th slot of vector v) belongs to type t. Using single-slot vector types together with intersection types, we can constrain multiple slots of a vector value, as with the type $\{\mathsf{const}(1) : \mathsf{int}\} \cap \{\mathsf{const}(2) : \mathsf{ref}\,(\mathsf{int})\}$. For readability, when there is no ambiguity, we write types of this form as $\{\,1 : \mathsf{int},\ 2 : \mathsf{ref}\,(\mathsf{int})\,\}$.

Since TML models scalar types and vector types using the same (higher-order logic) meta-type, we need to introduce a kinding system. For example, the type $\{t_1 : t_2\}$ does not make sense if t_1 is a single-slot vector type of the form $\{t_3 : t_4\}$. TML has a simple kinding system to rule out such pathological types. It has kinds *vector*, *scalar*, and *numeric*. A type of *vector* kind is one that judges a sequence of values, such as the contents of a register-bank or a formal-parameter list. A type of *scalar* kind is one that judges only slot zero of a vector value—for instance, the types int and $\mathsf{const}(n)$ are both scalar. A type has *numeric* kind if it behaves like $\mathsf{const}(n)$, for some n—so the type $\mathsf{const}(n)$ is numeric but the type int is not. Thus, the single-slot vector type $\{t_1 : t_2\}$ makes sense if and only if t_1 is a numeric type and t_2 is a scalar type.

[3]With N classes of variables, we would need N different shift operators and N^2 inference rules for commuting variables with shift operators.

We can calculate the kinds of various types:[4] for example, the type $\mathsf{box}\,(\mathsf{int})$ is scalar, while the type $\mathsf{plus}(\mathsf{const}(3), \mathsf{const}(4))$ is numeric. But what is the kind of the type variable $\underline{0}$? To handle type variables, we find it useful to apply an appropriate kind coercion on uses of type variables—that is, we have expressions containing $\mathsf{kd\_numeric}(\underline{0})$ or $\mathsf{kd\_scalar}(\underline{0})$ instead of a "bare" type variable $\underline{0}$. Specifically, we have kind-coercion types $\mathsf{kd\_numeric}$ (similarly, $\mathsf{kd\_scalar}$) such that $\mathsf{kd\_numeric}(t)$ is equivalent to t if t is indeed a numeric type; otherwise $\mathsf{kd\_numeric}(t)$ is equivalent to bottom. In some sense, the $\mathsf{kd\_numeric}(t)$ type is an intersection type that intersects t with a predicate that enforces the numeric property of types—we make this intuition precise in Section 4.9 when we present the semantics of kind-coercion types.

With kind-coercion types, we can *define* quantified types that quantify over numeric and scalar types. For example, a universal quantifier that quantifies over numeric types is defined as follows.

$$\mathsf{forall}_{\mathsf{num}}(t) \triangleq \mathsf{forall}(\mathsf{subst}(t, \mathsf{kd\_numeric}(\underline{0})))$$

That is, all occurrences of the variable $\underline{0}$ in t are substituted with its numeric restriction $\mathsf{kd\_numeric}(\underline{0})$. This approach is similar to that taken by Crary [2000] when defining inclusive subtyping.

No coercion is needed when quantifying over vector types since the set of vector types is semantically equal to the set of all types. That is, if we had a $\mathsf{kd\_vector}$ coercion it would be equivalent to the identity function.

3.2 Combining Type Constructors

We have discussed some simple examples of combining TML constructors, and we further demonstrate such uses of TML in this section.

Booleans, 32-bit Integers, and Pairs. Boolean types and 32-bit integer types can be encoded easily:

$$\begin{aligned} \mathsf{bool} &\triangleq \mathsf{const}(0) \cup \mathsf{const}(1) \\ \mathsf{int32} &\triangleq \mathsf{range}(0, 2^{32}) \end{aligned}$$

Next, we encode a pair type that has two immutable fields. To do this, we first define an offset type and a field type.

$$\begin{aligned} \mathsf{offset}(t_1, t_2) &\triangleq \mathsf{plus}(t_2, \mathsf{times}(\mathsf{const}(-1), t_1)) \\ \mathsf{field}(n, t) &\triangleq \mathsf{offset}(\mathsf{times}(\mathsf{const}(4), \mathsf{const}(n)), \mathsf{box}\,(t)) \\ \mathsf{pair}(t_0, t_1) &\triangleq \mathsf{field}(0, t_0) \cap \mathsf{field}(1, t_1) \end{aligned}$$

To understand offset types, let us consider the type $\mathsf{offset}(\mathsf{const}(n_1), t_2)$. Informally, the definition of offset types says that a value n has type $\mathsf{offset}(\mathsf{const}(n_1), t_2)$ if and only if $n + n_1$ has type t_2. Now let us relate this to what the definition actually says. Note that $\mathsf{offset}(\mathsf{const}(n_1), t_2)$ is essentially defined as $\mathsf{plus}(t_2, \mathsf{const}(-n_1))$—so, intuitively, this says that n has type $\mathsf{offset}(\mathsf{const}(n_1), t_2)$ if and only if n has type

[4]We can validate many kinding rules once we have defined the semantics of kinds. In Section 4.9, we will present our model of kinds together with some of these kinding lemmas (see Figure 4 on page 29).

"$t_2 - n_1$." The latter, if we "add" n_1 to both n and "$t_2 - n_1$," is the same as saying $n + n_1$ has type t_2.[5] Note that the plus operator can sensibly add a scalar type to a constant type, or a numeric type to a numeric type. Thus we can reason about address arithmetic as we do here in defining offset and field types. Finally, note that in the above definitions we assume that each field occupies four bytes[6]—hence, the use of const(4) in the definition of field(n, t).

ML Datatypes. The option type for integers, written in ML as

$$\texttt{datatype int\_option} \ = \ \texttt{None} \mid \texttt{Some of int},$$

can be encoded in TML as $\mathsf{const}(0) \cup \big(\mathsf{int}_{\neq}(\mathsf{const}(0)) \cap \mathsf{box}\,(\mathsf{int})\big)$, where we use integer 0 for the None case, and a boxed value (at a nonzero address) for the integer case.

More complicated datatypes such as the list type

$$\texttt{datatype List} \ = \ \texttt{Nil} \mid \texttt{Cons of Int} \ * \ \texttt{List}$$

can also be encoded. Compilers can choose to have either an untagged or tagged representation for this type. In an untagged representation, we can represent the list type as

$$\mathsf{rec}\Big(\underbrace{\mathsf{const}(0)}_{\texttt{Nil case}} \cup \underbrace{\big(\mathsf{int}_{\neq}(\mathsf{const}(0)) \cap \mathsf{field}(0, \mathsf{int}) \cap \mathsf{field}(1, \underline{0})\big)}_{\texttt{Cons case}}\Big)$$

For the Nil case, we use the value 0, hence the left side of the union type. For the Cons case, we have a pointer to the memory location containing a record of two fields, the first of which is the data, while the second is the pointer to the next cell. The rec operator binds a de Bruijn index—note that $\underline{0}$ is the type variable bound by the recursive type; it is not the integer 0.

In a tagged representation, each case in a datatype is uniquely associated with a tag, or a natural number. The representation has the tag as its first field, followed by the other fields in the datatype.

$$\mathsf{rec}\Big(\underbrace{\mathsf{field}(0, \mathsf{const}(0))}_{\texttt{Nil case}} \cup \underbrace{\big(\mathsf{field}(0, \mathsf{const}(1)) \cap \mathsf{field}(1, \mathsf{int}) \cap \mathsf{field}(2, \underline{0})\big)}_{\texttt{Cons case}}\Big)$$

Function Closures. A function closure is a package that has both a code pointer and an environment. We can encode function closures using existential types and the field constructor, where the existential abstracts over the type of the closure's environment. Consider, for example, a type cont(t) that models a closure of a continuation. The closure is a record of two fields: a code pointer and an environment. When called, the code pointer takes an argument of type t in register 1, and the environment in register 2. Here $\underline{0}$ is the type variable (for the type of the environment) bound by the existential type. (Recall that the type $\{\, \mathrm{r}_1 : t,\ \mathrm{r}_2 : \underline{0} \,\}$ below is shorthand for $\{\, \mathsf{const}(\mathrm{r}_1) : t,\ \mathsf{const}(\mathrm{r}_2) : \underline{0} \,\}$.)

$$\mathsf{cont}(t) \ \triangleq \ \mathsf{exists}\big(\mathsf{field}(0, \mathsf{codeptr}\,(\{\, \mathrm{r}_1 : t,\ \mathrm{r}_2 : \underline{0} \,\})) \cap \mathsf{field}(1, \underline{0})\big)$$

[5]The semantics of plus will be defined in Section 4.4; we encourage the reader to revisit the definition of offset types after reading that section.
[6]We have not attempted a general treatment of byte- and word-addressibility. Our prototype TML system treats word-addressed references, structure-fields, and arrays.

Dataflow Dependency. In Typed Machine Language, it is possible to specify dependencies between registers since TML types characterize vector values. For instance, the type $\mathsf{exists_{num}}(\{\, \mathrm{r}_1 : \mathsf{int}_{<}(\underline{0}), \mathrm{r}_2 : \mathsf{int}_{=}(\underline{0}) \,\})$ specifies that register 1 is always strictly less than register 2. (The type $\mathsf{exists_{num}}$ is the existential type that ranges over numeric types; its definition is analogous to that of $\mathsf{forall_{num}}$ given in Section 3.1.) The ability to encode dependencies between registers allows TML to encode dataflow analyses such as tag discrimination [Chen 2004].

4. A STEP-INDEXED MODEL OF TML TYPES

In this section, we describe the construction of a semantic model for TML types based on the foundation of higher-order logic and machine semantics. Our model is based on the *indexed model* of recursive types introduced by Appel and McAllester [2001]. We generalize that result to a language with general references—that is, mutable references that can store values of *any* type, including code pointers, references, recursive types, and impredicatively quantified types.

Our semantic model is essentially an instance of the method of *logical relations*. Specifically, we use *unary* logical relations, or *logical predicates*, which are predicates on terms, defined by induction on the structure of type expressions. The idea, roughly speaking, is to associate with each type a predicate (or invariant) that is preserved by the primitive operations of that type. Proof that the predicates are preserved by the relevant primitive operations is in turn sufficient for proving safety.

Logical relations may be based on denotational models [Plotkin 1973; Statman 1985; Pitts 1996] or on the operational semantics of a language [Tait 1967; Girard 1972; Pitts 1998; Birkedal and Harper 1997; Pitts 2000; Crary and Harper 2007]. Our semantics of TML is an instance of the latter—also known as *operational* or *syntactic* logical relations—since we associate with each type a predicate that defines the semantics (or meaning) of the type in terms of the operational semantics of the machine.

It is straightforward to construct logical relations for languages with simple type systems (e.g., with product, sum, and function types) since the relation can be defined by induction on the structure of types. However, for languages with richer type systems, such as those with recursive types or mutable references, the conditions required of the logical relation rule out definition by induction on types. In the case of recursive types, for instance, the predicate associated with the recursive type $\mu\alpha.\tau$ is naturally specified using the predicate for its one-step unfolding $\tau[\mu\alpha.\tau/\alpha]$, which is clearly not a strictly smaller type. In the absence of a well-founded definition of the logical relation, one has to prove that a relation that satisfies the "circular" specification actually exists. Unfortunately, establishing the existance of the logical relation is usually nontrivial [Pitts 1996; Birkedal and Harper 1997; Melliès and Vouillon 2004; Crary and Harper 2007].

The indexed model of recursive types introduced by Appel and McAllester [2001] takes a different approach, where logical relations are indexed not just by types, but also by the number of steps available for future evaluation. This permits well-founded definition of the logical relation: the predicate associated with $\mu\alpha.\tau$ at k available steps is specified using the predicate for $\tau[\mu\alpha.\tau/\alpha]$ at $k-1$ available steps, intuitively, since "unfolding" consumes a step. In essence, the extra information about available steps is sufficient for solving recursive equations on types.

Induction on types alone also fails to yield a well-founded definition of a logical relation for a language with general references—i.e., mutable references that may store other references, code pointers, recursive types, and even impredicative quantified types. In fact, a central challenge in building a semantic model for TML is how to construct a model of mutable references. We start below (Section 4.1) by explaining why induction on types does not suffice in the presence of references—we describe how one might naïvely construct a model for mutable references and show that such a model is not well founded. We then sketch how Appel and McAllester's step-indexing idea can be used to construct a well-founded model of mutable references that store values of any type. We shall see that the modeling of mutable references essentially dictates the overall shape of the semantic model for TML (yielding a model very different from that of Appel and McAllester [2001] who considered a purely functional language with recursive types). Unfortunately, this last model cannot directly be represented in higher-order logic as we shall explain. Therefore, in Section 4.2, we describe the structure of a slightly different model that *can* be encoded in higher-order logic; this is the model of TML that we have actually implemented as part of our FPCC prototype. In the rest of the section we give details of this model and the semantics of various TML types (Sections 4.3 to 4.11).

4.1 Towards a Model of Mutable References

We wish to construct a model of TML that supports updatable references. Proving type safety for a language that permits updates to *aliased* locations is not an easy task. Languages like ML and Java deal with the aliasing problem by allowing only *type-preserving updates* or *weak updates.* We adopt the same restriction for TML; that is, the TML type $\mathsf{ref}\,(t)$ introduced in Section 3 is the type ascribed to mutable references whose contents must always be of type t.

In this section, we motivate our semantics of TML by sketching out a naïve model and attempting to refine it. Let us consider how to model TML types. Intuitively, for each *closed* TML type t (i.e., one with no free type variables), we wish to define a predicate φ that specifies the set of values that belong to that type. We refer to φ as a (closed) semantic type, or as the semantic interpretation of t, also written $[\![t]\!]$. Modeling a semantic type as a predicate on values is sufficient when interpreting simple types (e.g., integers, arithmetic types), but it does not suffice when interpreting more advanced types such as references or code pointers. For instance, to ensure that a location l belongs to the semantic interpretation of $\mathsf{ref}\,(t)$, we must check that the contents of *memory* at location l belong to the interpretation of type t. Similarly, to ensure that l belongs to the interpretation of $\mathsf{codeptr}\,(t)$, we must check that the instructions in memory M at address l behave a certain way. This suggests that we should model semantic types φ as predicates on states $\langle R, M\rangle$ as well as values v. Unfortunately, for the semantics of type-invariant mutable references, even that will not suffice.

Modeling Permissible Updates. Type-preserving updates imply that only values of a particular type may be written at each allocated location. Thus, we need a model that, for each allocated location, keeps track of this type. For this, we introduce a *store type* Σ, a finite map from locations to closed semantic types. For each allocated location l, we keep track of the type of updates allowed at l. A semantic type φ would then be a predicate on three arguments: a store type Σ, a state $\langle R, M\rangle$, and a value v. Then a location l should belong to the interpretation of $\mathsf{ref}\,(t)$ if the store type Σ says that the permissible update type for location l is $[\![t]\!]$, and if the value in memory at location l satisfies $[\![t]\!]$.

$$[\![\mathsf{ref}\,(t)]\!](\Sigma, \langle R, M\rangle, v) \;\triangleq\; \Sigma(v) = [\![t]\!] \;\wedge\; [\![t]\!](\Sigma, \langle R, M\rangle, M(v))$$

An Inconsistent Model. Unfortunately, there is a problem with the above definition and proposed model. We want to model (closed) semantic types as predicates on store types Σ (a finite map from locations to closed semantic types), states S, and values v. The types of these logical objects are as follows.

$$\begin{array}{lcl} \textit{store-type} & = & \textit{location} \rightharpoonup \textit{closed-sem-type} \\ \textit{closed-sem-type} & = & \textit{store-type} \times \textit{state} \times \textit{value} \rightarrow \textit{bool} \end{array}$$

Notice that the metalogical type of *closed-sem-type* is recursive, and furthermore, it has an inconsistent cardinality—the set of closed semantic types must be bigger than itself.

Stratifying Semantic Types. Returning to our flawed semantic interpretation of $\mathsf{ref}\,(t)$ above, notice that t is a "smaller" type than $\mathsf{ref}\,(t)$, and that to define $[\![\mathsf{ref}\,(t)]\!]$, we only consider those locations in the store type whose permissible update types are $[\![t]\!]$, or more generally, are interpretations of types "smaller" than $\mathsf{ref}\,(t)$. This suggests a rationale for constructing a well-founded model: semantic types should be stratified so that a semantic type at level k relies only on a store type that maps locations to semantic types at level j for $j < k$. This leads to the following hierarchy of semantic types.

$$\begin{array}{lcl} \textit{closed-sem-type}_0 & = & \emptyset \\ \textit{store-type}_k & = & \textit{location} \rightharpoonup \textit{closed-sem-type}_k \\ \textit{closed-sem-type}_{k+1} & = & \textit{store-type}_k \times \textit{state} \times \textit{value} \rightarrow \textit{bool} \end{array}$$

By stratifying semantic types we have eliminated the circularity. We can now rewrite the interpretation of $\mathsf{ref}\,(t)$ so that semantic types and store types are annotated with levels to reflect the existence of a (semantic) type hierarchy. Intuitively, the interpretation of $\mathsf{ref}\,(t)$ at level k in the hierarchy (i.e., $[\![\mathsf{ref}\,(t)]\!] \in \textit{closed-sem-type}_k$) may be defined as a predicate on a store type $\Sigma \in \textit{store-type}_{k-1}$, a state $\langle R, M\rangle$, and a value v as shown below. We informally write φ^k and Σ^k in place of "φ such that $\varphi \in \textit{closed-sem-type}_k$" and "$\Sigma$ such that $\Sigma \in \textit{store-type}_k$" respectively.

$$[\![\mathsf{ref}\,(t)]\!]^k(\Sigma^{k-1}, \langle R, M\rangle, v) \;\triangleq\; \Sigma^{k-1}(v) = [\![t]\!]^{k-1} \;\wedge\; [\![t]\!]^{k-1}(\Sigma^{k-1}, \langle R, M\rangle, M(v))$$

We still need to formalize what the levels of the type hierarchy signify and when the semantic interpretation of a closed TML type t belongs to some level k—i.e., when $[\![t]\!] \in \textit{closed-sem-type}_k$.

Stratification via Syntax is Not Quite Good Enough. We observed above that t is a "smaller" type than $\mathsf{ref}\,(t)$—informally, t has fewer nested occurrences of ref than $\mathsf{ref}\,(t)$. In terms of the type hierarchy then, $[\![\mathsf{ref}\,(t)]\!]$ belongs to $\textit{closed-sem-type}_k$ if and only if $[\![t]\!]$ belongs to $\textit{closed-sem-type}_j$ for $j < k$. Informally, when $[\![t]\!]$ belongs to level k of the type hierarchy, we want that to mean that at level k we have sufficient "information" to conclude whether or not a value v has type t. The notion of having *sufficient* information at some level suggests that for each closed TML type t there exists a finite level k_{min} such that $[\![t]\!] \in \mathit{closed\text{-}sem\text{-}type}_{k_{min}}$ and $[\![t]\!] \notin \mathit{closed\text{-}sem\text{-}type}_k$ for $j \leq k_{min}$.

Consider for the moment a language with only primitive types (say int and $\mathsf{const}(n)$) and mutable references. We could use the number of nested occurrences of ref in t to determine whether the interpretation of the type t belongs to $\mathit{closed\text{-}sem\text{-}type}_k$ for some $k \geq 0$. Specifically, let the semantic interpretations of all primitive TML types (e.g., $[\![\mathsf{int}]\!]$ and $[\![\mathsf{const}(n)]\!]$) belong to level 0; if $[\![t]\!]$ belongs to level k, then $[\![\mathsf{ref}\,(t)]\!]$ belongs to level $k+1$; and if $[\![t]\!]$ belongs to level k, then $[\![t]\!]$ belongs to level $k+1$ (since if we have sufficient information at level k to determine if v has type t, then at level $k+1$ we still have sufficient information to conclude that v has type t). Thus, level 1 of the type hierarchy contains, for instance, $[\![\mathsf{ref}\,(\mathsf{int})]\!]$ and the interpretations of all level 0 types, but not $[\![\mathsf{ref}\,(\mathsf{ref}\,(\mathsf{int}))]\!]$. For any (finite) type expression there is some level in the type hierarchy powerful enough to contain its interpretation.

Unfortunately, stratification by syntax breaks down when we add quantified types to the language. Consider, for instance, the type $\exists\alpha.\mathsf{ref}\,(\alpha)$ (written in TML as $\mathsf{exists}(\mathsf{ref}\,(\underline{0}))$); we must determine if there exists some $k \geq 0$ such that $[\![\exists\alpha.\mathsf{ref}\,(\alpha)]\!]$ belongs to level k in the type hierarchy. If we know that the interpretation of α belongs to level k, then we can conclude that the interpretation of $\mathsf{ref}\,(\alpha)$ belongs to level $k+1$ and that of $\exists\alpha.\mathsf{ref}\,(\alpha)$ belongs to level $k+1$ (and that of $\mathsf{ref}\,(\exists\alpha.\mathsf{ref}\,(\alpha))$ belongs to level $k+2$ and so on). But α is a type variable, so we cannot know how complex the type that witnesses α is—or, in the case of universal types, how complex the type that instantiates α will be. Furthermore, we wish to model impredicative quantified types, which means that α, which we assumed is a level k type, may be witnessed by $\exists\alpha.\mathsf{ref}\,(\alpha)$ itself, which we just concluded is a level $k+1$ type—but this implies that α should be a level $k+1$ type, but that in turn would mean that $\exists\alpha.\mathsf{ref}\,(\alpha)$ must be a level $k+2$ type. Hence, there is no finite level of the type hierarchy that is guaranteed to be powerful enough to contain the interpretation of $\exists\alpha.\mathsf{ref}\,(\alpha)$.

In earlier work [Ahmed et al. 2002], we relied on the syntactic complexity of a type t in order to determine the level of the type hierarchy that is guaranteed to contain $[\![t]\!]$. As we have just seen, this approach cannot accomodate references to impredicative quantified types.

Stratification via Semantic Approximation. Instead of requiring that levels in the type hierarchy correspond to the syntactic complexity of type expressions, we will treat levels as an indication of how many more steps the program can safely execute. Informally, we want $(\Sigma^{k-1}, S, v) \in [\![t]\!]^k$ to mean that v "looks" like it has type t for k steps—perhaps v does not actually have type t, but any program that takes an argument of type t must execute for at least k steps on v before getting to a stuck state. Hence, levels in the type hierarchy correspond to approximations of a type's behavior.

We call k the *approximation index* or *step index* following Appel and McAllester's indexed model of types [Appel and McAllester 2001]. The latter gave a semantics of recursive types for a purely functional language. In this model, a value v belongs to $[\![t]\!]$ to approximation k if, in any computation running for no more than k steps, the value v behaves as if it has type t. We use this intuition to stratify types in our model of mutable references. In particular, consider the interpretation of $\mathsf{ref}\,(t)$ to approximation k. The assumption that a location l (with memory contents v) has type $\mathsf{ref}\,(t)$ cannot be proved wrong within k steps of execution if and only if (1) the assumption that v has type t cannot be be proved wrong within $k-1$ steps (since accessing v from l would require an extra dereferencing step) and (2) the store type Σ tells us that location l's permissible update type for $k-1$ steps is t (which guarantees that any value v' assigned to l must be such that the assumption that v' has type t cannot be proved wrong in $k-1$ steps, where $k-1$ suffices since the update consumes a step).

The use of semantic approximation to stratify type interpretations helps us model mutable references to quantified types by allowing us to "pick a level" (i.e., an appropriate approximation index) for a type variable. Suppose that in some execution the witness type for $\exists\alpha.\mathsf{ref}\,(\alpha)$ is $\mathsf{ref}^{40}(\mathsf{int})$ (where $\mathsf{ref}^{n}(t)$ denotes n applications of ref to t), but we intend to run the program for only 10 steps. Then it's all the same whether the witness type for α is $\mathsf{ref}^{40}(\mathsf{int})$ or $\mathsf{ref}^{10}(\bot)$ since in 10 execution steps the program cannot dereference more than 10 references, and thus, cannot *tell the difference* between values of these two types. The same idea applies in the presence of impredicative quantification, such as when the witness type for $\exists\alpha.\mathsf{ref}\,(\alpha)$ is $\exists\alpha.\mathsf{ref}\,(\alpha)$ itself. To define the interpretation of $\exists\alpha.\mathsf{ref}\,(\alpha)$ to approximation k—intuitively, the set of values that have type $\exists\alpha.\mathsf{ref}\,(\alpha)$ for k steps—we only need to know the interpretation of the witness type (i.e., $\exists\alpha.\mathsf{ref}\,(\alpha)$) to approximation $k-1$. Here $k-1$ suffices since one execution step is consumed by dereferencing (and perhaps one more step is consumed when unpacking the existential, though we shall discuss this point further in Section 4.8).

Representing Stratified Types in Higher-Order Logic. Using semantic approximation to stratify types, we model types as sets of tuples of the form (k, Σ, S, v), where k is the approximation index or step index, and Σ is a store type mapping locations to $k-1$ approximations of type interpretations. Notice that the metalogical type of Σ here depends on the step index k, suggesting that we need a dependent type theory to capture this dependency (of the type of Σ on the term k). In particular, the metalogical types of our closed semantic types and store types are as follows.

$$\begin{array}{lcl}
\textit{closed-sem-type}_0 & = & \emptyset \\
\textit{store-type}_k & = & \textit{location} \rightharpoonup \textit{closed-sem-type}_k \\
\textit{closed-sem-type}_{k+1} & = & \textit{natural} \times \textit{store-type}_k \times \textit{state} \times \textit{value} \rightarrow \textit{bool} \\
\textit{closed-sem-type} & = & \bigcup_{i \geq 0} \textit{closed-sem-type}_i
\end{array}$$

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

Since our FPCC implementation uses higher-order logic, we have to tweak the above model somewhat. In the rest of Section 4, we describe a model of TML that is representable in higher-order logic, but which tries to capture the spirit of the above model (and the idea of a hierarchy of semantic types) as closely as possible. In particular, we make store types Σ a finite map from locations to type expressions.

We still model closed semantic types φ as sets of tuples of the form (k, Σ, S, v), but now, since Σ maps locations to type syntax, the type of Σ no longer depends on k; hence, the model can be encoded in higher-order logic. To capture the essence of the stratified semantic model above, we ensure that the interpretation of any type t to approximation k—to be precise, the predicate $[\![t]\!](k, \Sigma, S, v)$—does not rely on the interpretation of types in the codomain of Σ (i.e., on any $[\![\Sigma(l)]\!]$) beyond approximation $k - 1$. The resulting model has only one weakness compared to the one with a hierarchy of semantic types; this has to do with quantified types and we discuss it further in Section 4.8.

Ahmed [2004, Chapter 3] gives a set-theoretic model (with a hierarchy of semantic types) for a λ-calculus with mutable references and impredicative quantified types, and briefly describes [Ahmed 2004, Chapter 5] a representation in the Calculus of Inductive Constructions (CiC). We do not describe that model here.

4.2 A Model of TML Representable in Higher-Order Logic

Intuitively, the predicate for a TML type t should define a set of values that belong to the type. For each type t, we define a predicate $[\![t]\!](\rho)(k, \Sigma, S, v)$. Informally, this says that "root value v in machine state S has type t to approximation k, given environment ρ to provide context for the free type variables of t, and given store type Σ to constrain the types of mutable references in S." The metalogical types of all the relevant components of the model are given below.

approx-index	k	:	*natural*
store-type	Σ	:	*location* $\rightharpoonup$ *type*
state	S	:	*register-bank* $\times$ *memory*
root-value	v	:	*number* $\to$ *number*
closed-sem-type	φ	:	*approx-index* $\times$ *store-type* $\times$ *state* $\times$ *root-value* $\to$ *bool*
environment	ρ	:	*natural* $\to$ *closed-sem-type*
sem-type	τ	:	*environment* $\to$ *closed-sem-type*
meaning function	$[\![\ ]\!]$	:	*type* $\to$ *environment* $\to$ *closed-sem-type*

Step Index k. The step index (or approximation index) k is a natural number that, informally, denotes the number of (future) computation steps available to the program. If a value belongs to a type t to approximation k, then the value is "good enough" for any program that runs for at most k more steps. As a concrete example, consider a location l in state S such that l contains an integer, and a program that is well typed assuming l has type ref (ref (int)). Strictly speaking, l has type ref (int). But if the program runs for only one more step then treating l's type as ref (ref (int)) is good enough since the program can perform at most one dereferencing step—thus, l belongs to type ref (ref (int)) to approximation 1. Similarly, l also belongs to the type ref (ref (ref (int))) to approximation 1.

Store Type Σ. The store type is a mapping from memory addresses to closed type expressions; we write *type* to denote the set of type expressions. The store type Σ prescribes the type of the contents of memory addresses. When a reference is allocated, we extend the current store type Σ so that it maps the newly allocated location l (where $l \notin \mathrm{dom}(\Sigma)$) to the intended type of its contents. We require that any future Σ' be a superset of Σ so that the permissible update types of locations cannot be changed. Finally, to ensure that updates of mutable references are type preserving, we check that the type of the new value being assigned to a location—at some point in the future when the store type is Σ'—matches the type prescribed by Σ'.

State S. As before, a state S is a pair of a register bank R and memory M. Recall from Section 4.1 that in order to determine if a value v belongs to the interpretation of $\mathsf{ref}\,(t)$ or $\mathsf{codeptr}\,(t)$ we need access to the current machine state. Thus, in the presence of references and code pointers, the semantic interpretation of a type t—given by the predicate $[\![t]\!]$—relies on the current machine state S.

Root Value v. Since TML types can type a whole register bank, we define a root value v to be a vector of integers. This vector of integers can be the vector of values in a register bank, or a vector of parameters passed to a function, or a list of addresses. Modeling root values as vectors allows us to support scalar types as well as vector types. A scalar type such as $\mathsf{int}_<(t)$ judges only one scalar value. As we will see, the model of a scalar type constrains only the scalar value in slot zero of the vector.

Environment ρ. TML types include type variables ($\underline{n}$) and type-variable binders ($\mathsf{rec}, \mathsf{forall}, \mathsf{exists}$). Thus, we need to parametrize the interpretation of TML types with an environment ρ that gives the meaning of type variables in the types. TML uses de Bruijn indices for type variables. Therefore, the environment ρ is a mapping from de Bruijn indices (natural numbers) to the meanings (interpretations) of closed types—i.e., to closed semantic types.

Semantic Type τ. As stated above, $[\![t]\!]$ is a predicate on $(\rho)(k, \Sigma, S, v)$, where ρ must map all de Bruijn indices (i.e., free type variables) that appear in t. We call any predicate on $(\rho)(k, \Sigma, S, v)$ a semantic type and use the metavariable τ to denote semantic types. Later, we will define the properties that a semantic type should satisfy, or what a *valid* semantic type is (see Section 4.10).

Closed Semantic Type φ. A semantic type applied to an environment ρ is a predicate on (k, Σ, S, v). We call a predicate on (k, Σ, S, v) a closed semantic type since it gives semantics to closed types. We use the metavariable φ for closed semantic types.

4.3 Preliminaries

We define semantic subtyping and equality (to some finite approximation k) for closed semantic types φ and lift that notion to environments ρ as well as semantic types τ.

Definition 4.1 (Semantic Subtyping and Equality).

$$\begin{aligned}
\varphi \subseteq_k \varphi' &\triangleq \forall j \leq k.\ \forall \Sigma, S, v.\ \varphi(j, \Sigma, S, v) \ \Rightarrow\ \varphi'(j, \Sigma, S, v) \\
\varphi \approx_k \varphi' &\triangleq (\varphi \subseteq_k \varphi') \ \wedge\ (\varphi' \subseteq_k \varphi) \\
\rho \approx_k \rho' &\triangleq \forall i.\ \rho(i) \approx_k \rho'(i), \\
\tau \subseteq_k \tau' &\triangleq \forall \rho.\ \tau(\rho) \subseteq_k \tau'(\rho) \\
\tau \approx_k \tau' &\triangleq \forall \rho.\ \tau(\rho) \approx_k \tau'(\rho)
\end{aligned}$$

Most properties discussed hereafter are defined up to some approximation—that is, they are parametrized by a step index k up to which the property holds but after which there are no guarantees. Unlike standard domain theory, which takes a limit of finitary approximations, we avoid taking limits. Everything remains at an approximation.

Definition 4.2 (Approx). The k-approximation of a closed semantic type φ is the set of tuples (j, Σ, S, v) in φ such that the index j is no greater than k.

$$(\lfloor \varphi \rfloor_k)(j, \Sigma, S, v) = \begin{cases} \varphi(j, \Sigma, S, v) & \text{if } j \leq k \\ \text{false} & \text{if } j > k \end{cases}$$

The important point about the k-approximation of a closed semantic type φ is that in order to determine the truth of $\lfloor \varphi \rfloor_k(j, \Sigma, S, v)$, we only need to know if $\varphi(j, \Sigma, S, v)$ holds for $j \leq k$; we need no information about how φ is defined for any $j > k$.

We write $\rho_\emptyset$ to denote an empty type environment, which maps every type variable to the (semantics of the) bottom (or empty) type.

Definition 4.3 (Environment Cons). For an environment ρ, we define a list cons operator to concatenate a closed semantic type φ to the beginning of ρ.

$$(\varphi \bullet \rho)(n) = \begin{cases} \varphi & \text{if } n = 0 \\ \rho(n-1) & \text{if } n > 0 \end{cases}$$

Finally, we say a *constant vector*, written $\mathsf{cv}(n)$, is a vector in which every slot is mapped to the number n.

Well-Founded Semantics. Next, we present the semantics of TML type constructors t. Our semantics is well founded since it is defined by induction on the step index and (nested induction on) the syntax of types. Put another way, whenever the semantic interpretation of a type t relies on semantic interpretations of other types, either the step index decreases, or the step index remains the same and the syntactic complexity of the type decreases.

4.4 Simple Type Constructors

The semantic interpretations of several TML type constructors can be defined simply by induction on the syntax of the type expression. The interpretations of these types are given in Figure 3.

The semantic interpretation of top admits every value (regardless of step index k, store type Σ, or state S) since every value v has type top, while that of bottom

$$
\begin{array}{rcl}
[\![\mathsf{top}]\!](\rho)(k,\Sigma,S,v) & \triangleq & \text{true} \\
[\![\mathsf{bottom}]\!](\rho)(k,\Sigma,S,v) & \triangleq & \text{false} \\
[\![\mathsf{int}]\!](\rho)(k,\Sigma,S,v) & \triangleq & \text{true} \\
[\![t_1 \cap t_2]\!](\rho)(k,\Sigma,S,v) & \triangleq & [\![t_1]\!](\rho)(k,\Sigma,S,v) \ \wedge\ [\![t_2]\!](\rho)(k,\Sigma,S,v) \\
[\![t_1 \cup t_2]\!](\rho)(k,\Sigma,S,v) & \triangleq & [\![t_1]\!](\rho)(k,\Sigma,S,v) \ \vee\ [\![t_2]\!](\rho)(k,\Sigma,S,v) \\
[\![\mathsf{const}(n)]\!](\rho)(k,\Sigma,S,v) & \triangleq & v(0) = n \\
[\![\mathsf{int}_<(t)]\!](\rho)(k,\Sigma,S,v) & \triangleq & \exists n.\ [\![t]\!](\rho)(k,\Sigma,S,\mathsf{cv}(n)) \ \wedge\ v(0) < n \\
[\![\mathsf{int}_=(t)]\!](\rho)(k,\Sigma,S,v) & \triangleq & \exists n.\ [\![t]\!](\rho)(k,\Sigma,S,\mathsf{cv}(n)) \ \wedge\ v(0) = n \\
[\![\mathsf{int}_>(t)]\!](\rho)(k,\Sigma,S,v) & \triangleq & \exists n.\ [\![t]\!](\rho)(k,\Sigma,S,\mathsf{cv}(n)) \ \wedge\ v(0) > n \\
[\![\mathsf{plus}(t_1,t_2)]\!](\rho)(k,\Sigma,S,v) & \triangleq & \exists n_1,n_2.\ [\![t_1]\!](\rho)(k,\Sigma,S,\mathsf{cv}(n_1)) \ \wedge\ [\![t_2]\!](\rho)(k,\Sigma,S,\mathsf{cv}(n_2)) \\
 & & \qquad \wedge\ n_1 + n_2 = v(0) \\
[\![\mathsf{times}(t_1,t_2)]\!](\rho)(k,\Sigma,S,v) & \triangleq & \exists n_1,n_2.\ [\![t_1]\!](\rho)(k,\Sigma,S,\mathsf{cv}(n_1)) \ \wedge\ [\![t_2]\!](\rho)(k,\Sigma,S,\mathsf{cv}(n_2)) \\
 & & \qquad \wedge\ n_1 * n_2 = v(0) \\
[\![\mathsf{mod}(t_1,t_2)]\!](\rho)(k,\Sigma,S,v) & \triangleq & \exists n_1,n_2.\ [\![t_1]\!](\rho)(k,\Sigma,S,\mathsf{cv}(n_1)) \ \wedge\ [\![t_2]\!](\rho)(k,\Sigma,S,\mathsf{cv}(n_2)) \\
 & & \qquad \wedge\ n_1 \% n_2 = v(0) \\
[\![\mathsf{readable}]\!](\rho)(k,\Sigma,S,v) & \triangleq & \text{readable\_loc}(v(0),S) \\
[\![\mathsf{writable}]\!](\rho)(k,\Sigma,S,v) & \triangleq & \text{writable\_loc}(v(0),S) \\
[\![\underline{n}]\!](\rho)(k,\Sigma,S,v) & \triangleq & \rho(n)(k,\Sigma,S,v) \\
[\![\mathsf{subst}(t_1,t_2)]\!](\rho)(k,\Sigma,S,v) & \triangleq & [\![t_1]\!]\Big((\lfloor [\![t_2]\!]\rho \rfloor_k) \bullet \rho\Big)(k,\Sigma,S,v) \\
[\![\{t_1 : t_2\}]\!](\rho)(k,\Sigma,S,v) & \triangleq & \exists n.\ [\![t_1]\!](\rho)(k,\Sigma,S,\mathsf{cv}(n)) \ \wedge\ [\![t_2]\!](\rho)(k,\Sigma,S,\mathsf{cv}(v(n)))
\end{array}
$$

Fig. 3. Semantic Interpretation of Simple TML Type Constructors

(the empty type) admits no values. The integer type int also admits every value because the values in any root v are already integers.[7]

The intersection type $t_1 \cap t_2$ (union type $t_1 \cup t_2$) admits values v that are in the interpretation of t_1 and (or) t_2. The singleton integer type $\mathsf{const}(n)$ admits root values v such that $v(0) = n$. Note that $\mathsf{const}(n)$ is a scalar type and judges only slot zero of the vector value v.

For the interpretation of $\mathsf{int}_<(t)$, we first assert that there exists some number n such that the constant root value $\mathsf{cv}(n)$ belongs to type t, and then check that $v(0) < n$. Consider, for instance, the interpretation of the type $\mathsf{int}_<(\mathsf{const}(10))$, which admits all integers less than 10.

For the interpretation of $\mathsf{plus}(t_1, t_2)$, we first require that there exist two numbers n_1 and n_2 such that $\mathsf{cv}(n_1)$ and $\mathsf{cv}(n_2)$ belong to types t_1 and t_2, respectively. Then, we check that $n_1 + n_2 = v(0)$. For the interpretation of type readable (or writable), we simply need to check that $v(0)$ is a readable (or writable) location.

For a de Bruijn index $\underline{n}$, we simply look up its interpretation in the environment ρ. For a substitution type $\mathsf{subst}(t_1, t_2)$, which substitutes t_2 for de Bruijn index $\underline{0}$ in t_1, we must interpret the type t_1 in a new environment that maps $\underline{0}$ to the interpretation of t_2 and maps de Bruijn indices $\underline{n}$ where $n > 0$ to $\rho(n-1)$. Specifically, to determine if root value v belongs to $\mathsf{subst}(t_1, t_2)$ for k steps given environment ρ, we check that v belongs to t_1 for k steps with the modified environment that maps $\underline{0}$ in t_1 to $\lfloor [\![t_2]\!]\rho \rfloor_k$, the k-approximation of $[\![t_2]\!]\rho$. By mapping $\underline{0}$ to $\lfloor [\![t_2]\!]\rho \rfloor_k$ instead of

[7]Readers may wonder why we need int at all if its model is the same as top. Though, in this paper, we have assumed for simplicity that all numbers are integers, in our FPCC prototype we actually do not make that assumption. Instead, we have a predicate isInt to constrain numbers. As a result, in our FPCC prototype, the interpretation of int is different from that of top.

$[\![t_2]\!]\rho$, we ensure that our definition is well founded. Moreover, $\lfloor [\![t_2]\!]\rho \rfloor_k$ suffices here due to a property of semantic types called *nonexpansiveness*. Intuitively, if the interpretation of t_1 is nonexpansive then, if we only want to know whether v belongs to t_1 for k steps (i.e., we want the k-approximation of t_1), it is all the same whether we map $\underline{0}$ to $[\![t_2]\!]\rho$ or to $\lfloor [\![t_2]\!]\rho \rfloor_k$. Every one of the TML type interpretations we define is nonexpansive.

The single-slot vector type $\{t_1 : t_2\}$ constrains only one slot of the root value, intuitively, the slot specified by t_1. For its semantics, we first check that there exists some number n such that the constant root value $\mathsf{cv}(n)$ belongs to type t_1; this tells us that the type constrains the nth slot of the root value v. We then check that the value in the nth slot of root value v belongs to type t_2. As an example, consider the interpretation of the type $\{\mathsf{const}(3) : \mathsf{readable}\}$ which specifies that the value in slot 3 of a root value is readable.

4.5 References

In Section 4.1, we informally explained when a value belongs to the mutable reference type $\mathsf{ref}\,(t)$. We now give a formal definition of the interpretation of $\mathsf{ref}\,(t)$. The semantic interpretation of mutable-reference types is as follows.

$$[\![\mathsf{ref}\,(t)]\!](\rho)(k, \Sigma, \langle R,\ M\rangle, v) \ \triangleq \begin{cases} [\![t]\!](\rho)(k-1, \Sigma, \langle R,\ M\rangle, \mathsf{cv}(M(v(0)))) \\ \quad \wedge\ [\![\Sigma(v(0))]\!](\rho_\emptyset) \approx_{k-1} [\![t]\!](\rho) & \text{if } k > 0 \\ \\ \text{true} & \text{if } k = 0 \end{cases}$$

When k equals 0, the type $\mathsf{ref}\,(t)$ admits any value, reflecting the intuition that there are no steps remaining in which the program could exhibit unsafe behavior.

When k is greater than 0, the interpretation of $\mathsf{ref}\,(t)$ admits a root value v for k steps if $v(0)$ is a location such that the following conditions are satisfied. First, the contents of memory M at location $v(0)$ must belong to the interpretation of t for $k-1$ steps. Here, the decremented step index $(k-1)$ suffices since the program must take one step to fetch the referenced value. Second, it must be the case that the location $v(0)$ can only be updated with values that belong to the interpretation of type t—that is, t must be the permissible update type for location $v(0)$. Hence, we require that the type t match the type prescribed for location $v(0)$ by the store type Σ, to approximation $k-1$. Here approximation $k-1$ suffices since the program must take one step to update the reference cell. Furthermore, since the store type Σ maps locations to *closed* type expressions, the closed TML type $\Sigma(v(0))$ can be interpreted in the empty environment $\rho_\emptyset$. The type t, meanwhile, may contain free type variables, so it must be interpreted in the environment ρ.

It is easy to see that the interpretation of $\mathsf{ref}\,(t)$ is well founded since the definition of $[\![\mathsf{ref}\,(t)]\!]$ to approximation k only makes use of the interpretations of types t and $\Sigma(v(0))$ (whatever these types may be) to approximation $k-1$. Furthermore, the presence of impredicative polymorphism does not complicate matters. Consider, for instance, the interpretation of type $\mathsf{ref}\,(\underline{0})$ in an environment ρ that maps de Bruijn index $\underline{0}$ to the closed semantic type $\varphi = [\![\mathsf{exists}(\mathsf{ref}\,(\underline{0}))]\!]\rho$ and where $\Sigma(v(0)) = \mathsf{exists}(\mathsf{ref}\,(\underline{0}))$. Syntactically, the type of the contents $\mathsf{exists}(\mathsf{ref}\,(\underline{0}))$ is bigger than the reference type being interpreted $\mathsf{ref}\,(\underline{0})$—which makes it difficult to establish well-foundedness for traditional logical relations that are defined by induction on types alone—but this is not problematic since our logical relation is defined by outer induction on the step index and nested induction on types.

Next, we give the semantic interpretation of immutable references.

$$[\![\mathsf{box}\,(t)]\!](\rho)(k, \Sigma, \langle R, M\rangle, v) \triangleq \begin{cases} [\![t]\!](\rho)(k-1, \Sigma, \langle R,\ M\rangle, \mathsf{cv}(M(v(0)))) \wedge \\ [\![\Sigma(v(0))]\!](\rho_\emptyset) \approx_{k-1} [\![\mathsf{const}(M(v(0)))]\!](\rho) & \text{if } k > 0 \\ \\ \text{true} & \text{if } k = 0 \end{cases}$$

The interpretation of $\mathsf{box}\,(t)$ admits any value v if $v(0)$ is a location whose current contents in memory M belong to the interpretation of t, and if these contents remain unchanged in the future. Thus, the crucial difference between the interpretations of $\mathsf{ref}\,(t)$ and $\mathsf{box}\,(t)$ lies in their uses of the store type Σ. In the mutable case, we may update location $v(0)$ with a new value only if that value has the type t; hence, we require that the type prescribed by Σ matches type t. In the immutable case, however, we may update location $v(0)$ with a new value only if that value is *equal* to the current contents, namely $M(v(0))$. We enforce the latter by checking that the type prescribed by Σ matches the singleton integer type $\mathsf{const}(M(v(0)))$.

4.6 Code Pointers

Informally, we say that an address l is of type $\mathsf{codeptr}\,(t)$ if it is safe to jump to l when the precondition t is satisfied. In formalizing this intuition, however, we encounter a problem. It may be the case that we know that an address l is a code pointer, but the program will not jump to l until some point in the future. The following example illustrates the problem.

$$\begin{array}{rl} & \{\mathrm{r}_7 : \mathsf{codeptr}\,(\{\mathrm{r}_1 : \mathsf{int}\})\} \\ \textit{now}: & \text{instruction}_1 \\ & \quad\vdots \\ & \text{instruction}_n \\ \textit{future}: & \texttt{jmp}\ \mathrm{r}_7 \end{array}$$

Suppose that before executing the first instruction we have established that register 7 is a code pointer that we can jump to if register 1 contains an integer. But it is only after executing a series of instructions—none of which write to r_7—that the program decides to jump to register 7. Between now and that future point in time, the program may update mutable references and construct additional data structures (by writing to currently unallocated parts of memory). We need to find a systematic way to establish that despite any memory allocation and updates, the address pointed to by register 7 is still a code pointer with precondition $\{\mathrm{r}_1 : \mathsf{int}\}$ at that future point in time. In essence, we have to show that the types of memory locations are preserved over time. Our solution to this problem is to use a *possible-worlds* model.

Possible Worlds. To restrict our attention to only those future states that are *possible* during the execution of the programs that we consider, we construct a possible-world extend-state relation ($\sqsubseteq$) that constrains the evolution of a state. This is a relation on pairs (S, Σ) of a machine state S and a store type Σ. Informally, it relates a current state S and store type Σ to a future state S' and store type Σ'.

$$\begin{aligned}
(S,\Sigma) \sqsubseteq_k (S',\Sigma') &\triangleq \Sigma \sqsubseteq \Sigma' \ \wedge\ \vdash S :_k \Sigma \ \wedge\ \vdash S' :_k \Sigma' \\
&\quad\ \wedge\ \text{convention}((S,\Sigma),(S',\Sigma')) \\[1em]
\Sigma \sqsubseteq \Sigma' &\triangleq \text{dom}(\Sigma) \subseteq \text{dom}(\Sigma') \ \wedge\ \forall l \in \text{dom}(\Sigma).\ \Sigma(l) = \Sigma'(l) \\[1em]
\vdash \langle R,\ M\rangle :_k \Sigma &\triangleq \forall l \in \text{dom}(\Sigma).\ [\![\Sigma(l)]\!](\rho_\emptyset)(k-1,\Sigma,\langle R,\ M\rangle,\mathsf{cv}(M(l))) \\[1em]
\text{convention}((S,\Sigma),(S',\Sigma')) &\triangleq \text{valid\_state}(S,\Sigma) \ \wedge\ \text{valid\_state}(S',\Sigma') \\
&\quad\ \wedge\ \text{convention\_extend}((S,\Sigma),(S',\Sigma'))
\end{aligned}$$

The definition of $(S,\Sigma) \sqsubseteq_k (S',\Sigma')$ is also indexed by k. First, the definition asserts that $\Sigma \sqsubseteq \Sigma'$, which says that the store type can be extended as long as all the old types are preserved.

Second, in all possible worlds, the store should satisfy its store type. The relation $\vdash S :_k \Sigma$ means S satisfies Σ to approximation k. That is, for each location l, we look up the type $\Sigma(l)$ and then check whether $[\![\Sigma(l)]\!]$ in the empty environment $\rho_\emptyset$ accepts value $\mathsf{cv}(M(l))$ to approximation $k-1$. Since dereferencing $M(l)$ must use one step in any computation, $k-1$ is good enough.

Finally, the machine code we are judging is with respect to some architecture, and comes from some compiler that manages their registers, stack frames, the allocation heap, and so on. The architecture and the compiler use certain low-level conventions, which are captured by convention$((S,\Sigma),(S',\Sigma'))$. The convention predicate first checks that each state is a valid state. The valid_state(S,Σ) predicate enforces conventions on a single state. For example, on SPARC, the register 0 always has value 0. The convention_extend$((S,\Sigma),(S',\Sigma'))$ predicate enforces conventions across states, and is reflexive and transitive. We omit the exact definitions of valid_state and convention_extend, since they depend on particular architectures and compilers.[8] We only note that it should be straightforward to adjust these definitions for a different compiler and architecture as long as the compiler uses the same heap-allocation model.

[8]For the SML/NJ compiler and the SPARC architecture, we list the set of conditions in the two predicates below. In valid_state: (1) the register 0 has value 0; (2) values in registers and memory are 32-bit natural numbers (our axiomatization of the SPARC does not assume 32-bit registers); (3) memory locations in the register-spilling area and the heap area (defined by the SML/NJ compiler) are readable and writable; (4) the register-spilling area and the heap area are disjoint; (5) values in virtual registers are 32-bit natural numbers (virtual registers are for memory allocation and include a base pointer, a limit pointer, and a boundary pointer; see Section 6).

In convention_extend: (1) reserved registers remain the same (stack pointer, frame pointer, return address; SML/NJ does not generate stack pushes and pops, because it uses heap-allocated activation records, but we preserve these registers in order to return cleanly to the operating system when the program finishes); (2) the set of readable (writable, respectively) locations in the second state is larger than the set of readable (writable, respectively) locations of the first state (that is, we can allocate heap memory but we never deallocate it); (3) The base and the limit virtual registers remain the same.

It is easy to verify the following properties of the extend-state relation.

Lemma 4.4 (Properties of $\sqsubseteq$).

(1) (Reflexivity) If $\text{valid\_state}(S, \Sigma)$ and $\vdash S :_k \Sigma$, then $(S, \Sigma) \sqsubseteq_k (S, \Sigma)$.

(2) (Transitivity) If $(S_1, \Sigma_1) \sqsubseteq_k (S_2, \Sigma_2)$ and $(S_2, \Sigma_2) \sqsubseteq_k (S_3, \Sigma_3)$, then $(S_1, \Sigma_1) \sqsubseteq_k (S_3, \Sigma_3)$.

Having defined the extend-state relation, we can now formalize the semantic interpretation of code-pointer types.

$$\begin{aligned}[\![\mathsf{codeptr}\,(t)]\!](\rho)(k, \Sigma, S, v) \;\triangleq\;& \forall j < k, \Sigma', R', M'.\\ & \big((S, \Sigma) \sqsubseteq_j (\langle R',\ M'\rangle, \Sigma') \;\;\wedge\;\; R'(\mathsf{pc}) = v(0)\\ & \;\wedge\;\; [\![t]\!](\rho)(j, \Sigma', \langle R',\ M'\rangle, R')\big)\\ & \Rightarrow \text{safe\_state}(\langle R',\ M'\rangle, j)\end{aligned}$$

This definition quantifies over all possible future states that are reachable from the current one. If the program counter of the future state $\langle R',\ M'\rangle$ points to the address $v(0)$, and the precondition t is met, then the future state should be safe. The future state is required safe only for some $j < k$ steps, since we assume it takes $k - j$ steps to get from the current state to the future state.

This definition is based directly on the notion of safety, and consequently we have the following theorem.

Lemma 4.5 (Code Pointers Imply Safety). In any well-typed state where the program counter has type $\mathsf{codeptr}\,(t)$ and the register bank has type t, it is safe to execute any number of steps.

$$\frac{\begin{array}{ll}\text{valid\_state}(\langle R,\ M\rangle, \Sigma) & \forall k.\ \vdash \langle R,\ M\rangle :_k \Sigma\\ \forall k.\ [\![t]\!](\rho)(k, \Sigma, \langle R,\ M\rangle, R) & \forall k.\ [\![\mathsf{codeptr}\,(t)]\!](\rho)(k, \Sigma, \langle R,\ M\rangle, \mathsf{cv}(R(\mathsf{pc})))\end{array}}{\forall k.\ \text{safe\_state}(\langle R,\ M\rangle, k)}$$

4.7 Recursive Types

Recall from Section 3.1 that a recursive type $\mathsf{rec}\,(t)$ introduces a new type variable. Since we use de Bruijn indices to represent type variables, given a recursive type $\mathsf{rec}\,(t)$, each occurrence of the type variable $\underline{0}$ in t represents the recursive type itself. Based on this observation, we could naïvely attempt to define the semantics of recursive types as follows.

$$\text{A naïve interpretation:}\quad [\![\mathsf{rec}\,(t)]\!](\rho)(k, \Sigma, S, v) \;\triangleq\; [\![t]\!]\Big(\big([\![\mathsf{rec}\,(t)]\!]\rho\big) \bullet \rho\Big)(k, \Sigma, S, v)$$

That is, to interpret $\mathsf{rec}\,(t)$ in ρ, we construct a new type environment $[\![\mathsf{rec}\,(t)]\!]\rho \bullet \rho$, where type variable $\underline{0}$ is mapped back to the meaning of $\mathsf{rec}\,(t)$, and the rest of the type variables are resolved in ρ. Unfortunately, this naïve interpretation does not work because the definition of $[\![\mathsf{rec}\,(t)]\!]$ refers back to the very thing it is defining—i.e., we have a circular definition.

In their indexed model of recursive types, Appel and McAllester [2001] eliminated the above circularity via induction on the number of computation steps remaining, or k. The intuition is that for a recursive type $\mathsf{rec}\,(t)$ to be useful, it should take at least one computation step before the recursive type is used in t, that is, before the de Bruijn variable $\underline{0}$ is "reached." As an example, consider the following recursive type which describes a chain of pointers (i.e., integers ≥ 256) with a non-pointer (i.e., an integer in $[0, 256)$) at the end of the chain.

$$\mathsf{rec}\,(\mathsf{range}(0, 256) \cup (\mathsf{int}_{\geq}(256) \cap \mathsf{ref}\,(\underline{0})))$$

Let v be a value of the above type. Note that to get to a component of v with the type $\underline{0}$—which represents the recursive type itself—we have to go through a dereferencing step, as dictated by the reference type $\mathsf{ref}\,(\underline{0})$.

Based on this intuition, we define a notion of *contractive semantic types*.

Definition 4.6 (Contractive Semantic Types).

$$\text{contractive}(\tau, k) \;\triangleq\; \forall j \leq k, \varphi, \rho.\ \tau(\lfloor\varphi\rfloor_j \bullet \rho) \approx_k \tau(\lfloor\varphi\rfloor_{(j-1)} \bullet \rho)$$

The above definition formalizes the notion that when using a value of a semantic type τ, it takes at least one step for a program to reach the part of the value represented by the type variable $\underline{0}$. Essentially, it says that if there are only $j \leq k$ steps remaining, then it makes no difference whether we interpret $\underline{0}$ as the j-th approximation of φ, written $\lfloor\varphi\rfloor_j$ (see Definition 4.2), or as the $(j-1)$-th approximation—since the first step will not reach the part of the value represented by $\underline{0}$, the $(j-1)$-th approximation of φ suffices.

With the above definition of contractive types in hand, we can define the interpretation of recursive types as follows.

$$[\![\mathsf{rec}\,(t)]\!](\rho)(k, \Sigma, S, v) \;\triangleq\; \text{contractive}([\![t]\!], k) \;\wedge\; [\![t]\!]\Big(\big(\lfloor[\![\mathsf{rec}\,(t)]\!]\rho\rfloor_{(k-1)}\big) \bullet \rho\Big)(k, \Sigma, S, v)$$

This interpretation is well defined because to give the semantics of $\mathsf{rec}\,(t)$ to approximation k, we map $\underline{0}$ to $\lfloor[\![\mathsf{rec}\,(t)]\!]\rho\rfloor_{(k-1)}$—that is, we rely only on the $(k-1)$-th approximation of the meaning of $\mathsf{rec}\,(t)$. In other words, the above interpretation is defined by induction on k.

Given the notion of contractiveness and the interpretation of recursive types, we can prove that a recursive type is semantically equivalent to its unfolded type.

Lemma 4.7. For all k, if contractive($[\![t]\!], k$), then $[\![\mathsf{rec}\,(t)]\!] \approx_k [\![\mathsf{subst}(t, \mathsf{rec}\,(t))]\!]$.

Proof. By the definition of $\approx_k$, we need to show that $[\![\mathsf{rec}\,(t)]\!](\rho)(j, \Sigma, S, v)$ if and only if $[\![\mathsf{subst}(t, \mathsf{rec}\,(t))]\!](\rho)(j, \Sigma, S, v)$, for all ρ, $j \leq k$, Σ, S, and v. The proof is as follows.

$$\begin{array}{lll}
 & [\![\mathsf{rec}\,(t)]\!](\rho)(j, \Sigma, S, v) & \\
\Leftrightarrow & [\![t]\!]\Big(\big(\lfloor[\![\mathsf{rec}\,(t)]\!]\rho\rfloor_{(j-1)}\big) \bullet \rho\Big)(j, \Sigma, S, v) & \text{by definition of } [\![\mathsf{rec}\,(t)]\!] \\
\Leftrightarrow & [\![t]\!]\Big(\big(\lfloor[\![\mathsf{rec}\,(t)]\!]\rho\rfloor_{j}\big) \bullet \rho\Big)(j, \Sigma, S, v) & \text{by contractive}([\![t]\!], k) \\
\Leftrightarrow & [\![\mathsf{subst}(t, \mathsf{rec}\,(t))]\!](\rho)(j, \Sigma, S, v) & \text{by definition of } [\![\mathsf{subst}(t, \mathsf{rec}\,(t))]\!]
\end{array}$$

□

The term "contractive types" comes from the ideal model of recursive types by MacQueen et al. [1986]. Appel and McAllester [2001] called these well-founded types because they justify the use of induction on the approximation index.

Not all types are contractive. For example, the type $\underline{0}$ is not contractive. To make it easier to prove that certain types are contractive, we introduce a related notion of *nonexpansive semantic types.* Informally, a nonexpansive type at approximation index k should not inspect its environment ρ in greater detail than k.

Definition 4.8 (Nonexpansive Semantic Types).

$$\begin{array}{lcl}\text{nonexpansive}(\tau, k) & \triangleq & \forall \rho, \rho', j \leq k, \Sigma, S, v. \\ & & \big(\rho \approx_j \rho' \;\wedge\; \tau(\rho)(j, \Sigma, S, v)\big) \;\Rightarrow\; \tau(\rho')(j, \Sigma, S, v)\end{array}$$

All the type interpretations that we have defined are nonexpansive.[9] There are also lemmas that combine contractive types and nonexpansive types to get contractive types [Appel and McAllester 2001]. For example, the composition of a nonexpansive type constructor with a contractive type constructor (in either order) is contractive.

4.8 Quantified Types

The difficulty of giving semantics to impredicatively quantified types arises from the fact that the quantification ranges over the very types that are being defined. For example, for the type $\mathsf{forall}(t)$, the type variable $\underline{0}$ in t may be instantiated with the type $\mathsf{forall}(t)$ itself.

We handle the circularity in impredicatively quantified types using the same technique used to deal with the circularity in the semantics of recursive types. That is, we assume that the semantic type $[\![t]\!]$ in $\mathsf{forall}(t)$ is contractive, so that it takes at least one computation step before the index $\underline{0}$ in t can be "reached." Based on this intuition, the semantics of polymorphic types and existential types is defined as follows.

$$\begin{array}{lcl}[\![\mathsf{forall}(t)]\!](\rho)(k, \Sigma, S, v) & \triangleq & \text{contractive}([\![t]\!], k) \;\wedge \\ & & \forall t_1.\; [\![t]\!]\Big(\big(\lfloor [\![t_1]\!]\rho \rfloor_{(k-1)}\big) \bullet \rho\Big)(k, \Sigma, S, v) \\ \\ [\![\mathsf{exists}(t)]\!](\rho)(k, \Sigma, S, v) & \triangleq & \text{contractive}([\![t]\!], k) \;\wedge \\ & & \exists t_1.\; [\![t]\!]\Big(\big(\lfloor [\![t_1]\!]\rho \rfloor_{(k-1)}\big) \bullet \rho\Big)(k, \Sigma, S, v)\end{array}$$

Since there is at least one step before t_1 is used, mapping $\underline{0}$ to $\lfloor [\![t_1]\!]\rho \rfloor_{(k-1)}$ is sufficient. Assuming contractiveness, the following standard introduction and elimination lemmas for quantified types are easy to verify.

Lemma 4.9. Assume contractive$([\![t]\!], k)$.

(1) If $[\![\mathsf{subst}(t, t')]\!](\rho)(k, \Sigma, S, v)$ holds for all type t', then $[\![\mathsf{forall}(t)]\!](\rho)(k, \Sigma, S, v)$.
(2) If $[\![\mathsf{forall}(t)]\!](\rho)(k, \Sigma, S, v)$, then for all type t', we have $[\![\mathsf{subst}(t, t')]\!](\rho)(k, \Sigma, S, v)$.
(3) If $[\![\mathsf{subst}(t, t')]\!](\rho)(k, \Sigma, S, v)$ holds for some type t', then $[\![\mathsf{exists}(t)]\!](\rho)(k, \Sigma, S, v)$.
(4) If $[\![\mathsf{exists}(t)]\!](\rho)(k, \Sigma, S, v)$, then for some type t', we have $[\![\mathsf{subst}(t, t')]\!](\rho)(k, \Sigma, S, v)$.

[9]The reader may have noticed that the definition of contractive(τ, k) requires τ to be contractive only at variable $\underline{0}$, while the definition of nonexpansive(τ, k) requires τ to be nonexpansive at every variable. We can easily give an alternate definition of contractive(τ, k) that requires τ to be contractive at every variable. However, our existing definition of contractiveness suffices since Lemma 4.7 requires only that $[\![t]\!]$ in $\mathsf{rec}\,(t)$ be contractive at variable $\underline{0}$.

An Alternative Semantics. Having presented our actual semantics for quantified types, we next discuss an alternative proposal. Our goal in discussing this alternative interpretation is to explain the impact of not building the contractiveness requirement into the definition of quantified types as we have done above. We discuss this alternative semantics using universal types; the case for existential types is similar.

An alternative (and naïve) interpretation:

$$[\![\mathsf{forall}(t)]\!](\rho)(k, \Sigma, S, v) \;\triangleq\; \forall t_1.\ [\![t]\!]\big(([\![t_1]\!]\rho) \bullet \rho\big)(k-1, \Sigma, S, v)$$

The above interpretation is well defined. The reader may wonder whether the use of $[\![t_1]\!]\rho$ makes sense, since t_1 may be the universal type $\mathsf{forall}(t)$ itself. It is sensible—i.e., it does not lead to a circular definition—since the type t at approximation $k-1$ (or, with $k-1$ steps remaining) will need its environment only to approximation $k-1$. Therefore, in the definition above, it would have been equivalent to use the $(k-1)$-th approximation of $[\![t_1]\!]\rho$ in place of $[\![t_1]\!]\rho$. (Technically, the reason that $[\![t]\!]$ with $k-1$ steps remaining only needs its environment to approximation $k-1$ is due to the nonexpansiveness property (Definition 4.8), which we defined in Section 4.7.)

Although our alternative interpretation is well defined, it fails to conform to the computation model of real architectures. This is due to the fact that real machines do not take a computation step when performing type application on a value of universal type (or when unpacking a value of existential type). That is, suppose that we are in a state where we can safely execute k more steps, and $[\![\mathsf{forall}(t)]\!](\rho)(k, \Sigma, S, v)$ is true. Now suppose we perform a type application. According to the alternative semantics above, we have $[\![t]\!]\big(([\![t_1]\!]\rho) \bullet \rho\big)(k-1, \Sigma, S, v)$, which indicates we can now safely execute only $k-1$ more steps. But real architectures do not perform type applications as computation steps. Hence, in terms of the real operational semantics we haven't actually consumed a step, so there are still k steps remaining in the computation—a mismatch. If we were to use this alternative model and still base our theory on the number of computation steps, we would have to execute a no-op in conjunction with every type application or unpack.

Our chosen model of quantified types "solves" this problem by requiring $[\![t]\!]$ to be contractive in the definition of $[\![\mathsf{forall}(t)]\!]$, so that there is no need for the index k to decrease in the interpretation (as it does in our first alternative semantics above). It is reasonable to ask whether this restriction might affect us in practice. The short answer is that for type-preserving compilation in our ML compiler, it is inconvenient but not fatal [Ahmed 2004, chapter 4.1]. In particular, the existential types we need are usually record types, which are contractive, and the universal types are code-pointer types, which are also contractive.

4.9 Scalar and Numeric Types and Kinds

We treat any predicate on (φ, k) as a kind—as before, φ is a closed semantic type and k a step index—and use the metavariable κ to denote a kind. The semantics

$$\frac{}{\text{haskind}([\![\mathsf{bottom}]\!], \text{numeric})} \qquad \frac{}{\text{haskind}([\![\mathsf{bottom}]\!], \text{scalar})} \qquad \frac{}{\text{haskind}([\![\mathsf{int}]\!], \text{scalar})}$$

$$\frac{}{\text{haskind}([\![\mathsf{const}(n)]\!], \text{numeric})} \qquad \frac{}{\text{haskind}([\![\mathsf{const}(n)]\!], \text{scalar})}$$

$$\frac{\text{haskind}([\![t_1]\!], \text{numeric}) \quad \text{haskind}([\![t_2]\!], \text{numeric})}{\text{haskind}([\![\mathsf{plus}(t_1, t_2)]\!], \text{numeric})} \qquad \frac{\text{haskind}([\![t_1]\!], \text{scalar})}{\text{haskind}([\![\mathsf{plus}(t_1, \mathsf{const}(n))]\!], \text{scalar})}$$

$$\frac{\text{haskind}([\![t]\!], \text{scalar})}{\text{haskind}([\![\mathsf{int}_=(t)]\!], \text{scalar})} \qquad \frac{\text{haskind}([\![t]\!], \text{scalar})}{\text{haskind}([\![\mathsf{ref}\ (t)]\!], \text{scalar})} \qquad \frac{}{\text{haskind}([\![\mathsf{codeptr}\ (t)]\!], \text{scalar})}$$

$$\frac{\text{haskind}([\![t]\!], \text{scalar})}{\text{haskind}([\![\mathsf{rec}\ (t)]\!], \text{scalar})} \qquad \frac{\text{haskind}([\![t]\!], \text{numeric})}{\text{haskind}([\![\mathsf{forall}(t)]\!], \text{numeric})} \qquad \frac{\text{haskind}([\![t]\!], \text{scalar})}{\text{haskind}([\![\mathsf{forall}(t)]\!], \text{scalar})}$$

$$\frac{\text{haskind}([\![t]\!], \text{numeric})}{\text{haskind}([\![\mathsf{exists}(t)]\!], \text{numeric})} \qquad \frac{\text{haskind}([\![t]\!], \text{scalar})}{\text{haskind}([\![\mathsf{exists}(t)]\!], \text{scalar})}$$

Fig. 4. Kinding lemmas (selected)

of scalar and numeric kinds are defined as follows.

$$\begin{aligned}\text{scalar}(\varphi, k) \ &\triangleq\ \forall j \le k, \Sigma, S, v, v'.\\ &\quad \big(v(0) = v'(0)\ \wedge\ \varphi(j, \Sigma, S, v)\big)\ \Rightarrow\ \varphi(j, \Sigma, S, v')\end{aligned}$$

$$\begin{aligned}\text{numeric}(\varphi, k) \ &\triangleq\ \forall j \le k, \Sigma, \Sigma', S, S', v, v'.\\ &\quad \big(\varphi(j, \Sigma, S, v)\ \wedge\ \varphi(j, \Sigma', S', v')\big)\ \Rightarrow\ v(0) = v'(0)\end{aligned}$$

The scalar kind accepts closed semantic types that care only about slot zero of a root value. The numeric kind accepts a closed semantic type φ if and only if all root values v in φ contain the same integer n in $v(0)$.

We define a predicate haskind(τ, κ) to denote that a semantic type τ has kind κ:

$$\text{haskind}(\tau, \kappa)\ \triangleq\ \forall k, \rho.\ \kappa(\tau(\rho), k).$$

With the semantics of kinds in place, we can prove many kinding lemmas. We present a few of these in Figure 4.

Next, we define the semantic interpretations of the kind-coercion types $\mathsf{kd\_scalar}(t)$ and $\mathsf{kd\_numeric}(t)$.

$$[\![\mathsf{kd\_scalar}(t)]\!](\rho)(k, \Sigma, S, v)\ \triangleq\ [\![t]\!](\rho)(k, \Sigma, S, v)\ \wedge\ \text{scalar}([\![t]\!]\rho, k)$$

$$[\![\mathsf{kd\_numeric}(t)]\!](\rho)(k, \Sigma, S, v)\ \triangleq\ [\![t]\!](\rho)(k, \Sigma, S, v)\ \wedge\ \text{numeric}([\![t]\!]\rho, k)$$

The interpretation of $\mathsf{kd\_scalar}(t)$ is equivalent to the interpretation of t if t is a scalar type; otherwise it is equivalent to bottom. The type $\mathsf{kd\_numeric}(t)$ is similarly defined. Note that we can prove that haskind$([\![\mathsf{kd_scalar}(t)]\!], \text{scalar})$ and haskind$([\![\mathsf{kd_numeric}(t)]\!], \text{numeric})$ for any t. Therefore, we can prove that $[\![\mathsf{forall}(\mathsf{plus}(\mathsf{kd\_numeric}(\underline{0}), \mathsf{kd\_numeric}(\underline{0})))]\!]$ is a numeric type (with lemmas in Figure 4). Finally, because the type bottom is a member of each kind, applying a kind coercion to bottom always results in a member of the corresponding kind, and each kind coercion is idempotent.

4.10 Valid Semantic Types

We have given the semantic interpretations of TML types above. The interpretation of a TML type t, written $[\![t]\!]$, is a predicate on $(\rho)(k, \Sigma, S, v)$. Recall that any such predicate is a semantic type (for which we use the metavariable τ), while any predicate on (k, Σ, S, v) is a closed semantic type (for which we use the metavariable φ). In this section, we define the notion of a *valid semantic type* which encompasses various properties that our TML type interpretations must satisfy.

First, if a value v belongs to the interpretation of a type t for k steps—meaning that v may not actually have type t, but in k steps of execution, the program cannot tell the difference—then it should also belong to that type interpretation for $j \leq k$ steps. That is, a type interpretation $[\![t]\!]$ should be *downward closed* with respect to the step index.

Definition 4.10 (Downward Closure).

$$\begin{aligned}
\text{downward-closed}(\varphi) &\triangleq \forall k, \Sigma, S, v.\ \forall j \leq k.\ \varphi(k, \Sigma, S, v) \ \Rightarrow\ \varphi(j, \Sigma, S, v) \\
\text{downward-closed}(\rho) &\triangleq \forall i.\ \text{downward-closed}(\rho(i)) \\
\text{downward-closed}(\tau) &\triangleq \forall \rho.\ \text{downward-closed}(\rho) \ \Rightarrow\ \text{downward-closed}(\tau(\rho))
\end{aligned}$$

We define downward closure for a closed semantic type φ and extend the notion pointwise to environments ρ. We say that a semantic type τ is downward closed if, given a downward closed environment ρ, the closed semantic type $\tau(\rho)$ is downward closed.

Second, the k approximation of a type interpretation $[\![t]\!]$ should not inspect its environment ρ beyond approximation k. In other words, every TML type interpretation should be *nonexpansive* (see Definition 4.8 on page 27).

Finally, if a value v belongs to type t in world (S, Σ), then v should continue to belong to type t in any world (S', Σ') that is reachable from (S, Σ)—that is, if $(S, \Sigma) \sqsubseteq_j (S', \Sigma')$, where j is the number of steps left to execute. As explained in Section 4.6, changes to the state—such as dynamic allocation or updates to mutable references—should not invalidate assumptions about the types of values. That is, type interpretations should be closed under the extend-state relation. This property, called *monotonicity*, is typical in possible-worlds models. It is also known as Kripke monotonicity.

Definition 4.11 (Monotonicity).

$$\begin{aligned}
\text{monotone}(\varphi, k) &\triangleq \forall j \leq k.\ \forall S, S', \Sigma, \Sigma', v. \\
&\qquad\qquad \big(\varphi(j, \Sigma, S, v) \ \wedge\ (S, \Sigma) \sqsubseteq_j (S', \Sigma')\big) \ \Rightarrow\ \varphi(j, \Sigma', S', v) \\
\text{monotone}(\rho, k) &\triangleq \forall i.\ \text{monotone}(\rho(i), k) \\
\text{monotone}(\tau, k) &\triangleq \forall \rho.\ \text{monotone}(\rho, k) \ \Rightarrow\ \text{monotone}(\tau(\rho), k)
\end{aligned}$$

We say that τ is a valid semantic type if it satisfies each of the three properties mentioned above.

Definition 4.12 (Valid Semantic Type).

$$\begin{aligned}
&\text{valid-sem-type}(\tau) \triangleq \\
&\qquad \forall k.\ \text{downward-closed}(\tau) \ \wedge\ \text{nonexpansive}(\tau, k) \ \wedge\ \text{monotone}(\tau, k)
\end{aligned}$$

$$\frac{t_1 <: t_1' \qquad t_2 <: t_2'}{t_1 \cap t_2 <: t_1' \cap t_2'} \qquad \frac{t <: t_1 \qquad t <: t_2}{t <: t_1 \cap t_2} \qquad \frac{}{t_1 \cap t_2 <: t_1}$$

$$\frac{}{t \cap t \doteq t} \qquad \frac{}{t_1 \cap t_2 \doteq t_2 \cap t_1} \qquad \frac{}{(t_1 \cap t_2) \cap t_3 \doteq t_1 \cap (t_2 \cap t_3)}$$

$$\frac{}{t_1 \cap (t_1 \cup t_2) \doteq t_1} \qquad \frac{}{t_1 \cup (t_1 \cap t_2) \doteq t_1}$$

$$\frac{}{t_1 \cap (t_2 \cup t_3) \doteq (t_1 \cap t_2) \cup (t_1 \cap t_3)} \qquad \frac{}{t_1 \cup (t_2 \cap t_3) \doteq (t_1 \cup t_2) \cap (t_1 \cup t_3)}$$

$$\frac{}{t \cap \mathsf{top} \doteq t} \qquad \frac{}{t \cap \mathsf{bottom} \doteq \mathsf{bottom}} \qquad \frac{}{\mathsf{forall}(t_1) \cap \mathsf{forall}(t_2) \doteq \mathsf{forall}(t_1 \cap t_2)}$$

Fig. 5. Subtyping and type-equality lemmas for intersection types

Each of the TML type interpretations presented above (Sections 4.4 through 4.9) is a valid semantic type.

THEOREM 4.13. *If t is a TML type, then* valid-sem-type($[\![t]\!]$).

To prove the theorem, we need to show that

$$\forall t.\ \forall k.\ \text{downward-closed}([\![t]\!]) \ \wedge\ \text{nonexpansive}([\![t]\!], k) \ \wedge\ \text{monotone}([\![t]\!], k).$$

The proof is by induction on the step index k and nested induction on the type t.

4.11 Subtyping and Type-Equality Lemmas

Subtyping relations are typically defined either syntactically by a formal system or semantically as the subset relation on the semantic interpretations of types. We take the latter approach in defining subtyping for TML. We say t_1 is a subtype of t_2, written $t_1 <: t_2$, if the interpretation of t_1 is a subset of the interpretation of t_2. We say types t_1 and t_2 are equal, written $t_1 \doteq t_2$, if t_1 is a subtype of t_2 and vice versa. The following definition makes use of semantic subtyping ($\subseteq_k$) which we defined in Section 4.3 (see Definition 4.1).

Definition 4.14 (SUBTYPING AND TYPE EQUALITY).

$$\begin{aligned} t_1 <: t_2 &\ \triangleq\ \forall k.\ [\![t_1]\!] \subseteq_k [\![t_2]\!] \\ t_1 \doteq t_2 &\ \triangleq\ t_1 <: t_2 \ \wedge\ t_2 <: t_1 \end{aligned}$$

Subtyping rules typically found in syntactic type theories can be proved as lemmas in TML. In fact, any subtyping rule can be added to TML, as long as it can be proved as a lemma given the semantic definition of subtyping and the model of TML types. We call these rules subtyping lemmas.

We have proved a large number of useful subtyping and type-equality lemmas. We do not list all of these lemmas here, but for illustrative purposes we present a subset of the lemmas involving intersection types in Figure 5.

5. A COMPOSITIONAL LOGIC FOR CONTROL FLOW

We have introduced TML, whose expressive type constructors can specify rich properties of machine states. In this section, we introduce another key abstraction, $\mathcal{L}_c$, for specifying properties of machine instructions.

At a first approximation, the logic $\mathcal{L}_c$ specifies properties of machines instructions in terms of pre- and post-conditions, similar to Hoare Logic. But unlike Hoare Logic, $\mathcal{L}_c$ can handle unstructured control flow in low-level programs. It introduces a single, unifying notion of *multiple-entry and multiple-exit program fragments.* The notion of program fragments unifies various structures of TALs, including instructions, basic blocks, and programs. An instruction is treated as a program fragment with one entry and one exit, or two exits if the instruction is a conditional branch. A basic block is a program fragment with one entry and multiple exits. A complete program, meanwhile, is a fragment with multiple entries and zero exits (if there are no indirect exits in registers). This unifying notion of program fragments permits $\mathcal{L}_c$ to have only one judgment for specifying properties of fragments, while TALs usually have different judgments for specifying properties of instructions, basic blocks, and programs. Furthermore, $\mathcal{L}_c$ has a set of simple composition rules for composing fragments. A TAL's rules for composing basic blocks or instructions can be explained in terms of $\mathcal{L}_c$'s composition rules.

Since our intermediate logic $\mathcal{L}_c$ is built to reason about program fragments, it enables modular verification of properties of partial programs. Each program fragment needs to be verified only once; program fragments are linked together by $\mathcal{L}_c$'s composition rules. In contrast, many TALs only support verification of complete programs, due to the fact that the TAL's judgments require a global map of preconditions of all basic blocks in the program. If two basic blocks are verified under different global maps, then they cannot be composed in such TALs. In this respect, $\mathcal{L}_c$ is more general.

In the rest of this section we present the syntax of the composition rules of $\mathcal{L}_c$ (Section 5.1), and then present its semantics in higher-order logic in a continuation-based interpretation (Section 5.2). We will illustrate the power of $\mathcal{L}_c$ in Section 6 by showing how to model the control flow of a simple typed assembly language.

5.1 Logic $\mathcal{L}_c$

We have implemented our control logic $\mathcal{L}_c$ on the SPARC. To convey the essential ideas of the logic, we present it here on the imaginary machine of Section 2. The syntax of $\mathcal{L}_c$ is as follows.

$$\begin{array}{rrcl}
(\textit{machine instructions}) & i & ::= & \texttt{add}\; r_d, r_s, n \mid \texttt{ld}\; r_d, r_s[n] \mid \texttt{st}\; r_d[n], r_s \mid \ldots \\
(\textit{program fragments}) & F & ::= & \{i_1@l_1, \ldots, i_n@l_n\} \\
(\textit{register-file types}) & \phi & ::= & \{\, r_1 : t_1, \ldots, r_n : t_n \,\} \\
(\textit{continuation sets}) & \Psi & ::= & \{l_1 \multimap \phi_1, \ldots, l_n \multimap \phi_n\}
\end{array}$$

The machine instructions i are exactly those given in Section 2; we do not list all of them here.

We write $i@l$ to denote that the instruction i is at the machine address l. A program fragment F is a set of instruction-at-location ($i@l$) specifications.

A register-file type ϕ specifies the types of the general-purpose registers. We use register-file types as our assertion language to specify invariants of machine states at particular program points. In the application of $\mathcal{L}_c$ to TML we choose register-file types as our assertion language, but any reasonable predicate on states can be used. Formally, a register-file type ϕ is simply a mapping from general-purpose registers to TML types. A register-file type can be encoded as a TML type as follows. (This encoding just formalizes the shorthand we were using in Section 3.)

$$\{\, r_1 : t_1, \ldots, r_n : t_n \,\} \;\triangleq\; \{\mathsf{const}(r_1) : t_1\} \cap \ldots \cap \{\mathsf{const}(r_n) : t_n\}$$

Since register-file types can be encoded as TML types, we reuse much of the notation from TML types, for instance, lifting the notion of TML subtyping up to register-file subtyping: $\phi_1 <: \phi_2$. We write $\phi[r \mapsto t] = \phi'$ to denote that ϕ' is identical to ϕ except for the fact that ϕ' maps register r to the type t.

In $\mathcal{L}_c$, we use the notation $l \mathrel{-\!\!\rhd} \phi$ (pronounced "l with ϕ") to associate a register-file type with an address. Informally, if $l \mathrel{-\!\!\rhd} \phi$ holds in a program, then whenever ϕ is satisfied, it is safe to "continue from l" (or, jump to l). In other words, l is a continuation with formal parameters whose types are specified by ϕ. Hence, we call ϕ a *precondition* of the address l, and call $l \mathrel{-\!\!\rhd} \phi$ a *continuation*. We use the metavariable Ψ to range over sets of continuations.

Generalized Hoare Tuples. In Hoare Logic [Hoare 1969], a triple $\{p\}s\{q\}$ describes the relationship between exactly two states—the normal entry and exit states—associated with a program execution. That is, if the state before the execution of s satisfies the assertion p, then the state after the execution of s satisfies q. For a high-level programming language with structured control flow, a program logic based on Hoare triples works fine. In low-level languages, however, the presence of jump instructions and conditional branch instructions means that low-level code may contain multiple entry and exit points. Since a Hoare logic triple $\{p\}s\{q\}$ is tailored to describe the relationship between the normal entry and normal exit states, it is not surprising that such Hoare logic triples are problematic when considering program fragments with more than one entry or exit point [O'Donnell 1982].

To facilitate reasoning about control flow in low-level programs, the logic $\mathcal{L}_c$ provides a judgment for specifying properties of multiple-entry and multiple-exit program fragments:

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

where F is a set of program fragments, and Ψ' and Ψ are sets of continuations. The meaning of the judgment may be explained as follows. Suppose

$$\begin{array}{lcl} \Psi' & = & \{l'_1 \mathrel{-\!\!\rhd} \phi'_1, \ldots, l'_m \mathrel{-\!\!\rhd} \phi'_m\} \text{ and} \\ \Psi & = & \{l_1 \mathrel{-\!\!\rhd} \phi_1, \ldots, l_n \mathrel{-\!\!\rhd} \phi_n\}. \end{array}$$

Here, addresses $l'_1, \ldots, l'_m$ in Ψ' are the exit points of F, while $l_1, \ldots, l_n$ in Ψ are the entry points of F. The following figure depicts the relationship between F, Ψ and Ψ'.

$$\begin{tikzcd}
l_1 \mathrel{-\!\!\rhd} \phi_1 \arrow[dr] & \cdots & l_n \mathrel{-\!\!\rhd} \phi_n \arrow[dl] & \Psi \\
 & \boxed{F} \arrow[dl] \arrow[dr] & & \\
l'_1 \mathrel{-\!\!\rhd} \phi'_1 & \cdots & l'_m \mathrel{-\!\!\rhd} \phi'_m & \Psi'
\end{tikzcd}$$

Thus, informally, the judgment $F\,;\,\Psi' \vdash \Psi$ says that for a set F of program fragments, if it is safe to continue from any of the exits, provided that the precondition associated with that exit is true, then it is safe to continue from any of the entries, provided that the precondition associated with that entry is true.[10] Note that in the judgment $F\,;\,\Psi' \vdash \Psi$ we have put the exit points on the left of $\vdash$, and the entries on the right. An alternative notation for this judgment is $\vdash \{\Psi\}F\{\Psi'\}$, which is perhaps easier to read as the entry points appear to the left of the exits. We decided to write $F\,;\,\Psi' \vdash \Psi$ instead because the judgment is meant to yield Ψ given F and Ψ'; the notation $F\,;\,\Psi' \vdash \Psi$ more closely reflects the logical interpretation of the judgment.

Using this judgment, we can provide $\mathcal{L}_c$ rules for individual machine instructions. As an example, consider the following rule for the `add` instruction.

$$\frac{}{\{(\mathsf{add}\ r_d, r_s, n)@l\};\ \{(l+1) \mathrel{-\!\!\triangleright} \{\, r_d : \mathsf{int}\,\}\} \vdash \{l \mathrel{-\!\!\triangleright} \{\, r_s : \mathsf{int}\,\}\}}$$

The fragment, $(\mathsf{add}\ r_d, r_s, n)@l$, has one entry, namely l, and one exit, namely $l+1$. The rule states that if it is safe to continue from the exit $l+1$ when $\{\, r_d : \mathsf{int}\,\}$ is true, then it is safe to continue from the entry l when $\{\, r_s : \mathsf{int}\,\}$ is true. The informal explanation for why it is safe to continue from l is as follows. Suppose we start from l in an initial state where the next instruction to execute is "$\mathsf{add}\ r_d, r_s, n$", and register r_s is of type int. The new state after the execution of the `add` instruction reaches the exit $l+1$, and based on the semantics of `add`, the destination register r_d is of type int. Since we have assumed that it is safe to continue from $l+1$ when $\{\, r_d : \mathsf{int}\,\}$ is true, we may conclude that it is safe to continue in the new state from $l+1$. Hence, it follows that the initial state can safely continue from l when $\{\, r_s : \mathsf{int}\,\}$ is true.

Notice that the `add` rule may be expressed in a Hoare-logic style (with pre- and postconditions) as follows.

$$\{\, r_s : \mathsf{int}\,\}(l : \mathsf{add}\ r_d, r_s, n)\{\, r_d : \mathsf{int}\,\}$$

Note that this Hoare-logic style rule for `add` expresses the same specification as the `add` rule above.

The Hoare triple $\{\phi\}(l : i)\{\phi'\}$, for a nonjump instruction i that has only one entry and one exit, has in $\mathcal{L}_c$ a corresponding judgment: $\{i@l\}\,;\,\{(l+1) \mathrel{-\!\!\triangleright} \phi'\} \vdash \{l \mathrel{-\!\!\triangleright} \phi\}$. We interpret the *direct style* Hoare triple $\{\phi\}(l : i)\{\phi'\}$ as follows: if ϕ is true in the state before i, and i terminates, then ϕ' will be true in the state after the execution of i. The direct-style semantics positively asserts that the exit state satisfies the postcondition. We interpret $\{i@l\}\,;\,\{(l+1) \mathrel{-\!\!\triangleright} \phi'\} \vdash \{l \mathrel{-\!\!\triangleright} \phi\}$ in *continuation style*: If $l+1$ is safe provided that ϕ' is satisfied, then l is safe provided that ϕ is satisfied. The two styles of interpretations are closely related—in fact, under certain assumptions they are equivalent [Tan 2005, Ch 2.4.2].

Hoare triples can accommodate only one entry and one exit, and thus cannot specify conditional-branch instructions such as `bnz`, which has two possible exits.[11]

[10] In this informal explanation of $F\,;\,\Psi' \vdash \Psi$ we have chosen to ignore, for the moment, an additional property of the judgment, which is the requirement that it must take at least one computation step to get from an entry to an exit point. We give the precise interpretation of $F\,;\,\Psi' \vdash \Psi$ in Section 5.2.

[11] Later variants of Hoare Logic can accommodate more than one exit; see the discussion of related work (Section 8).

Our judgment $F\,;\,\Psi' \vdash \Psi$ is more general. A rule for `bnz` is as below; it assumes two exit continuations.

$$\begin{array}{l}\{(\texttt{bnz}\ r_s, l_d)@l\};\\ \{l_d \mathrel{-\!\!\triangleright} \big(\phi[r_s \mapsto \phi(r_s) \cap \mathsf{int}_{\neq}(\mathsf{const}(0))]\big), (l+1) \mathrel{-\!\!\triangleright} \big(\phi[r_s \mapsto \phi(r_s) \cap \mathsf{const}(0)]\big)\}\\ \quad \vdash \{l \mathrel{-\!\!\triangleright} \phi\}\end{array}$$

When the branch is taken, the register r_s is nonzero; therefore the rule adds the information $\mathsf{int}_{\neq}(\mathsf{const}(0))$ to $\phi(r_s)$. Similarly, when the branch is not taken, the register r_s should additionally have type $\mathsf{const}(0)$.

We have used example rules to explain our judgment for program fragments. One important thing to note is that the logic $\mathcal{L}_c$ is really about how to compose program fragments using its composition rules. In some sense, the logic is parameterized over the set of rules for individual instructions. Rules for machine instructions can be soundly added to the logic, as long as they can be proved from the semantics of the judgment for program fragments (we will introduce the formal semantics later). As an example, the rule for the `add` instruction that we saw above is not realistic as it does not specify that the types of registers other than the destination register must be preserved. The following is a more realistic rule; it can be proved sound and can therefore be added to $\mathcal{L}_c$.

$$\frac{\phi <: \{\, r_s : \mathsf{int} \,\} \qquad \phi[r_d \mapsto \mathsf{int}] = \phi'}{\{(\texttt{add}\ r_d, r_s, n)@l\};\ \{(l+1) \mathrel{-\!\!\triangleright} \phi'\} \vdash \{l \mathrel{-\!\!\triangleright} \phi\}}$$

Composition Rules. The core of $\mathcal{L}_c$ is its composition rules. These rules can be used to compose judgments on individual instructions into properties of the whole collection. By internalizing control flow, these composition rules permit modular reasoning.

Figure 6 presents $\mathcal{L}_c$'s composition rules. The figure does not include the rules for $\phi_1 <: \phi_2$. The symbol $<:$ is the TML subtyping relation, and the logic $\mathcal{L}_c$ admits all TML subtyping lemmas.

We next illustrate the composition rules using a running example in Figure 7. For this example, we use informal graphs, but translating the latter into the formal syntax of $\mathcal{L}_c$ is straightforward. In this example, we use the shorthand $r_1 < 0$, $r_1 \leq 0$, and $r_1 = 0$ for $\{\, r_1 : \mathsf{int}_{<}(\mathsf{const}(0)) \,\}$, $\{\, r_1 : \mathsf{int}_{\leq}(\mathsf{const}(0)) \,\}$, and $\{\, r_1 : \mathsf{int}_{=}(\mathsf{const}(0)) \,\}$, respectively.

Assume we have two individual instructions, depicted in Figure 7. The first instruction increments register r_1 by one. If $r_1 < 0$ when entering this instruction, then $r_1 \leq 0$ after execution of the instruction. The second instruction is "`bnz` r_1, l", which has one entry but two exits. For the fall-through exit, we have $r_1 = 0$; when the branch is taken, we have $r_1 < 0$. Our goal is to combine these two instructions together to get a property of the two-instruction block. Notice that the two-instruction block is effectively a repeat-until loop: it keeps incrementing r_1 until r_1 is zero. For this repeat-until loop, we wish to prove that if $r_1 < 0$ before entering the block, then $r_1 = 0$ after completion of the block.

$$\boxed{F \,;\, \Psi_1 \vdash \Psi_2}$$

$$\frac{F_1 \,;\, \Psi_1' \vdash \Psi_1 \qquad F_2 \,;\, \Psi_2' \vdash \Psi_2}{F_1 \cup F_2 \,;\, \Psi_1' \cup \Psi_2' \vdash \Psi_1 \cup \Psi_2}\ \textsf{combine}$$

$$\frac{F \,;\, \Psi' \cup \{l \mathrel{-\!\triangleright} \phi\} \vdash \Psi \cup \{l \mathrel{-\!\triangleright} \phi\}}{F \,;\, \Psi' \vdash \Psi \cup \{l \mathrel{-\!\triangleright} \phi\}}\ \textsf{discharge}$$

$$\frac{\vdash \Psi_1' \Rightarrow \Psi_2' \qquad F \,;\, \Psi_2' \vdash \Psi_2 \qquad \vdash \Psi_2 \Rightarrow \Psi_1}{F \,;\, \Psi_1' \vdash \Psi_1}\ \textsf{weaken}$$

$$\boxed{\vdash \Psi_1 \Rightarrow \Psi_2}$$

$$\frac{m \geq n}{\vdash \{l_1 \mathrel{-\!\triangleright} \phi_1, \ldots, l_m \mathrel{-\!\triangleright} \phi_m\} \Rightarrow \{l_1 \mathrel{-\!\triangleright} \phi_1, \ldots, l_n \mathrel{-\!\triangleright} \phi_n\}}\ \textsf{s-width}$$

$$\frac{\phi' <: \phi}{\vdash \Psi \cup \{l \mathrel{-\!\triangleright} \phi\} \Rightarrow \Psi \cup \{l \mathrel{-\!\triangleright} \phi'\}}\ \textsf{s-depth}$$

Fig. 6. $\mathcal{L}_c$ composition rules

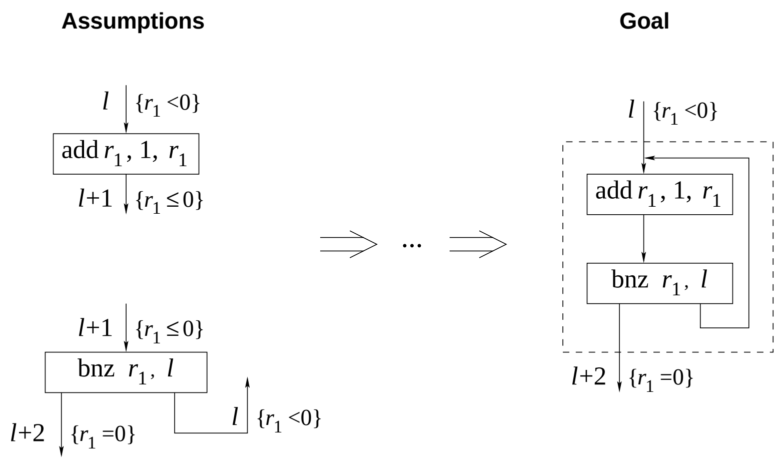


Fig. 7. An example to illustrate $\mathcal{L}_c$'s composition rules

The steps to derive the goal from the assumptions are presented in Figure 8. In step 1, we use the combine rule in Figure 6. When combining two fragments, F_1 and F_2, the combine rule joins the entries of F_1 and F_2; the same goes for the exits. For the example, since both instructions have only one entry, we end up with two entries after the combine rule. Since the first instruction has one exit, and the second instruction has two exits, we end up with three exits after the combine rule.

After application of the combine rule, we may end up with some address that is both an entry and an exit. For example, the address l after step 1 in Figure 8 is both an entry and an exit. Furthermore, the entry and the exit for l carry the same precondition, namely $r_1 < 0$. In this case, the discharge rule in Figure 6 can be used to eliminate the address l as an exit. Formally, the discharge rule states that if some $l \mathrel{-\!\triangleright} \phi$ appears on both the left and the right of the $\vdash$, then it can be removed from the left. Recall that exits are on the left, so this rule removes an

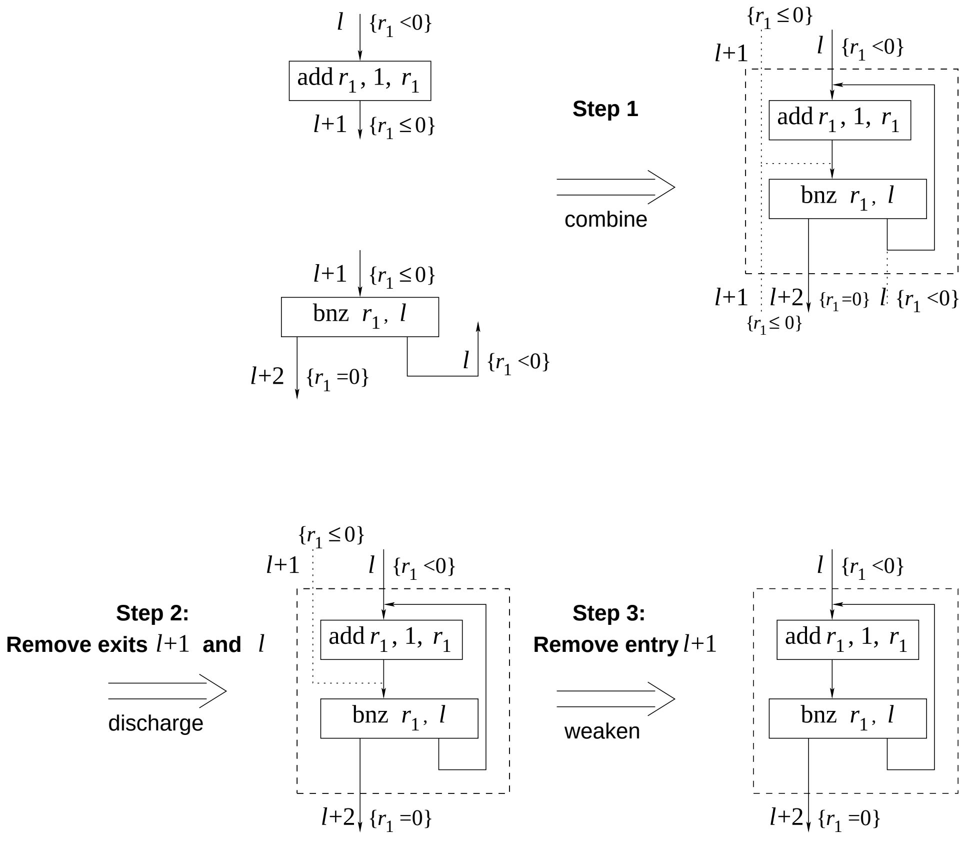


Fig. 8. The steps to derive the example in Figure 7

exit. At first sight, the discharge rule appears to be unsound. In the next section, we will present an index-based interpretation that justifies its soundness.

The address $l + 1$ is also both an entry and an exit, and the entry and the exit carry the same precondition. The discharge rule can remove $l + 1$ as an exit as well. Therefore, step 2 in Figure 8 is to apply the discharge rule twice to remove both l and $l + 1$ as exits. After this step, there is only one exit left.

In the last step, we remove $l + 1$ as an entry using the weaken rule. The weaken rule uses a relation between two sets of continuations:

$$\vdash \Psi_1 \Rightarrow \Psi_2,$$

which says that Ψ_1 is a stronger set of continuations than Ψ_2.

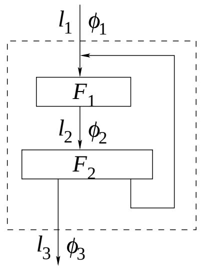
The rule s-width in Figure 6 states that a set of continuations is a stronger set than its subset. Therefore, $\vdash \{l + 1 \multimap (r_1 \le 0), l \multimap (r_1 < 0)\} \Rightarrow \{l \multimap (r_1 < 0)\}$ is derivable. Using this result and the weaken rule, step 3 in Figure 8 removes address $l + 1$ as an entry.

After these steps, we have one entry and one exit left for the repeat-until loop, and we have proved the desired property for the loop.

One natural question to ask is which addresses the logic should keep as entries. The example eliminates $l+1$ as an entry, while l remains. This is because it is the desired goal that tells us what the entries ought to be. In some other scenario we may want to keep $l+1$ as an entry as, for instance, in cases when other fragments need to jump to $l+1$. This is possible in unstructured control flow even though $l+1$ points to the middle of a loop. In general, the logic $\mathcal{L}_c$ itself does not dictate which entries to keep; that is the decision of the user of $\mathcal{L}_c$'s inference rules.

The example in Figure 8 has made use of all the composition rules with the exception of the s-depth rule. The s-depth rule states that a continuation with a weaker precondition is stronger than the continuation with a stronger precondition. The rule is contravariant in the preconditions. As an example, note that we can use this rule together with the weaken rule to derive $F\,;\,\{l \multimap \{r_1 : \mathsf{int}\}\} \vdash \Psi$ from $F\,;\,\{l \multimap \{r_1 : \mathsf{int}, r_2 : \mathsf{int}\}\} \vdash \Psi$.

Deriving Rules for Common Control-Flow Structures. We have shown a concrete example of using $\mathcal{L}_c$'s composition rules. In general, these composition rules can derive rules for common control-flow structures when these structures exist in machine code. To illustrate, next we derive a general rule for repeat-until loops. Suppose we have two program judgments. Fragment F_1 has one entry and one exit. Fragment F_2 has one entry, but two exits and one of the exits loops back to the entry of F_1. Therefore, when combined, these two fragments form a repeat-until loop. Below is the depiction of a repeat-until loop and the derivation of a general rule for repeat-until loops:



$$\cfrac{\cfrac{\cfrac{\cfrac{\begin{array}{c}F_1\,;\,\{l_2 \multimap \phi_2\} \vdash \{l_1 \multimap \phi_1\}\\ F_2\,;\,\{l_1 \multimap \phi_1, l_3 \multimap \phi_3\} \vdash \{l_2 \multimap \phi_2\}\end{array}}{F_1 \cup F_2\,;\,\{l_2 \multimap \phi_2, l_1 \multimap \phi_1, l_3 \multimap \phi_3\} \vdash \{l_1 \multimap \phi_1, l_2 \multimap \phi_2\}}\ \textsf{combine}}{F_1 \cup F_2\,;\,\{l_2 \multimap \phi_2, l_3 \multimap \phi_3\} \vdash \{l_1 \multimap \phi_1, l_2 \multimap \phi_2\}}\ \textsf{discharge}\ l_1}{F_1 \cup F_2\,;\,\{l_3 \multimap \phi_3\} \vdash \{l_1 \multimap \phi_1, l_2 \multimap \phi_2\}}\ \textsf{discharge}\ l_2}{F_1 \cup F_2\,;\,\{l_3 \multimap \phi_3\} \vdash \{l_1 \multimap \phi_1\}}\ \textsf{weaken}$$

In $\mathcal{L}_c$, we can analogously derive rules for many other control-flow structures, including sequential composition and while loops.

5.2 Semantics of $\mathcal{L}_c$

The semantics of $\mathcal{L}_c$ is centered on an interpretation of the judgment $F\,;\,\Psi' \vdash \Psi$. We have informally discussed its interpretation: for the set of fragments F, if Ψ' is true, then Ψ is true. If a continuation set Ψ is true, that means that it is safe to continue from any address in Ψ, provided that the associated precondition is true.

However, this naïve interpretation cannot justify the discharge rule. To see why, let us examine a special case of the discharge rule, where both Ψ' and Ψ are empty sets. Then the rule becomes

$$\frac{F\,;\,\{l \multimap \phi\} \vdash \{l \multimap \phi\}}{F\,;\,\{\,\} \vdash \{l \multimap \phi\}}$$

According to the naïve interpretation, the above rule is like stating that if "assuming $l \multimap \phi$ is a true continuation, we can derive $l \multimap \phi$ to be a true continuation", then "$l \multimap \phi$ is a true continuation". This is clearly unsound, because the premise is a tautology.

The problem is not that $\mathcal{L}_c$ is intrinsically unsound, but rather that the naïve interpretation has not captured the complete invariants implicit in $\mathcal{L}_c$. The interpretation that we must adopt is a stronger one. The idea is based on the notion of continuations being *approximately* true. The judgment $F\,;\,\Psi' \vdash \Psi$ is interpreted as Ψ being true at a higher approximation, by assuming the truth of Ψ' at a lower approximation. In this inductive interpretation, Ψ' and Ψ are treated differently, and it allows the discharge rule to be justified via induction.

Continuations Being Approximately True. We interpret the truth of $l \multimap \phi$ as that the address l is of type $\mathsf{codeptr}\,(\phi)$. The interpretation of code-pointer types in Section 4.6 is indexed by an approximation index k. By that definition, $l \multimap \phi$ being a true continuation in a state S and a store type Σ to approximation k means that the state is safe to execute for k steps. In other words, the state will not get stuck within k steps.

In a straightforward way, the semantics of a single continuation is then extended to a set of continuations. We define $\Sigma; S \models_k \Psi$ to mean that under a store type Σ and a state S, a continuation set Ψ is true to approximation k.

$$\Sigma; S \models_k \Psi \;\triangleq\; \forall (l \multimap \phi) \in \Psi.\; [\![\mathsf{codeptr}\,(\phi)]\!](\rho_\emptyset)(k, \Sigma, S, \mathsf{cv}(l))$$

In the definition, every $\mathsf{codeptr}\,(\phi)$ is applied to the empty environment $\rho_\emptyset$; we assume the precondition ϕ to be a closed type in this presentation. Our implementation also deals with open register-file types. Briefly, if ϕ is open and its type variables are collected in a type context Δ, then $\forall\Delta.\mathsf{codeptr}\,(\phi)$ will be closed. The type $\forall\Delta.\mathsf{codeptr}\,(\phi)$ can then be encoded and interpreted using TML polymorphic types. See Tan's thesis [Tan 2005] for details.

Loading Program Fragments. The predicate frag_loaded(F, S) describes the loading of a fragment set F into a state $S = \langle R,\, M\rangle$:

$$\text{frag\_loaded}(F, \langle R,\, M\rangle) \;\triangleq\; \forall i, l.\, (i@l) \in F \;\Rightarrow\; \mathsf{decode}(M(l), i)$$

Recall that $\mathsf{decode}(n, i)$ decodes a machine integer n into a machine instruction i. The decode relation was introduced in Section 2.

Semantics of the Judgment $F\,;\,\Psi' \vdash \Psi$. We define a relation, $F\,;\,\Psi' \models \Psi$, which is the semantic interpretation of $F\,;\,\Psi' \vdash \Psi$.

$$F\,;\,\Psi' \models \Psi \;\triangleq\; \forall S.\ \text{frag\_loaded}(F, S) \;\Rightarrow\; \forall k, \Sigma.\ \big(\Sigma; S \models_k \Psi' \;\Rightarrow\; \Sigma; S \models_{k+1} \Psi\big).$$

The definition quantifies over all states S such that F is loaded in the state. It derives the truth of Ψ to approximation $k+1$, from the truth of Ψ' to approximation k. In other words, if it is safe to continue from any of the addresses in Ψ', provided that the associated precondition is true for some number k of computation steps, then it is safe to continue from any of the addresses in Ψ, provided that the associated precondition is true for $k+1$ computation steps. This inductive definition allows the discharge rule to be proved by induction over k.

In order to prove the discharge rule, we have given $F\,;\,\Psi' \models \Psi$ a strong definition: we interpret continuations in Ψ at $(k+1)$-th approximation, not just k. But what about rules other than the discharge rule? Do they support such a strong semantics? The answer is yes, because of one implicit invariant—for any judgment $F\,;\,\Psi' \vdash \Psi$ that is derivable, it takes at least one computation step from addresses in Ψ before it will make use of the guarantees in Ψ'. Because of this invariant, despite the fact that it is safe to continue from exits for only k steps, we can still show that it is safe to continue from entries for $k+1$ steps.

Finally, since $\mathcal{L}_c$ also contains rules for deriving $\vdash \Psi \Rightarrow \Psi'$, we present its semantics below to complete the presentation.

$$\models \Psi \Rightarrow \Psi' \ \triangleq\ \forall \Sigma, S, k.\ (\Sigma; S \models_k \Psi) \Rightarrow (\Sigma; S \models_k \Psi')$$

The relation $\phi <: \phi'$ is the TML subtyping relation, and we introduced its semantics in Section 4.11.

Soundness and Type Safety. Based on the semantics we have developed, we next present soundness and type-safety theorems for $\mathcal{L}_c$.

THEOREM 5.1 (SOUNDNESS). *If $F\,;\,\Psi' \vdash \Psi$, then $F\,;\,\Psi' \models \Psi$.*

The proof is by induction on the derivation $F\,;\,\Psi' \vdash \Psi$. The most interesting case is the proof of the discharge rule, which is shown below.

Lemma 5.2 (SOUNDNESS OF discharge RULE).
If $F\,;\,\Psi' \cup \{l \multimap\!\triangleright\, \phi\} \models \Psi \cup \{l \multimap\!\triangleright\, \phi\}$, then $F\,;\,\Psi' \models \Psi \cup \{l \multimap\!\triangleright\, \phi\}$.

PROOF. For all S, assuming

$$\text{frag\_loaded}(F, S), \tag{5.2.1}$$

we need to show $\forall k, \Sigma.\ \big(\Sigma; S \models_k \Psi' \ \Rightarrow \Sigma; S \models_{k+1} \Psi \cup \{l \multimap\!\triangleright\, \phi\}\big)$. We prove this by induction on the natural number k.

For the base case, assume $\Sigma; S \models_0 \Psi'$ and show that $\Sigma; S \models_1 \Psi \cup \{l \multimap\!\triangleright\, \phi\}$, for any Σ.

By the definition of $[\![\mathsf{codeptr}\,(\phi)]\!]$, it is trivial to show $\Sigma; S \models_0 \{l \multimap\!\triangleright\, \phi\}$. Together with the assumption $\Sigma; S \models_0 \Psi'$, we have

$$\Sigma; S \models_0 \Psi' \cup \{l \multimap\!\triangleright\, \phi\} \tag{5.2.2}$$

Now from $F\,;\,\Psi' \cup \{l \multimap\!\triangleright\, \phi\} \models \Psi \cup \{l \multimap\!\triangleright\, \phi\}$, the result 5.2.1, and the result 5.2.2, we get $\Sigma; S \models_1 \Psi \cup \{l \multimap\!\triangleright\, \phi\}$, which is what we needed to show for the base case.

For the inductive case, assume

$$\forall \Sigma.\ \big(\Sigma; S \models_k \Psi' \ \Rightarrow \Sigma; S \models_{k+1} \Psi \cup \{l \multimap\!\triangleright\, \phi\}\big) \tag{5.2.3}$$

We must show that

$$\forall\Sigma.\ \big(\Sigma; S \models_{k+1} \Psi' \ \Rightarrow \Sigma; S \models_{k+2} \Psi \cup \{l \multimap \phi\}\big).$$

Thus, assume

$$\Sigma; S \models_{k+1} \Psi'. \tag{5.2.4}$$

Since $[\![\mathsf{codeptr}\,(\phi)]\!]$ is downward closed in terms of the index k, it is easy to show

$$\Sigma; S \models_{k} \Psi'. \tag{5.2.5}$$

From the induction hypothesis 5.2.3 and the result 5.2.5, we have

$$\Sigma; S \models_{k+1} \Psi \cup \{l \multimap \phi\}, \tag{5.2.6}$$

from which we have

$$\Sigma; S \models_{k+1} \{l \multimap \phi\}. \tag{5.2.7}$$

Together with 5.2.4, we have

$$\Sigma; S \models_{k+1} \Psi' \cup \{l \multimap \phi\}. \tag{5.2.8}$$

Now, use the assumption $F\,;\,\Psi' \cup \{l \multimap \phi\} \models \Psi \cup \{l \multimap \phi\}$, 5.2.1 and 5.2.8, to derive $\Sigma; S \models_{k+2} \Psi \cup \{l \multimap \phi\}$, which is what we needed to show for the inductive case. □

For type safety, we need to show that if a program type checks in $\mathcal{L}_c$ then it is safe—i.e., it will not get stuck. To present the formal theorem, we must first introduce a definition.

Definition 5.3. Given machine code $C = \{\, l_1 \mapsto n_1, \ldots, l_k \mapsto n_k \,\}$, we use $\mathsf{frag}(C)$ for the corresponding program fragment, that is,

$$\mathsf{frag}(C) \ \triangleq\ \{i_1@l_1, \ldots, i_k@l_k\},$$

where $\mathsf{decode}(n_j, i_j)$ holds for $1 \le j \le k$.

Note that $\mathsf{frag}(-)$ is a partial function since for some numbers n, there is no instruction i such that $\mathsf{decode}(n, i)$.

Theorem 5.4 (Type Safety of $\mathcal{L}_c$).
If $\mathsf{frag}(C)\,;\,\{\,\} \models \{\mathit{startLoc} \multimap \mathsf{top}\}$*, then* $\mathrm{safe\_code}(C, \mathit{startLoc})$.

Or, for the fragment $\mathsf{frag}(C)$, if our logic can derive that *startLoc* is a safe address with precondition top, under no assumptions about the exits, then the machine program C is a safe program.

Proof(Sketch). By the definition of $\mathrm{safe\_code}(C, \mathit{startLoc})$ (Definition 2.1 on page 7), we need to prove that $\mathrm{safe\_state}(\langle R,\, M\rangle, k)$ for any k, R, and M, assuming

$$\mathsf{loaded}(C, \langle R,\, M\rangle), \qquad R(\mathsf{pc}) = \mathit{startLoc}, \qquad \mathrm{init\_cond}(\langle R,\, M\rangle).$$

We use Lemma 4.5 (page 25) to prove the goal.

For the initial state, the store type is the map containing only the loaded program C, called Σ_C.[12] From $\mathsf{loaded}(C, \langle R,\, M\rangle)$, we can prove $\forall k.\ \vdash \langle R,\, M\rangle :_k \Sigma_C$.

[12]When $C = \{\, l_1 \mapsto n_1, \ldots, l_k \mapsto n_k \,\}$, the initial store type Σ_C is defined as follows:

$$\Sigma_C = \{\, l_1 \mapsto \mathsf{box}\,(\mathsf{int}_=(\mathsf{const}(n_1))), \ldots, l_k \mapsto \mathsf{box}\,(\mathsf{int}_=(\mathsf{const}(n_k)))\,\}$$

$$
\begin{array}{rrcl}
\textit{(programs)} & P & ::= & B; P \mid B \\
\textit{(instruction blocks)} & B & ::= & i; B \mid \texttt{goto}\ l \mid \texttt{bz}\ r_s, l \mid \texttt{jmp}\ r_d \\
\textit{(instructions)} & i & ::= & \texttt{add}\ r_d, r_s, n \mid \texttt{ld}\ r_d, r_s[n] \mid \texttt{st}\ r_d[n], r_s \\
\\
\textit{(code specification)} & \Psi & ::= & \{\, l_1 : \mathsf{codeptr}\,(\phi_1),\ \ldots,\ l_n : \mathsf{codeptr}\,(\phi_n)\,\} \\
\textit{(register-file types)} & \phi & ::= & \{\, r_1 : t_1, \ldots, r_n : t_n \,\} \\
\textit{(types)} & t & ::= & \mathsf{int} \mid \mathsf{codeptr}\,(\phi) \mid \mathsf{list}(t) \mid \mathsf{listcons}(t) \mid \mathsf{listnil}
\end{array}
$$

Fig. 9. TAL$_0$: Syntax

We have not presented definitions of init_cond$(-)$ and valid_state$(-,-)$, but it suffices to say that the definition of init_cond$(\langle R,\ M\rangle)$ is strong enough to prove valid_state$(\langle R,\ M\rangle, \Sigma_C)$.

By the definition of $\mathsf{frag}(C)\,;\,\{\,\}\ \models\ \{\mathit{startLoc} \multimapdotbothB \mathsf{top}\}$, we have that

$$\forall k.\ [\![\mathsf{codeptr}\,(\mathsf{top})]\!](\rho_\emptyset)(k, \Sigma_C, \langle R,\ M\rangle, \mathsf{cv}(\mathit{startLoc})).$$

Now using Lemma 4.5, we have $\forall k.$ safe_state$(\langle R,\ M\rangle, k)$. □

6. MODELING TYPED ASSEMBLY LANGUAGES

We have introduced an intermediate layer with two key abstractions: TML and $\mathcal{L}_c$. This layer can serve as a semantic foundation for a variety of typed assembly languages. To illustrate how TML and $\mathcal{L}_c$ can be leveraged to prove the type soundness of a TAL, in Section 6.1 we introduce a tiny TAL, dubbed TAL$_0$, and explain how TML and $\mathcal{L}_c$ can be used to model the data structures and control flow of this tiny TAL. In Section 6.2, we discuss some of the salient issues that arose when modeling LTAL, the low-level typed assembly language used in our FPCC system.

6.1 Modeling TAL$_0$

TAL$_0$ is a simple typed assembly language that manipulates lists. Specifically, TAL$_0$ can perform list updates, but it cannot perform list allocation and initialization (a point that we will return to when we discuss memory allocation in Section 6.2).

Figure 9 presents the syntax of TAL$_0$. We assume that the underlying machine is the same as the one in Section 2. A TAL$_0$ program consists of an assembly program P and a code specification Ψ. An assembly program P consists of a sequence of instruction blocks, each of which is a sequence of instructions ending in a control-transfer instruction. The code specification Ψ is a mapping from addresses to code-pointer types, which take register-file types as preconditions. Note that the notation $\{\, l : \mathsf{codeptr}\,(\phi)\,\}$ is essentially the same as $\{l \multimapdotbothB \phi\}$ in $\mathcal{L}_c$. The domain of a well-formed Ψ contains exactly those addresses that correspond to the beginnings of instruction blocks. Therefore, it effectively specifies a precondition for the beginning of every instruction block.

The types of TAL$_0$ include an integer type int, and a code-pointer type $\mathsf{codeptr}\,(\phi)$. We use code-pointer types to handle indirect jumps. To manipulate lists, TAL$_0$ has a list type $\mathsf{list}(t)$. The type $\mathsf{listcons}(t)$ contains those lists that have at least one cons cell, i.e., nonempty lists. The type $\mathsf{listnil}$ is the type ascribed to empty lists.

That is, the initial store type uses immutable-reference types to reflect the presence of C.

We will use the notation $|B|$ to denote the size of an instruction block B. Since each instruction on the imaginary machine is of size one, $|B|$ equals the number of instructions in B.

Type System. The type system of TAL_0 has judgments for programs, for instruction blocks, for instructions, and for subtyping:

—The judgment $\vdash_{\mathtt{p}} P : \Psi$ means that the program P is well formed with respect to the code specification Ψ.

—The judgment $\Psi; l \vdash_{\mathtt{f}} P$ means that the program fragment P, starting at address l, is well formed, assuming the global code specification Ψ. The code specification Ψ provides the preconditions associated with addresses to which P might jump.

—The judgment $\Psi; l; \phi \vdash_{\mathtt{b}} B$ means that the instruction block B, starting at address l, is well formed, assuming the precondition of B is ϕ and assuming the global code specification Ψ. The code specification Ψ provides the preconditions associated with addresses to which B might jump.

—The judgment $\vdash_{\mathtt{i}} \{\phi_1\}i\{\phi_2\}$ means that the instruction i is well formed with respect to the precondition ϕ_1 and the postcondition ϕ_2.

—The judgments $t_1 <: t_2$ and $\phi_1 <: \phi_2$ are the subtyping judgments between types and register-file types, respectively.

Figure 10 presents the type system of TAL_0. To check that a program P is well formed with respect to a code specification Ψ, the prog rule invokes $\Psi; 0 \vdash_{\mathtt{f}} P$. Then, the type system uses the frag1 and frag2 rules to check that each instruction block in P is well formed. The rules frag1 and frag2 look up the precondition of each block inside the global code specification Ψ. These two rules also make sure that Ψ is well formed by checking that Ψ associates preconditions with only those addresses that correspond to the beginning of instruction blocks. (Preconditions for other addresses are computed by the type system.)

When checking an instruction block B, the type system uses the seq rule to walk through the block to check that every instruction is well formed. The goto, jmp, and bz rules check the last instruction in a block. The goto rule checks that the current precondition, ϕ, is a subtype of the precondition of the destination address (which must be in the domain of Ψ).

The rule for "$\mathtt{jmp}\ r_d$" needs some explanation. Before we can jump to the register r_d, the rule requires that r_d be of a code-pointer type that takes the current register-file type ϕ as the precondition. As an example, suppose the instruction is "$\mathtt{jmp}\ \mathrm{r}_1$", and suppose the precondition is:

$$\phi = \{\, \mathrm{r}_1 : \mathsf{codeptr}\,(\{\, \mathrm{r}_2 : \mathsf{int}\,\}),\ \mathrm{r}_2 : \mathsf{int}\,\}.$$

This precondition requires r_1 to be a code pointer that the program can jump to when r_2 is of integer type. Intuitively, "$\mathtt{jmp}\ \mathrm{r}_1$" is a safe jump because $\{\, \mathrm{r}_2 : \mathsf{int}\,\}$ satisfies the precondition of the destination address r_1. Using the subtyping rules in Figure 10, we can derive $\phi <: \{\, \mathrm{r}_1 : \mathsf{codeptr}\,(\phi)\,\}$ as follows.

$\boxed{\vdash_{\mathtt{p}} P : \Psi}$

$$\frac{\Psi; 0 \vdash_{\mathtt{f}} P}{\vdash_{\mathtt{p}} P : \Psi}\ \textsf{prog}$$

$\boxed{\Psi; l \vdash_{\mathtt{f}} P}$

$$\frac{\begin{array}{c}\Psi(l) = \textsf{codeptr}\,(\phi) \quad \forall l < x < l + |B|.\ x \notin \mathrm{dom}(\Psi)\\ \Psi; l; \phi \vdash_{\mathtt{b}} B \quad \Psi; l + |B| \vdash_{\mathtt{f}} P\end{array}}{\Psi; l \vdash_{\mathtt{f}} (B; P)}\ \textsf{frag1}$$

$$\frac{\Psi(l) = \textsf{codeptr}\,(\phi) \qquad \forall x > l.\ x \notin \mathrm{dom}(\Psi) \qquad \Psi; l; \phi \vdash_{\mathtt{b}} B}{\Psi; l \vdash_{\mathtt{f}} B}\ \textsf{frag2}$$

$\boxed{\Psi; l; \phi \vdash_{\mathtt{b}} B}$

$$\frac{\vdash_{\mathtt{i}} \{\phi_1\} i \{\phi_2\} \qquad \Psi; l + 1; \phi_2 \vdash_{\mathtt{b}} B}{\Psi; l; \phi_1 \vdash_{\mathtt{b}} (i; B)}\ \textsf{seq}$$

$$\frac{\Psi(l_d) = \textsf{codeptr}\,(\phi_d) \qquad \phi <: \phi_d}{\Psi; l; \phi \vdash_{\mathtt{b}} (\mathtt{goto}\ l_d)}\ \textsf{goto}$$

$$\frac{\phi <: \{\, r_d : \textsf{codeptr}\,(\phi)\,\}}{\Psi; l; \phi \vdash_{\mathtt{b}} (\mathtt{jmp}\ r_d)}\ \textsf{jmp}$$

$$\frac{\begin{array}{c}\Psi(l_d) = \textsf{codeptr}\,(\phi_d) \quad \Psi(l+1) = \phi'\\ \phi <: \{\, r_s : \textsf{list}(t)\,\} \quad \phi[r_s \mapsto \textsf{listnil}] <: \phi_d \quad \phi[r_s \mapsto \textsf{listcons}(t)] <: \phi'\end{array}}{\Psi; l; \phi \vdash_{\mathtt{b}} (\mathtt{bz}\ r_s, l_d)}\ \textsf{bz}$$

$\boxed{\vdash_{\mathtt{i}} \{\phi_1\} i \{\phi_2\}}$

$$\frac{\phi <: \{\, r_s : \textsf{int}\,\} \qquad \phi[r_d \mapsto \textsf{int}] = \phi'}{\vdash_{\mathtt{i}} \{\phi\}(\mathtt{add}\ r_d, r_s, n)\{\phi'\}}\ \textsf{add} \qquad \frac{\phi <: \{\, r_d : \textsf{listcons}(t), r_s : t\,\}}{\vdash_{\mathtt{i}} \{\phi\}(\mathtt{st}\ r_d[0], r_s)\{\phi\}}\ \textsf{st-upd}$$

$$\frac{\begin{array}{c}\phi <: \{\, r_s : \textsf{listcons}(t)\,\}\\ \phi[r_d \mapsto t] = \phi'\end{array}}{\vdash_{\mathtt{i}} \{\phi\}(\mathtt{ld}\ r_d, r_s[0])\{\phi'\}}\ \textsf{ld0} \qquad \frac{\begin{array}{c}\phi <: \{\, r_s : \textsf{listcons}(t)\,\}\\ \phi[r_d \mapsto \textsf{list}(t)] = \phi'\end{array}}{\vdash_{\mathtt{i}} \{\phi\}(\mathtt{ld}\ r_d, r_s[1])\{\phi'\}}\ \textsf{ld1}$$

$\boxed{t_1 <: t_2,\ \phi_1 <: \phi_2}$

$$\frac{}{t <: t}\ \textsf{s-refl} \qquad \frac{\phi_2 <: \phi_1}{\textsf{codeptr}\,(\phi_1) <: \textsf{codeptr}\,(\phi_2)}\ \textsf{s-cptr}$$

$$\frac{}{\textsf{listnil} <: \textsf{list}(t)}\ \textsf{s-nil} \qquad \frac{}{\textsf{listcons}(t) <: \textsf{list}(t)}\ \textsf{s-cons}$$

$$\frac{\mathrm{dom}(\phi_2) \subseteq \mathrm{dom}(\phi_1) \qquad \forall r \in \mathrm{dom}(\phi_2).\ \phi_1(r) <: \phi_2(r)}{\phi_1 <: \phi_2}\ \textsf{s-rfile}$$

Fig. 10. TAL$_0$: Type system

$$\cfrac{\cfrac{\cfrac{}{\mathsf{int} <: \mathsf{int}}\ \textsf{s-refl}}{\{\,\mathrm{r}_1 : \mathsf{codeptr}\,(\{\,\mathrm{r}_2 : \mathsf{int}\,\}),\ \mathrm{r}_2 : \mathsf{int}\,\} <: \{\,\mathrm{r}_2 : \mathsf{int}\,\}}\ \textsf{s-rfile}}{\cfrac{\mathsf{codeptr}\,(\{\,\mathrm{r}_2 : \mathsf{int}\,\}) <: \mathsf{codeptr}\,(\{\,\mathrm{r}_1 : \mathsf{codeptr}\,(\{\,\mathrm{r}_2 : \mathsf{int}\,\}),\ \mathrm{r}_2 : \mathsf{int}\,\})}{\begin{array}{l}\{\,\mathrm{r}_1 : \mathsf{codeptr}\,(\{\,\mathrm{r}_2 : \mathsf{int}\,\}),\ \mathrm{r}_2 : \mathsf{int}\,\} \\ \quad <: \{\,\mathrm{r}_1 : \mathsf{codeptr}\,(\{\,\mathrm{r}_1 : \mathsf{codeptr}\,(\{\,\mathrm{r}_2 : \mathsf{int}\,\}),\ \mathrm{r}_2 : \mathsf{int}\,\})\,\}\end{array}}\ \textsf{s-rfile}}\ \textsf{s-cptr}$$

Since TAL_0 manipulates lists, the bz rule is specialized for the case when the source register r_s is of type $\mathsf{list}(t)$. TAL_0 implicitly assumes an untagged representation for lists: use value 0 for the empty list, and a nonzero pointer to a record of two fields for a cons cell; the first field is the data, and the second is the tail of the list. Therefore, in the typing rule for bz, when the branch is taken, we know that the list is an empty list, and we update r_s to be of type listnil. For the fall-through case, the list is nonempty, and r_s has type $\mathsf{listcons}(t)$.

The rules for $\vdash_{\mathtt{i}} \{\phi_1\}i\{\phi_2\}$ take a precondition ϕ_1 and an instruction i as inputs and calculate a postcondition ϕ_2 as an output. In the add rule, the postcondition is the precondition with the destination register updated with the int type. The st-upd rule deals with the case of updating the contents of the head of a list. The list must be nonempty; hence, the rule st-upd requires a $\mathsf{listcons}(t)$ type. The ld0 (ld1) rule deals with the case of loading data (the tail) in a list.

To demonstrate TML's expressive power in terms of encoding TAL_0 types, we have arbitrarily chosen to make the heads of lists mutable and the tails immutable. This is why we have two ld rules, but only one st-upd rule. We could easily make it the other way around, or make both mutable, or make both immutable.

As an example, consider the following TAL_0 program which takes an integer list as input and adds one to every integer in the list. For the reader's convenience, we put type specifications in front of every instruction, although TAL_0 only needs specifications at the beginnings of basic blocks.

```
begin :     { r1 : list(int) }
        bz r1, end
            { r1 : listcons(int) }
        ld r2, r1[0]
            { r1 : listcons(int), r2 : int }
        add r2, r2, 1
            { r1 : listcons(int), r2 : int }
        st r1[0], r2
            { r1 : listcons(int), r2 : int }
        ld r1, r1[1]
            { r1 : list(int) }
        goto begin
  end :     { }
        goto end
```

Modeling Data Types in TAL_0. We model TAL_0's data types using TML type constructors. Intuitively, a $\mathsf{list}(t)$ type in TAL_0 has two cases, either an empty list,

$$\begin{array}{lcl}
\models_{\mathtt{p}} P : \Psi & \triangleq & P@0; \{\,\} \models_{\mathcal{L}_c} \Psi \\
\Psi; l \models_{\mathtt{f}} P & \triangleq & P@l; \Psi \models_{\mathcal{L}_c} \Psi|_{\geq l} \\
\Psi; l; \phi \models_{\mathtt{b}} B & \triangleq & B@l; \Psi \models_{\mathcal{L}_c} \{l : \mathsf{codeptr}\,(\phi)\} \\
\models_{\mathtt{i}} \{\phi_1\} i \{\phi_2\} & \triangleq & \forall l.\ i@l; \{l + |i| : \mathsf{codeptr}\,(\phi_2)\} \models_{\mathcal{L}_c} \{l : \mathsf{codeptr}\,(\phi_1)\}
\end{array}$$

Fig. 11. TAL$_0$: Semantics of typing judgments

or a nonempty list with a head and a tail. Since TAL$_0$ uses an untagged representation, an empty list should be represented as integer zero, while a nonempty list is a nonzero pointer to a record that has the data as the first field and the tail as the second field. Therefore, we encode the list types in TAL$_0$ as follows:

$$\begin{array}{l}
\mathsf{list}(t) \quad \triangleq \\
\quad \mathsf{rec}\Big(\mathsf{const}(0) \cup \big(\mathsf{int}_{\neq}(\mathsf{const}(0)) \cap \mathsf{offset}(\mathsf{const}(0), \mathsf{ref}\,(t)) \cap \mathsf{offset}(\mathsf{const}(1), \mathsf{box}\,(\underline{0}))\big)\Big)
\end{array}$$

$$\begin{array}{lcl}
\mathsf{listnil} & \triangleq & \mathsf{const}(0) \\
\mathsf{listcons}(t) & \triangleq & \mathsf{int}_{\neq}(\mathsf{const}(0)) \cap \mathsf{offset}(\mathsf{const}(0), \mathsf{ref}\,(t)) \cap \mathsf{offset}(\mathsf{const}(1), \mathsf{box}\,(\mathsf{list}(t)))
\end{array}$$

We defined the type $\mathsf{offset}(t_1, t_2)$ in Section 3.2. Intuitively, a value n_1 has type $\mathsf{offset}(\mathsf{const}(n_2), t)$ if and only if $n_1 + n_2$ has type t.

TAL$_0$ also has type int, $\mathsf{codeptr}\,(\phi)$, and subtyping. We model these directly using their TML counterparts—that is why we have deliberately used the same syntax for int, $\mathsf{codeptr}\,(\phi)$, and subtyping.

With these semantics based on TML, the subtyping rules in TAL$_0$ can be easily proved based on the subtyping rules in TML. For example, the subtyping rule $\mathsf{listcons}(t)$ <: $\mathsf{list}(t)$ can be proved by using the fold-unfold subtyping lemma of recursive types in TML. As we can see, the TML layer provides a nice separation between low-level machine details and high-level types. When we model a new TAL, many proofs about types are easily reused.

Modeling Control Flow in TAL$_0$. We next model TAL$_0$'s control flow based on $\mathcal{L}_c$. We proceed in two steps. First, we develop models for TAL$_0$ judgments based on $\mathcal{L}_c$'s instruction judgment; we will use $F; \Psi' \models_{\mathcal{L}_c} \Psi$ for $\mathcal{L}_c$'s instruction judgment to avoid confusion. Next, we demonstrate how the TAL$_0$ rules and its type safety theorem can be proved from the rules in $\mathcal{L}_c$.

Let us first introduce some notation. In TAL$_0$, an instruction block B denotes a list of consecutive instructions. We will write $B@l$ to mean that B is at address l. When $B = i_0; \ldots; i_n$, the notation $B@l$ is an abbreviation for $\{\, i_0@l, \ldots, i_n@(l + n)\,\}$. Similarly, we will write $P@l$ to denote that the program P is at the address l.

Figure 11 presents semantics of judgments in TAL$_0$'s type system, based on $F; \Psi' \models_{\mathcal{L}_c} \Psi$. The first judgment is $\vdash_{\mathtt{p}} P : \Psi$, which we model as $P@0; \{\,\} \models_{\mathcal{L}_c} \Psi$. The program P has multiple entries: the beginnings of instruction blocks in P are possible entries; these beginnings correspond to the domain of Ψ. Furthermore, since P is the complete program, it does not depend on other exits and thus has no exits (except for some possible indirect exits in registers). Therefore, the semantics of $\vdash_{\mathtt{p}} P : \Psi$ in Figure 11 says that every address in the domain of Ψ is a code pointer with respect to the corresponding precondition prescribed in Ψ.

In the judgment $\Psi; l \vdash_{\mathtt{f}} P$, the P part is not a complete program, but only a partial program starting at the address l, and thus its entries include the beginnings of only those instruction blocks in P; these beginnings correspond to those addresses that are not less than l in the domain of Ψ. We use the notation $\Psi|_{\geq l}$ to denote the restriction of Ψ to those addresses greater than or equal to l. Meanwhile, P may jump outside of P to any instruction block in the complete program. Therefore, the semantics of $\Psi; l \vdash_{\mathtt{f}} P$ in Figure 11 says that every address in the domain of $\Psi|_{\geq l}$ is a code pointer, assuming the global code specification Ψ.

In the judgment $\Psi; l; \phi \vdash_{\mathtt{b}} B$, the block B has only one entry, namely l. But the last instruction in B may jump to any other block in the complete program. Therefore, the semantics of $\Psi; l; \phi \vdash_{\mathtt{b}} B$ in Figure 11 says that the address l is a code pointer with the precondition ϕ, assuming the global code specification Ψ.

In the judgment $\vdash_{\mathtt{i}} \{\phi_1\}i\{\phi_2\}$, the instruction i is not a control-transfer instruction and therefore has one entry and one exit. Its semantics in Figure 11 states that the address l is a code pointer with the precondition ϕ_1, assuming that the address $l + |i|$ is a code pointer with precondition ϕ_2.

Next, we show how the soundness of TAL_0's composition rules (prog, frag1, frag2 and seq) follows from $\mathcal{L}_c$'s rules. The two interesting cases are the prog and the seq rules, whose proofs are presented below.

Lemma 6.1 Soundness of prog Rule. If $\Psi; 0 \models_{\mathtt{f}} P$, then $\models_{\mathtt{p}} P : \Psi$.

Proof. From the definitions in Figure 11, we need to prove $P@0; \{\ \} \models_{\mathcal{L}_c} \Psi$, by assuming $P@0; \Psi \models_{\mathcal{L}_c} \Psi|_{\geq 0}$. Since the domain of Ψ is a subset of the natural numbers, we have $\Psi|_{\geq 0} = \Psi$, and thus

$$P@0; \Psi \models_{\mathcal{L}_c} \Psi.$$

Note that every $\{\, l : \mathsf{codeptr}\,(\phi)\,\}$ in Ψ appears on both the left and the right in the above judgment. Because $\mathrm{dom}(\Psi)$ is finite, we use the discharge rule of $\mathcal{L}_c$ multiple times to remove all continuations from the left of the judgment, and then get

$$P@0; \{\ \} \models_{\mathcal{L}_c} \Psi.$$

□

Lemma 6.2 Soundness of seq Rule. If $\models_{\mathtt{i}} \{\phi_1\}i\{\phi_2\}$, and $\Psi; l+1; \phi_2 \models_{\mathtt{b}} B$, then $\Psi; l; \phi_1 \models_{\mathtt{b}} i; B$.

Proof. From the definitions in Figure 11, we need to prove

$$(i; B)@l; \Psi \models_{\mathcal{L}_c} \{\, l : \mathsf{codeptr}\,(\phi_1)\,\},$$

by assuming

$$\forall l.\ i@l; \{\, l + |i| : \mathsf{codeptr}\,(\phi_2)\,\} \models_{\mathcal{L}_c} \{\, l : \mathsf{codeptr}\,(\phi_1)\,\} \quad (1)$$

$$B@(l+1); \Psi \models_{\mathcal{L}_c} \{\, l+1 : \mathsf{codeptr}\,(\phi_2)\,\} \quad (2)$$

Performing a universal elimination on (1) using the address l, and considering that $|i| = 1$, we have

$$i@l; \{\, l+1 : \mathsf{codeptr}\,(\phi_2)\,\} \models_{\mathcal{L}_c} \{\, l : \mathsf{codeptr}\,(\phi_1)\,\} \quad (3)$$

Using the combine rule in $\mathcal{L}_c$ on (3) and (2), and also taking into account that $i@l$ together with $B@(l+1)$ is the same as $(i;B)@l$, we get

$$(i;B)@l; \Psi \cup \{\, l+1 : \mathsf{codeptr}\,(\phi_2)\,\} \models_{\mathcal{L}_c} \{\, l : \mathsf{codeptr}\,(\phi_1), l+1 : \mathsf{codeptr}\,(\phi_2)\,\}. \tag{4}$$

Since $\{\, l+1 : \mathsf{codeptr}\,(\phi_2)\,\}$ appears both on the left and on the right of the above judgment, we use the discharge rule to remove it from the left:

$$(i;B)@l; \Psi \models_{\mathcal{L}_c} \{\, l : \mathsf{codeptr}\,(\phi_1), l+1 : \mathsf{codeptr}\,(\phi_2)\,\}. \tag{5}$$

Finally, we use the weaken rule to remove $\{\, l+1 : \mathsf{codeptr}\,(\phi_2)\,\}$ from the right of the judgment to prove our goal:

$$(i;B)@l; \Psi \models_{\mathcal{L}_c} \{\, l : \mathsf{codeptr}\,(\phi_1)\,\} \tag{6}$$

□

TAL$_0$ also contains rules for individual instructions, such as the add and ld0 rules. These rules are proved based on the operational semantics of the instructions, the definition of $F; \Psi' \models_{\mathcal{L}_c} \Psi$, and the model of TAL$_0$ types. We have proved many generic typing lemmas for machine instructions; these lemmas are useful for proving the specific typing rules for instructions in a TAL. We do not elaborate on these here, but refer readers to a Ph.D. dissertation [Tan 2005, Chapter 4].

Finally, we present the type-safety theorem of TAL$_0$, which is a direct result of our semantic encoding of "well-formed programs".

THEOREM 6.3 (TYPE SAFETY OF TAL$_0$). *If (1) the assembly-language program P type-checks with type Ψ, (2) the machine-language program C corresponds to the assembly-language program P, and (3) the initial entry precondition in Ψ for address startLoc is* true*, then it is safe to execute program C from address startLoc. That is, if (1) $\models_{\mathsf{p}} P : \Psi$, (2) $P@0 = \mathsf{frag}(C)$, and (3) $\{\mathit{startLoc} \multimap \mathsf{top}\} \in \Psi$, then* safe_code$(C, \mathit{startLoc})$.

PROOF. From the semantics of $\models_{\mathsf{p}} P : \Psi$, we have $P@0; \{\,\} \models_{\mathcal{L}_c} \Psi$. By assumption (2) and (3), we have

$$\mathsf{frag}(C); \{\,\} \models_{\mathcal{L}_c} \{\mathit{startLoc} \multimap \mathsf{top}\}.$$

Now we apply Theorem 5.4 to get safe_code$(C, \mathit{startLoc})$. □

6.2 Modeling LTAL

We have shown above how to model a simple typed assembly language TAL$_0$. The real typed assembly language in our system, LTAL, is much more feature-laden and complex. But still, the layer of TML and $\mathcal{L}_c$ is invaluable in modeling LTAL. In this section, we briefly sketch some important issues. Interested readers can find more details in Tan et al. [2004] and Tan [2005].

LTAL's type system is much more expressive than that of TAL$_0$. It contains polymorphic types, existential types, recursive types, and many others. All these types can be easily encoded using TML. The type refinement rules and substitution rules in LTAL are simply lemmas (or combinations of lemmas) provided by TML and are easy to prove.

Virtual Instructions. While encoding LTAL data structures using TML is straightforward, encoding control flow using $\mathcal{L}_c$ is more problematic due to the presence of *virtual instructions.* To accommodate recursive types (and also polymorphic and existential types) while still maintaining syntax-directed type checking, LTAL has virtual instructions that do not correspond to real machine instructions. For example, LTAL has an instruction that unfolds a recursive type. These virtual instructions only manipulate types, and perform no real computation. Their purpose is to guide the type-checking process.

These virtual instructions create problems for our control logic $\mathcal{L}_c$. This is due to the fact that the semantics of $\mathcal{L}_c$ relies on the one-step invariant: the program in question must take at least one computation step before it can make use of guarantees about the exits. A virtual instruction, however, takes no computation steps. Our solution to this mismatch is based on the following intuition. As long as the virtual instruction is followed by a real instruction, then it takes at least one computation step to reach an exit for *the whole block.* Thus, intuitively, the virtual instruction can borrow (or piggyback on) the execution of the next real instruction. Based on this intuition, our semantics of the LTAL instruction judgment has two cases. If the instruction is a real instruction, then there is a one-step requirement; if it is a virtual instruction, we only require that the precondition of the instruction be a subtype of the postcondition. Based on this semantics, when a virtual instruction is followed by a real instruction, we can recover the one-step invariant. Another way of saying this is that we type check virtual instructions by TML subtyping. We refer the reader to Tan [2005, chapter 3.4] for details.

Memory Allocation. The TAL_0 we considered above supports list updates but not list allocation and initialization. LTAL, in contrast, supports both allocation and initialization of data structures. In her thesis, Chen [2004, Chapter 4.4] explains the LTAL typing rules for heap allocation and initialization in great detail. Here we will explain the semantics of those rules.

For memory allocation and other purposes, we have extended states with a general facility of virtual registers. That is, instead of (S, Σ), a state is actually a tuple (S, Σ, V), where V is a store for virtual registers. For memory allocation, we use a pair of virtual registers $(V_{\text{lo}}, V_{\text{hi}})$. The address space contains allocated locations (i.e., locations in the domain of Σ), *available* locations (i.e., locations l such that $V_{\text{lo}} \leq l < V_{\text{hi}}$), and other locations. We maintain in the convention invariant (which is now a predicate on (S, Σ, V)) that the allocated locations are disjoint from the available locations, and that the available locations are readable and writable.

Types are now predicates on (S, Σ, V), and we add two new numeric types lo and hi with values equal to the corresponding virtual registers:

$$[\![\mathsf{lo}]\!](\rho)(k, \Sigma, S, V, v) \;\triangleq\; v(0) = V_{\text{lo}}$$

The new registers are virtual—that is, they are not part of the hardware machine model and, therefore, there are no hardware instructions to access or modify them directly. We design our TAL so that two *real* registers are used to track the low and high boundaries of the available area. For example, if we use registers 8 and 9 for this purpose, we can describe the heap convention in TML with the type

$$\mathsf{heap}(n) \;\triangleq\; \{\mathrm{r}_8 : \mathsf{int}_{=}(\mathsf{lo} - n) \cap \mathsf{int}_{>}(\mathsf{const}(0)), \mathrm{r}_9 : \mathsf{int}_{=}(\mathsf{hi} - B)\}.$$

Here we use the notation $\mathsf{lo} - n$ as shorthand for $\mathsf{minus}(\mathsf{lo}, \mathsf{const}(n))$. When we are not in the middle of an allocation, $n = 0$ and we have $\mathrm{r}_8 = V_{\mathrm{lo}}$ and $\mathrm{r}_9 = V_{\mathrm{hi}} - B$, where B is a constant larger than most of the blocks we will allocate. For LTAL we use $B = 4096$. Furthermore, since r_8 points to the next block to be allocated, it should always contain a nonzero address, hence the constraint $\mathsf{int}_{>}(\mathsf{const}(0))$.

Now suppose the program wants to allocate a cons cell. The new cell can go at location r_8 provided that $V_{\mathrm{lo}} + 2 < V_{\mathrm{hi}}$. Provided that $B \geq 2$, the program can simply execute a conditional branch to test whether $\mathrm{r}_8 < \mathrm{r}_9$, and at the branch target the following vector-type is satisfied:

$$\mathsf{heap}(0) \ \cap \ \mathsf{can\_alloc}(2)$$

where $\mathsf{can\_alloc}(b) \ \triangleq \ \mathsf{exists}_{\mathsf{num}}(\{\, \mathrm{r}_8 : \mathsf{int}_{=}(\underline{0}), \mathrm{r}_9 : \mathsf{int}_{>=}(\underline{0} + b - B) \,\})$. This guarantees that at least b words remain between V_{lo} and V_{hi}. To allocate large blocks $(b > B)$ we can establish $\mathsf{can\_alloc}(b)$ with the more complicated branch test, $\mathrm{r}_8 + (b - B) > \mathrm{r}_9$.

In addition to the "updating" LTAL store instruction used for modifying already-allocated mutable references, we add an "initializing" LTAL store instruction for creating new mutable references, as well as another "initializing" LTAL store instruction for creating new immutable references. These new assembly-language store instructions use exactly the same underlying machine instruction (`st`) but have different typing rules. They manipulate the virtual register V_{lo} to move the lower boundary of the available area, and at the same time add the new location to the domain of Σ.

We illustrate with a program that performs $\mathrm{r}_3 \leftarrow \mathsf{cons}(\mathrm{r}_1, \mathrm{r}_2)$, or branches to the label out_of_memory if the heap is exhausted.

```
    $\mathsf{heap}(0) \cap \{\mathrm{r}_1 : \mathsf{int}, \mathrm{r}_2 : \mathsf{listcons}(\mathsf{int})\}$
br $(\mathrm{r}_8 > \mathrm{r}_9)$, out_of_memory
    $\mathsf{heap}(0) \cap \mathsf{can\_alloc}(2) \cap \{\mathrm{r}_1 : \mathsf{int},\ \mathrm{r}_2 : \mathsf{listcons}(\mathsf{int})\}$
st $\mathrm{r}_8[0], \mathrm{r}_1$        // ref-initializing store.
    $\mathsf{heap}(1) \cap \mathsf{can\_alloc}(2) \cap \{\mathrm{r}_8 : \mathsf{ref}\,(\mathsf{int})\} \cap \{\mathrm{r}_1 : \mathsf{int},\ \mathrm{r}_2 : \mathsf{listcons}(\mathsf{int})\}$
st $\mathrm{r}_8[1], \mathrm{r}_2$        // box-initializing store.
    $\mathsf{heap}(2) \cap \mathsf{can\_alloc}(2) \cap \{\mathrm{r}_8 : \mathsf{ref}\,(\mathsf{int})\} \cap \{\mathrm{r}_8 : \mathsf{offset}(\mathsf{const}(1), \mathsf{box}\,(\mathsf{int}))\}$
      $\cap \{\mathrm{r}_1 : \mathsf{int},\ \mathrm{r}_2 : \mathsf{listcons}(\mathsf{int})\}$
add $\mathrm{r}_3, \mathrm{r}_8, 0$
    $\mathsf{heap}(2) \cap \{\mathrm{r}_3 : \mathsf{listcons}(\mathsf{int})\}\} \cap \{\mathrm{r}_1 : \mathsf{int}, \mathrm{r}_2 : \mathsf{listcons}(\mathsf{int})\}$
add $\mathrm{r}_8, \mathrm{r}_8, 2$
    $\mathsf{heap}(0) \cap \{\mathrm{r}_1 : \mathsf{int},\ \mathrm{r}_2 : \mathsf{listcons}(\mathsf{int}),\ \mathrm{r}_3 : \mathsf{listcons}(\mathsf{int})\}$
```

Every instruction precondition includes $\mathsf{heap}(n)$ to keep track of the offset n between r_8 and the actual V_{lo} value. The fall-through postcondition of the branch instruction guarantees that $\mathrm{r}_8 \leq \mathrm{r}_9$, which is exactly $\mathsf{can\_alloc}(B)$. In turn, since $B \geq 2$, we have $\mathsf{can\_alloc}(B) <: \mathsf{can\_alloc}(2)$.

The rule (in Hoare-logic style notation) for an initializing store is as follows. Provided that there is space available to allocate, and provided that the difference between r_8 and V_{lo} is exactly n, we can store at $\mathrm{r}_8 + n$ and (implicitly) add 1 to

V_{lo}.

$$\frac{\phi <: \{\, r_d : \tau \,\} \cap \mathsf{can\_alloc}(n+1) \qquad \phi[\mathrm{r}_8 \mapsto \mathsf{offset}(n, \mathsf{ref}\,(\tau))] = \phi'}{\{\phi \cap \mathsf{heap}(n)\}\ (\mathtt{st}\ \mathrm{r}_8[n], r_d)\ \{\phi' \cap \mathsf{heap}(n+1)\}}$$

This rule (and a similar rule for immutable references) takes us past the two store instructions. By straightforward subtyping, the postcondition of the second `st` instruction is converted to

$$\mathsf{heap}(2) \cap \{\mathrm{r}_8 : \mathsf{listcons(int)}\}\} \cap \{\mathrm{r}_1 : \mathsf{int}, \mathrm{r}_2 : \mathsf{listcons(int)}\}$$

This condition takes us past the first `add` instruction. Finally, the last `add` adjusts the offset of r_8.

The rule for initializing stores increments V_{lo}. Since the machine semantics of `st` does not increment V_{lo}, readers may wonder how this is possible. The V_{lo} is a virtual component, just like our store type. Therefore, the rule for initializing stores requires that there exist virtual registers and a store type in the new state after the initializing store, given the virtual registers and the store type in the old state before the store. In the case of initializing store, the virtual components in the new state can be easily constructed based on the components in the old state.

TML is expressive enough to model the state of a partially allocated record at each stage of the allocation: testing whether there's enough space, storing each individual field, copying the result pointer ($\mathrm{r}_3 \leftarrow \mathrm{r}_8$), and adjusting the allocation pointer. Compiler optimizations can be performed, and TML can still reason about the optimized program. For example, the LTAL compiler consolidates several out-of-memory checks into one when they occur in the same extended basic block. It is also possible to schedule unrelated instructions among the memory stores.

One limitation of memory management in our system is that it supports neither garbage collection nor explicit deallocation, though Ahmed [2004, Chapter 7] shows how to encode a region calculus.

7. OVERVIEW OF THE FPCC PROJECT

We have shown how to produce machine-checked safety proofs for machine-language programs compiled from a source language that uses pointer data structures and higher-order polymorphic functions. This has been possible in the early years of the 21st century because of progress throughout the 1980s and 1990s in type theory, in compilation of functional programs, and in mechanical theorem proving.

By 1997, Harper and Morrisett [1995] and Tarditi et al. [1996] had demonstrated the principle of *typed intermediate languages*—that each intermediate language of an optimizing compiler can be type-checkable in a provably sound type system. Although TILs were not particularly useful for their original intended purpose (improving the quality of compiler optimizations), Necula [1997] showed what TILs are really good for: producing safety proofs for machine-language programs compiled from type-safe source languages. Necula's Proof-Carrying Code sidestepped the two major difficulties with proving safety properties of optimized machine-language programs: (1) the source programs are not usually correct to begin with, and (2) the compiler is too complex to prove correct. By having the compiler produce type-checkable machine code from type-checked source code, both these problems are avoided.

But Necula's PCC had several weaknesses. The trusted base contained many ad-hoc components, particularly the verification-condition generators. The type system had no machine-checked soundness proof, only a paper proof of an abstraction of a subset. The original system could not allocate new data structures.

The Cornell TAL by Morrisett et al. [1999] was an influential implementation of a PCC-like system that demonstrated an important principle: communication of proofs from compiler to checker can be done very effectively by means of a type system for assembly language.

In 1999 at Princeton we began the Foundational Proof-Carrying Code project to build a PCC system for a full-scale language (core ML) with a fully machine-checked soundness proof with respect to the smallest possible trusted base. There were many design decisions to be made. Necula had represented typing rules using the LF notation of the Edinburgh Logic Framework, and we chose to do the same. The advantage of LF for this purpose was that the natural logic-programming interpretation of LF—implemented in the Twelf system [Pfenning and Schürmann 1999] can correspond directly to the operation of a type-checker, with typing rules as Horn clauses. Furthermore, Twelf constructed proof witnesses in LF for all of its derivations, and this looked promising as a means of checking with respect to a small trusted base.

But how should the soundness proof be done? By 1999, denotational semantics was completely out of fashion; the last vestiges of it had been washed away by the enormously influential paper of Wright and Felleisen [1994], which provided an elegant notation and formulation of syntactic progress-and-preservation proofs. Furthermore, Twelf has a metatheory system that should be usable for this kind of syntactic proof. But Appel [1985] had acquired the habit of thinking (denotational) semantically about machine language. Denotational semantics can lead to elegant and modular formulations, since the main idea is that each program construct is explained by one self-contained semantic object—one constructs the semantics of larger programs by composing smaller program fragments.

Appel and Felty [2000] carried out the experiment of defining types as predicates on states, and defining type operators in Church's Higher-Order Logic (HOL) as functions on these predicates. This approach led fairly easily to a type system for machine language, capable of expressing heap-allocated *immutable* record types, first-order continuations (code-pointers), address arithmetic, covariant recursive types, polymorphic (universally quantified) types, and existential types. But it could not accommodate mutable references or contravariant recursive types, and (although we did not realize it at the time) its treatment of loops and recursive functions was too weak.

After this experiment, we decided to use HOL to do the semantic modeling of type systems whose syntax was represented in LF. We represented HOL as an object logic in Twelf. A theorem in HOL is stated as a type in LF, and its proof is just a lambda-expression with that type. We did not use any sort of tactical prover; instead we relied on Twelf's type reconstruction to help us write these lambda-expressions. An informal tutorial [Appel 2000] explains the method. Later we did use the logic-programming facilities of Twelf to write some tactics for arithmetic identities and other special purposes.

The trusted base of our system was a design criterion from the beginning. We are proving theorems of the form, "This machine-language program obeys its safety policy." To believe our proof, you must trust the axioms of the logic [Church 1940]; you must trust the proof checker for the logic [Pfenning and Schürmann 1999]; and you must trust that the statement of the theorem correctly represents something about running machine-language programs. Therefore in addition to writing down the axioms of logic, we also constructed a representation of machine-language syntax (instruction decoding) and semantics (instruction execution) in higher-order logic [Michael and Appel 2000].

At the Pittsburgh CADE-2000 conference in which this last result was presented, the conference excursion was whitewater rafting on the Youghiogheny river, a full hour by bus from Pittsburgh. On the ride back, Appel happened to sit with David McAllester and explained the problem of contravariant recursive types in a type system for machine language. McAllester recalled a course he had taken years earlier in graduate school and suggested adapting D_∞ models [Scott 1976]; this conversation led to our indexed model of types [Appel and McAllester 2001]. Initially the indices were useful for contravariant recursive types; the current paper shows that they are useful for many other things as well.

By 2001 we had a reasonably clear idea of the specification of the theorem to be proved [Appel 2001]: "this machine-language program doesn't get stuck." The safety policy was embedded in the specification of the legal machine instructions. The clear specification and the motto "Foundational Proof-Carrying Code" inspired Hamid et al. [2002] and Crary [2003] to apply syntactic progress-and-preservation proof techniques to proving soundness of typed assembly languages.

Also in 2001 it began to be clear how to transmit proofs from the compiler to the verifier, which had previously been a mystery. The solution is to use the result of Morrisett et al. [1998], that is, Typed Assembly Language. We would prove the soundness of a TAL; the compiler would output TAL; a type-checker for TAL would verify both the well-typedness of the TAL and its correspondence to the machine-language program; and the trace of this type-checker would form the backbone of the PCC proof.

We used Standard ML of New Jersey [Appel and MacQueen 1991] as the basis for our type-preserving compiler. Shao [1997] had already rewritten the front end to be type-preserving; in fact, this was much more sophisticated than what we needed (it included the ML module system), so Hai Fang reimplemented type-preserving closure conversion for Core ML. Juan Chen designed the low-level typed intermediate languages and our Low-level Typed Assembly Language (LTAL), and carried types all the way down to the bottom. Meanwhile, Dinghao Wu was implementing the soundness proof of the LTAL from the TML layer described in the current paper; in negotiations between Chen and Wu the LTAL was revised until it was both usable (by the compiler) and sound. The compiler work was completed in less than two years [Chen et al. 2003].

By 2001 we had also identified the biggest challenge: semantic modeling of mutable references. At that time there were already syntactic progress-and-preservation soundness proofs for mutable references [Harper 1994], but the semantic models before 2001 [Pitts and Stark 1993; Stark 1994] supported only integer references. Contemporaneous work [Levy 2002] supported general references, but without quantified types. We absolutely needed quantified types to implement typed closure conversion (using existential types) and polymorphism (universal types). Our first result [Ahmed et al. 2002] modeled general references with quantified types, relying on syntactic complexity (nesting of ref type constructors) to stratify the types. We thought the problem was solved, but it took us a while to realize that we had only modeled mutable references that could store values of predicative quantified types, while we needed mutable references to impredicative quantified types to handle closure conversion. Another year's work led to the full solution to the problem of mutable references with impredicative quantified types [Ahmed et al. 2003; Ahmed 2004]. At this point it became increasingly clear that HOL is not sufficiently powerful to represent our model in a fully elegant and general way. For a step-indexed model of the polymorphic λ-calculus augmented with mutable references, Ahmed et al. [2003] (and Ahmed [2004]) show a representation in set theory that does not use syntax (it is fully semantic) and also a representation in the Calculus of Inductive Constructions without using syntax. But in the HOL representation of our model we had to use an internal Gödelization of the syntax of types, which Xinming Ou implemented. Thus, the current paper (which describes the system and machine-checked proof we actually built) uses a syntax of TML and a denotation relation, all represented in HOL. Aside from this one issue, HOL was expressive enough for a very natural modeling of all the other aspects of TML, of $\mathcal{L}_c$, and of LTAL.

Between 2001 and 2003 the modular structure of the proof became clear, with the TML serving as an abstraction layer to support the LTAL. The LTAL supports 100% syntax-directed type-checking, but is consequently quite specialized and rigid; TML is not at all syntax-directed, which permits it to be general and flexible. The TML and its semantic model (other than the mutable-reference issue) were designed by Swadi, Richards, and Virga. The $\mathcal{L}_c$ control logic, which is more elegant and general than was strictly necessary to support the LTAL, was designed by Tan during this period.

Also between 2001 and 2003 we refined our notion of the trusted base. Since Twelf can, in principle, produce independently checkable proof witnesses, we decided to investigate the simplest possible external verifier of Twelf (object-logic) proofs. An object-logic proof in Twelf, by the Curry-Howard isomorphism, is simply a series of type-checked definitions. Appel et al. [2003] implemented an 800-line C program that implements LF type-checking. This means that our trusted base is less than 3000 lines of source code: 1953 (nonblank, noncomment) lines of LF to specify HOL, arithmetic, SPARC, and the safety policy; and 800 lines of C to implement the proof-checker. Furthermore, the C program is trustworthy[13] because it is the straightforward implementation of a peer-reviewed algorithm [Harper and Pfenning 2005].

Our Proof-Carrying-Code proof (that is, the proof of memory safety that accom-

[13]The C program needs to be compiled by a C compiler, so it would appear that this compiler would need to be included in our TCB. Appel et al. [2003, Sec. 8.2] describes ways of avoiding this, for example, by manually comparing the assembly output of the compiler (around 3900 lines of SPARC code) with the C checker.

panies the machine code) is structured in two parts: first, the LTAL type system and its soundness proof; second, a derivation in that type system. The LTAL-soundness component is the same for any machine code being verified, and the deriviation is (roughly) proportional in size to the machine code, with a large constant of proportionality (about 1000). Therefore, Wu et al. [2003] added to the 800-line C program a 300-line nonbacktracking prolog interpreter, which can interpret the syntax-directed clauses of LTAL. Now the proof witness sent from the compiler to the checker is just the LTAL assembly-language program. The checker first verifies the soundness proof of LTAL, then type checks the LTAL program (which also has the effect of verifying that the LTAL is correctly assembled into machine language).

By Spring 2005 the LTAL soundness proof had stabilized into 165,000 lines of Twelf code, organized as follows.

Tokens	Lines	Definitions +Lemmas	Component
6 075	526	125	Core logic and arithmetic definitions
17 827	2 409	848	Specification of machine and safety policy
299 961	31 128	3 829	Lemmas about logic and arithmetic
114 406	13 641	640	Lemmas about compilation conventions, registers, frame layouts, safety policy
142 052	16 076	2 263	Defs, lemmas about SPARC instructions
472 961	58 740	4 392	TML
72 317	8 376	399	$\mathcal{L}_c$
247 063	31 906	2 716	LTAL (up to statement rules)
17 776*	2 200*	52*	LTAL statement rules*
1 390 438	165 002	15 264	TOTAL

The asterisk indicates that the soundness proofs of some of the LTAL statement rules were not completed by the time the project came to an end. In all, the FPCC project was the work of 6 PhD students [Swadi 2003; Ahmed 2004; Chen 2004; Tan 2005; Wu 2005; Richards 2009] with help from two other students (Neophytos Michael and Xinming Ou), two consecutive postdocs (Roberto Virga and Daniel Wang), and occasional outside collaborators (Amy Felty, David McAllester, and Aaron Stump).

Subsequent work by Appel et al. [2007] has further developed the semantic framework presented in this paper. The indexed model is viewed as a Kripke model of a Gödel-Löb modal logic; the result is that the numeric indices are no longer omnipresent, being hidden inside the Kripke model. The proof by induction Theorem 5.1 of the current paper is viewed as an application of the Löb rule. Machine-checked proofs are given in Coq, which permits a natural (dependently typed) representation of Ahmed's model of mutable references.

It is now clear that the use of Church's Higher-Order Logic was a design mistake, because it does not permit the dependent types necessary for a clean implementation of our semantic model. If we had used a more powerful logic, we could have eliminated the many thousands of lines that we devoted to the specification of a syntax for TML types and the Gödelization of this syntax. In addition, the contractiveness requirement for quantified types would be eliminated, because it is an artifact of fitting our model into the HOL straightjacket. This would make the TML type system simpler for its users.

8. RELATED WORK

We group related work into four categories: work on proof-carrying code, work on semantic models of types, recent work involving step-indexed logical relations, and work related to our control logic.

8.1 Proof-Carrying Code

Necula's early proof-carrying-code systems [Necula 1997; Colby et al. 2000] had no machine-checked soundness proof and were specialized to a particular architecture and compiler. Appel and Felty [Appel and Felty 2000; Appel 2001] introduced the foundational approach to PCC in which the soundness of the system is proved from first principles. Our LTAL [Chen et al. 2003] is also specialized to an architecture and compiler, but its machine-checked soundness proof factors the common parts (TML plus $\mathcal{L}_c$) from the specialization (LTAL).

The Syntactic Approach to FPCC. Both Hamid et al. [2002] and Crary [Crary 2003; Crary and Sarkar 2003] have demonstrated syntactic foundational PCC systems. Both systems prove a soundness theorem for an abstract machine first. They then establish a simulation relation between the abstract and concrete machines to prove that if the abstract machine is never stuck the concrete machine is never stuck. Recent work by Shao et al. on Certified Assembly Programming (CAP, XCAP) [Yu et al. 2003; Ni and Shao 2006] takes a Hoare-logic-based approach to PCC. Like our system, XCAP is extensible, supports separate verification of code modules, permits impredicative quantification and (with extensions to XCAP) mutable references, and is expressive enough to specify invariants for assembly code.

The syntactic approach to FPCC does not require development of denotational semantics for complicated types such as recursive types and mutable-reference types. It has been very successful in delivering foundational proof-carrying code. On the other hand, the real system built by Crary and Sarkar uses the metatheory engine of Twelf [Pfenning and Schürmann 1999] and has not so far produced a proof object (an independently checkable proof expressed in a general logic) that represents the soundness proof. (The system of Hamid et al. [2002] deals with a target machine with only a dozen instructions.) Furthermore, there is a difficulty for the syntactic approach when trying to relate results in different type systems. The syntactic approach treats the syntax of each type system abstractly. Since each system has its own syntax of terms and types, it is difficult for the syntactic approach to derive general theorems that relate different type systems[14]. The semantic approach embeds the meaning of terms and types into a common logic. A common semantic framework makes it possible to relate theorems in different type systems.

8.2 Semantic Models and Logical Relations

Logical relations were originally developed for denotational semantics of typed λ-calculi (e.g., [Plotkin 1973; Statman 1985]). Our step-indexed logical relations

[14]See the work of Feng et al. [2007] for progress on this. In some sense, they also use a "semantic" approach: they embed specifications of different systems into their common logic OCAP.

are based on the operational semantics of the language. Early examples of logical relations based on operational semantics include Tait's proof of strong normalization of the simply typed λ-calculus [Tait 1967], and Girard's method of reducibility candidates used to prove normalization for System F [Girard 1972].

Operational logical relations have also been used for reasoning about equivalence of terms. Wand and Sullivan [1997] describe a denotational semantics based on an operational term model and use the approach for proving the correctness of program transformations in a Scheme compiler. Pitts [2002] has made use of operational (or syntactic) logical relations to reason about the equivalence of programs written in a fragment of ML. Pitts has also shown how to handle existential types [Pitts 1998] and parametric polymorphism [Pitts 2000], but always in the absence of general references and recursive datatypes.

Birkedal and Harper [1997] and Crary and Harper [2007] developed operational logical relations for recursive types—the latter also supported polymorphic types—by adapting Pitts' minimal invariance technique [Pitts 1996] for use in a purely operational setting. A question that merits further investigation is the relationship between the different notions of approximation, namely Crary and Harper's syntactic projections and our step counts.

Our model accommodates general mutable references that can contain values of any type, including other references, recursive types, code pointers, and even impredicative quantified types. Abramsky et al. [1998] describe a game semantics of general references that is fully abstract but does not support quantified types. They model reference cells as pairs of a "read method" and a "write method" in the style of Reynolds [1981], while we have a location-based model. Levy [2002] describes a model of general references that, like ours, is based on possible-worlds semantics, and like our 2002 result [Ahmed et al. 2002] makes use of syntactic types to deal with the circularity that comes with modeling ML-style mutable references.

There has also been work on reasoning about equivalence of imperative programs, which we discuss briefly. Pitts and Stark [1993] [Stark 1994] introduced the ν-calculus, a call-by-value λ-calculus with dynamically generated names that can store values of ground type, and developed operational logical relations for this calculus. Benton and Leperchey [2005] developed a logical relation based on a monadic semantics in the category of FM-cpos for a higher-order language with recursive functions and dynamically allocated mutable references. Their references may store values of ground type as well as other references, but not functions (or recursive types)—that is, they cannot support cycles in the memory graph. Their logical relations are parameterized by store relations that "depend" on only part of the stores—thus, related stores continue to be related if they are updated in parts on which the store relation does not depend. Bohr and Birkedal [2006] have since shown how to extend Benton and Leperchey's model to support mutable references that can store functions and recursive types.

8.3 Step-Indexed Logical Relations

Step-indexed logical relations have been employed in various contexts in the last few years, both for proving type safety (via logical predicates) and for establishing equivalence of programs (using binary logical relations), in both functional and imperative settings, and for both typed as well as untyped languages.

Type Safety. Ahmed, Fluet, and Morrisett [Ahmed et al. 2007; Morrisett et al. 2005] show how to prove type safety for a language with both type-invariant (*shared*, aliasable) references as well as type-varying (*unique*, unaliased) references which may be deallocated or updated with values of different types. Essentially, in this setting, closed semantic types can be modeled as predicates on k, L, Σ, S, and v, where Σ is a store type that maps only shared references to their (closed semantic) types, while L keeps track of the set of unique reference locations that are reachable only from, or *owned* by, the value v. Furthermore, in this model, two values v_1 and v_2 belonging to types t_1 and t_2 can only be paired if the locations owned by v_1 are disjoint from the locations owned by v_2. Hence, the modeling of unique references is reminiscent of models of Separation Logic. Ahmed et al. also show how to model CQUAL's `restrict` feature [Aiken et al. 2003], which allows a computation to temporarily treat a shared reference as a unique reference and perform non-type-preserving updates, even though there may be unknown aliases to the cell.

Ahmed et al. [2005] present a model of a language with *substructural state*—that is, a language in which references are qualified as *linear* (must be used exactly once), *affine* (used at most once), *relevant* (used at least once), or *unrestricted* (used any number of times). To correctly reason about the storage of unique (e.g., affine) references in shared (e.g., unrestricted) references, Ahmed et al. [2005] essentially model semantic types as predicates on k, σ, S, and v, where σ is a *local* store type that keeps track of the ascribed types and qualifiers of only those locations that are immediately reachable from v (or in a λ-calculus setting, locations that appear as sub-expressions of v). The use of a local store type σ rather than global store type Σ affects the structure of the logical relation—for instance, it eliminates the $\sqsubseteq$ predicate as formulated in this paper—and makes it possible to distinguish locations reachable from a computation from those that are garbage.

In recent work, Hriţcu and Schwinghammer [2008] show how to build a step-indexed model of the imperative object calculus of Abadi and Cardelli [1996]. The authors effectively reuse the model of general references in Ahmed's thesis, extending it with subtyping and the semantics of object types. The latter is quite subtle due to the combination of method update and invocation, object cloning, and subtyping.

Relational Reasoning. Benton and Zarfaty [2007] describe a semantic approach to verifying the type soundness of a compiler. They interpret types in the high-level language as *binary* relations over configurations of the low-level machine, unlike the FPCC prototype where we interpret high-level types using *unary* predicates over low-level code and data. The use of binary relations is more powerful and elegant since the relations can specify not just the set of values in the interpretation of a high-level type, but also a type-specific notion of equality on that set of values. Hence, type soundness of the compiler can be specified in terms of observational equivalence instead of the usual notion of stuck states. This yields a strong (extensional) notion of memory safety—that is, compiled code may actually read or write locations that it shouldn't, as long as those reads and write do not affect the observable behavior of the program. This is a richer property than the notion of memory safety provided by syntactic approaches to type soundness or even by the semantic-but-with-unary-predicates approach that we have taken. This work builds on earlier work by Benton [2006] where he presents a framework for modular specification and verification of low-level code that employs both step-indexing and a notion of relations with disjoint supports (for reasoning about separation). Benton shows how to specify and verify the implementation of a simple memory manager and its clients in this framework, though the framework can also talk about *equivalence* of two programs with respect to a specification.

Ahmed [2006] presents a step-indexed logical relation that provides a sound and complete proof method for reasoning about contextual equivalence of programs in a language with recursive types and polymorphism. Recently, Ahmed and Blume [2008] used this logical relation to show that typed closure conversion (in a language with recursive types and quantified types) preserves contextual equivalences. Proving that a compiler preserves observational equivalence—intuitively, that all abstractions guaranteed by the source type system are preserved at the target—is a far stronger property than the one guaranteed by our FPCC system, namely that the compiler preserves type and memory safety.

Logical relations in the literature have always been synonymous with typed languages due to the fact that logical relations are almost always defined by induction on types. But since step-indexed logical relations can be defined by induction on future step counts, the method can be used to reason about untyped languages. In recent work, Acar et al. [2008] and Matthews and Ahmed [2008] have demonstrated the use of step-indexed logical relations in untyped settings.

8.4 Logics for Reasoning About Control and Low-Level Code

Since the Hoare triple $\{p\}s\{q\}$ describes only the relationship between the normal entry and the normal exit states, "it is not surprising that trouble arises in considering program segments with more than one mode of entry and/or exit" [O'Donnell 1982]. To verify programs with goto statements, many researchers have proposed variants [Clint and Hoare 1972; Kowaltowski 1977; Arbib and Alagic 1979; de Bruin 1981] of conventional Hoare Logic. All of this work is at the level of high-level languages; for example, they treat a WHILE loop as a syntactic construct and have a special rule for it. In comparison, $\mathcal{L}_c$ derives rules for control-flow constructs based on a simple set of composition rules and thus is suitable for verifying low-level programs.

The aforementioned work on variants of Hoare Logic also differs from $\mathcal{L}_c$ in terms of the form of the specification. The work by de Bruin [1981] is a typical example. In his system, the judgment for a statement s is

$$\langle L_1 : p_1, \ldots, L_n : p_n | \{p\}s\{q\}\rangle, \tag{7}$$

where $L_1, \ldots, L_n$ are labels in a program P, the assertion p_i is the invariant associated with the label L_i, and the statement s is a part of the program P. The judgment judges a triple $\{p\}s\{q\}$, but under all label invariants in a program. By explicitly supplying invariants for labels in the judgment, de Bruin's system can handle goto statements, and its rule for goto statements is $\langle L_1 : p_1, \ldots, L_n : p_n | \{p_i\}\texttt{goto } L_i\{\mathit{false}\}\rangle$.

The judgment (7) is sufficient for verifying properties of programs with goto statements. TAL by Morrisett et al. uses a similar judgment to verify type safety of assembly-language programs. However, the judgment assumes the availability of global information, because it judges a statement s under all label invariants of a program. Consequently, it is impossible for de Bruin's system or TAL to compose fragments with different sets of global label invariants. In comparison, $\mathcal{L}_c$ can judge s under only those label invariants associated with the exits of s. The form of specification in $\mathcal{L}_c$ makes fewer assumptions (i.e., requires fewer label invariants) about the rest of the program and makes separate verification possible.

Cardelli [1997] proposed a linking logic to formalize program linking. Glew and Morrisett [1999] defined a modular assembly language to perform type-safe linking. Our logic $\mathcal{L}_c$ is related to these systems because exit labels can be thought of as imported labels in a module, and entry labels as exported labels. In essence, we apply the idea of modular linking to verification of machine code.

We presented the control logic $\mathcal{L}_c$ in a previous conference paper [Tan and Appel 2006]. The logic $\mathcal{L}_c$ (its composition rules in particular) can be combined with any specification language for machine states, and can verify programs with both direct and indirect goto statements. In this paper, the specification language for machine states is the TML type system. We show that the combination of TML and $\mathcal{L}_c$ provides a simple yet powerful semantic foundation for TALs.

Benton [2005] presented a typed, compositional logic for an idealized stack-based abstract machine. His logic is closely related to our control logic. It has a link rule that can combine separately verified program fragments. It also uses the step-indexed idea to resolve the circularity in control flow. But there are also differences. In particular, in Benton's logic each label has both an associated precondition and postcondition (the postcondition is the invariant before the next return instruction). Meanwhile, since we assume CPS-transformed code, our control logic has only preconditions for labels and thus enjoys a simpler presentation.

Saabas and Uustalu [2005] recently proposed a Hoare-style logic to reason about low-level programs with direct jumps, a goal similar to ours. Their logic is based on a big-step operational semantics of a low-level language. We use a small-step operational semantics instead. The small-step semantics accommodates nonterminating programs. Another difference is that Saabas and Uustalu's semantics of Hoare triples are in direct style, while ours is in continuation style. It is unclear whether the direct-style semantics can accommodate indirect jumps.

Finally, to reason about code pointers, XCAP by Ni and Shao [2006] used a two-layer system. A purely syntactic layer is built on top of a meta-logic layer. Reasoning about code pointers is done purely at the syntactic layer, while reasoning about all other assertions is done at the meta-logic layer. An interpretation function is required to map code-pointer assertions at the syntactic layer to propositions at the meta-logic layer. The two-layer formulation is complicated and limits the applicability of their system. Our model of code pointers can be embedded directly into a meta logic such as higher-order logic or the Calculus of Inductive Constructions.

9. CONCLUSION

We have designed and implemented an expressive and convenient intermediate layer—TML plus $\mathcal{L}_c$—for proving the soundness of Typed Assembly Languages. The power of the intermediate layer has been witnessed by the soundness proof of LTAL.

Our semantic methodology—using a semantics indexed everywhere by approximations corresponding to the number of remaining steps that the types are "good" for—has turned out to be robust and tractable. As our type system (and proof) evolved from a weak model [Appel and Felty 2000] to one that could support recursion, impredicativity, and mutable references, this indexing of the model has repeatedly been *the* useful induction principle. With it we can engineer a large machine-checked proof of a full-featured modern type system.

Acknowledgments.

This research was funded in part by DARPA grant F30602-99-1-0519; NSF grants CCR-9974553, CCR-0208601, and CCF-0540914; and ARDA grant NBCHC030106. We are grateful to the anonymous referees for their many helpful comments and suggestions.

REFERENCES

Abadi, M. and Cardelli, L. 1996. *A Theory of Objects.* Springer, New York.

Abramsky, S., Honda, K., and McCusker, G. 1998. A fully abstract game semantics for general references. In *13th IEEE Symposium on Logic in Computer Science (LICS).* IEEE Computer Society, Washington, DC, USA, 334–344.

Acar, U., Ahmed, A., and Blume, M. 2008. Imperative self-adjusting computation. In *35th ACM Symposium on Principles of Programming Languages (POPL).* ACM Press, New York, 309–322.

Ahmed, A. 2006. Step-indexed syntactic logical relations for recursive and quantified types. In *15th European Symposium on Programming (ESOP).* Springer-Verlag, Berlin, 69–83.

Ahmed, A., Appel, A. W., and Virga, R. 2002. A stratified semantics of general references embeddable in higher-order logic. In *17th IEEE Symposium on Logic in Computer Science (LICS).* IEEE Computer Society, Washington, DC, USA, 75–86.

Ahmed, A., Appel, A. W., and Virga, R. 2003. An indexed model of impredicative polymorphism and mutable references. http://www.cs.princeton.edu/~appel/papers/impred.pdf.

Ahmed, A. and Blume, M. 2008. Typed closure conversion preserves observational equivalence. In *ACM International Conference on Functional programming (ICFP).* ACM Press, New York, 157–168.

Ahmed, A., Fluet, M., and Morrisett, G. 2005. A step-indexed model of substructural state. In *ACM International Conference on Functional programming (ICFP).* ACM Press, New York, 78–91.

Ahmed, A., Fluet, M., and Morrisett, G. 2007. L3 : A linear language with locations. *Fundamenta Informaticae 77,* 4 (June), 397–449.

Ahmed, A. J. 2004. Semantics of types for mutable state. Ph.D. thesis, Princeton University, Princeton, NJ. Tech Report TR-713-04.

Aiken, A., Foster, J. S., Kodumal, J., and Terauchi, T. 2003. Checking and inferring local non-aliasing. In *ACM Conference on Programming Language Design and Implementation (PLDI).* ACM Press, New York.

Appel, A. W. 1985. Semantics-directed code generation. In *12th ACM Symposium on Principles of Programming Languages (POPL).* ACM Press, New York, 315–24.

Appel, A. W. 2000. Hints on proving theorems in Twelf. www.cs.princeton.edu/~appel/twelf-tutorial.

Appel, A. W. 2001. Foundational proof-carrying code. In *16th IEEE Symposium on Logic in Computer Science (LICS).* IEEE Computer Society, Washington, DC, USA, 247–258.

Appel, A. W. and Felty, A. P. 2000. A semantic model of types and machine instructions for proof-carrying code. In *27th ACM Symposium on Principles of Programming Languages (POPL).* ACM Press, New York, 243–253.

Appel, A. W. and MacQueen, D. B. 1991. Standard ML of New Jersey. In *3rd International Symposium on Programming Language Implementation and Logic Programming*, M. Wirsing, Ed. Springer-Verlag, New York, 1–13.

Appel, A. W. and McAllester, D. 2001. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. on Programming Languages and Systems 23,* 5 (Sept.), 657–683.

Appel, A. W., Mellies, P.-A., Richards, C. D., and Vouillon, J. 2007. A very modal model of a modern, major, general type system. In *34th ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press, New York, 109–122.

Appel, A. W., Michael, N. G., Stump, A., and Virga, R. 2003. A trustworthy proof checker. *Journal of Automated Reasoning 31*, 231–260.

Arbib, M. and Alagic, S. 1979. Proof rules for gotos. *Acta Informatica 11*, 139–148.

Benton, N. 2005. A typed, compositional logic for a stack-based abstract machine. In *3rd Asian Symposium on Programming Languages and Systems (APLAS)*. LNCS, vol. 3780. Springer-Verlag, Berlin.

Benton, N. 2006. Abstracting allocation: The new new thing. In *Computer Science Logic, 20th International Workshop, CSL 2006*. LNCS, vol. 4207. Springer-Verlag, Berlin.

Benton, N. and Leperchey, B. 2005. Relational reasoning in a nominal semantics for storage. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications*. Springer-Verlag, Berlin, 86–101.

Benton, N. and Zarfaty, U. 2007. Formalizing and verifying semantic type soundness for a simple compiler. In *Proceedings of the Ninth International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP)*. ACM Press, New York.

Birkedal, L. and Harper, R. 1997. Relational interpretations of recursive types in an operational setting. In *Theoretical Aspects of Computer Software*. Springer-Verlag, Berlin.

Bohr, N. and Birkedal, L. 2006. Relational reasoning for recursive types and references. In *4th Asian Symposium on Programming Languages and Systems (APLAS)*. Springer-Verlag, Berlin.

Cardelli, L. 1997. Program fragments, linking, and modularization. In *24th ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press, New York, 266–277.

Chen, J. 2004. A low-level typed assembly language with a machine-checkable soundness proof. Ph.D. thesis, Princeton University, Princeton, NJ. Tech Report CS-TR-704-04.

Chen, J., Wu, D., Appel, A. W., and Fang, H. 2003. A provably sound TAL for back-end optimization. In *ACM Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, New York, 208–219.

Church, A. 1940. A formulation of the simple theory of types. *Journal of Symbolic Logic 5,* 2 (June), 56–68.

Clint, M. and Hoare, C. A. R. 1972. Program proving: Jumps and functions. *Acta Informatica 1*, 214–224.

Colby, C., Lee, P., Necula, G. C., Blau, F., Cline, K., and Plesko, M. 2000. A certifying compiler for Java. In *ACM Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, New York.

Crary, K. 2000. Typed compilation of inclusive subtyping. In *ACM International Conference on Functional programming (ICFP)*. ACM Press, New York, 68–81.

Crary, K. 2003. Toward a foundational typed assembly language. In *30th ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press, New York, 198–212.

Crary, K. and Harper, R. 2007. Syntactic logical relations for polymorphic and recursive types. *Computation, Meaning and Logic: Articles dedicated to Gordon Plotkin, Electronic Notes in Theoretical Computer Science 172*, 259–299.

Crary, K. and Sarkar, S. 2003. Foundational certified code in a metalogical framework. In *19th International Conference on Automated Deduction (CADE '03)*. Springer-Verlag, Berlin, 106–120.

de Bruin, A. 1981. Goto statements: Semantics and deduction systems. *Acta Informatica 15*, 385–424.

Feng, X., Ni, Z., Shao, Z., and Guo, Y. 2007. An open framework for foundational proof-carrying code. In *Proceedings of the ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI)*. ACM Press, New York, 67–78.

Girard, J.-Y. 1972. Interprétation fonctionnelle et elimination des coupures dans l'arithmétique d'ordre supérieur. Ph.D. thesis, University of Paris VII.

Glew, N. and Morrisett, G. 1999. Type-safe linking and modular assembly language. In *26th ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press, New York, 250–261.

Hamid, N., Shao, Z., Trifonov, V., Monnier, S., and Ni, Z. 2002. A syntactic approach to foundational proof-carrying code. In *17th IEEE Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society, Washington, DC, USA, 89–100.

Harper, R. 1994. A simplified account of polymorphic references. *Information Processing Letters 51*, 201–206.

Harper, R. and Morrisett, G. 1995. Compiling polymorphism using intensional type analysis. In *22nd ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press, New York, 130–141.

Harper, R. and Pfenning, F. 2005. On equivalence and canonical forms in the LF type theory. *ACM Trans. on Computational Logic 6,* 1 (Jan.), 61–101.

Hoare, C. A. R. 1969. An axiomatic basis for computer programming. *Communications of the ACM 12,* 10 (October), 578–580.

Hriţcu, C. and Schwinghammer, J. 2008. A step-indexed semantics of imperative objects. In *Workshop on Foundations of Object-Oriented Languages (FOOL), informal proceedings.*

Kowaltowski, T. 1977. Axiomatic approach to side effects and general jumps. *Acta Informatica 7,* 4, 357–360.

Levy, P. B. 2002. Possible world semantics for general storage in call-by-value. In *CSL '02: Proceedings of the 16th International Workshop and 11th Annual Conference of the EACSL on Computer Science Logic.* LNCS, vol. 2471. Springer, Edinburgh, Scotland, UK, 232–246.

MacQueen, D., Plotkin, G., and Sethi, R. 1986. An ideal model for recursive polymophic types. *Information and Computation 71,* 1/2, 95–130.

Matthews, J. and Ahmed, A. 2008. Parametric polymorphiscm through run-time sealing, or, theorems for low, low prices. In *17th European Symposium on Programming (ESOP).* Springer-Verlag, Berlin, 16–31.

Melliès, P.-A. and Vouillon, J. 2004. Semantic types: A fresh look at the ideal model for types. In *31st ACM Symposium on Principles of Programming Languages (POPL).* ACM Press, New York, 52–63.

Michael, N. G. and Appel, A. W. 2000. Machine instruction syntax and semantics in higher-order logic. In *17th International Conference on Automated Deduction (CADE).* Springer-Verlag, Berlin, 7–24. LNAI 1831.

Morrisett, G., Ahmed, A., and Fluet, M. 2005. L3 : A linear language with locations. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications.* Springer-Verlag, Berlin, 293–307.

Morrisett, G., Crary, K., Glew, N., Grossman, D., Samuels, R., Smith, F., Walker, D., Weirich, S., and Zdancewic, S. 1999. TALx86: A realistic typed assembly language. In *Second ACM SIGPLAN Workshop on Compiler Support for System Software.* ACM Press, New York, 25–35.

Morrisett, G., Crary, K., Glew, N., and Walker, D. 2002. Stack-based typed assembly language. *Journal of Functional Programming 12,* 1, 43–88.

Morrisett, G., Walker, D., Crary, K., and Glew, N. 1998. From System F to typed assembly language. In *25th ACM Symposium on Principles of Programming Languages (POPL).* ACM Press, New York, 85–97.

Morrisett, G., Walker, D., Crary, K., and Glew, N. 1999. From System F to typed assembly language. *ACM Trans. on Programming Languages and Systems 21,* 3 (May), 527–568.

Necula, G. C. 1997. Proof-carrying code. In *24th ACM Symposium on Principles of Programming Languages (POPL).* ACM Press, New York, 106–119.

Ni, Z. and Shao, Z. 2006. Certified assembly programming with embedded code pointers. In *33rd ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press, New York, 320–333.

O'Donnell, M. J. 1982. A critique of the foundations of Hoare style programming logics. *Communications of the ACM 25,* 12, 927–935.

Pfenning, F. and Schürmann, C. 1999. System description: Twelf—a meta-logical framework for deductive systems. In *The 16th International Conference on Automated Deduction (CADE)*. Springer-Verlag, Berlin.

Pitts, A. M. 1996. Relational properties of domains. *Information and Computation 127,* 2, 66–90.

Pitts, A. M. 1998. Existential types: Logical relations and operational equivalence. In *ICALP '98: Proceedings of the 25th International Colloquium on Automata, Languages and Programming*. LNCS, vol. 1443. Springer-Verlag, Berlin, Germany, 309–326.

Pitts, A. M. 2000. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science 10*, 321–359.

Pitts, A. M. 2002. Operational semantics and program equivalence. In *Applied Semantics, Advanced Lectures*, G. Barthe, P. Dybjer, and J. Saraiva, Eds. LNCS, vol. 2395. Springer-Verlag, London, UK, 378–412. International Summer School, APPSEM 2000, Caminha, Portugal, September 9–15, 2000.

Pitts, A. M. and Stark, I. D. B. 1993. Observable properties of higher order functions that dynamically create local names, or: What's *new*? In *Mathematical Foundations of Computer Science*, A. M. Borzyszkowski and S. Sokołowski, Eds. LNCS, vol. 711. Springer-Verlag, Berlin, 122–141.

Plotkin, G. D. 1973. Lambda-definability and logical relations. Memorandum SAI–RM–4, University of Edinburgh, Edinburgh, Scotland. October.

Reynolds, J. C. 1981. The essence of Algol. In *Algorithmic Languages*, J. W. de Bakker and J. C. van Vliet, Eds. North-Holland, Amsterdam, 345–372.

Richards, C. D. 2009. The approximation modality in models of higher-order types. Ph.D. thesis, Princeton University, Princeton, NJ. In preparation.

Saabas, A. and Uustalu, T. 2005. A compositional natural semantics and Hoare logic for low-level languages. In *Proceedings of the Second Workshop on Structured Operational Semantics (SOS'05)*.

Scott, D. S. 1976. Data types as lattices. *SIAM Journal on Computing 5,* 3, 522–87.

Shao, Z. 1997. An overview of the FLINT/ML compiler. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation*. ACM Press, New York.

Stark, I. D. B. 1994. Names and higher-order functions. Ph.D. thesis, University of Cambridge, Cambridge, England.

Statman, R. 1985. Logical relations and the typed λ-calculus. *Information and Control 65,* 2–3 (May–June), 85–97.

Swadi, K. 2003. Typed machine language. Ph.D. thesis, Princeton University, Princeton, NJ. Tech Report TR-676-03.

Tait, W. W. 1967. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic 32,* 2 (June), 198–212.

Tan, G. 2005. A compositional logic for control flow and its application to foundational proof-carrying code. Ph.D. thesis, Princeton University, Princeton, NJ. Tech Report CS-TR-731-05.

Tan, G. and Appel, A. W. 2006. A compositional logic for control flow. In *7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. LNCS, vol. 3855. Springer-Verlag, Berlin, 80–94.

Tan, G., Appel, A. W., Swadi, K. N., and Wu, D. 2004. Construction of a semantic model for a typed assembly language. In *5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. LNCS, vol. 2937. Springer-Verlag, Berlin, 30–43.

Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R., and Lee, P. 1996. TIL: A type-directed optimizing compiler for ML. In *ACM Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, New York, 181–192.

Wand, M. and Sullivan, G. T. 1997. Denotational semantics using an operationally-based term model. In *24th ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press, New York, 386–399.

Wright, A. K. and Felleisen, M. 1994. A syntactic approach to type soundness. *Information and Computation 115,* 1, 38–94.

Wu, D. 2005. Interfacing compilers, proof checkers, and proofs for foundational proof-carrying code. Ph.D. thesis, Princeton University, Princeton, NJ. Tech Report CS-TR-733-05.

Wu, D., Appel, A. W., and Stump, A. 2003. Foundational proof checkers with small witnesses. In *5th ACM International Conference on Principles and Practice of Declarative Programming (PADL)*. ACM Press, New York, 264–274.

Yu, D., Hamid, N. A., and Shao, Z. 2003. Building certified libraries for PCC: Dynamic storage allocation. In *12th European Symposium on Programming (ESOP)*. Springer-Verlag, London, UK.