Construction of a Semantic Model for a Typed Assembly Language ⋆

Gang Tan, Andrew W. Appel, Kedar N. Swadi, and Dinghao Wu

Department of Computer Science
Princeton University
{gtan,appel,kswadi,dinghao}@cs.princeton.edu

Abstract. Typed Assembly Languages (TALs) can be used to validate the safety of assembly-language programs. However, typing rules are usually trusted as axioms. In this paper, we show how to build semantic models for typing judgments in TALs based on an induction technique, so that both the type-safety theorem and the typing rules can be proved as lemmas in a simple logic. We demonstrate this technique by giving a complete model to a sample TAL. This model allows a typing derivation to be interpreted as a machine-checkable safety proof at the machine level.

1 Overview

Safety properties of machine code are of growing concern in both industry and academia. If machine code is compiled from a safe source language, compiler verification can ensure the safety of the machine code. However, it is generally prohibitive to do verification on an industrial-strength compiler due to its size and complexity.

In this paper, we do validation directly on machine code. Necula introduced Proof-Carrying Code (PCC) [1], where a low-level code producer supplies a safety proof along with the code to the consumer. He used types to specify loop invariants and limited the scope of the proof to type safety. Typed Assembly Language (TAL) [2] by Morrisett *et al.* refined PCC by proposing a full-fledged low-level type system and a type-preserving compiler that can automatically generate type annotations as well as the low-level code. Once an assembly-language program with type annotations is type checked, the code consumer is assured of type safety.

Take a typing rule from the Cornell TAL [2],

$$\frac{\Psi;\Delta;\Gamma \vdash r : \forall[\,].\Gamma' \qquad \Delta \vdash \Gamma \leq \Gamma'}{\Psi;\Delta;\Gamma \vdash \texttt{jmp}\ r}$$

which means that a "jump to register r" instruction type-checks if the value in r is a code pointer with precondition Γ', and the current type environment Γ is a subtype of Γ'. This rule is intuitively "correct" based on the semantics of the jump instruction. (In this paper we won't be concerned with Ψ, Δ, etc. of the Cornell TAL; we show this rule just to illustrate the complexity of that system's trusted axioms.)

⋆ To appear in *5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '04)*, January 2004. This research was supported in part by DARPA award F30602-99-1-0519 and by NSF grant CCR-0208601.

In the Cornell TAL system and its variant [3] , such typing rules are accepted as axioms. They are a part of the Trusted Computing Base (TCB). However, low-level type systems tend to be complex because of intricate machine semantics. Any misunderstanding of the semantics could lead to errors in the type system. League *et al.* [4] found an unsound proof rule in the SpecialJ [5] type system. In the process of refining our own TAL [6], we routinely find and fix bugs that can lead to unsoundness.

In systems that link trusted to untrusted machine code, errors in the TCB can be exploited by malicious code. A more foundational approach is to move the entire type system out of the TCB by proving the typing rules as lemmas, instead of trusting them as axioms; also by verifying the type-safety theorem: type checking of code implies the safety policy, or the slogan—well-typed programs do not go wrong.

1.1 A foundational approach

In the type-theory community, there is a long tradition of giving denotational semantics [7] to types and proving typing rules as lemmas. Appel and Felty [8] applied this idea to PCC and gave a semantic model to types and machine instructions in higher-order logic. In the following years, semantic models of types have been extended to include recursive types [9] and mutable references [10]. With these models, it is possible to reason locally about types and operations on values, but unfortunately no model has been provided to typing judgments such as $\Psi;\Delta;\Gamma \vdash \text{jmp } r$ and no method is provided to construct the safety proof for an entire program.

The main contribution of this paper is to use a good set of abstractions to give models to judgments so that both typing rules and the type-safety theorem can be mechanically verified in a theorem-proving system. Our approach is truly foundational in that only axioms in higher-order logic and the operational semantics of instructions need to be trusted; it has a minimal trusted base. Our approach can be viewed as a way to map a typing derivation to a machine-checkable safety proof at the machine level, because each application of a typing rule in the typing derivation can be seen as the use of a (proved) lemma.

1.2 Model of TALs

In this section, we give an informal overview of our model of a typed assembly language, particularly an induction technique to prove program safety. We will not refer to any specific TAL in this section.

A TAL program consists of two parts: code and type annotations. As an example, the following code snippet has a number of basic blocks; each one is given a label (like l_0) and is composed of a sequence of instructions. Each basic block has an associated type annotation (like ϕ_0) that is the basic block's precondition.

$$
\begin{array}{ll}
\phi_0 & l_0 : \texttt{add}\ 1,1,2 \\
 & \quad\ \texttt{jmp}\ l_3 \\
\phi_1 & l_1 : \texttt{ld}\ 2,3 \\
 & \quad\ \ldots \\
\phi_m & l_m : \ldots
\end{array}
$$

Type annotations are generated by a type-preserving compiler from source language types. They serve as a specification (types as specifications). For instance, if ϕ_0 is $\{r_1 : \mathsf{int}\} \cap \{r_2 : \mathsf{box}(\mathsf{int})\}$, it expresses that register one is of type integer and register two is a pointer to an integer.

Since we need to show the type-safety theorem, the first ingredient is to define what the safety of code means. Following the standard practice in type theory, we define code is safe if it will not get stuck before it naturally stops. Note the trick here is to define the machine's operational semantics in such a way that the machine will get stuck if it violates the safety policy (see Section 2). To simplify the presentation, we treat only nonterminating programs that do not stop naturally.[1]

We first informally explore how to prove a TAL program is safe. We define "a label l in a state is safe for k steps" to mean that when the control of the state is at l, the state can run for k steps. The goal then is to prove that if entry condition ϕ_0 holds, label l_0 is safe for k steps for any natural number k. A natural thought is to show it by induction over k. The base case ($k = 0$) is trivial; the inductive case is to show label l_0 is safe for $k+1$ steps given that it is safe for k steps. But at this moment we have no idea in which state the code will be after k steps, so we cannot prove that the state can go ahead one more step.

The solution is to do induction simultaneously over all labels, i.e. prove each label l_i is safe for $k+1$ steps with respect to its precondition ϕ_i, assuming all labels l_j are safe for k steps with respect to ϕ_j. Let us take label l_0 in the example to see why this new induction works. Basic block l_0 has length two, and ends with a jump to label l_3, which has been assumed to be safe for k steps, provided that precondition ϕ_3 is met. Suppose by inspecting the two instructions in block l_0, we have concluded that l_0 is safe for two steps and after two steps ϕ_3 will be true. Combined with the assumed k-step safety of label l_3, label l_0 is safe for $k+2$ steps, which implies that it is safe for $k+1$ steps.

In this proof method, we still need to inspect each instruction in every block. For example, in block l_0, we check if the precondition ϕ_0 is enough to certify the safety of its two instructions and if ϕ_3 will be met after their execution. What we have described essentially is a proof method to combine small proofs about instructions into a proof about the safety of the whole program. In the rest of this section, we informally give models to typing judgments based on this technique. Before that, we first motivate what kind of typing judgments a TAL would usually have.

To type check a TAL program, the type system must have a wellformedness judgment for programs. Since a TAL program is composed of basic blocks, the wellformedness judgment for programs requires another judgment, a wellformedness judgment for basic blocks. Similarly, the type system should also have a wellformedness judgment for instructions.

The model of the wellformedness judgment for programs can be that all labels are safe with respect to their preconditions. In the following sections, we will develop abstractions so that this model can be written down in a succinct subtyping formula: $\Delta(C) \subset \Gamma$. The model of wellformedness of a basic block can be that this particular ba-

[1] Every program can be transformed into this form by giving it a continuation at the beginning and letting the last instruction be a jump to this continuation. The continuation could return to the operating system, for example.

sic block is safe for $k+1$ steps assuming all the other basic blocks are safe for k steps. Based on the induction technique and this model, we can prove the typing rule that concludes the wellformedness of a program from the wellformedness of basic blocks. The model of wellformedness of instructions is similar to the one of basic blocks and we do not go into details at this stage.

In Section 2, we present the model of a RISC architecture (Sparc) and formally define the safety of code based on a particular safety policy (memory safety); Section 3 shows the syntax of a sample TAL; Section 4 shows an indexed model of types and its intuition. The material in these three sections has been described by other papers [11, 6, 9] as part of the foundational PCC project; we briefly sketch them to set up a framework within which our proof method can be formally presented in section 5.

2 Safety specification

Our machine model consists of a set of formulas in higher-order logic that specify the decoding and operational semantics of instructions. Our safety policy specifies which addresses may be loaded and stored by the program (memory safety) and defines what the safety of code means. Our machine model and safety policy are trusted and are small enough to be "verifiable by inspection".

A machine state (r,m) consists of a register bank r and a memory m, which are modeled as functions from numbers to contents (also numbers). A machine instruction is modeled by a relation between machine states (r,m) and (r',m') [11]. For example, a load instruction (`ld`) is specified by[2]

$$\texttt{ld}\ s,d \equiv \lambda r,m,r',m'.\ r'(d) = m(r(s)) \ \wedge\ \big(\forall x \neq d.\ r'(x) = r(x)\big) \ \wedge\ m' = m \ \wedge\ \text{readable}(r(s))$$

The machine operational semantics is modeled by a step relation $\mapsto$ that steps from one state (r,m) to another state (r',m') [11], where the state (r',m') is the result of first decoding the current machine instruction, incrementing the program counter and then executing the machine instruction.

The important property of our step relation is that it is deliberately partial: it omits any step that would be illegal under the safety policy. For example, suppose in some state (r,m) the program counter points to a `ld` instruction that would, if executed, load from an address that is unreadable according to the safety policy. Then, since our `ld` instruction requires that the address must be readable, there will not exist (r',m') such that $(r,m) \mapsto (r',m')$.

The mixing of machine semantics and safety policy is to follow the standard practice in type theory so that we can get a clean and uniform definition of code safety. For instance, we can define that a state is safe if it cannot lead to a stuck state.

$$\text{safe}(r,m) \equiv \forall r',m'.\ (r,m) \mapsto^* (r',m') \ \Rightarrow \exists r'',m''.\ (r',m') \mapsto (r'',m'')$$

where $\mapsto^*$ denotes zero or more steps.

[2] Our step relation first increments *pc*, then executes an instruction. Thus, the semantics of `ld` does not include the semantics of incrementing the *pc*.

To show safe(r,m), it suffices to prove that the state is "safe for n steps," for any natural number n.

$$\text{safe\_n}(n,r,m) \equiv \forall r',m'.\ \forall j < n.\ (r,m) \mapsto^j (r',m') \ \Rightarrow\ \exists r'',m''.\ (r',m') \mapsto (r'',m'')$$

where $\mapsto^j$ denotes j steps being taken.

An assembly-language program C is a list of assembly instructions. For example,

$$C = \texttt{add r1,r1,r2;}\ \ \texttt{jmp } l_3\texttt{;}\ \ \texttt{ld [r2],r3;}\ \ \ldots$$

We use predicate prog_loaded(m,C) to mean that code C is loaded in memory m:

$$\text{prog\_loaded}(m,C) \equiv \forall\ 0 \le k < |C|.\ \text{decode}(m(4k),C_k)$$

where $|C|$ is the length of the list; predicate decode(x,C_k) means that word x is decoded into instruction C_k (the k-th instruction in C). In this paper, we assume code C is always loaded at start address 0 and thus the k-th instruction will be at address $4k$ in the memory (Sparc instructions are four bytes long).

We define that an assembly program C is safe if any initial state (r,m) satisfying the following is a safe state: code C is loaded inside m; the program counter initially points to address 0; when the program begins executing, entry condition ϕ_0[3] holds on the state (r,m).

$$\text{safe\_code}(C) \equiv \forall r,m.\ \big(\text{prog\_loaded}(m,C)\ \wedge\ r(\text{pc}) = 0\ \wedge\ (r,m) : \phi_0\big)\ \Rightarrow\ \text{safe}(r,m)$$

3 Typed Assembly Language

In this section, we introduce a typed assembly language. We will show here a small subset of our actual implementation. Our full language has hundreds of operators and rules, as necessary for production-scale safety checking of real software (so it's a good thing that our full soundness proof is machine-checkable).

3.1 Syntax

Figure 1 shows our TAL syntax. A TAL program consists of assembly program C and type annotation Γ. Assembly program C is a sequence of assembly instructions, which include addition (`add`), load (`ld`) and branch always (`ba`).[4] Type annotation Γ takes the form of a label environment, which summarizes the preconditions of all the labels of the program. Our TAL has 32 registers, and labels are divisible by 4.

[3] In our implementation, the initial condition ϕ_0 is simple enough to be described directly in our underlying logic so that semantic model of types is not a part of the specification of the safety theorem; it is contained entirely within the *proof* of the theorem.

[4] Since our sample TAL cannot deal with delay slots, the `ba` instruction is really a `ba` followed by a `nop` in Sparc.

$$
\begin{array}{rl}
(\textit{program}) & C ::= i \mid i;C \\
(\textit{instruction}) & i ::= \mathtt{add}\ r,r,r \mid \mathtt{ld}\ r,r \mid \mathtt{ba}\ l \\
(\textit{label env}) & \Gamma ::= \{l : \mathsf{codeptr}(\phi)\} \cap \ldots \\
(\textit{register num}) & r ::= 0 \mid 1 \mid \ldots \mid 31 \\
(\textit{label}) & l ::= 0 \mid 4 \mid 8 \mid \ldots
\end{array}
$$

Fig. 1. Syntax: Typed assembly language syntax

$$
\begin{array}{rl}
(\textit{types}) & \tau ::= \mathsf{int} \mid \mathsf{int}_{=}(n) \\
& \quad \mid \mathsf{box}(\tau) \mid \mathsf{codeptr}(\phi) \\
(\textit{type env}) & \phi ::= \top_\phi \mid \bot_\phi \mid \{n : \tau\} \\
& \quad \mid \phi_1 \cap \phi_2 \mid \phi[n \mapsto \tau] \\
(\textit{nat}) & n ::= 0 \mid 1 \mid 2 \mid \ldots
\end{array}
$$

Fig. 2. Syntax: Types

Figure 2 lists the type and type environment constructors. They include integer type int and immutable reference type $\mathsf{box}(\tau)$. The language also has singleton type $\mathsf{int}_{=}(n)$ containing only value n. An address l has type $\mathsf{codeptr}(\phi)$ if it is safe to pass the control to address l provided that precondition ϕ is met.

A type environment ϕ specifies types of slots in a vector, such as a register bank or the list of program labels. Any vector satisfies environment $\top_\phi$, and no vector can satisfy $\bot_\phi$. A singleton environment $\{n : \tau\}$ means slot n (e.g., register n or label n) has type τ. Intersection type $\phi_1 \cap \phi_2$ can be used to type several slots of a vector, e.g. $\{n_1 : \tau_1\} \cap \{n_2 : \tau_2\}$ specifies that slots n_1 and n_2 have type τ_1 and τ_2 , respectively.

We use $\phi \subset \{n : \tau\}$ to describe that $\{n : \tau\}$ is one of the conjuncts in ϕ. We write $\phi(n)$ for the type of slot n in ϕ. Notation $\phi[n \mapsto \tau]$ updates the type of slot n to τ, by first removing the old entry for n in ϕ (if one exists), then intersecting it with $\{n : \tau\}$.

The type annotation Γ, or the label environment, specifies the type of each label in terms of the code pointer type, so it is also a type environment; we use the same operators for Γ as for ϕ.

3.2 Type checking

There are three kinds of judgments in our type system:

- **Program** judgment $\vdash_{\mathtt{p}} C : \Gamma$ means that assembly program C is wellformed with respect to type annotation Γ.
- **Block** judgment $\Gamma; l \vdash_{\mathtt{b}} C : \Gamma'$ means that assembly program C, starting at address l, is wellformed with respect to Γ', assuming the global label-environment Γ. Environment Γ provides preconditions of labels to which C might jump; Environment Γ' is the collection of preconditions of labels inside C and is a part of Γ. Superficially, it seems semantically circular to judge the wellformedness of some labels (Γ') by assuming the wellformedness of all labels (Γ). However, as indicated in the overview section, the model of $\vdash_{\mathtt{b}}$ is that from a weaker assumption about Γ (every label inside is safe for k steps), we prove a stronger result about Γ' (every label inside is safe for $k+1$ steps).
- **Instruction** judgment $\Gamma; l \vdash_{\mathtt{i}} \{\phi_1\}\, i\, \{\phi_2\}$ means that assembly instruction i, at address l, is wellformed with respect to precondition ϕ_1 and postcondition ϕ_2. As in $\vdash_{\mathtt{b}}$, Γ provides label preconditions. The purpose of having location l in the judgment is to be able to compute the destination address for pc-relative jump instructions.

Typing rules except for instructions are shown in Figure 3. To check that program C is wellformed, the PROG rule will call $\Gamma;0 \vdash_{\mathtt{b}} C:\Gamma$, thus recursively call BLOCK_1 and BLOCK_2 rules to check that each basic block in C is wellformed.

Rule BLOCK_1 first looks up precondition ϕ_1 and postcondition ϕ_2 in Γ for the current block, composed of one instruction i; checks the wellformedness of instruction i with respect to ϕ_1 and ϕ_2; then checks the rest of the code with respect to Γ'. Without loss of generality, we assume that each block has exactly one instruction. In rule BLOCK_2, the postcondition of the last instruction i is $\bot_\phi$, because the control is not allowed to be beyond the last instruction (an unconditional branch satisfies this postcondition).

$$\frac{\Gamma;0 \vdash_{\mathtt{b}} C:\Gamma}{\vdash_{\mathtt{p}} C:\Gamma}\ \text{PROG} \qquad \frac{\Gamma(l)=\mathsf{codeptr}(\phi_1) \qquad \Gamma(l+4)=\mathsf{codeptr}(\phi_2) \qquad \Gamma;l \vdash_{\mathtt{i}} \{\phi_1\}\,i\,\{\phi_2\} \qquad \Gamma;l+4 \vdash_{\mathtt{b}} C:\Gamma'}{\Gamma;l \vdash_{\mathtt{b}} i;C:\{l:\mathsf{codeptr}(\phi_1)\}\cap\Gamma'}\ \text{BLOCK\_1}$$

$$\frac{\Gamma(l)=\mathsf{codeptr}(\phi) \qquad \Gamma;l \vdash_{\mathtt{i}} \{\phi\}\,i\,\{\bot_\phi\}}{\Gamma;l \vdash_{\mathtt{b}} i:\{l:\mathsf{codeptr}(\phi)\}}\ \text{BLOCK\_2}$$

Fig. 3. Syntax: Typing rules (except for instructions)

Figure 4 shows typing rules for instructions. The rule for instruction `add` requires that the source registers are of type int beforehand and the destination register gets type int afterward. The rule for instruction `ba` needs to look up the type of the destination label through Γ and check the current precondition ϕ matches the destination one (A TAL usually has subtyping rules allowing the current precondition to be stronger than the destination one).

$$\frac{\phi \subset \{s_1:\mathsf{int}\} \qquad \phi \subset \{s_2:\mathsf{int}\}}{\Gamma;l \vdash_{\mathtt{i}} \{\phi\}\,\mathtt{add}\ s_1,s_2,d\,\{\phi[d \mapsto \mathsf{int}]\}} \qquad \frac{\phi \subset \{s:\mathsf{box}(\tau)\}}{\Gamma;l \vdash_{\mathtt{i}} \{\phi\}\,\mathtt{ld}\ s,d\,\{\phi[d \mapsto \tau]\}} \qquad \frac{\Gamma(l+d)=\mathsf{codeptr}(\phi)}{\Gamma;l \vdash_{\mathtt{i}} \{\phi\}\,\mathtt{ba}\ d\,\{\bot_\phi\}}$$

Fig. 4. Syntax: Typing rules for instructions

4 Indexed model of types

In this section, we give a brief description of the *indexed* model of types, which is introduced in [9] to model general recursive types. Our induction technique in the overview section is also inspired by the intuition behind the indexed model.

In the indexed model, a type is a set of indexed values $\{\langle k,m,x\rangle\}$, where k is a natural number ("approximation" index), m is a memory, and x is an integer.

The indexed model of the types and type environments are listed below. For example, type $\mathsf{int}_{=}(3)$ would contain all the $\langle k,m,x\rangle$ such that x is 3. Memory m is a part of a value $\langle k,m,x\rangle$ because to express that x is of type $\mathsf{box}(\tau)$ we need to say that the content in the memory, or $m(x)$, is related to type τ.

$$\begin{aligned}
\mathsf{int} &\equiv \{\langle k,m,x\rangle \,|\, \text{true}\} & \mathsf{int}_{=}(n) &\equiv \{\langle k,m,x\rangle \,|\, x = n\}\\
\mathsf{box}(\tau) &\equiv \{\langle k,m,x\rangle \,|\, x \in \text{dom}(m) \wedge \forall j < k.\, \text{readable}(x) \wedge \langle j,m,m(x)\rangle \in \tau\}\\
\mathsf{codeptr}(\phi) &\equiv \{\langle k,m,x\rangle \,|\, \forall j,r.\ j < k \wedge r(\text{pc}) = x \wedge (m,r) :_j \phi \Rightarrow \text{safe_n}(j,r,m)\}
\end{aligned}$$

$$\begin{aligned}
\top_\phi &\equiv \{\langle k,m,\vec{x}\rangle \,|\, \text{true}\} & \bot_\phi &\equiv \{\langle k,m,\vec{x}\rangle \,|\, \text{false}\}\\
\{n:\tau\} &\equiv \{\langle k,m,\vec{x}\rangle \,|\, \langle k,m,x_n\rangle \in \tau\}\\
\phi_1 \cap \phi_2 &\equiv \{\langle k,m,\vec{x}\rangle \,|\, \langle k,m,\vec{x}\rangle \in \phi_1 \wedge \langle k,m,\vec{x}\rangle \in \phi_2\}\\
\phi[n \mapsto \tau] &\equiv \{\langle k,m,\vec{x}\rangle \,|\, \exists y.\ \langle k,m,\vec{x}[n \mapsto y]\rangle \in \phi \wedge \langle k,m,x_n\rangle \in \tau\}
\end{aligned}$$

We use $(m,x) :_k \tau$ as a syntactic sugar for $\langle k,m,x\rangle \in \tau$. We write $(m,x) : \tau$ to mean $(m,x) :_k \tau$ is true for any k, or (m,x) is a real member of type τ. Now we explain the purpose of index k in the model. In general, if $(m,x) :_k \tau$, value (m,x) may be a real member of type τ, or it may be a "fake" member that only k-approximately belongs to τ. Any program taking such a "fake" member as an input cannot tell the difference within k steps.

Let type τ be $\mathsf{box}(\mathsf{box}(\mathsf{int}))$. Suppose $(m,x) : \tau$, then x is a two-fold pointer and $m(m(x))$ is of type int. However, suppose we only know that $(m,x) : \mathsf{box}(\mathsf{int})$, then for one step (one dereference), (m,x) safely simulates membership in $\mathsf{box}(\mathsf{box}(\mathsf{int}))$. In this case, we can say $(m,x) :_1 \mathsf{box}(\mathsf{box}(\mathsf{int}))$.

One property of types is that they are closed under decreasing approximations, that is, if $(m,x) :_k \tau$ and $j < k$, then $(m,x) :_j \tau$.

Another example to understand the approximation index k is the type $\mathsf{codeptr}(\phi)$. A real member (m,l) of type $\mathsf{codeptr}(\phi)$ means that if condition ϕ is met, it is safe to jump to location l. Then $(m,l) :_k \mathsf{codeptr}(\phi)$ would mean that it is safe to execute k steps after jumping to l. Therefore, the definition of $\mathsf{codeptr}(\phi)$ says that for any j and r such that j is less than k, if the control is at location l and the current state satisfies ϕ, the state should be safe for j steps. In some sense, this definition only guarantees partial safety: safe within k steps. To show that location l is a safe location, we have to prove that it belongs to $\mathsf{codeptr}(\phi)$ under any k.

Sometimes we need to judge not only scalar values such as $\langle k,m,x\rangle$ but also vector values $\langle k,m,\vec{x}\rangle$ (a vector is a function from numbers to values). One use is to write $(m,r) :_k \phi$, which means that the contents of machine registers satisfy ϕ. In this case, $\vec{x}$ is the register bank r. Another use of vector types is the label environment Γ, which summarizes the types of all program labels. In this case, $\vec{x}$ would be the identity vector id (map l to l). For example, $(m,\text{id}) : \{l : \mathsf{codeptr}(\phi)\}$ means that address l itself has type $\mathsf{codeptr}(\phi)$.

With the semantic model of types, all the subtyping rules and introduction/elimination rules of types (not shown in the paper) can be proved as lemmas [8, 9]. In particular, the codeptr elimination rule CPTR_E is useful for the proof of Theorem 2 in Section 5.3.

$$\frac{(m,x) :_{k+1} \mathsf{codeptr}(\phi) \qquad r(pc) = x \qquad (m,r) :_k \phi}{\text{safe\_n}(k,r,m)}\ \text{CPTR\_E}$$

5 Semantic Models of typing judgments

In this section, we develop models for typing judgments based on the induction technique in Section 1.2. From their models, each of the typing rules is proved as a derived lemma. Finally, the type-safety theorem is also proved as a derived lemma.

5.1 Subtype induction

Our goal is to provide a proof that code C obeys our safety policy, or a proof of safe_code(C), which means that any state containing the code C is safe arbitrarily many steps under condition ϕ_0 — exactly what the type $\mathsf{codeptr}(\phi_0)$ denotes. Thus, our goal is formalized as $(m,0) : \mathsf{codeptr}(\phi_0)$, for any m containing code C.

As outlined in the overview section, we will prove a stronger result instead: all labels are of code-pointer types under corresponding preconditions. Formally, for any label l in the domain of label environment Γ, we will prove $(m,l) : \Gamma(l)$; another way to state this is $(m,\mathrm{id}) : \Gamma$.

A condition on m is that it must contain the code C. This condition can also be formalized as types. After all, in our model, types are predicates over states and can be used to specify invariants of states. Type $\mathsf{instr}(i)$ expresses that instruction i is in the memory.

$$\mathsf{instr}(i) \equiv \{\langle k,m,x\rangle \,|\, \mathrm{decode}(m(x),i)\}$$

Type environment constructor Δ turns a sequence of assembly instructions into a type environment that describes the code.

$$\Delta(i_0;i_1;\ldots) \equiv \{0 : \mathsf{instr}(i_0)\} \cap \{4 : \mathsf{instr}(i_1)\} \cap \ldots$$

With constructor Δ, judgment $(m,\mathrm{id}) : \Delta(C)$ formally states that memory m contains code C. For such a value (m,id), we want to show it also satisfies Γ, or $(m,\mathrm{id}) : \Gamma$. Now if we define

$$\Delta(C) \subset \Gamma \equiv \forall k,m.\ (m,\mathrm{id}) :_k \Delta(C) \ \Rightarrow (m,\mathrm{id}) :_k \Gamma$$

then $\Delta(C) \subset \Gamma$ expresses that any state containing code C respects invariants Γ under any approximation k.

We explore how to prove $\Delta(C) \subset \Gamma$. Assuming $(m,\mathrm{id}) :_k \Delta(C)$, we have to show $(m,\mathrm{id}) :_k \Gamma$. We will prove it by induction over k. When k is zero, judgment $(m,l) :_0 \Gamma(l)$ is trivially true since $\Gamma(l)$ is a code pointer type, which is always true at index zero by its definition. The inductive case is that, assuming $(m,\mathrm{id}) :_k \Delta(C) \Rightarrow (m,\mathrm{id}) :_k \Gamma$, we have to show that $(m,\mathrm{id}) :_{k+1} \Delta(C) \Rightarrow (m,\mathrm{id}) :_{k+1} \Gamma$. Since code environment $\Delta(C)$ ignores the index, $(m,\mathrm{id}) :_{k+1} \Delta(C)$ is equivalent to $(m,\mathrm{id}) :_k \Delta(C)$. Therefore, to prove $(m,\mathrm{id}) :_{k+1} \Gamma$, we have both $(m,\mathrm{id}) :_k \Delta(C)$ and $(m,\mathrm{id}) :_k \Gamma$.

Our intention is to give models to the typing judgments in our TAL based on this proof technique. To make the models concise, we abstract away from the indexes by defining a subtype-plus predicate $\Gamma_1 \Subset \Gamma_2$ to simulate the inductive case.

$$\Gamma_1 \Subset \Gamma_2 \equiv \forall k,m.\ (m,\mathrm{id}) :_k \Gamma_1 \ \Rightarrow (m,\mathrm{id}) :_{k+1} \Gamma_2$$

With the subtype-plus operator, the inductive case to prove $\Delta(C) \subset \Gamma$ can be written as $\Delta(C) \cap \Gamma \Subset \Gamma$: assuming code C is in the memory under index k and Γ is true under index k, prove that Γ is true under index $k+1$. The following subtype induction theorem formalizes what we have explained.

Theorem 1. *(Subtype Induction)*
$$\frac{\Delta(C) \cap \Gamma \Subset \Gamma}{\Delta(C) \subset \Gamma}$$

5.2 Semantic model of typing judgments

At the heart of our semantic model is a set of concise definitions for the typing judgments based on the abstractions (especially $\Subset$) we have developed. We hereby exhibit such definitions:

$$\begin{array}{ll}
\vdash_{\mathtt{p}} C : \Gamma & \equiv \Delta(C) \subset \Gamma \\
\Gamma; l \vdash_{\mathtt{b}} C : \Gamma' & \equiv \mathsf{offset}_l(\Delta(C)) \cap \Gamma \Subset \Gamma' \\
\Gamma; l \vdash_{\mathtt{i}} \{\phi_1\} i \{\phi_2\} & \equiv \{l : \mathsf{instr}(i)\} \cap \Gamma \cap \{l+4 : \mathsf{codeptr}(\phi_2)\} \Subset \{l : \mathsf{codeptr}(\phi_1)\}
\end{array}$$

where $\mathsf{offset}_l(\{0 : \tau_1\} \cap \{4 : \tau_4\} \cap \ldots) \;=\; \{l : \tau_1\} \cap \{l+4 : \tau_4\} \cap \ldots$.

The model of $\vdash_{\mathtt{p}} C : \Gamma$ means that any state having the code inside respects invariants Γ.

The judgment $\Gamma; l \vdash_{\mathtt{b}} C : \Gamma'$ judges the validity of Γ' by assuming Γ. Since Γ' is the collection of preconditions of labels inside C and is a part of Γ, the judgment itself has a superficial semantic circularity. We solve this circularity by giving it a model based on operator $\Subset$. By assuming Γ to approximation k, we prove Γ' to approximation $k+1$. Also, since code C starts at address l, we need to use $\mathsf{offset}_l(\Delta(C))$ to make a code environment that starts at address l.

The semantics of $\Gamma; l \vdash_{\mathtt{i}} \{\phi_1\} i \{\phi_2\}$ follows the same principle as the one for the model of $\vdash_{\mathtt{b}}$. We assume that instruction i is at location l and Γ holds to approximation k, we prove location l is a code pointer to approximation $k+1$. In the model of $\vdash_{\mathtt{i}}$, we have an extra assumption $\{l+4 : \mathsf{codeptr}(\phi_2)\}$. In our sample TAL, since every basic block has only one instruction, every address would have a precondition in Γ. Therefore, $\{l+4 : \mathsf{codeptr}(\phi_2)\}$ is a part of Γ and need not have been specified as a separate conjunct. However, in the case of multi-instruction basic blocks, Γ would only have preconditions for each basic block, ϕ_2 in this case would be reconstructed by the type system and not available in Γ.

5.3 Semantic proofs of typing rules

Using these models, we can prove both the type-safety theorem and the typing rules in Fig.3.

Theorem 2. *(Type Safety)*
$$\frac{\vdash_{\mathtt{p}} C : \Gamma \qquad \Gamma \subset \{0 : \mathsf{codeptr}(\phi_0)\}}{\mathrm{safe\_code}(C)}$$

Proof. From the definition of safe_code(C), we have the following assumptions for state (r,m) and want to show that safe(r,m).

$$i)\ \ \text{prog\_loaded}(C,m) \qquad ii)\ \ r(pc)=0 \qquad iii)\ \ (m,r):\phi_0$$

On the other hand, the model of $\vdash_{\mathtt{p}} C:\Gamma$ is $\Delta(C)\subset\Gamma$. The deduction steps from $\Delta(C)\subset\Gamma$ to safe(r,m) are summarized by the following proof tree.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{prog\_loaded}(C,m)}{(m,\mathrm{id}):\Delta(C)}\ (7) \qquad \Delta(C)\subset\Gamma \qquad \Gamma\subset\{0:\mathsf{codeptr}(\phi_0)\}}{(m,\mathrm{id}):\{0:\mathsf{codeptr}(\phi_0)\}}\ (6)}{(m,0):\mathsf{codeptr}(\phi_0)}\ (5)}{\forall k.\ (m,0):_k \mathsf{codeptr}(\phi_0)}\ (4)}{\forall k.\ (m,0):_{k+1}\mathsf{codeptr}(\phi_0)}\ (3a) \qquad \dfrac{(m,r):\phi_0}{\forall k.\ (m,r):_k\phi_0}\ (3b) \qquad r(pc)=0}{\dfrac{\forall k.\ \text{safe\_n}(k,r,m)}{\text{safe}(r,m)}\ (1)}\ (2)$$

Step (1) says to prove (r,m) is safe, it suffices to prove that (r,m) is safe for an arbitrary k steps. Step (2) is justified by rule CPTR_E in Section 4. Step (3a) is by universal instantiation. Step (4) is just the unfolding of the syntactic sugar of $(m,0):\mathsf{codeptr}(\phi_0)$. Step (5) is by the definition of the singleton environment. Step (6) is by the transitivity of subtyping. Step (7) can be easily proved by unfolding definitions. □

Theorem 3.
$$\frac{\Gamma;0\vdash_{\mathtt{b}} C:\Gamma}{\vdash_{\mathtt{p}} C:\Gamma}\ \text{PROG}$$

Proof. From the models of $\vdash_{\mathtt{p}}$ and $\vdash_{\mathtt{b}}$, we have to prove $\Delta(C)\subset\Gamma$ from $\Delta(C)\cap\Gamma\Subset\Gamma$ — exactly the subtype induction theorem (Theorem 1). □

Theorem 4.
$$\frac{\Gamma(l)=\mathsf{codeptr}(\phi_1) \qquad \Gamma(l+4)=\mathsf{codeptr}(\phi_2) \qquad \Gamma;l\vdash_{\mathtt{i}}\{\phi_1\}\,i\,\{\phi_2\} \qquad \Gamma;l+4\vdash_{\mathtt{b}} C:\Gamma'}{\Gamma;l\vdash_{\mathtt{b}} i;C:\{l:\mathsf{codeptr}(\phi_1)\}\cap\Gamma'}\ \text{BLOCK\_1}$$

Proof. The models of $\Gamma;l\vdash_{\mathtt{i}}\{\phi_1\}\,i\,\{\phi_2\}$ [5] and $\Gamma;l+4\vdash_{\mathtt{b}} C:\Gamma'$ gives us that

$$\{l:\mathsf{instr}(i)\}\cap\Gamma\Subset\{l:\mathsf{codeptr}(\phi_1)\} \qquad\qquad \mathsf{offset}_{l+4}(\Delta(C))\cap\Gamma\Subset\Gamma'$$

The goal $\mathsf{offset}_l(\Delta(i;C))\cap\Gamma\Subset\{l:\mathsf{codeptr}(\phi_1)\}\cap\Gamma'$ is proved by the following lemmas:

$$\frac{\Gamma\Subset\Gamma_1 \qquad \Gamma\Subset\Gamma_2}{\Gamma\Subset\Gamma_1\cap\Gamma_2} \qquad\qquad \mathsf{offset}_l(\Delta(i;C))=\{l:\mathsf{instr}(i)\}\cap\mathsf{offset}_{l+4}(\Delta(C))$$

□

The proof of rule BLOCK_2 is similar.

[5] The model of $\vdash_{\mathtt{i}}$ has another clause $\{l+4:\mathsf{codeptr}(\phi_2)\}$ on the left of $\Subset$. However, since $\Gamma(l+4)=\mathsf{codeptr}(\phi_2)$, we can prove that $\Gamma\cap\{l+4:\mathsf{codeptr}(\phi_2)\}=\Gamma$.

5.4 Semantic Proofs of Machine Instructions

What remains is the proofs of typing rules for instructions (Fig.4). We will show the technique by informally proving the LOAD rule.

$$\frac{\phi \subset \{s : \mathsf{box}(\tau)\}}{\Gamma; l \vdash_{\mathtt{i}} \{\phi\}\ \mathtt{ld}\ s,d\ \{\phi[d \mapsto \tau]\}}\ \text{LOAD}$$

The precondition states that register s is a pointer to type τ; the postcondition is that register d gets type τ and types of other registers remain the same. This typing rule is intuitively "correct" since operationally $\mathtt{ld}\ s,d$ loads the content at address $r(s)$ in the memory into register d.

The semantic model of $\vdash_{\mathtt{i}}$ tells us what the "correctness" of the rule LOAD means:

$$\{l : \mathsf{instr}(\mathtt{ld}\ s,d)\} \cap \Gamma \cap \{l+4 : \mathsf{codeptr}(\phi')\} \Subset \{l : \mathsf{codeptr}(\phi)\}$$

where $\phi' = \phi[d \mapsto \tau]$. That is, forall k and m, we should prove $(m,l) :_{k+1} \mathsf{codeptr}(\phi)$, or location l is safe within $k+1$ steps. We can assume (1) there is an instruction $\mathtt{ld}\ s,d$ at address l in m; (2) all the labels in Γ are code pointers to approximation k; (3) label $l+4$ is of type $\mathsf{codeptr}(\phi')$ to approximation k.

By the definition of $(m,l) :_{k+1} \mathsf{codeptr}(\phi)$, we start at (r,m) with condition ϕ met and control at l. By the semantics of $\mathtt{ld}\ s,d$, if the location (location $r(s)$) to read is readable, we can find a succeeding state (r',m') such that $(r,m) \mapsto (r',m')$. Not by coincidence, rule LOAD has a premise that $\phi \subset \{s : \mathsf{box}(\tau)\}$; together with that ϕ is met on state (r,m), readable$(r(s))$ can be shown.

Therefore, there is a state (r',m') that (r,m) can step to because of the execution of the instruction $\mathtt{ld}\ s,d$. By the semantics of $\mathtt{ld}$, condition ϕ' can be proved to hold on state (r',m') and the control of (r',m') is at label $l+4$. Because label $l+4$ is of type $\mathsf{codeptr}(\phi')$ to approximation k, state (r',m') is safe within k steps. Taking the step by $\mathtt{ld}$ into account, the first state (r,m) is safe within $k+1$ steps.

Proofs for other typing rules of instructions follow the same scheme. In the case of control-transfer instructions like $\mathtt{ba}$, the assumption (2) about Γ guarantees that it is safe to jump to the destination address.

6 Implementation

Our work is a part of the Foundational Proof-Carrying Code project [12], which includes a compiler from ML to Sparc, a typed assembly language called LTAL [6], and semantic proofs of typing rules. We have successfully given models to typing judgments in LTAL based on proof techniques in this paper, with a proved type-safety theorem and nearly complete semantic proofs of the typing rules.

All the proofs are written and machine-checked in a theorem-proving system — Twelf [13]: 1949 lines of axioms of the logic, arithmetic, and the specification of the SPARC machine; 23562 lines of lemmas of logic and arithmetic, theories of mathematical foundations such as sets and lists; 72036 lines of lemmas about conventions of machine states and semantic model of types; 18895 lines of lemmas about machines

instructions and the LTAL calculus. Most of the incomplete semantic proofs are about machines instruction semantics.

To focus the presentation on the essential ideas, we have not shown many features of our actual implementation. In this paper we have used immutable reference types to describe data structures in the memory and proved that programs can safely access these data structures. Our implementation can also deal with mutable references [10] so that programs can safely update data structures in the memory. Allocation of new data structures in the memory have also been taken into account by following the allocation model of SML/NJ. However, we do not currently support either explicit deallocation (`free`) or garbage collection. LTAL is also more expressive, including type variables, quantified types and condition-code types. It can also type-check position-independent code and multi-instruction basic blocks. Our semantic model supports all these features.

7 Related Work

There has been some work in the program-verification community to use a semantic approach to prove Hoare-logic rules as lemmas in an underlying logic [7, 14, 15]. Such proofs have been mechanized in HOL [14]. These works use first-order or higher-order logic to specify the invariants and have the difficulty that loop invariants cannot be derived automatically, so the approach does not scale to large programs.

Hamid *et al.* [16] and Crary [17] use a syntactic approach to prove type soundness. The first stage of their approach develops a typed assembly language, which is also given an operational semantics on an abstract machine. Then syntactic type-soundness theorems are proved on this abstract machine following the scheme presented by Wright and Felleisen [18]. The second stage uses a simulation relation between the abstract machine and the concrete architecture. The syntactic approach does not need the building of denotational semantics for complicated types such as recursive types, and it can also have machine-checkable proofs. However, the simulation step between the abstract machine and a full-fledged architecture is not a trivial task.

In some sense, the problem we solve in this paper is to give models to unstructured programs with goto statements and labels. There has been work by de Bruin [19] to give goto statements a domain-theoretic model. His approach to prove that code respects invariants is by approximations over code behavior, i.e. any k-th approximations of code behavior respects invariants. In our approach, we use types as invariants and do approximations over types, i.e. code respects any k-th approximations of types.

In conclusion, we have shown how to build end-to-end foundational safety proofs of programs on a real machine. We have constructed a semantic model for typing judgments of a typed assembly language and given proofs for both the type-safety theorem and typing rules. Our approach allows a typing derivation to be interpreted as a machine-checkable safety proof at the machine level.

References

1. Necula, G.: Proof-carrying code. In: 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New York, ACM Press (1997) 106–119

2. Morrisett, G., Walker, D., Crary, K., Glew, N.: From System F to typed assembly language. ACM Trans. on Programming Languages and Systems **21** (1999) 527–568
3. Morrisett, G., Crary, K., Glew, N., Grossman, D., Samuels, R., Smith, F., Walker, D., Weirich, S., Zdancewic, S.: TALx86: A realistic typed assembly language. In: Second ACM SIGPLAN Workshop on Compiler Support for System Software, Atlanta, GA (1999) 25–35 INRIA Technical Report 0288, March 1999.
4. League, C., Shao, Z., Trifonov, V.: Precision in practice: A type-preserving Java compiler. In: Proc. Int'l. Conf. on Compiler Construction. (2003)
5. Colby, C., Lee, P., Necula, G.C., Blau, F., Cline, K., Plesko, M.: A certifying compiler for Java. In: Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00), New York, ACM Press (2000)
6. Chen, J., Wu, D., Appel, A.W., Fang, H.: A provably sound TAL for back-end optimization. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '03). (2003) 208–219
7. Schmidt, D.A.: Denotational Semantics: A Methodology for Language Development. Allyn and Bacon, Boston (1986)
8. Appel, A.W., Felty, A.P.: A semantic model of types and machine instructions for proof-carrying code. In: POPL '00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press (2000) 243–253
9. Appel, A.W., McAllester, D.: An indexed model of recursive types for foundational proof-carrying code. ACM Trans. on Programming Languages and Systems **23** (2001) 657–683
10. Ahmed, A., Appel, A.W., Virga, R.: A stratified semantics of general references embeddable in higher-order logic. In: 17th Annual IEEE Symposium on Logic in Computer Science (LICS 2002). (2002) 75–86
11. Michael, N.G., Appel, A.W.: Machine instruction syntax and semantics in higher-order logic. In: 17th International Conference on Automated Deduction, Berlin, Springer-Verlag (2000) 7–24 LNAI 1831.
12. Appel, A.W.: Foundational proof-carrying code. In: Symposium on Logic in Computer Science (LICS '01), IEEE (2001) 247–258
13. Pfenning, F., Schürmann, C.: System description: Twelf — a meta-logical framework for deductive systems. In: The 16th International Conference on Automated Deduction, Berlin, Springer-Verlag (1999)
14. Gordon, M.: Mechanizing programming logics in higher-order logic. In G.M. Birtwistle, P.A. Subrahmanyam, eds.: Current Trends in Hardware Verification and Automatic Theorem Proving, Banff, Canada, Springer-Verlag, Berlin (1988) 387–439
15. Wahab, M.: Verification and abstraction of flow-graph programs with pointers and computed jumps. Research Report CS-RR-354, Department of Computer Science, University of Warwick, Coventry, UK (1998)
16. Hamid, N., Shao, Z., Trifonov, V., Monnier, S., Ni, Z.: A syntactic approach to foundational proof-carrying code. In: Proc. 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02). (2002) 89–100
17. Crary, K.: Toward a foundational typed assembly language. In: The 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press (2003) 198–212
18. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation **115** (1994) 38–94
19. de Bruin, A.: Goto statements: Semantics and deduction systems. Acta Informatica **15** (1981) 385–424