Foundational Proof-Carrying Code

Andrew W. Appel*
Princeton University

Abstract

Proof-carrying code is a framework for the mechanical verification of safety properties of machine language programs, but the problem arises of quis custodiat ipsos custodes*—who will verify the verifier itself? Foundational proof-carrying code is verification from the smallest possible set of axioms, using the simplest possible verifier and the smallest possible runtime system. I will describe many of the mathematical and engineering problems to be solved in the construction of a foundational proof-carrying code system.*

1 Introduction

When you obtain a piece of software – a shrink-wrapped application, a browser plugin, an applet, an OS kernel extension – you might like to ascertain that it's safe to execute: it accesses only its own memory and respects the private variables of the API to which it's linked. In a Java system, for example, the byte-code verifier can make such a guarantee, but only if there's no bug in the verifier itself, or in the just-in-time compiler, or the garbage collector, or other parts of the Java virtual machine (JVM).

If a compiler can produce Typed Assembly Language (TAL) [14], then just by type-checking the low-level representation of the program we can guarantee safety – but only if there's no bug in the typing rules, or in the type-checker, or in the assembler that translates TAL to machine language. Fortunately, these components are significantly smaller and simpler than a Java JIT and JVM.

Proof-carrying code (PCC) [15] constructs and verifies a mathematical proof about the machine-language program itself, and this guarantees safety – but only if there's no bug in the verification-condition generator, or in the logical axioms, or the typing rules, or the proof-checker.

What is the minimum possible size of the components that must be trusted in a PCC system? This is like asking, what is the minimum set of axioms necessary to prove a particular theorem? A foundational proof is one from just the foundations of mathematical logic, without additional axioms and assumptions; foundational proof-carrying code is PCC with trusted components an order of magnitude smaller than previous PCC systems.

Conventional proof-carrying code. Necula [15] showed how to specify and verify safety properties of machine-language programs to ensure that an untrusted program does no harm – does not access unauthorized resources, read private data, or overwrite valuable data. The provider of a PCC program must provide both the executable code and a machine-checkable proof that this code does not violate the safety policy of the host computer. The host computer does not run the given code until it has verified the given proof that the code is safe.

In most current approaches to PCC and TAL [15, 14], the machine-checkable proofs are written in a logic with a built-in understanding of a particular type system. More formally, type constructors appear as primitives of the logic and certain lemmas about these type constructors are built into the verification system. The semantics of the type constructors and the validity of the lemmas concerning them are proved rigorously but without mechnical verification by the designers of the PCC verification system. We will call this type-specialized PCC.

A PCC system must understand not only the language of types, but also the machine language for a particular machine. Necula's PCC systems [15, 7] use a verification-condition generator (VCgen) to derive, for each program, a *verification condition* – a logical formula that if true guarantees the safety of the program. The code producer must prove, and the code consumer must check the proof of, the verification condition. (Both producer and consumer independently run the VCgen to derive the right formula for the given program.)

The VCgen is a fairly large program (23,000 lines of C in the Cedilla Systems implementation [7]) that examines the machine instructions of the program, expands the substitutions of its machine-code Hoare logic, examines the formal parameter declarations to derive function precon-

*This research was supported in part by DARPA award F30602-99-1-0519 and by National Science Foundation grant CCR-9974553.

0-7695-1281-X/01 $10.00 © 2001 IEEE

ditions, and examines result declarations to derive post-conditions. A bug in the VCgen will lead to the wrong formula being proved and checked.

The soundness of a PCC system's typing rules and VCgen can, in principle, be proved as a metatheorem. Human-checked proofs of type systems are almost tractable; the appendices of Necula's thesis [16] and Morrisett et al.'s paper [14] contain such proofs, if not of the actual type systems used in PCC systems, then of their simplified abstractions. But constructing a mechanically-checkable correctness proof of a full VCgen would be a daunting task.

Foundational PCC. Unlike type-specialized PCC, the foundational PCC described by Appel and Felty [3] avoids any commitment to a particular type system and avoids using a VC generator. In foundational PCC the operational semantics of the machine code is defined in a logic that is suitably expressive to serve as a foundation of mathematics. We use higher-order logic with a few axioms of arithmetic, from which it is possible to build up most of modern mathematics. The operational semantics of machine instructions [12] and safety policies [2] are easily defined in higher-order logic. In foundational PCC the code provider must give both the executable code plus a proof in the foundational logic that the code satisfies the consumer's safety policy. The proof must explicitly define, down to the foundations of mathematics, all required concepts and explicitly prove any needed properties of these concepts.

Foundational PCC has two main advantages over type-specialized PCC — it is more flexible and more secure. Foundational PCC is more flexible because the code producer can "explain" a novel type system or safety argument to the code consumer. It is more secure because the trusted base can be smaller: its trusted base consists only of the foundational verification system together with the definition of the machine instruction semantics and the safety policy. A verification system for higher-order logic can be made quite small [10, 17].

In our research project at Princeton University (with the help of many colleages elsewhere) we are building a foundational PCC system, so that we can specify and automatically prove and check the safety of machine-language programs. In this paper I will explain the components of the system.

2 Choice of logic and framework

To do machine-checked proofs, one must first choose a logic and a logical framework in which to manipulate the logic. The logic that we use is Church's higher-order logic with axioms for arithmetic; we represent our logic, and check proofs, in the LF metalogic [10] implemented in the Twelf logical framework [18]. We have chosen LF because it naturally produces proof objects that we can send to a "consumer."

The Twelf system allows us to specify constructors of our object logic. Our object logic has types `tp`; its primitive types are propositions `o` and numbers `num`; there is an `arrow` constructor to build function types, and `pair` to build tuples. For any object-logic type T, object-logic expressions of that type have metalogical type `tm` T. Finally, for any formula A we can talk about proofs of A, which belong to the metalogical type `pf` (A).

```
tp   : type.
tm   : tp -> type.
o: tp.   num: tp.
arrow: tp -> tp -> tp.
   %infix right 14 arrow.
pair: tp -> tp -> tp.
pf   : tm o -> type.
```

We have object-logic constructors `lam` (to construct functions), `@` (to apply a function to an argument, written infix), `imp` (logical implication), and `forall` (universal quantification):

```
lam: (tm T1 -> tm T2) -> tm (T1 arrow T2).
@  : tm (T1 arrow T2) -> tm T1 -> tm T2.
  %infix left 20 @.
imp    : tm o -> tm o -> tm o.
  %infix right 10 imp.
forall : (tm T -> tm o) -> tm o.
```

The trick of using `lam` and `@` to coerce between metalogical functions `tm T1 -> tm T2` and object-logic functions `tm (T1 arrow T2)` is described by Harper, Honsell, and Plotkin [10]. We need object-logic functions so that we can quantify over them using `forall`; that is, the type of `F` in `forall [F]` *`predicate`*`(F)` must be `tm T` for some `T` such as `num arrow num`, but cannot be `tm T1 -> tm T2`.

We have introduction and elimination rules for these constructors (rules for pairing omitted here):

```
beta_e: {P: tm T -> tm o}
         pf(P (lam F @ X)) -> pf(P (F X)).
beta_i: {P: tm T -> tm o}
         pf(P (F X)) -> pf(P (lam F @ X)).

imp_i: (pf A -> pf B) -> pf (A imp B).
imp_e: pf (A imp B) -> pf A -> pf B.
```

```
forall_i:
      ({X:tm T}pf(A X)) -> pf(forall A).
forall_e:
      pf(forall A) -> {X:tm T}pf(A X).
not_not_e: pf ((B imp forall [A] A)
                 imp forall [A] A)
           -> pf B.
```

Our proofs don't need extensionality or the general axiom of choice.

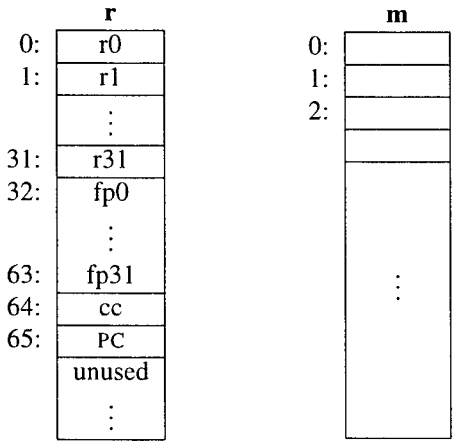
Once we have defined the constructors of the logic, we can define lemmas and new operators as definitions in Twelf:

```
and : tm o -> tm o -> tm o =
  [A][B]
  forall [C] (A imp B imp C) imp C.

  %infix right 12 and.

and_i    : pf A -> pf B -> pf (A and B) =
 [p1: pf A][p2: pf B]
 forall_i [c: tm o]
 imp_i [p3] imp_e (imp_e p3 p1) p2.

and_e1   : pf (A and B) -> pf A =
  [p1: pf (A and B)]
  imp_e (forall_e p1 A)
  (imp_i [p2: pf A] imp_i [p3: pf B] p2).
```

Of course, the defined lemmas are checked by machine (the Twelf type checker), and need not be trusted in the same way that the core inference rules are. Our interactive tutorial [1] provides an informal introduction to our object logic.

3 Specifying machine instructions

We start by modeling a specific von Neumann machine, such as the Sparc or the Pentium. A machine state comprises a *register bank* and a *memory*, each of which is a function from integers (addresses) to integers (contents). Every register of the instruction-set architecture (ISA) must be assigned a number in the register bank: the general registers, the floating-point registers, the condition codes, and the program counter. Where the ISA does not specify a number (such as for the PC) we use an arbitrary index:



A single step of the machine is the execution of one instruction. We can specify instruction execution by giving a step relation $(r,m) \mapsto (r',m')$ that describes the relation between the prior state (r,m) and the state (r',m') of the machine after execution.

For example, to describe the instruction $r_1 \leftarrow r_2 + r_3$ we might start by writing,

$$(r,m) \mapsto (r',m') \equiv$$
$$r'(1) = r(2) + r(3) \wedge (\forall x \neq 1.\, r'(x) = r(x)) \wedge m' = m$$

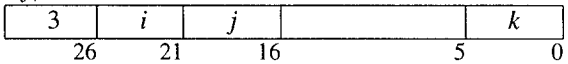
In fact, we can define $\mathrm{add}(i,j,k)$ as this predicate on four arguments (r,m,r',m'):

$$\mathrm{add}(i,j,k) =$$
$$\lambda r,m,r',m'.\ r'(i) = r(j) + r(k) \wedge (\forall x \neq i.\, r'(x) = r(x)) \wedge m' = m$$

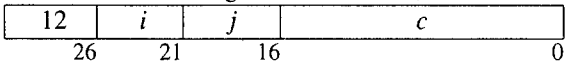
Similarly, we can define the instruction $r_i \leftarrow m[r_j + c]$ as

$$\mathrm{load}(i,j,c) =$$
$$\lambda r,m,r',m'.\ r'(i) = m(r(j)+c) \wedge (\forall x \neq i.\, r'(x) = r(x)) \wedge m' = m$$

But we must also take account of instruction fetch and decoding. Suppose, for example, that the add instruction is encoded as a 32-bit word, containing a 6-bit field with opcode 3 denoting *add*, a 5-bit field denoting the destination register i, and 5-bit fields denoting the source registers j,k:



The load instruction might be encoded as,



Then we can say that some number w decodes to an instruction *instr* iff,

$$
\begin{aligned}
&\text{decode}(w, instr) \equiv \\
&\quad (\exists i, j, k. \\
&\qquad 0 \le i < 2^5 \ \wedge\ 0 \le j < 2^5 \ \wedge\ 0 \le k < 2^5 \ \wedge \\
&\qquad w = 3 \cdot 2^{26} + i \cdot 2^{21} + j \cdot 2^{16} + k \cdot 2^0 \ \wedge \\
&\qquad instr = \text{add}(i, j, k)) \\
&\quad \vee (\exists i, j, c. \\
&\qquad 0 \le i < 2^5 \ \wedge\ 0 \le j < 2^5 \ \wedge\ 0 \le c < 2^{16} \ \wedge \\
&\qquad w = 12 \cdot 2^{26} + i \cdot 2^{21} + j \cdot 2^{16} + c \cdot 2^0 \ \wedge \\
&\qquad instr = \text{load}(i, j, \text{sign-extend}(c))) \\
&\quad \vee \ \ldots
\end{aligned}
$$

with the ellipsis denoting the many other instructions of the machine, which must also be specified in this formula.

Neophytos Michael and I have shown [12] how to scale this idea up to the instruction set of a real machine. Real machines have large but semiregular instruction sets; instead of a single global disjunction, the decode relation can be factored into operands, addressing modes, and so on. Real machines don't use integer arithmetic, they use modular arithmetic, which can itself be specified in our higher-order logic. Some real machines have multiple program counters (e.g., Sparc) or variable-length instructions (e.g., Pentium), and these can also be accommodated.

Our description of the decode relation is heavily factored by higher-order predicates (this would not be possible without higher-order logic). We have specified the execution behavior of a large subset of the Sparc architecture (without register windows or floating-point). For PCC, it is sufficient to specify a subset of the machine architecture; any unspecified instruction will be treated by the safety policy as illegal, which may be inconvenient for compilers that want to generate that instruction, but which cannot compromise safety.

Our Sparc specification has two components, a "syntactic" part (the decode relation) and a semantic part (the definitions of *add*, *load*, etc.). The syntactic part is derived from a 151-line specification written in the SLED language of the New Jersey Machine-Code Toolkit [19]; our translator expands this to 1035 lines of higher-order logic, as represented in Twelf; but we believe that a more concise and readable translation would produce only 500–600 lines. The semantic part is about 600 lines of logic, including the definition of modular arithmetic.

4 Specifying safety

Our step relation $(r, m) \mapsto (r', m')$ is deliberately partial; some states have no successor state. In these states the program counter $r(\text{PC})$ points to an illegal instruction. Now we will proceed to make it even more partial, by defining as illegal those instructions that violate our safety policy.

For example, suppose we wish to specify a safety policy that "only *readable* addresses will be loaded," where the predicate *readable* is given some suitable definion such as

$$\text{readable}(x) = 0 \le x < 1000$$

(see Appel and Felten [2] for descriptions of security policies that are more interesting than this one).

We can add a new conjunct to the semantics of the *load* instruction,

$$
\begin{aligned}
&\text{load}(i, j, c) \ = \\
&\quad \lambda r, m, r', m'.\ r'(i) = m(r(j) + c) \\
&\qquad\qquad \wedge\ \text{readable}(r(j) + c) \\
&\qquad\qquad \wedge\ (\forall x \neq i.\, r'(x) = r(x)) \ \wedge\ m' = m.
\end{aligned}
$$

Now, in a machine state where the program counter points to a load instruction that violates the safety policy, our step relation $\mapsto$ does not relate this state to any successor state (even though the real machine "knows how" to execute it).

Using this partial step relation, we can define safety; a given state is safe if, for any state reachable in the Kleene closure of the step relation, there is a successor state:

$$
\begin{aligned}
&\text{safe-state}(r, m) \ = \\
&\quad \forall r', m'.\ (r, m \mapsto^* r', m') \ \Rightarrow\ \exists r'', m''.\ r', m' \mapsto r'', m''
\end{aligned}
$$

A program is just a sequence of integers (representing machine instructions); we say that a program p is loaded at a location *start* in memory m if

$$\text{loaded}(p, m, start) \ = \ \forall i \in \text{dom}(p).\ m(i + start) = p(i)$$

Finally (assuming that programs are written in position-independent code), a program is *safe* if, no matter where we load it in memory, we get a safe state:

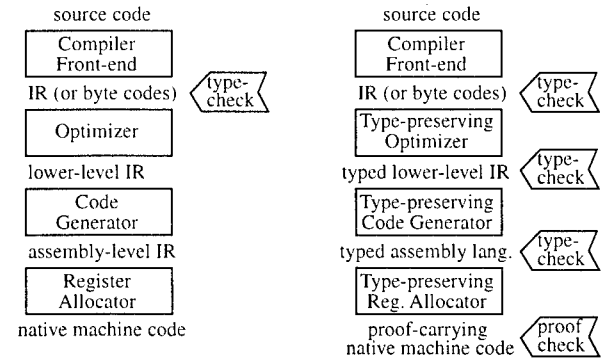
$$
\begin{aligned}
&\text{safe}(p) \ = \\
&\quad \forall r, m, start.\ \ \text{loaded}(p, m, start) \ \wedge\ r(\text{PC}) = start \ \Rightarrow \\
&\qquad\qquad \text{safe-state}(r, m)
\end{aligned}
$$

The important thing to notice about this formulation is that *there is no verification-condition generator*. The syntax and semantics of machine instructions, implicit in a VCgen, have been made explicit – and much more concise – in the step relation. But the Hoare logic of machine instructions and typing rules for function parameters, also implicit in a VCgen, must now be proved as lemmas – about which more later.

5 Proving safety

In a sufficiently expressive logic, as we all know, proving theorems can be a great deal more difficult than merely stating them – and higher-order logic is certainly expressive. For guidance in proving safety of machine-language programs we should not particularly look to previous work in formal verification of program correctness. Instead, we should think more of type checking: automatic proofs of decidable safety properties of programs.

The key advances that makes it possible to generate proofs automatically are *typed intermediate languages* [11] and *typed assembly language* [14]. Whereas conventional compilers type-check the source program, then throw away the types (using the lambda-calculus principle of *erasure*) and then transform the program through progressively lower-level intermediate representations until they reach assembly language and then machine language, a type-preserving compiler uses typed intermediate languages at each level. If the program type-checks at a low level, then it is safe, regardless of whether the previous (higher-level) compiler phases might be buggy on some inputs. As the program is analyzed into smaller pieces at the lower levels, the type systems become progressively more complex, but the type theory of the 1990's is up to the job of engineering the type systems.



Conventional Compiler **Type-preserving Compiler**

TAL was originally designed to be used in a certifying compiler, but one that certifies the assembly code and uses a trusted assembler to translate to machine code. But we can use TAL to help generate proofs in a PCC system that directly verifies the machine code. In such a system, the proofs are typically by induction, with induction hypotheses such as, "whenever the program-counter reaches location l, the register 3 will be a pointer to a pair of integers." These local invariants can be generated from the TAL formulation of the program, but in a PCC system they can be checked in machine code without needing to trust the assembler.

Typing rules for machine language. In important insight in the development of PCC is that one can write type-inference rules for machine language and machine states. For example, Necula [15] used rules such as

$$\frac{m \vdash x : \tau_1 \times \tau_2}{m \vdash m(x) : \tau_1 \wedge m(x+1) : \tau_2}$$

meaning that if x has type $\tau_1 \times \tau_2$ in memory m – meaning that it is a pointer to a boxed pair – then the contents of location x will have type τ_1 and the contents of location $x+1$ will have type τ_2.

Proofs of safety in PCC use the local induction hypotheses at each point in the program to prove that the program is typable. This implies, by a type-soundness argument, that the program is therefore safe.

If the type system is given by syntactic inference rules, the proof of type soundness is typically done by syntactic subject reduction – one proves that each step of computation preserves typability and that typable states are safe. The proof involves structural induction over typing derivations. In conventional PCC, this proof is done in the metatheory, by humans.

In foundational PCC we wish to include the type-soundness proof inside the proof that is transmitted to the code consumer because (1) it's more secure to avoid reliance on human-checked proofs and (2) that way we avoid restricting the protocol to a single type system. But in order to do a foundational subject-reduction theorem, we would need to build up the mathematical machinery to manipulate typing derivations as syntactic objects, all represented inside our logic using foundational mathematical concepts – sets, pairs, and functions. We would need to do case analyses over the different ways that a given type judgement might be derived. While this can all be done, we take a different approach to proving that typability implies safety.

We take a semantic approach. In a semantic proof one assigns a meaning (a semantic truth value) to type judgements. One then proves that if a type judgement is true then the typed machine state is safe. One further proves that the type inference rules are sound, i.e., if the premises are true then the conclusion is true. This ensures that derivable type judgements are true and hence typable machine states are safe.

The semantic approach avoids formalizing syntactic type expressions. Instead, one formalizes a type as a set of semantic values. One defines the operator $\times$ as a function taking two sets as arguments and returning a set. The

above type inference rule for pair projection can then be replaced by the following semantic lemma in the foundational proof:

$$\frac{\models x :_m \tau_1 \times \tau_2}{\models m(x) :_m \tau_1 \wedge m(x+1) :_m \tau_2}$$

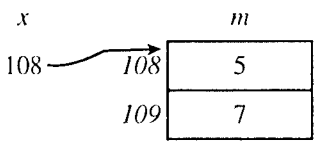
Although the two forms of the application type-inference rule look very similar they are actually significantly different. In the second rule τ_1 and τ_2 range over semantic sets rather than type expressions. The relation $\models$ in the second version is defined directly in terms of a semantics for assertions of the form $x :_m \tau$. The second "rule" is actually a lemma to be proved while the first rule is simply a part of the definition of the syntactic relation $\vdash$. For the purposes of foundational PCC, we view the semantic proofs as preferable to syntactic subject-reduction proofs because they lead to shorter and more manageable foundational proofs. The semantic approach avoids the need for any formalization of type expressions and avoids the formalization of proofs or derivations of type judgements involving type expressions.

5.1 Semantic models of types

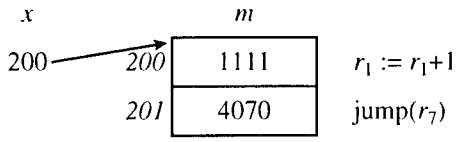
Building semantic models for type systems is interesting and nontrivial. In a first attempt, Amy Felty and I [3] were able to model a pure-functional (immutable datatypes) call-by-value language with records, address arithmetic, polymorphism and abstract types, union and intersection types, continuations and function pointers, and covariant recursive types.

Our simplest semantics is set-theoretic: a type is a set of values. But what is a value? It is not a syntactic construct, as in lambda-calculus; on a von Neumann machine we wish to use a more natural representation of values that corresponds to the way procedures and data structures are represented in practice. This way, our type theory can match reality without a layer of simulation in between. We can represent a value as a pair (m,x), where m is a memory and x is an integer (typically representing an address).

To represent a pointer data structure that occupies a certain portion of the machine's memory, we let x be the root address of that structure. For example, the boxed pair of integers $\langle 5,7\rangle$ represented at address 108 would be represented as the value $(\{108 \mapsto 5, 109 \mapsto 7\}, 108)$.



To represent a function value, we let x be the entry address of the function; here is the function $f(x) = x+1$, assuming that arguments and return results are passed in register 1:



This model of values would be sufficient in a semantics of statically allocated data structures, but to have dynamic heap allocation we must be able to indicate the set a of allocated addresses, such that any modification of memory outside the allocated set will not disturb already allocated values. A *state* is a pair (a,m), and a value is a pair $((a,m),x)$ of state and root-pointer. The alloc set a is virtual: it is not directly represented at run time, but is existentially quantified.

Limitations. In the resulting semantics [3] we could model heap allocation, but we could not model mutable record-fields; and though our type system could describe

```
datatype 'a list = nil
                 | :: of 'a * 'a list
```

we could not handle recursions where the type being defined occurs in a negative (contravariant) position, as in

```
datatype exp = APP of exp * exp
             | LAM of [exp] -> exp
```

where the boxed occurrence of `exp` is a negative occurrence. Contravariant recursion is occasionally useful in ML, but it is the very essence of object-oriented programming, so these limitations (no mutable fields, no contravariant recursion) are quite restrictive.

5.2 Indexed model of recursive types

In more recent work, David McAllester and I have shown how to make an "indexed" semantic model that can describe contravariant recursive types [4]. Instead of saying that a type is a set of values, we say that it is a set of pairs $\langle k,v\rangle$ where k is an approximation index and v is a value. The judgement $\langle k,v\rangle \in \tau$ means, "v approximately has type τ, and any program that runs for fewer than k instructions can't tell the difference." The indices k allow the construction of a well founded recursion, even when modeling contravariant recursive types.

The type system works both for von Neumann machines and for λ-calculus; here I will illustrate the latter. We define a *type* as a set of pairs $\langle k,v\rangle$ where k is a non-negative integer and v is a value and where the set τ is

such that if $\langle k, v\rangle \in \tau$ and $0 \leq j \leq k$ then $\langle j, v\rangle \stackrel{\cdot}{\in} \tau$. For any closed expression e and type τ we write $e :_k \tau$ if e is safe for k steps and if whenever $e \mapsto^j v$ for some value v with $j < k$ we have $\langle k-j, v\rangle \in \tau$; that is,

$$e :_k \tau \equiv \forall j \forall e'.\ 0 \leq j < k \wedge e \mapsto^j e' \wedge \mathrm{nf}(e') \Rightarrow \langle k-j, e'\rangle \in \tau$$

where $\mathrm{nf}(e')$ means that e' is a normal form — has no successor in the call-by-value small-step evaluation relation.

We start with definitions for the sets that represent the types:

$$\begin{array}{rcl} \bot & \equiv & \{\} \\ \top & \equiv & \{\langle k, v\rangle \mid k \geq 0\} \\ \textbf{int} & \equiv & \{\langle k, \mathbf{0}\rangle, \langle k, \mathbf{1}\rangle, \ldots \mid k \geq 0\} \\ \tau_1 \times \tau_2 & \equiv & \{\langle k, (v_1, v_2)\rangle \mid \forall j < k.\ \langle j, v_1\rangle \in \tau_1 \wedge \langle j, v_2\rangle \in \tau_2\} \\ \sigma \to \tau & \equiv & \{\langle k, \lambda x.e\rangle \mid \forall j < k \forall v.\ \langle j, v\rangle \in \sigma \Rightarrow e[v/x] :_j \tau\} \\ \mu F & \equiv & \{\langle k, v\rangle \mid \langle k, v\rangle \in F^{k+1}(\bot)\} \end{array}$$

Next we define what is meant by a typing judgement. Given a mapping Γ from variables to types, we write $\Gamma \vDash_k e : \alpha$ to mean that

$$\forall \sigma.\, \sigma :_k \Gamma \Rightarrow \sigma(e) :_k \alpha$$

where $\sigma(e)$ is the result of replacing the free variables in e with their values under substitution σ. To drop the index k, we define

$$\Gamma \vDash e : \alpha \equiv \forall k.\, \Gamma \vDash_k e : \alpha$$

Soundness theorem: It is trivial to prove from these definitions that if $\vDash e : \alpha$ and $e \mapsto^* e'$ then e' is not stuck, that is, $e' \mapsto e''$.

Well founded type constructors. We define the notion of a well founded type constructor. Here I will not give the formal definition, but state the informal property that if F is well founded and $x : F(\tau)$, then to extract from x a value of type τ, or to apply x to a value of type τ, must take at least one execution step. The constructors $\times$ and $\to$ are well founded.

Typing rules. Proofs of theorems such as the following are not too lengthy:

$$\frac{\Gamma \vDash \pi_1(e) : \tau_1 \qquad \Gamma \vDash \pi_2(e) : \tau_2}{\Gamma \vDash e : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vDash e : \tau_1 \times \tau_2}{\Gamma \vDash \pi_1(e) : \tau_1}$$

$$\frac{\Gamma \vDash e_1 : \alpha \to \beta \qquad \Gamma \vDash e_2 : \alpha}{\Gamma \vDash e_1 e_2 : \beta}$$

Finally, for any well founded type-constructor F, we have equirecursive types: $\mu F = F(\mu F)$.

Our paper [4] proves all these theorems and shows the extension of the result to types and values on von Neumann machines.

5.3 Mutable fields

Our work on mutable fields is still in a preliminary stage. Amal Ahmed, Roberto Virga, and I are investigating the following idea. Our semantics of immutable fields viewed a "state" as a pair (a, m) of a memory m and a set a of allocated addresses. To allow for the update of existing values, we enhance a to become a finite map from locations to types. The type $a(l)$ at some location l specifies what kinds of updates at that location will preserve all existing typing judgements. Then, as before, a type is a predicate on states (a, m) and root-pointers x of type integer. In our object logic, we would write the types of these logical objects as,

$$\begin{array}{rcl} \mathit{allocset} & = & \mathit{num} \xrightarrow{\text{fin}} \mathit{type} \\ \mathit{value} & = & \mathit{allocset} \times \mathit{memory} \times \mathit{num} \\ \mathit{type} & = & \mathit{num} \times \mathit{value} \ \to \ o \end{array}$$

The astute reader will notice that the metalogical type of "type" is recursive, and in a way that has an inconsistent cardinality: the set of types must be bigger than itself. This problem had us stumped for over a year, but we now have a tentative solution that replaces the type (in the allocset) with the Gödel number of a type. We hope to report on this result soon; we are delayed by our general practice of machine-checking our proofs in Twelf before submitting papers for publication, which in this case has saved us from some embarrassment.

5.4 Typed machine language

Morrisett's typed assembly language [14] is at too high a level to do proof-carrying code directly. Kedar Swadi, Gang Tan, Roberto Virga, and I have been designing a lower-level representation, called *typed machine language*, that will serve as the interface between compilers and our prover. In fact, we hope that a clean enough definition of this language will shift most of the work from the prover to the compiler's type-checker.

In order to avoid overspecializing the typed machine language (TML) with language-specific constructs such as records and disjoint-union variants, our TML will use very low-level typing primitives such as union types, intersection types, offset (address-arithmetic) types, and de-

pendent types. This will make type-checking of TML difficult; we will need to assume that each compiler will have a source language with a decidable type system, and that translation of terms (and types) will yield a witness to the type-checking of the resultant TML representation.

Abstract machine instructions. One can view machine instructions at many levels of abstraction:

1. At the lowest level, an instruction is just an integer, an opcode encoding.

2. At the next level, it implements a relation on raw machine states $(r,m) \mapsto (r',m')$.

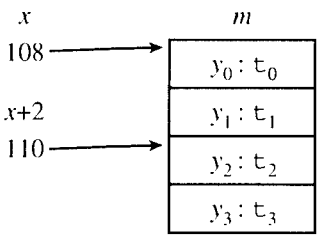
3. At a higher level, we can say that the Sparc *add* instruction implements a machine-independent notion of *add*, and similarly for other instruction.

4. Then we can view *add* as manipulating not just registers, but local variables (which may be implemented in registers or in the activation record).

5. We can view this instruction as one of various typed instructions on typed values; in the usual view, *add* has type $\text{int} \times \text{int} \rightarrow \text{int}$, but the address-arithmetic *add* has type

   $$(\tau_0 \times \tau_1 \times \ldots \times \tau_n) \times \text{const}(i) \rightarrow (\tau_i \times \tau_{i-1} \times \ldots \times \tau_n)$$

   for any i.

6. Finally, we can specialize this typed *add* to the particular context where some instance of it appears, for example by instantiating the i, n, and τ_i in the previous example.

Abstraction level 1 is used in the statement of the theorem (safety of a machine-language program p). Abstraction level 5 is implicitly used in conventional proof-carrying code [15]. Our ongoing research involves finding semantic models for each of these levels, and then proving lemmas that can convert between assertions at the different levels.

Hoare logic. In reasoning about machine instructions at a higher level of abstraction, notions from Hoare logic are useful: preconditions, postconditions, and substition. Without adding any new axioms, we can define a notion of predicates on states to serve as preconditions and postconditions, and substitution as a relation on predicates. But this can rapidly become inefficient, leading to proofs that are quadratic or exponential in size. Kedar Swadi, Roberto Virga, and I have taken some steps in lemmatizing substitution so that proofs don't blow up [5]; interesting related work has been done in Compaq SRC's extended static checker [9].

Software engineering practices. We define all of these abstraction levels in order to modularize our proofs. Since our approach to PCC shifts most of the work to the human prover of static, machine-checkable lemmas about the programming language's type system, we find it imperative to use the same software engineering practices in implementing proofs as are used in building any large system. The three most important practices are (1) abstraction and modularity, (2) abstraction and modularity, and (3) abstraction and modularity. At present, we have about thirty thousand lines of machine-checked proofs, and we would not be able to build and maintain the proofs without a well designed modularization.

6 Pruning the runtime system

Just as bugs in the compiler (of a conventional system) or the proof checker (of a PCC system) can create security holes, so can bugs in the runtime system: the garbage collector, debugger, marshaller/unmarshaller, and other components. An important part of research in Foundational PCC is to move components from the runtime system to the type-checkable user code. Then, any bugs in such components will either be detected by type-checking (or proof-checking), or will be type-safe bugs that may cause incorrect behavior but not insecure behavior.

Garbage collectors do two strange things that have made them difficult to express in a type-safe language: they allocate and deallocate arenas of memory containing many objects of different types, and they traverse (and copy) objects of arbitrary user-chosen types. Daniel Wang has developed a solution to these problems [22], based on the motto,

$$\text{Garbage collection} = \text{Regions} + \text{Intensional types}.$$

That is, the region calculus of Tofte and Talpin [20] can

be applied to the problem of garbage collection, as noticed in important recent work by Walker, Crary, and Morrisett [21]; to traverse objects of unknown type, the intensional type calculi of originally developed by Harper and Morrisett [11] can be applied. Wang's work covers the region operators and management of pointer sharing; related work by Monnier, Saha, and Shao [13] covers the intensional type system.

Other potentially unsafe parts of the runtime system are ad hoc implementations of *polytypic* functions – those that work by induction over the structure of data types – such as polymorphic equality testers, debuggers, and marshallers (a.k.a. serializers or picklers). Juan Chen and I have developed an implementation of polytypic primitives as a transformation on the typed intermediate representation in the SML/NJ compiler [6]. Like the λ_R transformation of Crary and Weirich [8] it allows these polytypic functions to be typechecked, but unlike their calculus, ours does not require dependent types in the typed intermediate language and is thus simpler to implement.

7 Conclusion

Our goal is to reduce the size of the trusted computing base of systems that run machine code from untrusted sources. This is an engineering challenge that requires work on many fronts. We are fortunate that during the last two decades, many talented scientists have built the mathematical infrastructure we need – the theory and implementation of logical frameworks and automated theorem provers, type theory and type systems, compilation and memory management, and programming language design. The time is ripe to apply all of these advances as engineering tools in the construction of safe systems.

References

[1] Andrew W. Appel. Hints on proving theorems in Twelf. www.cs.princeton.edu/~appel/twelf-tutorial, February 2000.

[2] Andrew W. Appel and Edward W. Felten. Models for security policies in proof-carrying code. Technical Report TR-636-01, Princeton University, March 2001.

[3] Andrew W. Appel and Amy P. Felty. 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*, pages 243–253. ACM Press, January 2000.

[4] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. Technical Report TR-629-00, Princeton University, October 2000.

[5] Andrew W. Appel, Kedar N. Swadi, and Roberto Virga. Efficient substitution in hoare logic expressions. In *4th International Workshop on Higher-Order Operational Techniques in Semantics (HOOTS 2000)*. Elsevier, September 2000. Electronic Notes in Theoretical Computer Science 41(3).

[6] Juan Chen and Andrew W. Appel. Dictionary passing for polytypic polymorphism. Technical Report CS-TR-635-01, Princeton University, March 2001.

[7] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Ken Cline, and Mark Plesko. A certifying compiler for Java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00)*. ACM Press, June 2000.

[8] Karl Crary and Stephanie Weirich. Flexible type analysis. In *ACM SIGPLAN International Conference on Functional Programming Languages*, September 1999.

[9] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 193–205. ACM Press, January 2001.

[10] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[11] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-second Annual ACM Symp. on Principles of Prog. Languages*, pages 130–141, New York, Jan 1995. ACM Press.

[12] Neophytos G. Michael and Andrew W. Appel. Machine instruction syntax and semantics in higher-order logic. In *17th International Conference on Automated Deduction*, June 2000.

[13] Stefan Monnier, Bratin Saha, and Zhong Shao. Principled scavenging. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*, page to appear, June 2001.

[14] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Trans. on Programming Languages and Systems*, 21(3):527–568, May 1999.

[15] George Necula. Proof-carrying code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, New York, January 1997. ACM Press.

[16] George Ciprian Necula. *Compiling with Proofs*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, September 1998.

[17] Frank Pfenning. Elf: A meta-language for deductive systems. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, pages 811–815, Nancy, France, June 1994. Springer-Verlag LNAI 814.

[18] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *The 16th International Conference on Automated Deduction*. Springer-Verlag, July 1999.

[19] Norman Ramsey and Mary F. Fernández. Specifying representations of machine instructions. *ACM Trans. on Programming Languages and Systems*, 19(3):492–524, May 1997.

[20] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Twenty-first ACM Symposium on Principles of Programming Languages*, pages 188–201. ACM Press, January 1994.

[21] David Walker, Karl Crary, and Greg Morrisett. Typed memory management via static capabilities. *ACM Trans. on Programming Languages and Systems*, 22(4):701–771, July 2000.

[22] Daniel C. Wang and Andrew W. Appel. Type-preserving garbage collectors. In *POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 166–178. ACM Press, January 2001.