A Semantic Model of Types and Machine Instructions for Proof-Carrying Code

Andrew W. Appel
Bell Laboratories * and Princeton University

Amy P. Felty†
Bell Laboratories

Abstract

Proof-carrying code is a framework for proving the safety of machine-language programs with a machine-checkable proof. Previous PCC frameworks have defined type-checking rules as part of the logic. We show a universal type framework for proof-carrying code that will allow a code producer to choose a programming language, prove the type rules for that language as lemmas in higher-order logic, then use those lemmas to prove the safety of a particular program. We show how to handle traversal, allocation, and initialization of values in a wide variety of types, including functions, records, unions, existentials, and covariant recursive types.

1 Introduction

When a host computer runs an untrusted program, the host may want some assurance that the program does no harm: does not access unauthorized resources, read private data, or overwrite valuable data. Proof-carrying code [Nec97] is a technique for providing such assurances. With PCC, the host – called the “code consumer” – specifies a safety policy, which tells under what conditions a word of memory may be read or written or how much of a resource (such as CPU cycles) may be used. The provider of the program – the “code producer” – must also provide a program-verification-style proof that the program satisfies these conditions. The host computer mechanically checks the proof before running the program.

Two significant advantages of PCC are that (1) these proofs can be performed on the native machine code, so that no unsoundness can be introduced in translation from the proved program to the program that will execute (in contrast, a JIT compiler can be 10^6 lines of code and therefore cannot possibly be free of bugs), and (2) for sufficiently simple safety policies and for programs compiled from type-safe source languages, the proofs can be constructed fully automatically. Unlike typed assembly language [MWCG98], PCC can use both types and dataflow to prove safety.

Necula has demonstrated two instances of PCC safety policies: one for a subset of C [Nec98] and another for an extremely restricted subset of ML [Nec97]. In our work we have generalized the approach and removed many restrictions:

1. Instead of building type-inference rules into the safety policy, we model the types via definitions from first principles, then prove the typing rules as lemmas. This makes the safety policy independent of the type system used by the program, so that programs compiled from different source languages can be sent to the same code consumer.

2. We show how to prove safe the allocation and initialization of data structures, not just the traversal of data.

3. We show how to handle a much wider variety of types, including records, tagged variants, first-class functions, first-class labels, existential types (i.e. abstract data types), union types, intersection types, and covariant recursive types.

4. We move the machine instruction semantics from the verification-condition generator to the safety policy; this simplifies the trusted computing base at the expense of complicating the proofs, which is the right trade-off to make.

---

†Current address: University of Ottawa, afelty@site.uottawa.ca

 upd(f, d, x, f') = def ∀z. (d = z ∧ f'(z) = x) ∨ (d ≠ z ∧ f'(z) = f(z))

add(d, s, c)(r, m, r', m') = def upd(r, d, r(s1) + r(s2), r') ∧ m = m'.
add(d, s, c)(r, m, r', m') = def upd(r, d, r(s) + c, r') ∧ m = m'
load(d, s, c)(r, m, r', m') = def readable(r(s) + c) ∧ upd(r, d, m(r(s) + c), r') ∧ m = m'
store(s1, s2, c)(r, m, r', m') = def writable(r(s2) + c) ∧ upd(m, r(s2) + c, r(s1), m') ∧ r = r'
jump(d, s, c)(r, m, r', m') = def 3α. upd(r, 17, r(s) + c, r') ∧ upd(r', d, r(17), r') ∧ m = m'
bgt(s1, s2, c)(r, m, r', m') = def (r(s1) > r(s2) ∧ upd(r, 17, r(17) + c, r') ∧ m = m') ∨ (r(s1) ≤ r(s2) ∧ r = r' ∧ m = m')
beq(s1, s2, c)(r, m, r', m') = def (r(s1) = r(s2) ∧ upd(r, 17, r(17) + c, r') ∧ m = m') ∨ (r(s1) ≠ r(s2) ∧ r = r' ∧ m = m')

\[\begin{array}{|c|c|c|}
\hline
\text{OPCODE} & \text{add} & \text{addi} \\
\hline
\text{d} & \text{s1} & \text{s2} \\
\hline
0 & 1 & d \\
\hline
\text{rd} & \text{rs1} & \text{rs2} \\
\hline
\text{rd} & \text{rd} & \text{rd} \\
\hline
\text{rd} & \text{rs} & \text{rs} \\
\hline
\text{rd} & \text{rd} & \text{rd} \\
\hline
\text{rs} & \text{rs} & \text{rd} \\
\hline
\text{if} & \text{rs2} & \text{rs2} \\
\hline
\text{if} & \text{rs2} & \text{rs2} \\
\hline
\end{array}\]

Figure 1: Semantic definition of machine instructions.

2 Example

To illustrate, we use an imaginary word-addressed machine with a simple instruction set and instruction encoding.

Example 1. We wish to verify the safety of the following short program.

\[\begin{align*}
100: & \quad 2210 \quad r_3 \leftarrow m(r_1) \\
101: & \quad 4070 \quad \text{jump}(r_7); \quad r_0 \leftarrow pc
\end{align*}\]

The code producer will provide the program (i.e., in this case the sequence of hexadecimal integers (2210,4070)) and a proof that if these integers are loaded at address 100 then it will be safe to jump there. The program’s precondition is that register 1 points to a record of two integers and register 7 points to a return address.

The logic comprises a set of inference rules and a set of axioms. The inference rules are standard natural-deduction rules of higher-order logic with natural number arithmetic and induction, augmented with just a few predicates and rules concerning the readability, writability, and “jumpability” of machine addresses, and the decoding and semantics of machine instructions.

We refer to the axioms as the safety policy. For Example 1, we will use the following safety policy:

1. \(∀v. (v ≥ 50) → \text{readable}(v)\)
2. \(∀v. (v ≥ 100) → \text{writable}(v)\)
3. \(∀r, m, (r(17) = r_0(7)) → \text{safe}(r, m)\)
4. \(r_0(1) = 50\)
5. \(r_0(17) = 100\)

Axioms 1 and 2 describe what addresses are readable and writable. Axioms 3–7 describe the initial state of the machine, comprising a register-bank \(r_0\) and a memory \(m_0\), each of which is a function from integers to integers. Axiom 3 says that any future state \(r, m\) whose program counter \(r(17)\) is equal to what’s in \(r_0(7)\) is a safe state; or in common terms, initially \(r_7\) is a valid return address (we write \(r(7)\) and \(r_7\) interchangeably). Axiom 4 says that \(r_1\) is an address in the readable range, and axiom 5 says that the program counter \(r_{17}\) is initially 100.

The predicates readable, writable, and safe are primitives. They are not defined; instead their meaning is encapsulated by inference rules that are used to build proofs about them. A sound safety policy would insure that from any machine state \(r, m\), if safe\((r, m)\), then execution proceeding from that state will never load from an unreadable location or store to an unwritable location.

We also include axioms to describe the untrusted code that has just been loaded. Although these are not part of the predefined safety policy, the code producer and code consumer can each calculate these axioms from the code itself:

6. \(m_0(100) = 2210\)
7. \(m_0(101) = 4070\)

We have implemented our logic in Twelf [PS99], which is an implementation of the Edinburgh logical framework [HHP93]. All the theorems in this paper have been checked in Twelf.

**Theorem.** safe\((r_0, m_0)\).

This theorem is the one that the code producer must prove; the code consumer will check the proof before jumping to address 100. But before describing the proof, we must show the inference rules for reasoning about machine instructions.

3 Instruction execution

Each instruction defines a relation between the machine state (registers, memory) before execution and the ma-
format(w,a,b,c,d) =def 0 ≤ a < 16 ∧ 0 ≤ b < 16 ∧ 0 ≤ c < 16 ∧ 0 ≤ d < 16 ∧ w = a * 16^3 + b * 16^2 + c * 16 + d.

decode(v,m,i) =def
  (∃d,s_1,s_2 . format(m,v),0,d,s_1,s_2) ∧ i = add(d,s_1,s_2))
∨  (∃d,s_1,c . format(m,v),1,d,s_1,c) ∧ i = addi(d,s_1,c)) ∨ ...
\[(r,m) \mapsto (r',m') = \text{def } \exists i, r'. \text{decode}(r(17), m, i) \land \text{upd}(r, 17, r(17) + 1, r') \land i(r', m, r', m')\]

\[\begin{align*}
\text{Inv}(r,m) & \quad \forall r^1, m^1. \text{Inv}(r^1, m^1) \rightarrow (\text{safe}(r^1, m^1) \lor (\exists r^2, m^2. (r^1, m^1) \rightarrow (r^2, m^2) \land \text{Inv}(r^2, m^2))) \\
\text{safe}(r,m) & \quad \text{multistep}
\end{align*}\]

Figure 3: The `multistep` inference rule of the logic.

## 5 Types

We have demonstrated that it is possible to prove a program safe. But for applications in proof-carrying code, it will be necessary to prove safety of large programs completely automatically. Such proofs can be based on dataflow or on types.

Although it is possible to construct proofs by purely dataflow-based techniques such as software fault isolation [WLAG93], in this paper we will concentrate on types. Necula’s PCC logic for an ML subset [Nec97] has inference rules such as the following (expressed in slightly different notation):

\[
v : m \vdash \tau_1 \times \tau_2 \quad \text{record2}_{\mathbb{E}}
\]

\[
\vdash \text{readable}(v) \land \text{readable}(v + 1) \\
\land m(v) : m(v + 1) : m \vdash \tau_1 \land \tau_2
\]

\[
v : m \vdash \tau \land m(v + 1) : m \vdash \text{list}_{\mathbb{E}}
\]

These rules relate typing judgements directly to the layout of typed values in machine memory, which is essential to proofs about machine-language programs. We write the judgement \(v : m : \tau\) with the colon subscripted by a machine memory \(m\), since a judgement that holds in one memory state might not hold in another. (Necula writes \(m \vdash v : \tau\)).

The disadvantage of inference rules for types. Necula’s PCC system includes typing rules in the safety policy, that is, in the trusted computing base. He proves the soundness of these rules by a metatheorem. Such a safety policy will require the code producer to use a particular type system, with values laid out in memory in a particular way – in effect, the safety policy will force the use of a single programming language and a single compiler.

Our approach allows each code producer to define the type system that its own mobile code uses. Of course, the type system must be sound; we allow the code producer to prove the typing rules as lemmas (provable in the object logic) rather than define new inference rules with a soundness metatheorem (which would be difficult for the code consumer to check).

We view the judgement \(v : m : \tau\) as syntactic sugar for \(\tau(m)(v)\), an application of the predicate \(\tau\) to memory \(m\) and integer (or address) \(v\). To illustrate, we will define the untagged integer type, cartesian product type, and list type as predicates, and prove (as theorems) the typing rules shown above.

Any one-word bit pattern qualifies as an untagged integer, so the \(\text{int}\) predicate accepts any value in any memory:

\[\text{int}(m)(v) = \text{def true}\]

Cartesian products can be defined in terms of the contents of two adjacent memory words:

\[
\text{record2}(\tau_1, \tau_2) = \text{def}
\]

\[
\Rightarrow \text{readable}(v) \land \text{readable}(v + 1) \\
\land \tau_1(m(v)) \land \tau_2(m(v + 1))
\]

Now the `record2_{\mathbb{E}}` rule shown above can be proved as a theorem, directly from the definition of `record2`.

We can go on to define union types, list types, and so on, with corresponding traversal theorems. But Necula’s PCC system gives no rules for creation (i.e., allocation and initialization) of data structures such as records and lists. From our definition of `record2` we could certainly prove the theorem,

\[
\vdash \text{readable}(v) \land \text{readable}(v + 1) \\
\land m(v) : m(v + 1) : m \vdash \tau_1 \land m \vdash \tau_2
\]

\[
\Rightarrow v : m : \tau_1 \times \tau_2
\]

But this is not enough! Any program that creates a new record value must initialize it by storing two values into memory. The step rule for the store instruction is

\[
\text{store}(s_1, s_2, c)(r, m, r', m') = \text{def}
\]

\[
\text{readable}(r(s_2) + c) \land
\text{upd}(m, r(s_2) + c, r(s_1), m') \land r = r'
\]

which relates a memory \(m\) (before the store) to a memory \(m'\) (after the store). Now suppose we have the following program fragment:

\[
103: \begin{align*}
I_{103}(r, m) & = r_1 : m \vdash \text{int} \times (\text{int} \times \text{int}) \land r_3 : m \vdash \text{int}
\end{align*}
\]

\[
104: \begin{align*}
I_{104}(r, m) & = r_1 : m \vdash \text{int} \times (\text{int} \times \text{int}) \land r_3 : m \vdash \text{int} \times \text{int}
\end{align*}
\]
prove \( r_2 \vdash_{\text{min}} \text{int} \times \text{int} \) with respect to the new memory \( m' \) to which \( I_{105} \) will be applied. But unfortunately, we cannot prove \( r_1 \vdash_{\text{min}} \text{int} \times (\text{int} \times \text{int}) \), because \( I_{103} \) establishes that fact about \( r_1 \) in a different version of \( m \). Practically speaking, we don’t know whether one of the store instructions overwrites a field of the record at \( r_1 \) so as to invalidate the typing judgement.

The following theorem is certainly provable:

\[
\frac{v \vdash_{m} \tau_1 \times \tau_2 \quad \text{upd}(m, x, y, m') \quad x \neq v \quad x \neq v + 1}{v \vdash_{m'} \tau_1 \times \tau_2}
\]

but how can we organize the proof so as to establish that \( x \neq v \)?

The solution is to reason carefully about heap allocation, distinguishing the allocated region of the heap from the unallocated region, as the next section will explain.

## 6 Heap Allocation

A call-by-value pure functional program allocates new data-structure values on a heap, and never updates old values. (Imperative languages are much harder to reason about, so we leave that for future work.) The program (and run-time environment) keeps track of which locations are allocated and which are free on the heap. In a very simple system an allocation pointer – a register or memory location – points to the boundary between allocated and unallocated memory. A more complex system might use a data structure to keep track of which blocks of memory are allocated.

The typing judgement \( v \vdash_{m} \tau_1 \times \tau_2 \) should imply that the addresses \( v \) and \( v + 1 \) are in the allocated set. We can make this explicit by making the allocated set \( a \) a parameter of the typing judgement: \( v \vdash_{a,m} \tau \). Now we define record types a bit differently than in the previous section (where \( a \) is an allocation predicate and \( v \in a \) is syntactic sugar for \( a(v) \)):

\[
\text{record}_2(\tau_1, \tau_2) \vdash_{a,m} v = \text{def} \quad v \in a \land (v + 1) \in a \\
\land \text{readable}(v) \land \text{readable}(v + 1) \\
\land \tau_1(a,m)(m,v) \land \tau_2(a,m)(m(v + 1))
\]

**Maintaining the allocation pointer.** Consider a program that uses register \( r_6 \) as an allocation pointer, so that the “standard” allocation predicate is

\[
a(v) = \text{def} \quad v < r_6
\]

Abstracting over \( r \) and \( m \), we say that

\[
\text{stda}(r,m)(v) = \text{def} \quad v < r(6)
\]

If all memory beyond address 100 is readable and writable, and the program itself occupies addresses 100–299, then we might start with \( r_6 = 300 \) and increase \( r_6 \) as the program executes. The program will initialize (i.e., store) new data structures beyond \( r_6 \); to ensure that the \( \text{prog} \) invariant holds, we must continually maintain the invariant \( r_6 \geq 300 \).

**Allocating a record.** Figure 4 shows a program that creates a new record value by storing the two fields at locations \( r_6 \) and \( r_6 + 1 \) and then increasing \( r_6 \) by 2. Clearly, at the point \( I_{109} \) \( r_2 \) satisfies all the conditions in the right-hand side of the definition of \( \text{record}_2(\tau, \tau)(\text{stda}(r,m), m) \), proving the judgement \( r_2 \vdash_{\text{stda}(r,m), m} \tau \times \tau \).

But at the same time, there is a pre-existing record value in \( r_1 \) that will still be needed after the new record is created – that is, both the precondition \( I_{106} \) and the post-condition \( I_{110} \) mention \( r_1 \vdash_{a,m} \tau \). The trick is to maintain this judgement even as the stores create “different” \( m \)'s and increasing \( r_6 \) creates “different” \( a \) sets.

We will define a valid type as one satisfying these conditions:

\[
\text{valid}(\tau) = \text{def} \quad \forall a, d', m, v. (a \subset d') \rightarrow \tau(a,m) v \rightarrow \tau(d', m) v \\
\land \forall a, m, m', v. (\forall x \in a. m(x) = m'(x)) \rightarrow \tau(a,m) v \rightarrow \tau(a,m') v
\]

The first condition is that a typing judgement \( v \vdash_{a,m} \tau \) is invariant under increasing the size of the allocated set; the second is that the judgement is invariant under storing any value at any unallocated location.

This valid predicate is strong enough to enable safety proofs for programs that traverse, allocate, and initialize data structures built from valid types. On the other hand, valid does not guarantee preservation of typing judgments through updating of already allocated fields, so we are restricted to type systems that use immutable data structures.

If \( \tau \) is a valid type, then the judgement \( r_1 \vdash_{a,m} \tau \) will be preserved through all the operations between \( I_{106} \) and \( I_{110} \). Each typing predicate that we wish to use in our proof of safety must be proved valid. We will show such theorems in the next section.

Morrisett et al. [MWCG98] show how to prove safety of allocation based on a type system for partially initialized records. We have not chosen to do this; instead, the approach we have shown in this section uses dataflow analysis to reason about the contents of the partially initialized record. We believe this will work well, since record initialization is an essentially local phenomenon.

**Heap exhaustion.** We can model a bounded-size heap as follows. Axiom 2 of the safety policy would be written as

\[ 2. \forall v. (100 \leq v < 1000) \rightarrow \text{writable}(v) \]

At each initializing store, it would be necessary to prove that the address is \( \geq 100 \) and \( < 1000 \). The former can be
proved as before from the invariant (e.g. $I_{107}$ of Figure 4).
The latter requires the program to include an instruction of the form if $r_6 + 2 \geq 1000$ goto exit. Immediately after such an instruction (which is the heap-limit check that any reasonable program would have to perform anyway), it is easy to establish that $r_6 < 1000$.

7 Type constructors

Almost all the types used in ML programs can be defined and proved valid in our system: record types, tagged union datatypes, function types, abstract types, polymorphic types, and covariant recursive types. We have not yet succeeded in defining contravariant recursive types, as the next section will discuss.

We start with some primitives:

- **constty i (a, m) $v = def$ v = i**
  The constant type, that is, $6 \cdot \text{constty}(6)$.

- **char (a, m) $v = def$ 0 $\leq v < 256**
  The character (or tiny integer) type.

- **boxed (a, m) $v = def$ v $\geq 256**
  The type of boxed (noncharacter) values.

- **ref $\tau (a, m) v = def$ v $\in a \land \text{readable}(v) \land \tau (a, m) (m v)**
  The type of (immutable) references to memory words containing values of type $\tau$.

- **aref $\tau (a, m) v = def$ v $\in a \land \forall v \notin d \land \tau (d, m) (m v)**
  The type of acyclic references, that is, the referenced data structure does not contain pointers back to address $v$.

- **offset i $\tau (a, m) v = def$ $\tau (a, m) (v + i)**
  The type of values $v$ such that $v + i$ has type $\tau$.

- **field i $\tau = def$ offset i (ref $\tau$)**
  The type of a record field at offset $i$ containing a value of type $\tau$. If acyclic records are desired, then aref can be used instead of ref.

union($\tau_1, \tau_2$) $\langle a, m \rangle v = def \ \tau_1 (a, m) v \lor \tau_2 (a, m) v$
  The type $\tau_1 \cup \tau_2$ of values that belong either to $\tau_1$ or $\tau_2$.

intersection($\tau_1, \tau_2$) $\langle a, m \rangle v = def \ \tau_1 (a, m) v \land \tau_2 (a, m) v$
  The type $\tau_1 \cap \tau_2$.

record1($\tau_1, \tau_2$) = def field 0 $\tau_1 \cap$ field 1 $\tau_2$
  A definition of the two-element record type equivalent to the one given in section 6 but more concise.

sum($\tau_1, \tau_2$) = def
record1(constty 0, $\tau_1$) $\cup$ record1(constty 1, $\tau_2$)
  A tagged disjoint sum type.

money = def
record1(constty 0, int) $\cup$
record1(constty 1, int) $\cup$
record1(constty 2, int, int)
  Equivalent to the ML datatype,

\[
\text{money} = \text{COIN of int} \\
| \text{BILL of int} \\
| \text{CHECK of int * int}
\]

existential($F$) $\langle a, m \rangle v = def \ \exists \tau. (F \tau) (a, m) v \land \text{valid} (\tau)$
  An existential type, useful in defining abstract data types [MP88] and function closures [MMH96].

universal($F$) $\langle a, m \rangle v = def \ \forall \tau. \text{valid} (\tau) \rightarrow (F \tau) (a, m) v$
  An universal type, useful for polymorphic functions.

Now we must prove all these types and constructors valid. The types constty, char, boxed are invariant with respect to increasing $a$ or updating $m$ at an unallocated location because their definitions don’t use the $a$ or $m$ argument at all.

Type ref($\tau$) is valid if $\tau$ is valid:

1. $a \subseteq a' \rightarrow \tau(a, m) w \rightarrow \tau(a', m) w$ for all $w$, so the implication will hold for the particular $w = m(v)$.

2. if $m = m'$ at all allocated locations, then $\tau(a, m) (m(v)) \rightarrow \tau(a, m') (m(v))$ by validity of $\tau$. And since $v \in a$, then $m(v) = m'(v)$, so $\tau(a, m') (m(v)) \rightarrow \tau(a, m') (m'(v))$ by congruence.
Offset $i \tau$ is valid if $\tau$ is valid by instantiation of $v + i$ for $v$ in the definition of validity of $\tau$.

Union and intersection types are valid (if their components are valid) by an equally simple argument.

A valid type constructor is one that preserves validity, as do ref and offset($r$). It is easy to show that the composition of valid constructors preserves validity; therefore, field types, record types, and sum types are valid if their component types are valid.

It is trivial to prove that the type existential $F$ is valid if $F$ is a valid constructor.

From these definitions, we can derive introduction and elimination for all of our type constructors. For example,

$$v \in a \land v + 1 \in a \land \text{readable}(v) \land \text{readable}(v + 1) \land m(v) : a, m \tau_1 \land m(v + 1) : a, m \tau_2 \Rightarrow \text{record2}$$

8 Function types

We will build function values (and function types) in three stages. First-order continuations – that is, machine-code addresses with arguments – belong to the codeptr type. A continuation closure (cont) is a code pointer with an environment. And a function closure (func) is also a code pointer with an environment, but the arguments of this code pointer include a cont. A compiler could generate these closures by following the typed closure conversion algorithm of Morrisett et al. [MWCG98].

A codeptr is an address to which control may be passed provided that its precondition is met. In a type-based proof, the precondition is mainly in the form of typing judgements. We can take address 106 from Figure 4 as an example; we can jump to location 106 from any machine state satisfying $I_{106}$ and the prog invariant. Let us separate this invariant into two parts, the “standard” invariant and the part specific to entry-point 106:

$$\text{stdp}(r, m) = \text{prog}(r, m) \land r_6 \geq 300$$

$P_{106}(r, m) = r_1 : \text{stda}(r, m), m \tau$

Notice that entry-point 106 uses the “standard” representation of the allocated set, that is, stda$(r, m)$. Not all program locations do; a program is free to spill $r_6$ to a memory location, or to defer incrementing $r_6$ until a series of allocations is complete. In such cases, a program point’s allocated-set would be represented as $v < m(ap)$ or $v < r_6 + k$ instead of stda$(r, m)(v) = v < r(6)$. However, we can make the restriction that any address to which we attribute the codeptr type must use stda.

We can abstract stda from $P_{106}$ to yield the component of the invariant that deals just with the formal-parameter type(s) of that entry point:

$$P_{106}(a, m) r = r_1 : a, m \tau$$

This predicate has almost the form of a type, except with an $r$ parameter instead of $v$. That is, it specifies the “type” of the register bank, or rather, the types of some subset of the registers – the formal parameter types.

For any such parameter-precondition $P$, we define

$$\text{codeptr}(P) (a, m) v = \text{def}$$

$$\forall \tau', m', r'(17) = v$$

$$\land \text{stdp}(r', m')$$

$$\land P(\text{stda}(r', m'), m')(r')$$

$$\Rightarrow \text{safe}(r', m')$$

This says that $v$ is a codeptr with formal parameters $P$ if, for any future register-bank $r'$ and memory $m'$, if the program-counter is at location $v$, the standard precondition stdp holds, and the types of the registers satisfy $P$, then it’s safe to continue.

In order for codeptr($P$) to be a valid type, $P$ must be a valid register-type – that is, it must be invariant with respect to increasing the allocated set or modifying memory at unallocated locations. It is easy to show that $P_{106}$ is valid if $\tau$ is valid. In general if $\tau_1, \tau_2, \ldots$ are valid types, then the predicate

$$r_{i_1} : a, m \tau_1 \land \ldots \land r_{i_k} : a, m \tau_k$$

is a valid formal-parameters predicate.

Let us define a family of predicates params$\tau_k$ – for various $k$ – as the standard calling sequence of $k$ arguments:

$$\text{params}_1 (\tau_1) (a, m) r = \text{def}$$

$$r_1 : a, m \tau_1$$

$$\text{params}_2 (\tau_1, \tau_2) (a, m) r = \text{def}$$

$$r_1 : a, m \tau_1 \land r_2 : a, m \tau_2$$

$$\text{params}_3 (\tau_1, \tau_2, \tau_3) (a, m) r = \text{def}$$

$$r_1 : a, m \tau_1 \land r_2 : a, m \tau_2 \land r_3 : a, m \tau_3$$

Thus, with respect to the program of Figure 4 we can make the following judgement:

$$\text{stdp}(r, m) \Rightarrow 106 : \text{stda}(r, m), m \text{ codeptr}(\text{params}_1 (\tau))$$

Continuation closures. In a programming language with nested lexical scopes for function definitions, an inner function may have free variables (which are bound only in an outer scope). The implementation of such a function must include both control (e.g., a code pointer) and environment (a data structure in which values for the free variables can be found). Since two functions of the same type may have different sets of free variables, the type of the environment should not be part of the function type. We solve this problem in the standard way: we use an existential type to hide the type of the environment [MMH96].

A continuation is a function that never returns (or rather, its return is the completion of the whole program).
Continuations, like functions, need closures and environments. For any type $\tau$, $\text{cont}(\tau)$ is the continuation taking a $\tau$ argument in register 1. However, the code entry point will also have to take an environment (of type $\sigma$) in register 2.

$$\text{cont}(\tau) = \text{def} \space \text{existential}(\lambda \sigma. \text{record}_2(\text{codeptr}(\text{params}_2(\tau, \sigma)), \sigma))$$

To apply a continuation value $v$, one must first fetch the codeptr $c$ from $m(v + 0)$ and put it in some register, say $r_5$. One must put a value of type $\tau$ in $r_1$. One must fetch the environment $e$ from $m(v + 1)$ into $r_2$. One must ensure that the standard precondition $\text{stdp}(r, m)$ holds. **Theorem:** Then it is safe to jump to the address contained in $r_5$. **Proof:** by expansion of definitions.

**Function closures.** A function is just a continuation with an additional argument that is itself a continuation. That is, the function type $\alpha \to \beta$ takes one argument that is a value of type $\alpha$, and another argument of type $\text{cont}($β$)$. Since functions may have free variables, we make function closures in the same way as for continuations – so the codeptr component of a function has another argument of type $\sigma$, the environment type.

$$\text{func}(\alpha, \beta) = \text{def} \space \text{existential}(\lambda \sigma. \text{record}_2(\text{codeptr}(\text{params}_2(\alpha, \text{cont}(\beta), \sigma)), \sigma))$$

Calling a function is done almost exactly as calling a continuation, except that $r_1$ contains the argument, $r_2$ contains the continuation-closure, and $r_3$ contains the function environment.

The type-constructors $\text{cont}$ and $\text{func}$ are valid because they are just compositions of other valid constructors (existential, record, codeptr, params).

We have described functions with heap-allocated continuations – not stack-allocated frames – because they are easier to reason about, easier to implement, suitably efficient, and used by a compiler [Sha98] that can plausibly serve as a front-end for our PCC system. Of course it is also possible to reason effectively about stack-allocated frames [MCGW98, KKR86].

### 9 Recursive Datatypes

In order to define recursive datatypes, we introduce a subtyping relation defined as logical implication:

$$\text{subtype}(\tau_1, \tau_2) = \text{def} \forall a, m, v. \tau_1(a, m)(v) \to \tau_2(a, m)(v).$$

We write $\tau_1 \sqsubseteq \tau_2$ to denote this relation. Using this relation, we define the following rec predicate:

$$\text{rec}(f) = \text{def} \forall \tau. \text{valid}(\tau) \to f(\tau) \sqsubseteq \tau \to \tau(a, m)(v)$$

The recursive types are all types $\text{rec}(f)$ for which the least fixed point of the argument function $f$ is $\text{rec}(f)$. It can be shown that any function $f$ that preserves validity and also satisfies the following monotone predicate has this property.

$$\text{monotone}(f) = \text{def} \forall \tau_1, \tau_2. \tau_1 \sqsubseteq \tau_2 \to f(\tau_1) \sqsubseteq f(\tau_2)$$

In particular, we prove that whenever $f$ satisfies these properties, both $f(\text{rec}(f)) \sqsubseteq \text{rec}(f)$ and $\text{rec}(f) \sqsubseteq f(\text{rec}(f))$ hold, and thus the following theorem holds.

$$\text{preserves}\_\text{validity}(f) \land \text{monotone}(f) \Rightarrow \text{roll}\_\text{unroll}$$

This theorem allows us to fold and unfold recursive types. Unfolding is useful for proofs of safety for programs that traverse recursive datatypes, while folding is useful in proofs involving allocation. Using the rec operator we can define (for example) polymorphic lists:

$$\text{list}(\tau) = \text{def} \text{rec}(\lambda \tau'. \text{consty} 0 \sqcup \text{boxed}(\text{rec}_2(\text{int}, \tau)))$$

The address used for pointers to cons cells must not be 0, so we use a boxed address to point to cons cells.

In order to build arbitrary recursive datatypes using any of the constructors of section 7, we have proved that they preserve both validity and monotonicity. For the constructor ref, for example, we proved monotone(ref). For constructors that take two arguments, we must show that the constructor is monotone in both. For example, we showed monotone2(union), where:

$$\text{monotone}_2(f) = \text{def} \forall \tau_1, \tau_2, \tau_1', \tau_2'. \tau_1 \sqsubseteq \tau_2 \to \tau_1' \sqsubseteq \tau_2' \to f(\tau_1, \tau_1') \sqsubseteq f(\tau_2, \tau_2')$$

We want to be able to automate the proofs that show that any datatype built from these constructors is monotonic and preserves validity. This automation is in fact easy as long as we prove the right set of lemmas. The lemmas we have proved allow us to structure proofs for arbitrary datatypes so that they contain exactly one lemma application for each constructor that appears in the datatype. The following lemma about union illustrates the form of the lemmas that we use for this purpose:

$$\text{valid}\_\text{mono}(f) \Rightarrow \text{valid}\_\text{mono}(g) \Rightarrow \text{vm}\_\text{union}$$

where valid mono is:

$$\text{valid}\_\text{mono}(f) = \text{def} \text{preserves}\_\text{validity}(f) \land \text{monotone}(f).$$

We prove analogous lemmas for ref, aref, offset, field, intersection, sum, and records of any number of arguments.

Allowing function types in recursive datatypes presents a further challenge. Not all types satisfy the
monotone criterion; only covariant types do. In these types occurrences of the type being defined can only appear positively, that is, they must appear to the left of an even number of function arrows in an ML declaration.

For instance, in the following examples:

\[ \tau_1 = c_1 \text{ of int} | c_2 \text{ of int} \rightarrow \tau_1 \]
\[ \tau_2 = c_1 \text{ of int} | c_2 \text{ of } \tau_2 \rightarrow \text{int} \]
\[ \tau_3 = c_1 \text{ of int} | (c_2 \text{ of } \tau_3 \rightarrow \text{int}) \rightarrow \tau_3 \]
\[ \tau_4 = c_1 \text{ of int} | c_2 \text{ of } ((\tau_4 \rightarrow \text{int}) \times \text{int}) \rightarrow (\tau_4 \times \text{int}) \]

the first, third, and fourth satisfy the restriction. Proving that they do requires proving antimonotone (codeptr), where we define:

\[ \text{antimonotone}(f) = \text{def } \forall \tau_1, \tau_2. \tau_1 \sqsubseteq \tau_2 \rightarrow f(\tau_2) \sqsubseteq f(\tau_1). \]

The antimonotonicity of codeptr results from the appearance of the argument-type predicate to the left of an implication arrow in codeptr’s definition.

We prove the composition of a monotone with an antimonotone operator, or vice versa, is antimonotone; and that the composition of antimonotone operators is monotone. Then it follows easily that cont is antimonotone, and that func(\(\alpha\), \(\tau_2\)) is monotone in \(\alpha\) if \(\tau_1\) is antimonotone in \(\alpha\) and \(\tau_2\) is monotone in \(\alpha\). Note that \(\tau_2\) appears inside two nested cont operators, establishing its monotonicity.

These and similar results allow us to prove the validity of the recursive types \(\tau_1, \tau_2, \tau_4\) shown above. We must prove (anti)monotonicity lemmas for all the constructors of section 7. The next section includes an example of their use.

## 10 Implementation in Twelf

Our encoding of higher-order logic (the object logic) is illustrated by the following declarations in Twelf (the metalogic).

```
tp: type.
int: ttp.
form: ttp.
arrow: ttp -> ttp -> ttp.
%infix right 14 arrow.
tm: ttp -> type.
form: ttp.
pf: ttp form -> type.
lam: (tm T -> tm U) -> tm (T arrow U).
\$: tm (T arrow U) -> tm T -> tm U.
%infix left 20 $.
and: ttp form -> ttp form -> ttp form.
%infix right 12 and.
forall: (tm T -> ttp form) -> ttp form.
```

A metalogic (Twelf) type is a type, an object-logic type is a ttp, and a programming-language type is a ttp (which is not in the core logic since it is a definition at the discretion of the code producer). Object-logic types are constructed from int, the type form of formulas of the object logic, and the arrow constructor. Object-level terms of type \(T\) have type \(\text{tm } T\) in the metalogic. Quantifying at the metalevel allows us to encode polymorphic object-level types. Terms of type \(\text{pf } A\) are terms representing proofs of object formula \(A\).

The declarations beginning with \(\text{lam}\) introduce constants for constructing terms and formulas. Note that the universal quantifier \(\text{forall}\) is polymorphic; uppercase letters denote variables, and free variables are implicitly quantified at the outermost level. Braces are used for explicit quantification. The last five declarations encode the introduction and elimination rules of natural deduction for conjunction and universal quantification. The complete encoding (about 100 lines of Twelf) includes the remaining inference rules of higher-order logic, an encoding of integers (including arithmetic operators and natural number induction), the multistep rule, and the axioms of the safety policy. All other objects are definitions and theorems built from this core signature.

The following are the Twelf definitions of some of the type constructors as well as the polymorphic lists presented in section 9.

```
ty : ttp = state arrow int arrow form.
ref : tm (ty arrow ty) =
    lam3 [T][S][V] fst S @ V and
    readable @ V and T @ S @ (and S @ V).
offset : tm (int arrow ty arrow ty) =
    lam4 [I][T][S][V](T @ S @ (V + I)).
field : tm (int arrow ty arrow ty) =
    lam2 [I][T](offset @ I @ (ref @ T)).
record2 : tm (ty arrow ty arrow ty) =
    lam2 [T][U](intersect @ (field @ S @ T @ (field @ I @ U)) @ T).
listf : tm ty -> tm (ty arrow ty) =
    [T](lam [T'](union @ (constty @ (const 0)) @
        (intersect @ boxed @ (record2 @ T @ T')))).
list : tm (ty arrow ty) =
    (lam [T](rec @ (listf T))).
```

The ty declaration gives the type for predicates representing ML types. Some definitions are omitted.
example lam4 is defined in terms of lam and binds 4 variables, and state is (pair allocset memory) where pair is polymorphic, defined in the usual way with \( \lambda \)-calculus. The following theorem justifies the use of rec in the definition of list.

\[
\text{vm_listf : pf (validtype @ T) \rightarrow pf (valid_mono @ (listf T)) =}
\begin{cases}
P:pf (validtype @ T) \\
(vm\_union vm\_constty \\
(vm\_intersect vm\_boxed \\
(vm\_record2 \\
(vm\_validtype P) vm\_id))).
\end{cases}
\]

The \( \text{vm\_lemmas} \) state that the constructors preserve validity and monotonicity. The \( \text{vm\_union} \) theorem, for instance, was presented in the previous section.

11 Conclusion and Future Work

We have described a framework for proof-carrying code which should be sufficiently general to accommodate real programming languages on real machines.

Any PCC system must be concerned with keeping the size of proofs small. Our lemmatization of the typing rules adds a constant size to any proof, but no multiplicative factor. Also, instead of expanding out Hoare-logic substitutions before proving – as Necula does in his verification-condition generator – we avoid this potentially exponential blowup in theorem size by using a step-relation machine semantics. This gives us the potential for smaller proofs.

Future Work

A variety of directions remain to be explored. We summarize a few here.

Machine instruction sets. To handle real machines, we plan to encode instruction set architectures such as the Sparc and Pentium; we will have to handle variable size instructions and byte addressing.

Contravariant recursive types. Many real programming languages—ML, Java, C—have contravariant recursive types such as this one:

\[
\text{datatype exp} = \text{APP of exp * exp} \\
| \text{LAM of exp \rightarrow exp}
\]

Our current type framework cannot handle this type because of the occurrence of exp to the left of the arrow in the LAM constructor. We plan to adapt the model of types in MacQueen et al. [MPS86] or in Mitchell and Viswanathan [MV96] to our notion of types as predicates on machine states. Doing so requires the formalization of metric spaces or partial equivalence relations, respectively.

Mutable fields. We also plan to describe mutable data structures, such as ML refs and Java objects. Handling references will involve allowing for mutable memory locations, which will require a more complex notion of allocation, and thus a more complex valid type predicate.

F\( \omega \). Our longer-range plan is to cover more of the types used by a production compiler for a language such as ML. In particular, we plan to incorporate the type system of the FLINT intermediate language [Sha98] (which will also compile Java [LST99]), for which we will have to encode the types and kinds of the F\( \omega \) polymorphic \( \lambda \)-calculus [Gir72, Rey74].

Other type systems. To show that our approach to safety policies (which moves information from the trusted computing base into a semantic model built from first principles) is truly universal, we plan to build a model of a type system that is possibly quite different from that of ML. One possibility is the type system of Touchstone [Nec98] which has mutable records, but no function types or union types; or the typed assembly language of Morrisett et al. [MWCG98].

Concurrency. Our model is sequential. Concurrency and asynchronous exceptions can be handled by assuming (in the step relation) that some portions of memory can change between successive machine instructions, and some portions will not. The safety policy must guarantee that certain memory locations (i.e. unshared variables of this thread) are preserved unchanged.

Automating proof. In an earlier version of our system, we built a prototype theorem prover which automatically proved safety of simple programs that traverse and allocate lists. We have lots of ideas about how to augment this prover. In doing so, it will be necessary to keep proofs small. Our goal is to develop a set of lemmas that allow us to build proofs fully automatically that are linear in the size of the type-annotated intermediate representation of the compiled program; we believe this is possible for the kinds of safety proofs we are considering. An example illustrating this idea is the \( \text{vm\_list} \) theorem in the previous section whose proof uses exactly as many lemma constructors as the description of the \( \text{listf} \) type uses type constructors.
Runtime code generation. Because our machine semantics treats machine instructions as data in the Von Neumann style, there is the potential to prove the safety of programs that do runtime code generation.

Garbage collection. We have left garbage collection for future work, but the approach of Wang and Appel [WA99] looks promising and fits into our framework.

Acknowledgements. We thank Neophytos Michael for assistance in implementing the toy-machine decode function in Twelf; Robert Harper, Frank Pfenning, Carsten Schürmann for advice about encoding logics in Twelf; Doug Howe, David MacQueen, and Jon Riecke for advice about recursive types; Greg Morrisett for comments on an early draft of the paper.

References


