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

July 16, 1999

Abstract

Proof-carrying code is a framework for proving the safety of machine-language programs with a machine-checkable proof. Such proofs have previously 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 (e.g., by a just-in-time compiler) in translation from the proved program to the program that will execute, and (2) for sufficiently simple safety policies and for programs compiled from type-safe source languages, the proofs can be constructed fully automatically.

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.

*On sabbatical 1998-99.

$$\mathrm{upd}(f,d,x,f') =_{\mathrm{def}} \forall z.\, d = z \wedge f'(z) = x \vee d \neq z \wedge f'(z) = f(z)$$

$$\begin{aligned}
&\mathrm{add}(d,s_1,s_2)(r,m,r',m') =_{\mathrm{def}} \mathrm{upd}(r,d,r(s_1)+r(s_2),r') \wedge m = m'. \\
&\mathrm{addi}(d,s,c)(r,m,r',m') =_{\mathrm{def}} \mathrm{upd}(r,d,r(s)+c,r') \wedge m = m' \\
&\mathrm{load}(d,s,c)(r,m,r',m') =_{\mathrm{def}} \mathrm{readable}(r(s)+c) \wedge \mathrm{upd}(r,d,m(r(s)+c),r') \wedge m = m' \\
&\mathrm{store}(s_1,s_2,c)(r,m,r',m') =_{\mathrm{def}} \mathrm{writable}(r(s_2)+c) \wedge \mathrm{upd}(m,r(s_2)+c,r(s_1),m') \wedge r = r' \\
&\mathrm{jump}(d,s,c)(r,m,r',m') =_{\mathrm{def}} \exists r''.\, \mathrm{upd}(r,17,r(s)+c,r'') \wedge \mathrm{upd}(r'',d,r(17),r') \wedge m = m' \\
&\mathrm{bgt}(s_1,s_2,c)(r,m,r',m') =_{\mathrm{def}} r(s_1) > r(s_2) \wedge \mathrm{upd}(r,17,r(17)+c,r') \wedge m = m' \vee r(s_1) \leq r(s_2) \wedge r = r' \wedge m = m' \\
&\mathrm{beq}(s_1,s_2,c)(r,m,r',m') =_{\mathrm{def}} r(s_1) = r(s_2) \wedge \mathrm{upd}(r,17,r(17)+c,r') \wedge m = m' \vee r(s_1) \neq r(s_2) \wedge r = r' \wedge m = m'
\end{aligned}$$

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.

	OPCODE	
add	$0\ d\ s_1\ s_2$	$r_d \leftarrow r_{s_1} + r_{s_2}$
addi	$1\ d\ s\ c$	$r_d \leftarrow r_s + c$
load	$2\ d\ s\ c$	$r_d \leftarrow m(r_s + c)$
store	$3\ s_1\ s_2\ c$	$m(r_{s_2} + c) \leftarrow r_{s_1}$
jump	$4\ d\ s\ c$	$r_d \leftarrow pc;\ pc \leftarrow r_s + c$
bgt	$5\ s_1\ s_2\ c$	if $r_{s_1} > r_{s_2}$ then $pc \leftarrow pc + c$
beq	$6\ s_1\ s_2\ c$	if $r_{s_1} = r_{s_2}$ then $pc \leftarrow pc + c$

Example 1. We wish to verify the safety of the following short program. The code producer will provide the program (i.e., in this case the sequence of 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.

$$\begin{array}{lll}
100: & 2210 & r_2 \leftarrow m(r_1) \\
101: & 4070 & \mathit{jump}(r_7);\ r_0 \leftarrow pc
\end{array}$$

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. $\forall v.\ (v \geq 50) \rightarrow \mathrm{readable}(v)$
2. $\forall v.\ (v \geq 100) \rightarrow \mathrm{writable}(v)$
3. $\forall r,m.(r(17) = r^0(7)) \rightarrow \mathrm{safe}(r,m)$
4. $r^0(1) > 50$
5. $r^0(17) = 100$
6. $m^0(100) = 2210$
7. $m^0(101) = 4077$

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^0_1 is an address in the readable range, and axiom 5 says that the program counter r_{17} is initially 100. The remaining axioms describe the untrusted code that has just been loaded.

We have implemented our logic in Twelf [PS99], which is an implementation of the Edinburgh logical framework [Pfe91]. All the theorems in this paper have been checked in Twelf.

Theorem. $\mathrm{safe}(r^0,m^0)$.

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.

$$\text{format}(w,a,b,c,d) =_{\text{def}} \ 0 \le a < 16 \wedge 0 \le b < 16 \wedge 0 \le c < 16 \wedge 0 \le d < 16 \wedge w = a*16^3 + b*16^2 + c*16 + d.$$

$$\begin{array}{l} \text{decode}(v,m,i) =_{\text{def}} \\ \quad (\exists d, s_1, s_2.\ \text{format}(m(v), 0, d, s_1, s_2) \wedge i = \text{add}(d, s_1, s_2)) \\ \vee \ \ (\exists d, s_1, c.\ \text{format}(m(v), 1, d, s_1, c) \wedge i = \text{addi}(d, s_1, c)) \vee ... \end{array}$$

Figure 2: Instruction decoding.

3 Instruction execution

Each instruction defines a relation between the machine state (registers, memory) before execution and the machine state afterwards. We treat the program counter as part of the register set (r_{17}) even though it's not really namable in an instruction opcode. Figure 1 shows the definition of this relation for each of the instructions add, addi, load, and so on.

On a von Neumann machine, each instruction is represented in memory by an integer. Our *decode* relation (Figure 2) is a predicate on three arguments (v,m,i) and says that address v in memory m contains the encoding of instruction i.

We can now write relation $step(r,m,r',m')$ (Figure 3) which says that the execution of one instruction in state (r,m) leads to state (r',m'). This holds only for *safe and legal* instruction executions, because the definition of the *load* relation requires that the loaded address be readable, and the definition of *store* requires that the stored address be writable, and the *decode* relation fails to hold at all for illegal instructions.

Finally, we capture the notion of continued execution by the inference rule *multistep* (Figure 3), which is a coinduction principle based (loosely) on the Floyd-Hoare *while* rule.

4 The global invariant

To prove our program safe, we construct an invariant *Inv* that holds at all times. We start by informally annotating each instruction with a precondition.

$$\begin{array}{lll} & I_{100}(r,m) = \text{jumpable}(r_7) \wedge \text{readable}(r_1) & \\ 100: & 2210 \quad r_2 \leftarrow m(r_1) & \\ & I_{101}(r,m) = \text{jumpable}(r_7) & \\ 101: & 4077 \quad \textit{jump}(r_7) & \end{array}$$

where

$$\text{jumpable}(v) =_{\text{def}} \ \forall r', m'. r'(17) = v \rightarrow \text{safe}(r', m')).$$

Our definitions allow for the possibility that a store instruction will overwrite the program, which allows us to prove the safety of self-modifying code. But our simple example does not overwrite itself, and this fact is a necessary part of our invariant:

$$\begin{array}{ll} \text{prog}(m) =_{\text{def}} & \text{decode}(100, m, \text{load}(2,1,0)) \\ & \wedge \text{decode}(101, m, \text{jump}(0,7,0)) \end{array}$$

Now our global invariant is just the combination of the *prog* invariant with all the local ones:

$$\begin{array}{l} Inv(r,m) =_{\text{def}} \ \text{prog}(m) \wedge \\ \quad (r(17) = 100 \wedge I_{100}(r,m) \\ \ \vee \ r(17) = 101 \wedge I_{101}(r,m) \\ \ \vee \ \text{jumpable}(r(17))) \end{array}$$

To prove our theorem $\text{safe}(r^0,m^0)$ we use the *multistep* rule. First we show $Inv(r^0,m^0)$, then that *Inv* is preserved under the *step* relation.

Axioms 6 and 7, along with the definition of the decode relation, prove that $\text{prog}(m^0)$ holds. Axiom 5 ($r^0(17) = 100$) means that the remaining proof obligation for $Inv(r^0,m^0)$ is $I_{100}(r^0,m^0)$, which can be proved directly from axioms 3, 4, and 1.

To show that the invariant is conserved, we work by cases:

- $r_{17} = 100 \wedge I_{100}(r^1,m^1)$. By $prog(m^1)$ we have $\text{decode}(r^1_{17}, m^1, \text{load}(2,1,0))$. Letting $r^2 = r^1[17 \mapsto r^1_{17} + 1, 2 \mapsto m^1(r^1_1)]$ and $m^2 = m^1$, and using $\text{readable}(r^1_1)$ from I_{100}, we have $\text{step}(r^1,m^1,r^2,m^2)$. Since $r^1_7 = r^2_7$, by I_{100} we have $\text{jumpable}(r^2_7)$. Thus $r^2_{17} = 101 \wedge I_{101}(r^2,m^2)$ is proved. Since $m^1 = m^2$, $prog(m^2)$ holds.
- $r^1_{17} = 101 \wedge I_{101}(r^1,m^1)$. By $prog(m^1)$ we have $\text{decode}(r^1_{17}, m^1, \text{jump}(0,7,0))$. Letting $r^2 = r^1[17 \mapsto r^1_7, 0 \mapsto r^1_{17}]$, we have $\text{jump}(0,7,0)(r^1[17 \mapsto r^1_{17} + 1], m^1, r^2, m^1)$ by the definition of jump. Thus we have $\text{step}(r^1,m^1,r^2,m^1)$. We can use the definition of the upd relation, along with $\text{jumpable}(r^1_7)$, to show

$$\text{step}(r,m,r',m') =_{\text{def}} \exists i, r''.decode(r(17),m,i) \wedge \text{upd}(r,17,r(17)+1,r'') \wedge i(r'',m,r',m')$$

$$\frac{Inv(r,m) \qquad \forall r^1,m^1.\ Inv(r^1,m^1) \to (\text{safe}(r^1,m^1) \vee (\exists r^2,m^2.\text{step}(r^1,m^1,r^2,m^2) \wedge Inv(r^2,m^2)))}{\text{safe}(r,m)}\text{multistep}$$

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

jumpable(r^2_{17}), which satisfies one of the disjuncts of the *Inv* relation.

• jumpable(r^1_{17}) implies safe(r^1,m^1) directly by the definition of jumpable with r',m' instantiated by r^1,m^1.

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):

$$\frac{v :_m \tau_1 \times \tau_2}{\begin{array}{l}\text{readable}(v) \wedge \text{readable}(v+1)\\ \wedge m(v) :_m \tau_1 \wedge m(v+1) :_m \tau_2\end{array}}\text{record2\_e}$$

$$\frac{v :_m \text{list}(\tau) \qquad v \neq 0}{\begin{array}{l}\text{readable}(v) \wedge \text{readable}(v+1)\\ \wedge m(v) :_m \tau \wedge m(v+1) :_m \text{list}(\tau)\end{array}}\text{list\_e}$$

These rules relate typing judgements directly to the layout of typed values in machine memory, which is essential to proofs of 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 that 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 an application of the predicate τ to memory m and integer (or address) v, that is, $\tau(m)(v)$. 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 *int* predicate accepts any value in any memory:

$$\text{int}(m)(v) =_{\text{def}} \text{true}.$$

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

$$\begin{array}{l}\text{record}_2(\tau_1,\tau_2)\,m\,v =_{\text{def}}\\ \quad \text{readable}(v) \wedge \text{readable}(v+1)\\ \quad \wedge \tau_1\,m\,(m\,v) \wedge \tau_2\,m\,(m(v+1))\end{array}$$

Now the *record2_e* rule shown above can be proved as a theorem, directly from the definition of record_2.

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 record_2 we could certainly prove the theorem,

$$\frac{\begin{array}{l}\text{readable}(v) \wedge \text{readable}(v+1)\\ m(v) :_m \tau_1 \wedge m(v+1) :_m \tau_2\end{array}}{v :_m \tau_1 \times \tau_2}\text{record2\_i}$$

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

$$\begin{array}{l}\mathrm{store}(s_1,s_2,c)(r,m,r',m') =_{\mathrm{def}}\\ \quad \mathrm{writable}(r(s_2)+c)\wedge\\ \quad\ \mathrm{upd}(m,r(s_2)+c,r(s_1),m')\wedge r=r'\end{array}$$

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

$$\begin{array}{ll} & I_{103}(r,m) = r_1 :_m \mathrm{int}\times(\mathrm{int}\times\mathrm{int})\wedge r_3 :_m \mathrm{int}\\ 103: & \mathbf{m}(\mathbf{r_2}) \leftarrow \mathbf{r_3}\\ & I_{104}(r,m) = r_1 :_m \mathrm{int}\times(\mathrm{int}\times\mathrm{int})\\ & \qquad\qquad\qquad \wedge r_3 :_m \mathrm{int}\wedge m(r_2)=r_3\\ 104: & \mathbf{m}(\mathbf{r_2}+\mathbf{1}) \leftarrow \mathbf{r_3}\\ & I_{105}(r,m) = r_1 :_m \mathrm{int}\times(\mathrm{int}\times\mathrm{int})\wedge r_2 :_m \mathrm{int}\times\mathrm{int}\end{array}$$

After storing two integers into memory at addresses r_2 and r_2+1 we can legitimately use the record2_i rule to prove $r_2 :_{m'} \mathrm{int}\times\mathrm{int}$ with respect to the new memory m' to which I_{105} will be applied. But unfortunately, we cannot prove $r_1 :_{m'} \mathrm{int}\times(\mathrm{int}\times\mathrm{int})$, 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:

$$\frac{v :_m \tau_1\times\tau_2 \quad \mathrm{upd}(m,x,y,m') \quad x\neq v \quad x\neq v+1}{v :_{m'} \tau_1\times\tau_2}$$

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 $v :_m \tau_1\times\tau_2$ 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: $v :_{a,m} \tau$. 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 $a(v)$):

$$\begin{array}{l}\mathrm{record}_2(\tau_1,\tau_2)\,(a,m)\,v =_{\mathrm{def}}\\ \quad v\in a\wedge(v+1)\in a\\ \quad \wedge\,\mathrm{readable}(v)\wedge\mathrm{readable}(v+1)\\ \quad \wedge\,\tau_1\,(a,m)\,(m\,v)\wedge\tau_2\,(a,m)\,(m(v+1))\end{array}$$

Maintaining the allocation pointer. Consider a program that uses register r_6 as an allocation pointer, so that the "standard" allocated predicate is

$$a(v) =_{\mathrm{def}}\ v < r_6$$

Abstracting over r and m, we say that

$$\mathrm{stda}(r,m)(v) =_{\mathrm{def}}\ v < r(6)$$

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 *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 $\mathrm{record}_2(\tau,\tau)(\mathrm{stda}(r,m),m)$, proving the judgement $r_2 :_{\mathrm{stda}(r,m),m} \tau\times\tau$.

But at the same time, there is a pre-existing record value in r_1 that will still be needed to use after the new value – that is, both the precondition I_{106} and the postcondition I_{110} mention $r_1 :_{a,m} \tau$. The trick is to maintain this judgement even as the stores create "different" m's and increasing r_6 creates "different" a sets.

$$
\begin{array}{ll}
 & I_{106}(r,m) = r_6 \geq 300 \wedge r_1 :_{\mathrm{stda}(r,m),m} \tau \\
106: & \mathbf{m(r_6) \leftarrow r_1} \\
 & I_{107}(r,m) = r_6 \geq 300 \wedge r_1 :_{\mathrm{stda}(r,m),m} \tau \wedge m(r_6) :_{\mathrm{stda}(r,m),m} \tau \\
107: & \mathbf{m(r_6+1) \leftarrow r_1} \\
 & I_{108}(r,m) = r_6 \geq 300 \wedge r_1 :_{\mathrm{stda}(r,m),m} \tau \wedge m(r_6) :_{\mathrm{stda}(r,m),m} \tau \wedge m(r_6+1) :_{\mathrm{stda}(r,m),m} \tau \\
108: & \mathbf{r_2 \leftarrow r_6 + 0} \\
 & I_{109}(r,m) = r_6 \geq 300 \wedge r_1 :_{\mathrm{stda}(r,m),m} \tau \wedge m(r_2) :_{\mathrm{stda}(r,m),m} \tau \wedge m(r_2+1) :_{\mathrm{stda}(r,m),m} \tau \wedge r_2 = r_6 \\
109: & \mathbf{r_6 \leftarrow r_6 + 2} \\
 & I_{110}(r,m) = r_6 \geq 300 \wedge r_1 :_{\mathrm{stda}(r,m),m} \tau \wedge r_2 :_{\mathrm{stda}(r,m),m} \tau \times \tau
\end{array}
$$

Figure 4: A program that allocates and initializes a record.

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

$$
\begin{array}{l}
\mathrm{valid}(\tau) =_{\mathrm{def}} \\
\quad \forall a, a', m, v.\ (a \subset a') \rightarrow \tau\,(a,m)\,v \rightarrow \tau\,(a',m)\,v \\
\wedge\ \forall a, m, m', v.\ (\forall x \in a.\ m(x) = m'(x)) \rightarrow \\
\qquad\qquad \tau\,(a,m)\,v \rightarrow \tau\,(a,m')\,v
\end{array}
$$

The first condition is that a typing judgement $v :_{a,m} \tau$ 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.

If τ is a valid type, then the judgement $r_1 :_{a,m} \tau$ 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.

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:

$\mathrm{constty}\, i\, (a,m)\, v =_{\mathrm{def}}\ v = i$
: The constant type, that is, $6 :_{a,m} \mathrm{constty}(6)$.

$\mathrm{char}\, (a,m)\, v =_{\mathrm{def}}\ 0 \leq v < 256$
: The character (or tiny integer) type.

$\mathrm{boxed}\, (a,m)\, v =_{\mathrm{def}}\ v \geq 256$
: The type of boxed (noncharacter) values.

$\mathrm{ref}\ \tau\, (a,m)\, v =_{\mathrm{def}}\ v \in a \wedge \mathrm{readable}(v) \wedge \tau\,(a,m)\,(m\,v)$
: The type of (immutable) references to memory words containing values of type τ.

$\mathrm{aref}\ \tau\, (a,m)\, v =_{\mathrm{def}}\ v \in a \wedge \mathrm{readable}(v) \wedge$
$\quad \exists a'.\ a' \subset a \wedge v \notin a' \wedge \tau\,(a',m)\,(m\,v)$
: The type of acyclic references, that is, the referenced data structure does not contain pointers back to address v.

$\mathrm{offset}\ i\ \tau\, (a,m)\, v =_{\mathrm{def}}\ \tau\,(a,m)\,(v+i)$
: The type of values v such that $v+i$ has type τ.

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

$\mathrm{union}(\tau_1, \tau_2)\,(a,m)\,v =_{\mathrm{def}}\ \tau_1\,(a,m)\,v \vee \tau_2(a,m)\,v$
: The type $\tau_1 \cup \tau_2$ of values that belong either to τ_1 or τ_2.

$\mathrm{intersection}(\tau_1, \tau_2)\,(a,m)\,v =_{\mathrm{def}}\ \tau_1\,(a,m)\,v \wedge \tau_2(a,m)\,v$
: The type $\tau_1 \cap \tau_2$.

$\mathrm{record}_2(\tau_1, \tau_2) =_{\mathrm{def}}\ \mathrm{field}\ 0\,\tau_1\ \cap\ \mathrm{field}\ 1\,\tau_2$
: A definition of the two-element record type equivalent to the one given in section 6 but more concise.

$\mathrm{sum}(\tau_1,\tau_2) =_{\mathrm{def}}$
$\quad \mathrm{record}_2(\mathrm{constty}\ 0,\ \tau_1) \cup \mathrm{record}_2(\mathrm{constty}\ 1,\ \tau_2)$
A tagged disjoint sum type.

$$\begin{array}{lll} \mathrm{money} & =_{\mathrm{def}} & \mathrm{record}_2(\mathrm{constty}\ 0,\ \mathrm{int}) \\ & \cup & \mathrm{record}_2(\mathrm{constty}\ 1,\ \mathrm{int}) \\ & \cup & \mathrm{record}_3(\mathrm{constty}\ 2,\ \mathrm{int},\ \mathrm{int}) \end{array}$$

Equivalent to the ML datatype,

```
money = COIN of int
      | BILL of int
      | CHECK of int * int
```

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

$\mathrm{universal}(F)\,(a,m)\,v =_{\mathrm{def}} \forall\tau.\ \mathrm{valid}(\tau) \rightarrow (F\tau)(a,m)\,v$
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 $\mathrm{ref}(\tau)$ is valid if τ is valid:

1. $a \subset a' \rightarrow \tau(a,m)\,w \rightarrow \tau(a',m)\,w$ 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 τ. 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 τ is valid by instantiation of $v+i$ for v in the definition of validity of τ.

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 $\mathrm{offset}(i)$. 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 $\mathrm{existential}(F)$ is valid if F is a valid constructor.

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:

$$\begin{array}{rcl} \mathrm{stdp}(r,m) & = & \mathrm{prog}(r,m) \wedge r_6 \geq 300 \\ I'_{106}(r,m) & = & r_1 :_{\mathrm{stda}(r,m),m} \tau \end{array}$$

Notice that entry-point 106 uses the "standard" representation of the allocated set, that is, $\mathrm{stda}(r,m)$. 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 $\mathrm{stda}(r,m)(v) = v < r(6)$. However, we can make the restriction that any address to which we attribute the *codeptr* type must use *stda*.

We can abstract *stda* from I'_{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

$$\begin{array}{l} \mathrm{codeptr}(P)\,(a,m)\,v =_{\mathrm{def}} \\ \quad \forall r',m'.\ r'(17) = v \\ \qquad\qquad \wedge\,\mathrm{stdp}(r',m') \\ \qquad\qquad \wedge\,P(\mathrm{stda}(r',m'),m')(r') \\ \qquad\qquad \rightarrow \mathrm{safe}(r',m') \end{array}$$

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 τ is valid. In general if $\tau_1, \tau_2, \ldots$ are valid types, then the predicate

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

is a valid formal-parameters predicate.

Let us define a family of predicates params_k – for various k – as the standard calling sequence of k arguments:

$$\begin{aligned}
&\mathrm{params}_1\ (\tau_1)\ (a,m)\, r =_{\mathrm{def}} \\
&\quad r_1 :_{a,m} \tau_1 \\
&\mathrm{params}_2\ (\tau_1,\tau_2)\ (a,m)\, r =_{\mathrm{def}} \\
&\quad r_1 :_{a,m} \tau_1 \wedge r_2 :_{a,m} \tau_2 \\
&\mathrm{params}_3\ (\tau_1,\tau_2,\tau_3)\ (a,m)\, r =_{\mathrm{def}} \\
&\quad r_1 :_{a,m} \tau_1 \wedge r_2 :_{a,m} \tau_2 \wedge r_3 :_{a,m} \tau_3
\end{aligned}$$

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

$$\mathrm{stdp}(r,m) \ \rightarrow\ 106 :_{\mathrm{stda}(r,m),m} \mathrm{codeptr}(\mathrm{params}_1(\tau)).$$

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 τ, cont(τ) is the continuation taking a τ argument in register 1. However, the code entry point will also have to take an environment (of type σ) in register 2.

$$\begin{aligned}
&\mathrm{cont}(\tau) =_{\mathrm{def}} \\
&\ \mathrm{existential}(\lambda\sigma.\ \mathrm{record}_2(\mathrm{codeptr}(\mathrm{params}_2(\tau,\sigma)),\ \sigma))
\end{aligned}$$

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 τ in r_1. One must fetch the environment e from $m(v+1)$ into r_2. One must ensure that the standard precondition 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 $\alpha \rightarrow \beta$ takes one argument that is a value of type α, and another argument of type 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 σ, the environment type.

$$\begin{aligned}
&\mathrm{func}(\alpha,\beta) =_{\mathrm{def}} \\
&\ \mathrm{existential}(\lambda\sigma. \\
&\qquad \mathrm{record}_2(\mathrm{codeptr}(\mathrm{params}_3(\alpha,\mathrm{cont}(\beta),\sigma)),\ \sigma))
\end{aligned}$$

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 cont and 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, KKR+86].

9 Recursive Datatypes

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

$$\mathrm{subtype}(\tau_1,\tau_2) =_{\mathrm{def}} \forall a,m,v.\ \tau_1(a,m)(v) \rightarrow \tau_2(a,m)(v).$$

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

$$\mathrm{rec}(f) =_{\mathrm{def}} \forall\tau.\ \mathrm{valid}(\tau) \rightarrow f(\tau) \sqsubseteq \tau \rightarrow \tau(a,m)(v)$$

The recursive types are all types rec(f) for which the least fixed point of the argument function f is 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 $f(\mathit{rec}(f)) \sqsubseteq \mathit{rec}(f)$ and $\mathit{rec}(f) \sqsubseteq f(\mathit{rec}(f))$ hold, and thus the following theorem holds.

$$\frac{\text{preserves\_validity}(f) \qquad \text{monotone}(f)}{\mathit{rec}(f)(a,m)(v) \leftrightarrow f(\mathit{rec}(f))(a,m)(v)}\text{roll\_unroll}$$

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:

$$\begin{aligned}\text{list}(\tau) =_{\text{def}}\ & \text{rec}(\lambda\tau'.\ \text{constty}\ 0 \\ & \quad \cup \quad \text{boxed} \cap \text{record}_2(\text{int}, \tau))\end{aligned}$$

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 $\text{monotone}_2(\text{union})$, where:

$$\begin{aligned}&\text{monotone}_2(f) =_{\text{def}} \forall \tau_1, \tau_2, \tau_1', \tau_2'. \\ &\quad \tau_1 \sqsubseteq \tau_2 \to \tau_1' \sqsubseteq \tau_2' \to f(\tau_1, \tau_1') \sqsubseteq f(\tau_2, \tau_2')\end{aligned}$$

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:

$$\frac{\text{valid\_mono}(f) \qquad \text{valid\_mono}(g)}{\text{valid\_mono}(\lambda\tau.\ (f\ \tau) \cup (g\ \tau))}\text{vm\_union}$$

where valid_mono is:

$$\text{valid\_mono}(f) =_{\text{def}} \text{preserves\_validity}(f) \wedge \text{monotone}(f).$$

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:

$$\begin{aligned}\tau_1 &= c_1 \text{ of int} \mid c_2 \text{ of int} \to \tau_1 \\ \tau_2 &= c_1 \text{ of int} \mid c_2 \text{ of } \tau_2 \to \text{int} \\ \tau_3 &= c_1 \text{ of int} \mid c_2 \text{ of } (\tau_3 \to \text{int}) \to \tau_3 \\ \tau_4 &= c_1 \text{ of int} \mid c_2 \text{ of } ((\tau_4 \to \text{int}) \times \text{int}) \to (\tau_4 \times \text{int})\end{aligned}$$

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

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

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 $\text{func}(\tau_1, \tau_2)$ is monotone in α if τ_1 is antimonotone in α and τ_2 is monotone in α. Note that τ_2 appears inside two nested cont operators, establishing its monotonicity.

These and similar results allow us to prove the validity of the recursive types τ_1, τ_2, τ_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 meta logic).

```
tp: type.
int: tp.
form: tp.
arrow: tp -> tp -> tp.
%infix right 14 arrow.
```

```
tm: tp -> type.
form: tp.
pf: tm form -> type.

lam: (tm T -> tm U) -> tm (T arrow U).
@: tm (T arrow U) -> tm T -> tm U.
%infix left 20 @.
and: tm form -> tm form -> tm form.
%infix right 12 and.
forall: (tm T -> tm form) -> tm form.

and_i:  pf A -> pf B -> pf (A and B).
and_e1:  pf (A and B) -> pf A.
and_e2:  pf (A and B) -> pf B.
forall_i: ({X:tm T}pf (A X))
  -> pf (forall A).
forall_e: pf(forall A)
  -> {X:tm T}pf (A X).
```

A metalogic (Twelf) type is a `type`, an object-logic type is a `tp`, and a programming-language type is a `ty` (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 `(tm T)` in the metalogic. Quantifying at the metalevel allows us to encode polymorphic object-level types. Terms of type `(pf A)` are terms representing proofs of object formula `A`.

The declarations beginning with `lam` introduce constants for constructing terms and formulas. Note that the universal quantifier `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 : tp = state arrow int arrow form.
```

```
ref : tm (ty arrow ty) =
 lam3 [T][S][V] fst S @ V and
 readable @ V and T @ S @ (snd 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 @ 0 @ T) @ (field @ 1 @ U)).

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. For 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 λ-calculus. The following theorem justifies the use of `rec` in the definition of `list`.

```
vm_listf : pf (validtype @ T) ->
 pf (valid_mono @ (listf T)) =
 [P:pf (validtype @ T)]
 (vm_union vm_constty
  (vm_intersect vm_boxed
   (vm_record2
    (vm_validtype P) vm_id))).
```

This theorem illustrates the application of one lemma per type constructor. These are the lemmas stating that the constructor preserves validity and monotonicity. The `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.

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:

```
datatype exp = APP of exp * exp
             | LAM of exp -> 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 the Banach fixed point theorem on complete metric spaces.

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_ω. Our longer-range plan is to cover more of 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_ω polymorphic λ-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 recursive 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 `vm_list` theorem in the previous section whose proof uses exactly as many lemma constructors as the description of the `listf` type uses type constructors.

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

[Gir72] J.-Y. Girard. *Interprétation Fonctionnelle et Elimination des Coupures dans l'Arithmétique d'Ordre Supérieur*. PhD thesis, University of Paris VII, 1972.

[KKR+86] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. ORBIT: An optimizing compiler for Scheme. *SIGPLAN Notices (Proc. Sigplan '86 Symp. on Compiler Construction)*, 21(7):219–33, July 1986.

[LST99] Christopher League, Zhong Shao, and Valery Trifonov. Representing java classes in a typed intermediate language. In *Proc. 1999 ACM SIGPLAN International Conference on Functional Programming (ICFP '99)*, page (to appear), New York, 1999. ACM Press.

[MCGW98] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *ACM Workshop on Types in Compilation*, Kyoto, Japan, March 1998.

[MMH96] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 271–283. ACM Press, January 1996.

[MP88] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502, July 1988.

[MPS86] David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Computation*, 71(1/2):95–130, 1986.

[MV96] J.C. Mitchell and R. Viswanathan. Effective models of polymorphism, subtyping and recursion. In *23rd International Colloquium on Automata, Languages, and Programming*. Springer-Verlag, 1996.

[MWCG98] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *POPL '98: 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–97. ACM Press, January 1998.

[Nec97] 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.

[Nec98] George Ciprian Necula. *Compiling with Proofs*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, September 1998.

[Pfe91] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[PS99] 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.

[Rey74] John C. Reynolds. Towards a theory of type structure. In *Proc. Paris Symp. on Programming*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425, Berlin, 1974. Springer.

[Sha98] Zhong Shao. Typed cross-module compilation. In *Proc. 1998 ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, pages 141–152, New York, 1998. ACM Press.

[WLAG93] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proc. 14th ACM Symposium on Operating System Principles*, pages 203–216, New York, 1993. ACM Press.