Proof-Carrying Code

George C. Necula

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213–3891

necula@cs.cmu.edu

Abstract

This paper describes *proof-carrying code* (PCC), a mechanism by which a host system can determine with certainty that it is safe to execute a program supplied (possibly in binary form) by an untrusted source. For this to be possible, the untrusted code producer must supply with the code a *safety proof* that attests to the code's adherence to a previously defined safety policy. The host can then easily and quickly validate the proof without using cryptography and without consulting any external agents.

In order to gain preliminary experience with PCC, we have performed several case studies. We show in this paper how proof-carrying code might be used to develop safe assembly-language extensions of ML programs. In the context of this case study, we present and prove the adequacy of concrete representations for the safety policy, the safety proofs, and the proof validation. Finally, we briefly discuss how we use proof-carrying code to develop network packet filters that are faster than similar filters developed using other techniques and are formally guaranteed to be safe with respect to a given operating system safety policy.

1 Introduction

High-level programming languages are designed and implemented with the assumption of a closed world. Taking ML as an example, the programmer must normally assume that all components of the program are written in ML in order to establish that the program will have the properties conferred by type safety. In practice, however, programs often have some components written in ML and others in a different language (perhaps C or even assembly language). In such situations, we lose the guarantees provided by the design of ML unless extremely expensive mechanisms (such as sockets and processes) are employed. In implementation terms, it is extremely difficult to determine whether the invariants of the ML heap will be respected by the foreign code, and so we must use some kind of expensive firewall or simply live dangerously.

This problem is exacerbated in the realms of distributed and web computing, particularly when mobile code is allowed. In this kind of situation, agent A on one part of the network might write a component of the software system in ML, compile it to native machine code, and then transmit it to an agent B on another node for execution. How does agent A convince agent B that the native code has the type-safety properties shared by all ML programs, and furthermore that it respects the representation invariants chosen for maintaining the state of B's heap?

There are many other manifestations of the same problem. For example, in the realm of operating systems, it is often profitable to allow application programs to run within the same address space as the operating-system kernel. Once again, the problem is how can the kernel know that the inherently untrusted application code respects the kernel's internal invariants. The problem here seems even worse in practice, because the kinds of properties required of the application code are difficult in the sense that standard type systems cannot express them easily. For example, in the SPIN kernel [1], there are often basic requirements about the proper use of synchronization locks that would be hard, if not impossible, to express in the ML or Modula-3 type systems.

This research was sponsored in part by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software," ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.

To appear in the Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97), Paris, France, January 15–17, 1997.

In the situations described above, a *code consumer* must somehow become convinced that the code supplied by an untrusted *code producer* has some (previously agreed upon) set of properties. Sometimes this is referred to as establishing "trust" between the consumer and producer. Cryptography can be used to ensure that the code was produced by a trusted person or compiler [1, 13]. This scheme is weak because of its dependency on personal authority—even trusted persons, or compilers written by them, can make errors occasionally or even act maliciously.

In this paper, we present *proof-carrying code* (PCC for short), which is a mechanism for dealing with these problems. With proof-carrying code, the code producer is required to create a *safety proof* that attests to the fact that the code respects a formally defined safety policy. Then, the code consumer is able to use a simple and fast *proof validator* to check, with certainty, that the proof is valid and hence the foreign code is safe to execute.

There is an analogy between safety proofs and types. The analogy carries over to proof validation and type checking. With this analogy in mind we note that most attempts to tamper with either the code or the proof result in a validation error. In the few cases when the code and the proof are modified such that validation still succeeds, the new code is also safe. This is why we consider proof-carrying code to be intrinsically safe, without need for external authentication or cryptography.

In a previous paper [11], we have already shown how proof-carrying code can be used to implement safe and very fast network packet filters. In this paper, we provide more of the necessary technical details and theorems that establish the soundness and adequacy of our certification scheme, as well as present a second case study involving the extension of a run-time system for an ML implementation. We begin with an overview of the stages involved in the creation and use of proof-carrying code. Then, we present the case study of extending a simplified form of the run-time system of the TIL compiler [15] for Standard ML. In doing so, we show a sample formal system for PCC and state the necessary theorems for soundness and adequacy of the methodology. We continue with a brief description of the network packet filter example from our previous paper. After these case studies, we discuss some of the problems involved in generating the proofs, as well as some other engineering matters. Finally, we summarize what has been accomplished so far and where we see the most interesting directions for further research.

2 Proof-Carrying Code

Figure 1 shows the typical process of generating and using proof-carrying code. The whole process is centered around the *safety policy*, which is defined and made public by the code consumer. Through this policy, the code consumer specifies precisely under what conditions it considers the execution of a foreign program to be safe.

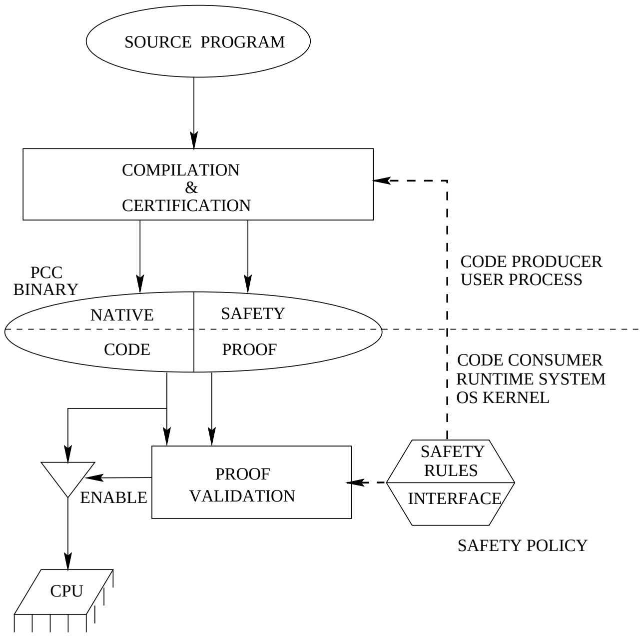

Figure 1: Overview of Proof-Carrying Code.

The safety policy consists of two main components: the *safety rules* and the *interface*. The safety rules describe all authorized operations and their associated safety preconditions. The interface describes the calling conventions between the code consumer and the foreign program, that is the invariants holding when the consumer invokes the foreign code and the invariants that the foreign code must establish before calling functions provided by the consumer, or before returning to the consumer. In the analogy with types, the safety rules are the typing rules and the interface is the signature against which the foreign module is compiled.

The life of a PCC binary spans three stages. In the first stage—called *certification*—the code producer compiles (or assembles) and generates a proof that a source program adheres to the safety policy. In the general case, certification is essentially a form of program verification with respect to the specification described by the safety policy. In addition, a proof of successful verification is produced and suitably encoded to yield the safety proof, which together with the native code component forms the PCC binary. The code producer can store the resulting PCC binary for future use, or can deliver it to code consumers for execution.

In the second stage—called *validation*—a code consumer validates the proof part of a PCC binary presented for execution and loads the native code component for execution. The validation is quick and driven by a straightforward algorithm. It is only the implementation of this simple algorithm that the consumer must trust in addition to the soundness of its safety policy.

The existence of the proof allows for the verification process to be performed off-line and only once for a given program, independently of the number of times it is executed. This has important engineering advantages, especially in cases where verification is hard and time consuming or requires user interaction. In such cases it would be undesirable to perform verification at the consumer site.

Finally, in the last stage of the process the code consumer executes the machine-code program possibly many times. This stage can proceed without performing additional run-time checks because the previous validation stage ensures that the code obeys the safety policy.

This completes our overview of the general proof-carrying code technique. Before we can attempt a practical implementation of PCC, we must decide on concrete representations for the safety policy, safety proofs and their validation procedure. We present next a summary of our current choices and continue in the next section with the details and formal adequacy theorems.

In our current experiments we use extensions of first-order predicate logic as the basis for formalizing the safety policy. The extensions are predicates denoting application-specific safety requirements, together with their derivation rules. In this setup, the interface part of the safety policy consists of a set of precondition and postcondition predicates for the foreign function and the functions exported by the code consumer. The safety rules are expressed as a Floyd-style verification condition generator, which given the program and a set of preconditions and postconditions produces a verification condition predicate (VC) in our logic. The VC has the property that if it can be proved using the proof rules in our logic, then the program satisfies the safety invariants. In this case the safety proof is an appropriate encoding of a proof of the VC predicate, proof is reduced to theorem proving in our logic and validation to proof checking. For the particular safety policy of the extensions to the TIL run-time system, we show that the above choices are adequate.

3 Case Study: Safe Extensions of the TIL Run-Time System

The practice of software development in languages such as ML and Haskell often involves extending the run-time system, usually by writing C code, to implement new primitive types and operations or functionality that is not easily programmed in the high-level language. This raises the question of how to ensure that the foreign code respects the basic assumptions of the run-time system. Even without considering user-extensions, the run-time systems of high-level languages usually include a sizeable part written in unsafe languages such as C or even assembly language. The mechanism that allows an untrusted user to safely extend the run-time can also be used by a small kernel of the run-time system to bootstrap the rest, increasing the level of confidence in the system.

We propose the use of proof-carrying code to allow arbitrary untrusted users to safely link foreign functions to a safe programming language run-time system. For this to be possible the compiler designer defines the safety policy, which is basically a formal description of the data-representation invariants to be preserved and calling conventions to be obeyed by foreign functions. Then, the user produces and attaches to the foreign code a safety proof attesting to the preservation of the invariants.

To make the presentation more concrete we show in detail how we use PCC to develop safe DEC Alpha assembly-language [14] extensions to a simplified version of the run-time system of the TIL [15] compiler for Standard ML [8]. For the purposes of this paper, we consider here only a small example and make some simplifying assumptions about TIL. (These are described later.) Scaling the technique to the entire Standard ML language is subject of current research.

```
datatype T = Int of int | Pair of int * int

fun sum (l : T list) =
 let
  fun foldr f nil a = a
    | foldr f (h::t) a = foldr f t (f(a, h))
 in
  foldr (fn (acc, Int i) => acc + i
          | (acc, Pair (i, j)) => acc + i + j)
        l 0
 end
```

Figure 2: The Standard ML source program.

Consider the Standard ML program fragment shown in Figure 2. This program defines a union type `T` and a function `sum` that adds all the integers in a `T list`. The plan for the rest of this section is to define a safety policy for extensions to the TIL run-time system and then prove the type safety of a hand-optimized assembly language version of the `sum` function.

Establishing a Safety Policy

The first order of business is to define the safety policy for the TIL run-time system in the presence of foreign functions. This is the job of the compiler designer, or a trusted person that is familiar with the data-representation conventions and basic invariants maintained by the TIL compiler and run-time system.

The safety policy in our case requires that foreign code maintains the data-representation invariants chosen by the TIL compiler. Data representation in TIL is type directed and the types involved in our example are the following:

$$\tau ::= \texttt{int} \mid \tau_1 * \tau_2 \mid \tau_1 + \tau_2 \mid \tau\ \texttt{list}$$

For convenience we use `T` as an abbreviation for the type $\texttt{int} + (\texttt{int} * \texttt{int})$. For this subset of ML types, the TIL data-representation rules are as follows: an integer value is represented as an untagged 32-bit machine word; a pair is represented as a pointer to a sequence of two memory locations containing values of appropriate types; a value of type $\tau_1 + \tau_2$ is represented as a pointer to a pair of locations containing respectively the constructor value (0 for $\texttt{inj}_l$ and 1 for $\texttt{inj}_r$) and the value carried by the constructor; the empty list is represented as the value 0 and the non-empty list as a pointer to a list cell. See Figure 4 for examples of TIL representations of several SML values.

The compiler designer describes formally the data-representation strategy by means of a typing judgment, $m \vdash e : \tau$, where e is an expression and m is a memory state. The memory state is part of the judgment because some of the types are represented as pointers, which are only valid in certain states of the memory (e.g., after the underlying value has been allocated). The sets of expressions and memory states for our example, are defined as follows:

$$\begin{array}{lcl} e & ::= & n \mid \mathbf{r}_i \mid \mathbf{sel}(m, e) \mid e_1 + e_2 \\ m & ::= & \mathbf{r_m} \mid \mathbf{upd}(m, e_1, e_2) \end{array}$$

where n is a 32-bit integer literal, $\mathbf{r}_i$ are the DEC Alpha machine registers, $\mathbf{r_m}$ is a pseudo register holding the state of memory during the computation, $\mathbf{sel}(m, e)$ denotes the contents of the location e in memory state m, and $\mathbf{upd}(m, e_1, e_2)$ denotes a memory state obtained from the old state m after updating the location e_1 with

```
val r0 : int = 5
val r1 : int * int = (2, 3)
val r2 : T = Pair r1
val r3 : T = Int 6
val r4 : T list = [r3, r2]
```

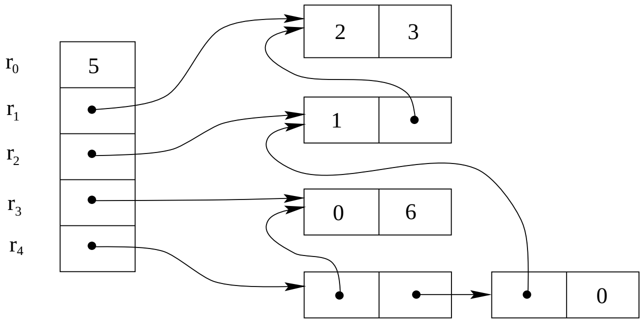

Figure 4: Data Representation in TIL. Each box represents a machine word.

e_2. To simplify the memory-safety aspect of the safety policy, the compiler designer introduces an additional type, called `addr`. This type is used for expressions whose value is a memory address that can be safely read.

Based on the known data-representation strategy used by the TIL compiler, the compiler designer defines the typing judgment by a set of inference rules, as shown in Figure 3. We only show the elimination rules for the typing judgment because for the example at hand we are not concerned with constructing values of non-base types. Also, for the purpose of this paper we ignore the overflow semantics of addition in Standard ML.

There is more to the safety policy than just the typing rules presented so far. For illustration purposes, we shall present the remaining mechanisms of the safety policy in the context of a concrete example of a foreign function.

The Foreign Function

In our experiment, the code producer writes a DEC Alpha assembly language implementation of the `sum` function, as shown in Figure 5. This code assumes that register $\mathbf{r}_0$ contains the argument of type `T list` on entry and the integer result on exit. The registers $\mathbf{r}_1$, $\mathbf{r}_2$ and $\mathbf{r}_3$ are used as temporaries. The purpose of the `INV` instruction will be explained shortly. This code is written to obey the TIL data-representation strategy, and it is this fact that must be proved for the consumer.

Note that the above assembly-language program is optimized by hand. One of our goals is to show that proof-carrying code does not pose restrictions on using

$$\frac{m \vdash e : \tau_1 * \tau_2}{m \vdash e : \texttt{addr} \ \wedge \ m \vdash e+4 : \texttt{addr} \ \wedge \ m \vdash \mathbf{sel}(m,e) : \tau_1 \quad \wedge \quad m \vdash \mathbf{sel}(m,e+4) : \tau_2}$$

$$\frac{m \vdash e : \tau_1 + \tau_2}{m \vdash e : \texttt{addr} \ \wedge \ m \vdash e+4 : \texttt{addr} \ \wedge \ \mathbf{sel}(m,e) = 0 \supset m \vdash \mathbf{sel}(m,e+4) : \tau_1 \ \wedge \ \mathbf{sel}(m,e) \neq 0 \supset m \vdash \mathbf{sel}(m,e+4) : \tau_2}$$

$$\frac{m \vdash e : \tau \ \texttt{list} \qquad\qquad e \neq 0}{m \vdash e : \texttt{addr} \ \wedge \ m \vdash e+4 : \texttt{addr} \ \wedge \ m \vdash \mathbf{sel}(m,e) : \tau \quad \wedge \quad m \vdash \mathbf{sel}(m,e+4) : \tau \ \texttt{list}}$$

$$\frac{m \vdash e_1 : \texttt{int} \qquad m \vdash e_2 : \texttt{int}}{m \vdash e_1 + e_2 : \texttt{int}} \qquad \frac{}{m \vdash 0 : \texttt{int}}$$

Figure 3: The typing rules.

				%$\mathbf{r}_0$ is `l`
0	`sum` :	`INV`	$\mathbf{r_m} \vdash \mathbf{r}_0 : \texttt{T list}$	
				%$\mathbf{r}_1$ is `acc`
1		`MOV`	$\mathbf{r}_1, 0$	%Initialize `acc`
2	L_2	`INV`	$\mathbf{r_m} \vdash \mathbf{r}_0 : \texttt{T list} \wedge \mathbf{r_m} \vdash \mathbf{r}_1 : \texttt{int}$	
				%Loop invariant
3		`BEQ`	$\mathbf{r}_0, L_{14}$	%Is list empty?
4		`LD`	$\mathbf{r}_2, 0(\mathbf{r}_0)$	%Load head
5		`LD`	$\mathbf{r}_0, 4(\mathbf{r}_0)$	%Load tail
6		`LD`	$\mathbf{r}_3, 0(\mathbf{r}_2)$	%Load constructor
7		`LD`	$\mathbf{r}_2, 4(\mathbf{r}_2)$	%Load data
8		`BEQ`	$\mathbf{r}_3, L_{12}$	%Is an integer?
9		`LD`	$\mathbf{r}_3, 0(\mathbf{r}_2)$	%Load `i`
10		`LD`	$\mathbf{r}_2, 4(\mathbf{r}_2)$	%Load `j`
11		`ADD`	$\mathbf{r}_2, \mathbf{r}_3, \mathbf{r}_2$	%Add `i` and `j`
12	L_{12}	`ADD`	$\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_1$	%Do the addition
13		`BR`	L_2	%Loop
14	L_{14}	`MOV`	$\mathbf{r}_0, \mathbf{r}_1$	%Copy result in $\mathbf{r}_0$
15		`RET`		%Result is in $\mathbf{r}_0$

Figure 5: DEC Alpha assembly language implementation of the `sum` function.

register allocation, scheduling or other low level optimization techniques.

For the rest of this section we consider the foreign program represented as a vector Π of instructions. With this convention a program point is an index in the vector Π.

Computing the Verification Condition

Our basic method to check compliance with the safety rules is based on computing a Floyd-style verification condition [4] for the foreign function. This is a predicate in first-order logic with the property that its validity with respect to the first-order logic rules and the typing rules is a sufficient condition for ensuring compliance with the safety policy. Both the code producer and the code consumer compute the verification condition: the code producer for the purpose of proving it, and the code consumer to ensure that the foreign code is accompanied contains a valid proof of it.

To use a Floyd-style verification condition generator, all of the loop invariants must be given as well as the interfaces for all functions being called. The invariant associated with the loop starting at L_2 is

$$\mathbf{r_m} \vdash \mathbf{r}_0 : \texttt{T list} \ \wedge \ \mathbf{r_m} \vdash \mathbf{r}_1 : \texttt{int}$$

and the interface for the function `sum` is given as a precondition and a postcondition as shown below:

$$\begin{array}{rcl} \mathit{Pre} & \equiv & \mathbf{r_m} \vdash \mathbf{r}_0 : \texttt{T list} \\ \mathit{Post} & \equiv & \mathbf{r_m} \vdash \mathbf{r}_0 : \texttt{int} \end{array}$$

In general, the loop invariant for type-safety policies is the conjunction of all typing predicates for the registers that are live at the invariant point. The interfaces are derived similarly from the function types.

For flexibility, we allow invariants to be associated with arbitrary points in the program, not necessarily part of loops. These points are marked in the program by `INV` pseudo instructions,[1] and their set is denoted by *Inv*. For such a point i, we write Inv_i to denote the corresponding invariant. To simplify the presentation, we assume that the code consumer prepends the instruction `INV` *Pre* to each untrusted program before analyzing it. With this convention we have that $0 \in \mathit{Inv}$ and $\mathit{Inv}_0 = \mathit{Pre}$.

The verification condition generator defined in Figure 6 computes a vector VC of predicates, one for each instruction. The notation $[e/\mathbf{r}_i]P$ stands for the predicate obtained from P by substituting the expression e for all occurrences of $\mathbf{r}_i$.

[1] In practice the invariants are kept separate from the code, allowing the code to be executed directly by the physical processor.

$$VC_i = \begin{cases} [\mathbf{r}_s + op/\mathbf{r}_d]\,VC_{i+1}, & \textit{if } \Pi_i = \text{ADD } \mathbf{r}_s, op, \mathbf{r}_d \\ \mathbf{r_m} \vdash \mathbf{r}_s + n : \texttt{addr} \wedge [\mathbf{sel}(\mathbf{r_m}, \mathbf{r}_s + n)/\mathbf{r}_d]\,VC_{i+1}, & \textit{if } \Pi_i = \text{LD } \mathbf{r}_d, n(\mathbf{r}_s) \\ (\mathbf{r}_s = 0 \supset VC_{i+n+1}) \wedge (\mathbf{r}_s \neq 0 \supset VC_{i+1}), & \textit{if } \Pi_i = \text{BEQ } \mathbf{r}_s, n \\ Post, & \textit{if } \Pi_i = \text{RET} \\ \mathcal{I}, & \textit{if } \Pi_i = \text{INV } \mathcal{I} \end{cases}$$

Figure 6: The verification condition generator.

The VC function is well defined if every loop in the program contains at least one invariant instruction. Our current implementation requires that every backward-branch target be an invariant instruction. In these conditions the entire vector VC can be computed in one pass through the program.

Based on the vector VC, we define the verification condition for the entire program as follows:

$$VC(\Pi, Inv, Post) = \forall \mathbf{r}_i. \bigwedge_{i \in Inv} Inv_i \supset VC_{i+1}$$

For our example program, the VC predicate has two conjuncts, one for the precondition and another for the invariant associated with L_2. The first conjunct corresponds to the control path from the function entry point to the start of the loop. This conjunct says that the loop invariant is established when the loop conditional is first executed:

$$\mathbf{r_m} \vdash \mathbf{r}_0 : \texttt{Foo list} \supset (\ \mathbf{r_m} \vdash \mathbf{r}_0 : \texttt{Foo list} \wedge \\ \mathbf{r_m} \vdash 0 : \texttt{int})$$

The second conjunct corresponds to the rest of the program and says both that the the loop invariant is preserved around the loop and that it entails the postcondition when the loop finishes. This part of the VC predicate is more complicated and we do not show it here.

Soundness of VC-based Certification

The VC predicate as defined above is proved by the code producer and a proof of it constitutes the safety proof. We discuss in this section the adequacy of using the proof of the VC predicate as a safety proof, and we defer to Sections 5 and 6 the more difficult problem of proof search.

We write $\triangleright P$ when the predicate P can be proved using the inference rules from Figure 3 and the proof rules of first-order predicate logic, a few of which are shown in Figure 7. Note that the implication introduction rule is hypothetical in the assumption u and the $\forall$ introduction rule is schematic in v. These side conditions must be satisfied for a proof to be valid.

$$\frac{\triangleright P \qquad \triangleright R}{\triangleright P \wedge R}\,\mathbf{and\_i} \qquad \frac{\begin{array}{c}\overline{\triangleright P}^{\,u} \\ \vdots u \\ \triangleright R\end{array}}{\triangleright P \supset R}\,\mathbf{impl\_i}^u \qquad \frac{\begin{array}{c} v \\ \vdots v \\ \triangleright [v/x]P\end{array}}{\triangleright \forall x\,.\,P}\,\mathbf{all\_i}^v$$

Figure 7: Fragment of the first-order predicate logic proof rules.

In Appendix A we prove the soundness of using the proof of the VC predicate as the basis for safety certification. We first formalize the execution of assembly-language programs on the DEC Alpha processor using an abstract machine. Then, we show that any program with a valid verification condition, when executed on the abstract machine starting in a state that satisfies the precondition, will reference only memory locations that are defined valid by the typing rules. Furthermore, if the program terminates then the final state satisfies the postcondition. This is stated here informally as Theorem 3.1 and then formalized in Appendix A.

Theorem 3.1 *For any program Π, set of invariants Inv and postcondition Post such that $\Pi_0 = \texttt{INV}\ Pre$, if $\triangleright\ VC(\Pi, Inv, Post)$ and the initial state satisfies the precondition Pre, then the program reads only from valid memory locations as they are defined by the typing rules, and if it terminates, it does so in a state satisfying the postcondition.*

The proof of soundness of VC-based certification from Appendix A is much simpler than other correctness arguments for Floyd's VC generators, mainly because of the more precise definition of programs, invariants and program points for assembly language programs than for flowcharts.

Safety Proofs

We argued in the previous section and we prove in Appendix A that a proof of validity of the VC predicate is sufficient to ensure compliance with the safety policy.

The safety proof must therefore be a suitable encoding of a derivation $\triangleright\, VC(\Pi, \mathit{Inv}, \mathit{Post})$.

We use a two-stage encoding of derivations. In the first stage we represent predicates and proofs as objects in the Edinburgh Logical Framework (also referred to as LF) [5]. In the second stage, we encode LF objects in a compact binary format, suitable for storage or transmission to code consumers. We shall discuss here in detail only the LF representation.

LF has been designed as a meta language for high-level specification of logics and provides natural support for the management of binding operators and hypothetical and schematic judgments. The LF type theory is a language with entities of three levels: objects, types and kinds. The abstract syntax of these entities is shown below:

$$\begin{array}{llcl} \text{Kinds} & K & ::= & \mathtt{Type} \mid \Pi x{:}A.K \\ \text{Types} & A & ::= & a \mid A\,M \mid \Pi x{:}A_1.A_2 \\ \text{Objects} & M & ::= & c \mid x \mid M_1\,M_2 \mid \lambda x{:}A.M \end{array}$$

We represent our logic in LF by means of a signature L that assigns types to a set of constants describing the syntax of expressions and predicates, and the proof rules of our logic. In LF, judgments are represented as types and judgment derivations as objects whose type is the representation of the judgments they prove. Type checking in the LF type discipline can then be used to check logic proofs.

We start now to present the signature L. First, we define the LF types `exp` of expressions, `pred` of predicates and `tp` of ML types. All of these are atomic LF types:

$$\begin{array}{lcl} \mathtt{exp} & : & \mathtt{Type} \\ \mathtt{pred} & : & \mathtt{Type} \\ \mathtt{tp} & : & \mathtt{Type} \end{array}$$

For each expression and predicate constructor we define an LF constant as shown below.

$$\begin{array}{lcl} 0 & : & \mathtt{exp} \\ > & : & \mathtt{exp} \to \mathtt{exp} \to \mathtt{exp} \\ + & : & \mathtt{exp} \to \mathtt{exp} \to \mathtt{exp} \\ \mathtt{true} & : & \mathtt{pred} \\ \mathtt{and} & : & \mathtt{pred} \to \mathtt{pred} \to \mathtt{pred} \\ \mathtt{impl} & : & \mathtt{pred} \to \mathtt{pred} \to \mathtt{pred} \\ \mathtt{all} & : & (\mathtt{exp} \to \mathtt{pred}) \to \mathtt{pred} \\ \mathtt{hastype} & : & \mathtt{exp} \to \mathtt{exp} \to \mathtt{tp} \to \mathtt{pred} \end{array}$$

Note that binding predicate constructors are represented as higher-order LF constants, effectively shifting the machinery related to bound variables and substitution from our logic to LF. This relies on representing variables as LF variables.

The LF representation function $\ulcorner \cdot \urcorner$ is inductively defined on the structure of expressions and predicates. A few definition cases are shown below:

$$\begin{array}{rcl} \ulcorner P \land R \urcorner & = & \mathtt{and}\, \ulcorner P \urcorner\, \ulcorner R \urcorner \\ \ulcorner \forall x.P \urcorner & = & \mathtt{all}\, (\lambda x{:}\mathtt{exp}.\ulcorner P \urcorner) \\ \ulcorner x \urcorner & = & x \end{array}$$

For the representation of derivations we define a type family indexed by representation of predicates:

$$\mathtt{pf} \quad : \quad \mathtt{pred} \to \mathtt{Type}$$

Following the model of expressions and predicates we define a constant for each proof schema. A few of these constants are shown below:

$$\begin{array}{lcl} \mathtt{and\_i} & : & \Pi p{:}\mathtt{pred}.\Pi r{:}\mathtt{pred}. \\ & & \quad \mathtt{pf}\ p \to \mathtt{pf}\ r \to \mathtt{pf}\ (\mathtt{and}\ p\ \ r) \\ \mathtt{impl\_i} & : & \Pi p{:}\mathtt{pred}.\Pi r{:}\mathtt{pred}. \\ & & \quad (\mathtt{pf}\ p \to \mathtt{pf}\ r) \to \mathtt{pf}\ (\mathtt{impl}\ p\ \ r) \\ \mathtt{all\_i} & : & \Pi p{:}\mathtt{exp} \to \mathtt{pred}. \\ & & \quad (\Pi v{:}\mathtt{exp}.\mathtt{pf}\ (p\ v)) \to \mathtt{pf}\ (\mathtt{all}\ \ p) \end{array}$$

We then extend the representation function $\ulcorner \cdot \urcorner$ to derivations. When doing so care must be taken with hypothetical and schematic judgments, such as the implication introduction and the universal quantification introduction rules shown below. The representation of the conjunction introduction is typical for most other rules.

$$\left\ulcorner \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \triangleright P_1\end{array} \qquad \begin{array}{c}\mathcal{D}_2 \\ \triangleright P_2\end{array}}{\triangleright P_1 \land P_2} \right\urcorner = \mathtt{and\_i}\, \ulcorner P_1 \urcorner\, \ulcorner P_2 \urcorner\, \ulcorner \mathcal{D}_1 \urcorner\, \ulcorner \mathcal{D}_2 \urcorner$$

$$\left\ulcorner \dfrac{\begin{array}{c}\overline{\triangleright P_1}\,u \\ \vdots\ \mathcal{D}_u \\ \triangleright P_2\end{array}}{\triangleright P_1 \supset P_2}\,u \right\urcorner = \mathtt{impl\_i}\, \ulcorner P_1 \urcorner\, \ulcorner P_2 \urcorner\, (\lambda u{:}\mathtt{pf}\, \ulcorner P_1 \urcorner.\ulcorner \mathcal{D}_u \urcorner)$$

$$\left\ulcorner \dfrac{\begin{array}{c}\mathcal{D}_v \\ \triangleright [v/x]P_x\end{array}}{\triangleright \forall x.P_x} \right\urcorner = \mathtt{all\_i}\, (\lambda x{:}\mathtt{exp}.\ulcorner P_x \urcorner)\, (\lambda v{:}\mathtt{exp}.\ulcorner \mathcal{D}_v \urcorner)$$

The implication introduction rule introduces the hypothesis labelled u for the purpose of deriving P_2. Checking an instance of this rule schema involves verifying that it discharges properly the hypothesis u. Equivalently, the derivation $\mathcal{D}_u$ must be hypothetical in u. This is expressed naturally in LF by representing the hypothesis as a variable bound in $\mathcal{D}_u$. Finally, the LF representation of our logic contains also the representation of the application-specific proof rules from Figure 3. Their representation is straightforward because they do not involve hypothetical judgments. As an example we show below the LF representation of the typing rules for lists:

$$
\begin{array}{l}
\texttt{tp\_list} : \Pi m\!:\!\texttt{exp}.\Pi e\!:\!\texttt{exp}.\Pi t\!:\!\texttt{tp}. \\
\quad \texttt{pf}\ (\texttt{hastype}\ m\ e\ (\texttt{list}\ t)) \rightarrow \texttt{pf}\ (\texttt{neq}\ e\ 0) \rightarrow \\
\quad \texttt{pf}\ (\texttt{and}\ (\texttt{and}\ (\texttt{hastype}\ m\ e\ \texttt{addr}) \\
\qquad\qquad\qquad\quad (\texttt{hastype}\ m\ (\texttt{sel}\ m\ e)\ t)) \\
\qquad\quad (\texttt{and}\ (\texttt{hastype}\ m\ (+\ e\ 4)\ \texttt{addr}) \\
\qquad\qquad\qquad\quad (\texttt{hastype}\ m\ (\texttt{sel}\ m\ (+\ e\ 4))\ (\texttt{list}\ t))))
\end{array}
$$

The purpose of the LF representation is to use the LF type-checking algorithm for checking the validity of proofs. This has the advantage that the code consumer need only trust one implementation of proof checking. Other logics can be encoded and their derivations checked by the same type checker just by changing the signature. Furthermore, the LF typing rules are so simple that a naive implementation takes only a couple of pages of code. This is important because it minimizes the concern that the type checker must be trusted.

We do not show here the typing rules for full LF. Instead we define in Appendix B a fragment of LF that is expressive enough to encode first-order and higher-order logics but is strictly simpler and less expressive than full LF. For this fragment, called LF_0, we show the typing rules and the adequacy of the encoding of predicates and derivations. The statement of the adequacy theorem is shown below. The LF_0 typing judgment $\Gamma \vdash_L M :_c A$ says that the object M is canonical of type A with respect to the type assignment Γ and the signature L.

Theorem 3.2 (Adequacy for first-order logic) *There is a bijection $\ulcorner\cdot\urcorner$ between derivations $\mathcal{D} :: \triangleright P$ with parameters v_i $(i = 1, \ldots, n)$ and from hypotheses $u_j :: \triangleright P_j$ $(j = 1, \ldots, m)$ and canonical LF objects $\ulcorner\mathcal{D}\urcorner$ such that*

$$v_i :_a \texttt{exp}, u_j :_a \texttt{pf}\ \ulcorner P_j \urcorner \vdash_L \ulcorner\mathcal{D}\urcorner :_c \texttt{pf}\ \ulcorner P \urcorner$$

The following corollary of the adequacy theorem states that LF type checking is a sufficient procedure for checking safety proofs.

Corollary 3.3 *If P is a closed predicate and M is a canonical LF object such that $\cdot \vdash_L M :_{LF} \texttt{pf}\ \ulcorner P\urcorner$, then there exists a derivation of $\mathcal{D} :: \triangleright P$, that is P is valid. Furthermore, $M = \ulcorner\mathcal{D}\urcorner$.*

A similar theorem is proved by Harper, Honsell and Plotkin [5] for canonical forms in full LF. In LF_0 the proofs are somewhat simpler because of the syntax directed form of typing judgments and canonical forms. A practical advantage of LF_0 over full LF is that definitional equality, responsible for the exponential worst case complexity of LF type checking, is replaced with a localized syntax-directed normalization judgment. For the particular signature L, normalization is only involved in checking instances of the universal quantification elimination rule schema.

Quantitative Results

One motivation for our experiments was to measure the size of safety proofs and the time it takes to validate them for a few simple examples. The safety proof contains the LF representations of the invariants—less the precondition, which is supplied by the code producer—and the proof of the VC predicate. All these LF objects are encoded in a portable and compact binary format.

Recall that the PCC binary contains also the native code. For the example presented in this section, the size of the entire PCC binary is 730 bytes. Of these, the safety proof occupies 420 bytes and the code 60 bytes. The rest (250 bytes) is a fixed-size overhead.

With our implementation of the LF_0 type-checking algorithm, validating the proof for our example takes 1.9ms on a DEC Alpha workstation running at 175MHz. This time is significantly less than it would take a trusted optimizing compiler to generate the same safe extension by compilation of SML source.

4 Case Study: Safe Packet Filters

In another case study, we used proof-carrying code to implement a collection of network packet filters. The details of this experiment are described elsewhere [11], and so here we give only a brief summary of the experiment and our results.

Many modern operating systems provide a facility for allowing application programs to receive packets directly from the network device. Typically, an application is not interested in receiving every packet from the network, but only the small fraction that exhibit a specific property (e.g., an application might want only TCP packets destined for a Telnet port). In such cases, it is highly profitable to allow the application program to specify a boolean function on network packets, and then have this filter run within the kernel's address space. The kernel can then avoid delivering uninteresting packets to the application, thereby saving the cost of many unnecessary context switches. Packet filters are supported by most of today's workstation operating systems [9].

The main technical problem is that application programs are inherently untrusted, and so the kernel must employ some method for ensuring safety. One popular solution, exemplified by the BSD Packet Filter architecture (BPF) [6], is to define a safe programming language for writing packet filters, and then use an interpreter in the kernel to execute them. In the BPF language, for example, filter programs are restricted to be loop-free, and all references to memory are checked at run time to be within the bounds of either the packet data or a statically allocated scratch memory.

In our experiment, we were able to use PCC to define the same safety policy as defined by BPF, and then write a collection of typical packet filters in hand-coded DEC Alpha assembly language. (The VC generator and the abstract machine we use are essentially the same as those shown in Figures 6 and 8, but with branches restricted to be forward-only.) Since the packet filters are written in hand-tuned assembler instead of an interpreted language, they are ten times faster than functionally equivalent packet filters written using BPF, two times faster than packet filters written in the safe subset of Modula-3, and 30% faster than filters developed using software fault isolation [16]. Furthermore, the proofs are small, ranging from 300 to 900 bytes in size, and the validation times are negligible, ranging from 0.3ms to 1.3ms. Note that we use exactly the same implementation of LF type checking as the previous application, with only the signature modified.

5 Generating Safety Proofs

The remaining aspect of our PCC experiment to be discussed is the generation of the safety proofs. There are still many open questions about proof generation, such as scalability to large programs. We currently obtain the proofs by using a very simple theorem prover that produces a witness for every successful proof. There are other possible methods that are likely to work better, especially for larger programs. We discuss some of these in Section 6.

For our experiments, we use the programming language Elf [12] to prove VC predicates and produce LF representation of their proofs. Elf is a logic programming language based on LF. A program in Elf is an LF signature and execution in Elf is search for canonical LF objects inhabiting an LF type in the context of a signature. In our case the program is the signature L and we are interested in finding a closed object M of type $\mathtt{pf}\ \ulcorner VC \urcorner$ for some verification condition VC. If such an object is found, according to Corollary 3.3, it constitutes the canonical LF representation of a proof $\triangleright\ VC$. Incidentally, this is exactly the required safety proof.

Proof search in Elf is performed in depth-first fashion, as in Prolog. With this operational view, the natural deduction style presentation of our logic is not appropriate for proof search, because any of the elimination rules would lead to non-termination. Our solution to this problem is based on the observation that all of the VC predicates in our current experiments are either first-order Horn clauses or first order hereditary Harrop formulas. These fragments of first-order logic admit a complete sequent-style proof system where the declarative meaning of logical connectors coincides with their search-related reading [7]. The resulting proofs are called uniform. The LF representation of a uniform proof system for our logic can then be used as a logic program to perform proof search.

We represent in LF the uniform derivation rules for our logic in a manner similar to the natural deduction representation. We use this representation in Elf to perform a goal-directed search for a uniform derivation of the validity of the VC predicate. We also represent in LF the proof of soundness of uniform derivations with respect to the natural deduction formulation of our logic. We exploit the operational reading of this soundness proof in Elf to convert the uniform derivation of the VC predicate to a natural deduction proof of it.

Each of the LF signatures representing our logic, the uniform proof derivations for it and the soundness of uniform proofs, consist of about 15 constant declarations.

6 Discussion and Future Work

For the type-safety example presented in this paper, we were able to employ simple rules for finding sufficiently strong loop invariants and the interfaces for all functions called. However, in the general case, this is a very difficult problem and the main factor that makes certification hard. One engineering advantage of PCC in this regard is that all of the hard work is done off-line, by the code producer who can employ a variety of tools including costly program analyses or even user interaction.

Another factor that makes the problem simpler than general program verification is that the code producer can allow the certification process to alter the code, perhaps by inserting run-time checks in strategic locations. This would have the tendency to make it easier to generate the proof automatically. For example, if we insert run-time bounds checks before some array operations then it becomes easy to verify that no out-of-bounds array accesses are performed.

Still, it seems unlikely that such a verification-based approach will scale up to programs of a more realistic size. We believe that a more promising technique for producing the proofs would be to rely on a compiler to prove that the target code preserves interesting properties of the source program, such as termination, lack of deadlock, or type safety. This would be generally achieved by instrumenting the compiler to generate proofs of safety in parallel with code transformations.

Currently, we have very little experience with such "certifying compilers." We have implemented a compiler for a small type-safe imperative language with sum and product types. The target language is similar to the source language except that it has only products. In target programs the sum-type values are represented as pairs, in a manner similar to the TIL representation strategy. We employ typing rules similar to those presented in this paper to prove type safety for the target program. We are able to implement the compiler in such a way that the type-safety proof for the source code is transformed into a proof in parallel with the transformation of the code into the target language.

We are exploring the feasibility of extending a more realistic compiler, such as TIL, to generate type-safety proofs. TIL currently preserves the proof of type safety through most of the high-level optimizations by means of typed intermediate languages. To make TIL a certifying compiler, we would need to extend the safety policy presented in this paper to the entire Standard ML language, and then modify the compiler so that it preserves the safety proof through all back-end optimizations and translations. While this seems quite a daunting task, it is encouraging that code scheduling and register allocation, when done correctly, do not change the safety predicate. In fact, it may be a simple correctness criterion for these optimizations that they preserve any safety predicate that the code has initially.

Another aspect of our PCC experiments that deserves some discussion is the choice of the underlying logic for the safety policy and the representation of derivations. A somewhat delicate point is the choice of axioms and proof rules for reasoning about arithmetic. In our experiments, we have chosen the rules and axioms a bit haphazardly, extending the logic as the need arose. While this approach might be workable in some circumstances, widespread use of PCC for, say, safe applets would require that all proof validators adopt the same logic. How to choose the right system may be a difficult task, though in practice this amounts to establishing a kind of standard basis library.

Beyond the matter of arithmetic, we plan to experiment with logics that are more expressive than first-order logic, such as linear logic or temporal logic. Such logics can provide more expressive mechanisms for defining practical safety policies. For example, linear logic might be useful for expressing revocation and single threading of capabilities. Temporal logic could be used possibly to express fairness or lack of deadlock. Also, higher-order logic could be the basis for reasoning about code-generating code.

Working with more expressive logics might require more meta-language machinery than provided by LF. This is the case for linear logic, for example. Another reason to experiment with other representation techniques, is that there are no known decidable criteria for ensuring that an LF signature is a proof of some theorem about a deductive system. If we could encode in an easy-to-check representation theorems like "all type-safe code is also memory safe", then we would have a mechanism by which untrusted users could safely define safety policies.

7 Conclusion

We have presented *proof-carrying code,* a mechanism that allows a code consumer to interact safely with native code supplied by untrusted code producers. PCC does not incur the run-time overhead of previous solutions to this problem. Instead, the code producer is required to generate a proof that attests to the code's safety properties. The kernel can easily check the proofs for validity, after which it is absolutely certain that the code respects the safety policy. Furthermore, PCC binaries are completely tamper-proof; any attempt to alter either the native code or proof in an PCC binary is either detected or harmless with respect to the safety policy.

The main contribution of the work presented in this paper is the principle of staging program verification into certification and proof validation, with the proof acting as a witness that the certification was performed correctly. This staging has great engineering advantages, all based on the intuition that proof checking is believed to be much easier than proof generation.

Application-specific proving strategies—goal directed search, interactive theorem proving or just brute-force search guided by heuristics—and their associated complexity and computational costs are moved off-line to the certification stage. In the validation stage, we only need a simple and reliable proof checker which in many cases is inexpensive enough to be used in performance critical paths. Moreover, the same proof checker covers many practical applications, which increases the reliability of the methodology. Lastly, the certification must be done only once independently how many times the code is used.

We have also shown a way to use standard verification techniques to check type safety at the assembly-language level. This is important for certifying extensions to safe programming languages and as a main building block in constructing certifying compilers. Similar techniques have been applied to assembly language before [2, 3] but neither as a basis for creating safety proofs nor for checking type safety.

We show an encoding of safety proofs as first-order logic derivations in LF. Our contribution in this area is to identify a fragment of LF which is both sufficient for many applications of PCC and also admits a simple and fast type-checking algorithm.

Proof-carrying code is an application of ideas from program verification, logic and type theory, in this case to extend to low-level languages safety properties that are normally enjoyed only by high-level languages. We have shown that this technique is useful both for safe interoperability of programming languages and operating system components. With the growth of interest in highly distributed computing, web computing, and extensible kernels, it seems clear to us that ideas from programming languages are destined to become increasingly critical for robust and good-performing systems.

8 Acknowledgments

This paper is an outgrowth of work done jointly with Peter Lee, who also suggested numerous improvements of the presentation. I also thank Robert Harper who made valuable comments on earlier drafts.

References

[1] Bershad, B., Savage, S., Pardyak, P., Sirer, E. G., Becker, D., Fiuczynski, M., Chambers, C., and Eggers, S. Extensibility, safety and performance in the SPIN operating system. In *Symposium on Operating System Principles* (Dec. 1995), pp. 267–284.

[2] Boyer, R. S., and Yu, Y. Automated proofs of object code for a widely used microprocessor. *J. ACM 43*, 1 (Jan. 1996), 166–192.

[3] Clutterbuck, D., and Carré, B. The verification of low-level code. *IEEE Software Engineering Journal 3*, 3 (May 1988), 97–111.

[4] Floyd, R. W. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, J. T. Schwartz, Ed. American Mathematical Society, 1967, pp. 19–32.

[5] Harper, R., Honsell, F., and Plotkin, G. A framework for defining logics. *Journal of the Association for Computing Machinery 40*, 1 (Jan. 1993), 143–184.

[6] McCanne, S., and Jacobson, V. The BSD packet filter: A new architecture for user-level packet capture. In *The Winter 1993 USENIX Conference* (Jan. 1993), USENIX Association, pp. 259–269.

[7] Miller, D., Nadathur, G., Pfenning, F., and Scedrov, A. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic 51* (1991), 125–157.

[8] Milner, R., Tofte, M., and Harper, R. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990.

[9] Mogul, J. C., Rashid, R. F., and Accetta, M. J. The packet filter: An efficient mechanism for user-level network code. In *ACM Symposium on Operating Systems Principles* (Nov. 1987), ACM Press, pp. 39–51. An updated version is available as DEC WRL Research Report 87/2.

[10] Necula, G. C., and Lee, P. Proof-carrying code. Technical Report CMU-CS-96-165, Computer Science Department, Carnegie Mellon University, Sept. 1996. Also appeared as FOX memorandum CMU-CS-FOX-96-03.

[11] Necula, G. C., and Lee, P. Safe kernel extensions without run-time checking. In *Second Symposium on Operating Systems Design and Implementations* (Oct. 1996), Usenix.

[12] Pfenning, F. Elf: A language for logic definition and verified meta-programming. In *Fourth Annual Symposium on Logic in Computer Science* (Pacific Grove, California, June 1989), IEEE Computer Society Press, pp. 313–322.

[13] Rouaix, F. A Web navigator with applets in Caml. *Proceedings of the 5th International World Wide Web Conference, in Computer Networks and Telecommunications Networking 28*, 7–11 (May 1996), 1365–1371.

[14] Sites, R. L. *Alpha Architecture Reference Manual.* Digital Press, 1992.

[15] Tarditi, D., Morrisett, J. G., Cheng, P., Stone, C., Harper, R., and Lee, P. TIL: A type-directed optimizing compiler for ML. In *PLDI'96 Conference on Programming Language Design and Implementation* (May 1996), pp. 181–192.

[16] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient software-based fault isolation. In *14th ACM Symposium on Operating Systems Principles* (Dec. 1993), ACM, pp. 203–216.

$$
(\rho, pc) \rightarrow \begin{cases} (\rho \circ [\mathbf{r}_s + op/\mathbf{r}_d], pc+1), & \textit{if} \quad \Pi_{pc} = \text{ADDQ} \;\; \mathbf{r}_s, op, \mathbf{r}_d \\ (\rho \circ [\mathbf{sel}(\mathbf{r_m}, \mathbf{r}_s + n)/\mathbf{r}_d], pc+1), & \textit{if} \quad \Pi_{pc} = \text{LDQ} \;\; \mathbf{r}_d, n(\mathbf{r}_s) \quad \textit{and} \quad \boxed{\mathbf{r_m} \vdash \mathbf{r}_s + n : \mathtt{addr}} \\ (\rho, pc+n+1), & \textit{if} \quad \Pi_{pc} = \text{BEQ} \;\; \mathbf{r}_s, n \quad \textit{and} \quad \mathbf{r}_s = 0 \\ (\rho, pc+1), & \textit{if} \quad \Pi_{pc} = \text{BEQ} \;\; \mathbf{r}_s, n \quad \textit{and} \quad \mathbf{r}_s \neq 0 \\ (\rho, pc+1), & \textit{if} \quad \Pi_{pc} = \text{INV} \;\; \mathcal{I} \end{cases}
$$

Figure 8: The abstract machine for the soundness proof.

A Soundness of the VC-based Certification

In this appendix we prove that the VC predicate as defined in the body of the paper is indeed sufficient to ensure compliance with the safety policy. In the context of our example, this means that every memory read operation references a readable address—as defined by the typing rules—and also that upon termination, the postcondition holds. This soundness proof can be easily extended to other examples.

In order to formalize the soundness property, we define an abstract machine that defines safety formally. The state of the machine consists of the value of the program counter (pc) and the state of the machine registers, including the pseudo register $\mathbf{r_m}$.

We view the current state of registers as a substitution (ρ) from register names to values. We write $\rho(\mathbf{r}_i)$ to denote the value of the register $\mathbf{r}_i$ in state ρ. We write $\rho(e)$ and $\rho(P)$ to express substitution of register names with their values in state ρ. We write $\rho \circ [e/\mathbf{r}_i]$ for the state obtained after executing the assignment $\mathbf{r}_i := e$ in state ρ. By definition, a state ρ satisfies a predicate P if and only if $\triangleright \rho(P)$.

With this notation, we define the abstract machine by the set of state transition rules shown in Figure 8. There are several interesting aspects of this abstract machine definition. Firstly, the machine specifies explicit safety conditions, shown boxed in Figure 8. For example, the safety condition for the memory read rule, $\mathbf{r_m} \vdash \mathbf{r}_s + n : \mathtt{addr}$, is satisfied in the current state ρ if $\triangleright \rho(\mathbf{r_m}) \vdash \rho(\mathbf{r}_s) + n : \mathtt{addr}$. This assumes that the application-specific extension of predicate logic (the set inference rules in Figure 3 for our example) is sound with respect to the given safety policy.

Secondly, this machine does not return errors explicitly. Instead the execution halts—due to the lack of appropriate rules—in cases when the safety conditions are not satisfied or invalid instructions are encountered. Lastly, the machine ignores the invariant instructions. This is appropriate because a physical machine does not know how to execute them. Therefore, if we ignore the boxed safety condition in the memory read rule, we obtain a faithful abstraction of the DEC Alpha processor. We show in the rest of this appendix that a valid VC predicate for a program guarantees that, at any moment during its execution, the safety conditions are satisfied, or equivalently, the execution does not halt. Such a program has the same effect if executed on the physical machine, which does not perform the safety checks.

The central result in this appendix is the progress lemma. Informally, this lemma says that if the current state satisfies the VC predicate for the current instruction then either the execution terminates immediately in a state the satisfies the postcondition, or else there is a subsequent state (the execution does not halt there.)

Lemma A.1 (Progress) *For any program Π such that $\Pi_0 = \mathtt{INV}$ Pre, if $\triangleright$ $VC(\Pi, Inv, Post)$ and $\triangleright \rho(VC_{pc})$ then either:*

- *$\Pi_{pc} = RET$, and $\triangleright \rho(Post)$, or*
- *Exists a new state ρ' such that $(\rho, pc) \rightarrow (\rho', pc')$ and $\triangleright \rho'(VC_{pc'})$.*

Proof: The proof is by case analysis of the current instruction.

Case: $\Pi_{pc} = \mathtt{RET}$. Because $\triangleright \rho(VC_{pc})$ and $VC_{pc} = Post$, we conclude that $\triangleright \rho(Post)$.

Case: $\Pi_{pc} = \mathtt{ADD}$ $\mathbf{r}_s, op, \mathbf{r}_d$. From hypothesis $\triangleright \rho([\mathbf{r}_s + n/\mathbf{r}_d] VC_{pc+1})$. By simple substitution manipulation we get that $\triangleright (\rho \circ [\mathbf{r}_s + n/\mathbf{r}_d])(VC_{pc+1})$. The conclusion follows immediately if we pick $pc' = pc + 1$ and $\rho' = \rho \circ [\mathbf{r}_s + n/\mathbf{r}_d]$.

Case: $\Pi_{pc} = \mathtt{LD}$ $\mathbf{r}_d, n(\mathbf{r}_s)$. From hypothesis $\triangleright \rho(\mathbf{r_m} \vdash \mathbf{r}_s + n : \mathtt{addr} \wedge [\mathbf{sel}(\mathbf{r_m}, \mathbf{r}_s + n)/\mathbf{r}_d] VC_{pc+1})$.
From here, using the conjunction elimination rules, it follows that $\triangleright$ $\rho(\mathbf{r_m} \vdash \mathbf{r}_s + n : \mathtt{addr})$, which means that the side condition in the memory read rule of the abstract machine is satisfied. If we pick $pc' = pc+1$ and $\rho' = \rho \circ [\mathbf{sel}(\mathbf{r_m}, \mathbf{r}_s + n)/\mathbf{r}_d]$ we deduce that $\triangleright \rho'(VC_{pc'})$.

Case: $\Pi_{pc} = \mathtt{BEQ}$ $\mathbf{r}_s, n$. We distinguish two cases depending on the value of $\rho(\mathbf{r}_s)$. We only show here the case when $\triangleright \rho(\mathbf{r}_s \neq 0)$. The other case is similar. From the hypothesis we get $\triangleright \rho(\mathbf{r}_s = 0 \supset VC_{pc+1} \wedge \mathbf{r}_s \neq 0 \supset VC_{pc+n+1})$. Using conjunction and then implication elimination we get that $\triangleright \rho(VC_{pc+n+1})$, which is exactly what we have to prove.

Case: $\Pi_{pc} = \texttt{INV}\ \mathcal{I}$. From the hypothesis we have that $\triangleright \rho(\mathcal{I})$. Now we use the validity of the VC predicate. By universal quantification elimination (with the instantiation ρ) and conjunction elimination on the proof of the VC predicate we get that $\triangleright \rho(\mathcal{I} \supset VC_{pc+1})$. Now using implication elimination we get the desired conclusion.

□

Lemma A.2 *For any program Π, set of invariants Inv, and postcondition Post such that $\Pi_0 = \texttt{INV}$ Pre, if $\triangleright VC(\Pi, Inv, Post)$ and the initial state ρ_0 satisfies the precondition Pre, then for any subsequent state ρ of the abstract machine such that $(\rho_0, 0) \to^* (\rho, pc)$, we have that $\triangleright \rho(VC_{pc})$.*

Proof: By induction on the length of the derivation $(\rho_0, 0) \to^* (\rho, pc)$. The base case follows immediately from the hypothesis observing that $VC_0 = Pre$. The inductive case is Lemma A.1.

□

Lemmas A.2 and A.1 can be easily used to show that at any point during the execution of a program with a valid VC predicate, the safety check in the memory load rule is satisfied, and furthermore whenever the program terminates, it does so in a state that satisfies the postcondition. This proves Theorem 3.1 from the main body of the paper.

B Adequacy of the LF Representation of Proofs

In this appendix we introduce LF_0, a fragment of full LF as defined in [5]. The benefits of using LF_0 instead of full LF for proof representation and validation is that LF_0 admits a simpler type-checking algorithm.

When using LF for checking proofs, the signature and the kinds involved can be trusted, as they are designed by the code consumer. This eliminates the need for type checking kinds in LF_0 . Also there are no dependent kinds allowed in LF_0 . Another distinguishing feature of LF_0 is that it only allows second-order constants and first-order abstractions. This is enough for representing a wide array of first-order and higher-order logics [5]. The benefit gained is that the normalization judgment is syntax directed and admits simple and efficient implementations.

Finally, by examination of the LF encoding functions we notice that only LF objects in canonical form are produced. This is in fact a crucial technical detail in the proofs of adequacy in [5]. In LF_0 we define typing judgments only for objects in canonical form, thus simplifying the typing rules and the adequacy proofs. An object is in canonical form if it is in $\beta\eta$–long-normal-form. We write $\Gamma \models_\Sigma M :_c A$ if the object M is in canonical form of type A with respect to the type assignment Γ and the LF_0 signature Σ. This judgment is defined in Figure 9 in terms of the atomic typing judgment $\Gamma \models_\Sigma M :_a A$. An object is atomic if it is a constant or a variable applied to zero or more arguments. Enough arguments must be present such that the application has a non-functional (atomic) type.

One variation from typical presentations of LF is that instead of a definitional equivalence judgment we use a normalization judgment. Furthermore, use of normalization is localized to the `at_app` rule. This makes both the canonical and atomic typing judgments syntax directed, which simplifies the adequacy proofs below.

Abstractions are restricted to first order by the `can_pi` rule, because an atomic type cannot be functional. This in turn, justifies the syntax-directed form of the normalization judgment. In particular, in the `nm_beta` rule, the term $[N'/x]M'$ is known to be in canonical form if M' is canonical and N' has an atomic type.

The following theorem relates the typing judgments of LF_0 with the typing judgment in LF and justifies the claim that LF_0 is a fragment of LF.

Theorem B.1 (Soundness of LF_0)

1. *If $\Gamma \models_\Sigma M :_c A$ then $\Gamma \models_\Sigma M :_{LF} A$.*
2. *If $\Gamma \models_\Sigma M :_a A$ then $\Gamma \models_\Sigma M :_{LF} A$.*
3. *If $\Gamma \models_\Sigma A :_a K$ then $\Gamma \models_\Sigma A :_{LF} K$.*
4. *If $\Gamma \models_\Sigma A :_{LF} \texttt{Type}$ and $A \Downarrow A'$ then $A \equiv_{LF} A'$ and $\Gamma \models_\Sigma A' :_{LF} \texttt{Type}$.*
5. *If $\Gamma \models_\Sigma M :_{LF} A$ and $M \Downarrow M'$ then $M \equiv_{LF} M'$ and $\Gamma \models_\Sigma M' :_{LF} A$.*

Proof: The proof is by simultaneous induction on the structure of LF_0 derivations.

□

We state below the adequacy theorems for expression, predicate and derivation representation as defined by the signature L. The proofs for the adequacy theorems follow closely the model of similar adequacy theorems in [5] and can be found in [10]. Technically, the proofs are somewhat simpler for LF_0 because of the syntax-directed form of the typing judgments and canonical forms. If we extend the signature of first-order predicate logic with first-order proof constants, the adequacy still holds. This means that LF is an adequate representation not only for first-order predicate logic but for all first-order extensions of it.

Canonical Objects

$$\frac{\Gamma, x :_a A \vdash_\Sigma M :_c B \qquad \Gamma \vdash_\Sigma A :_a \texttt{Type}}{\Gamma \vdash_\Sigma \lambda x\!:\!A.M :_c \Pi x\!:\!A.B}\texttt{can\_pi}$$

$$\frac{\Gamma \vdash_\Sigma M :_a A \qquad \Gamma \vdash_\Sigma A :_a \texttt{Type}}{\Gamma \vdash_\Sigma M :_c A}\texttt{can\_at}$$

Atomic Objects

$$\frac{\Gamma(x) = A}{\Gamma \vdash_\Sigma x :_a A}\texttt{at\_var} \qquad \frac{\Sigma(c) = A}{\Gamma \vdash_\Sigma c :_a A}\texttt{at\_ct} \qquad \frac{\Gamma \vdash_\Sigma M :_a \Pi x\!:\!A.B \qquad \Gamma \vdash_\Sigma N :_c A \qquad [N/x]B \Downarrow B'}{\Gamma \vdash_\Sigma M\ N :_a B'}\texttt{at\_app}$$

Atomic Types

$$\frac{\Sigma(a) = K}{\Gamma \vdash_\Sigma a :_a K}\texttt{t\_a} \qquad \frac{\Gamma \vdash_\Sigma A :_a B \to K \qquad \Gamma \vdash_\Sigma M :_c B}{\Gamma \vdash_\Sigma A\ M :_a K}\texttt{t\_pi}$$

Normalization

$$\frac{}{a \Downarrow a}\texttt{na\_a} \qquad \frac{A \Downarrow A' \qquad M \Downarrow M'}{A\ M \Downarrow A'\ M'}\texttt{na\_app} \qquad \frac{A \Downarrow A' \qquad B \Downarrow B'}{\Pi x\!:\!A.B \Downarrow \Pi x\!:\!A'.B'}\texttt{na\_pi}$$

$$\frac{}{x \Downarrow x}\texttt{nm\_var} \qquad \frac{}{c \Downarrow c}\texttt{nm\_c} \qquad \frac{M \Downarrow M' \qquad N \Downarrow N'}{M\ N \Downarrow M'\ N'}\texttt{nm\_app} \qquad \frac{M \Downarrow \lambda x\!:\!A.M' \qquad N \Downarrow N'}{M\ N \Downarrow [N'/x]M'}\texttt{nm\_beta}$$

Figure 9: Typing rules for LF_0

Theorem B.2 (Adequacy of Expression Representation.) *There is a compositional bijection $\ulcorner\cdot\urcorner$ between expressions e with free variables among $x_1, \ldots, x_n$ and atomic LF objects $\ulcorner e\urcorner$ such that $x_1 :_a \texttt{exp}, \ldots, x_n :_a \texttt{exp} \vdash_L \ulcorner e\urcorner :_a \texttt{exp}$. The bijection is compositional in the sense that $\ulcorner[e_1/x]e_2\urcorner = [\ulcorner e_1\urcorner/x]\ulcorner e_2\urcorner$.*

Theorem B.3 (Adequacy of Predicate Representation.) *There is a compositional bijection $\ulcorner\cdot\urcorner$ between predicates P with free variables among $x_1, \ldots, x_n$ and canonical LF objects $\ulcorner P\urcorner$ such that $x_1 :_a \texttt{exp}, \ldots, x_n :_a \texttt{exp} \vdash_L \ulcorner P\urcorner :_c \texttt{pred}$. The bijection is compositional in the sense that $\ulcorner[e/x]P\urcorner = [\ulcorner e\urcorner/x]\ulcorner P\urcorner$.*

Theorem B.4 (Adequacy of Derivation Representation.) *There is a bijection $\ulcorner\cdot\urcorner$ between derivations $\mathcal{D} :: \rhd P$ with parameters v_i $(i = 1, \ldots, n)$ and from hypotheses $u_j :: \rhd P_j$ $(j = 1, \ldots, m)$ and canonical LF objects $\ulcorner\mathcal{D}\urcorner$ such that $v_i :_a \texttt{exp}, u_j :_a \texttt{pf}\ \ulcorner P_j\urcorner \vdash_L \ulcorner\mathcal{D}\urcorner :_c \texttt{pf}\ \ulcorner P\urcorner$.*

The adequacy of derivation representation is the central result that justifies the use of LF_0 type checking as a sufficient procedure for checking validity of proofs. This is stated formally in the following corollary.

Corollary B.5 *If P is a closed predicate and M is a canonical LF object such that $\cdot \vdash_L M :_c \texttt{pf}\ \ulcorner P\urcorner$, then there exists a derivation $\mathcal{D} :: \rhd P$, that is P is valid. Furthermore, $M = \ulcorner\mathcal{D}\urcorner$.*