5 *Proof-Carrying Code*

George Necula

In the previous chapter we saw that one can adapt many of the ideas from type systems for high-level languages to assembly language. In this chapter, we describe yet another technique that can be used to type check assembly language programs. This time, however, we are going to depart from traditional type-checking approaches and see how one can adapt ideas from program verification to this problem. In the process of doing so, we are going to obtain a framework that can be adapted more easily to the verification of code properties that go beyond type safety.

5.1 Overview of Proof Carrying Code

Proof-Carrying Code (PCC) (Necula, 1997; Necula and Lee, 1996) is a *general* framework that allows the host to check *quickly* and *easily* that the agent has certain safety properties. The key technical detail that makes PCC powerful is a requirement that the agent producer cooperates with the host by attaching to the agent code an "explanation" of why the code complies with the safety policy. Then all that the host has to do to ensure the safe execution of the agent is to define a framework in which the "explanation" must be conducted, along with a simple yet sufficiently strong mechanism for checking that (a) the explanation is acceptable (i.e., is within the established framework), that (b) the explanation pertains to the safety policy that the host wishes to enforce, and (c) that the explanation matches the actual code of the agent.

There are a number of possible forms of explanations each with its own advantages and disadvantages. Safety explanations must be precise and comprehensive, just like formal proofs. In fact, in this chapter, the explanations are going to be formal proofs encoded in such a way that they can be checked easily and reliably by a simple proof checker.

There are several ways to implement the PCC concept, and all share the common requirement that the untrusted code contains information whose purpose is to simplify the verification task. At one extreme, we have the JVML and CLI verifiers, which rely on typing declarations present in the untrusted code to check the safety of the code. The KVM (Sun) implementation of the JVML verifier does further require that the code contains loop invariants in order to simplify and speed up the verification. Typed Assembly Language (described in Chapter 4) pushes these ideas to the level of assembly language. The most general instance of PCC, called Foundational Proof-Carrying Code (FPCC) (Appel, 2001; Appel and Felty, 2000), reduces to a minimum the size of the verifier and puts almost the entire burden of verification on the agent producer, who now has to produce and send with the agent detailed proofs of safety. In this chapter, we describe an instantiation of PCC that is similar to TAL in that it operates on agents written in assembly language, and is similar to FPCC in that it requires detailed proofs of safety to accompany the agent code. However, the architecture that we describe here uses a verifier that is more complex than that of FPCC, and thus somewhat less trustworthy. However, the advantage of this architecture is that is places a smaller burden on the agent producer than FPPC, and has been shown to scale to verifying even very large programs (Colby et al., 2000). We are going to refer to this architecture as the Touchstone PCC architecture.

A high-level view of the architecture of the Touchstone PCC system is shown in Figure 5-1. The agent contains, in addition to its executable content, checking-support data that allows the PCC infrastructure resident on the receiving host to check the safety of the agent. The PCC infrastructure is composed of two main modules. The verification-condition generator (VCGen) scans the executable content of the agent and checks directly simple syntactic conditions (e.g., that direct jumps are within the code boundary). Each time VCGen encounters an instruction whose execution could violate the safety policy, it asks the Checker module to verify that the dangerous instruction executes safely in the actual current context.

In order to construct a formal proof of a program, we need to reason about them using mathematical concepts. VCGen "compiles" programs to logical formulae in such a way that the aspects of the execution of the program that are relevant to the security policy are brought out.

VCGen can be quite simple because it relies on the Checker to verify complex safety requirements. There are some cases, however, when VCGen might have to understand complex invariants of the agent code in order to follow its control and data flow. For example, VCGen must understand the loop structure of the agent in order to avoid scanning the loop body an unbounded number of times. Also, VCGen must be able to understand even obscure con-

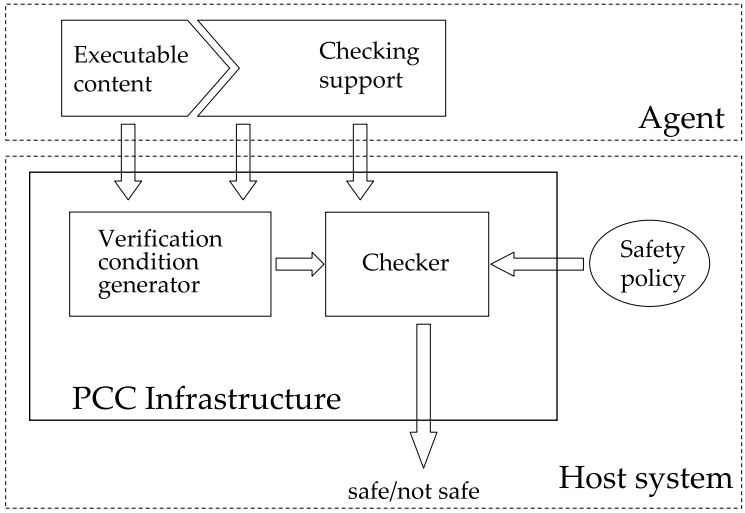

Figure 5-1: The Touchstone PCC architecture

trol flow, as in the presence of indirect jumps or function pointers. In such situations, VCGen relies on *code annotations* that are part of the checking support and are packaged with the agent. This puts most of the burden of handling the complex control-flow issues on the agent producer and keeps the VCGen simple.

The Checker module verifies for VCGen that all dangerous instructions are used in a safe context. The Checker module described in this chapter requires that VCGen formulates the safety preconditions of the dangerous instructions as formulas in a logic. We call these formulas the *verification conditions*. The Checker expects to find in the checking-support data packaged with the agent a formal proof that the safety precondition is met. For the verification to succeed, the Checker must verify the validity of the verification-condition proofs for all dangerous instructions identified by VCGen.

The Touchstone PCC infrastructure described here can be customized to check various safety policies. The "Safety Policy" element in Figure 5-1 is a collection of configuration data that specifies the precise logic that VCGen uses to encode the verification conditions, along with the trusted proof rules that can be used in the safety proofs supplied by the agent producer. For example, the host might require that the untrusted code interacts correctly with the runtime system of a Java Virtual Machine. This can be enforced in

```
type maybepair = Int of int | Pair of int * int
let rec sum(acc : int, x : maybepair list) =
  match x with
  | nil → acc
  | (Int i) :: tail → sum(acc + i, tail)
  | (Pair (l, r)) :: tail → sum (acc + l + r, tail)
```

Figure 5-2: OCaml source for the example agent

our system by a safety policy requiring that the code is well-typed with respect to the typing rules of Java. It is important to separate the safety policy configuration data from the rest of the infrastructure both for conceptual and for engineering reasons. This architecture allows the infrastructure to work with multiple safety policies, without changing most of the implementation.

An Example Agent

In the rest of this chapter, we explore the design and implementation details of the PCC infrastructure. The infrastructure can be configured to check many safety policies. In the example that we use here, we check a simple type-safety policy for an agent written in a generic assembly language. The agent is a function that adds all the elements in a list containing either integers or pairs of integers. If this agent were written in OCaml, its source code might be as shown Figure 5-2.

In order to write the agent in assembly language, we must decide what is the representation strategy for lists and for the `maybepair` type. For the purpose of this example, we represent a list as either the value 0 (for the empty list), or a pointer to a two-word memory area. The first word of the memory area contains a list element, and the second element contains the tail of the list. In order to represent an element of type `maybepair` in an economical way we ensure that any element of kind `Pair`(x, y) is an even-valued pointer to a two-word memory area containing x and y. We represent an element of kind `Int` x as the integer $2x + 1$ (to ensure that it is odd and thus distinguishable from a pair). For example, the representation of the list `[Int 2; Pair (3, 4)]` has the concrete representation shown in Figure 5-3. Notice the tagged representation of the `Int 2` element of the list.

In our examples, we will use the simple subset of a generic assembly language shown below. The expressions e contain arithmetic and logic operations involving constants and registers. This is the same assembly language

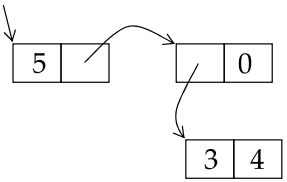

Figure 5-3: Concrete representation of the list `[Int 2; Pair (3, 4)]`

that was used in Chapter 4, except that we relax slightly the syntax of memory addresses, and we replace the general form of indirect jump with a return instruction.

$\mathtt{r}_x := e$	assign the result of e to register $\mathtt{r}_x$
$\mathtt{r}_x := \mathtt{Mem}[e]$	load $\mathtt{r}_x$ from address e
$\mathtt{Mem}[e'] := e$	store the result of e to address e'
`jump` L	jump to a label L
`if` e `jump` L	branch to label L if e is true
`return`	return from the current function

Given our representation strategy and the choice of assembly language instructions, the code for the agent is shown in Figure 5-4. On entry to this code fragment, registers $\mathtt{r}_x$ and $\mathtt{r}_{acc}$ contain the value of the formal arguments `x` and `acc` respectively. The code fragment also uses temporary registers $\mathtt{r}_t$ and $\mathtt{r}_s$. To simplify the handling of the return instruction, we use the convention that the return value is always contained in the register $\mathtt{r}_R$.

The safety policy in this case requires that all memory reads be from pointers that are either non-null lists, in which case we can read either the first or the second field of a list cell, or from pointers to elements of the `Pair` kind. In the case of a memory write, the safety policy constrains the values that can be written to various addresses as follows: in the first word of a list cell we can write either an odd value or an even value that is a pointer to an element of `Pair` kind, and in the second word of a list cell we can write either zero or a pointer to some list cell. There are no constraints on what we can write to the elements of a `Pair`. The restrictions on memory writes ensure that the contents of the accessible memory locations is consistent with the type assigned to their addresses.

The safety policy specifies not only requirements on the agent behavior but can also specify assumptions that the agent can make about the context of the execution. In the case of our agent, the safety policy might specify that the contents of the register $\mathtt{r}_x$ on entry is either zero or a pointer to a list

```
1  sum:                                                  ;r_x : maybepair list
2  Loop:
3          if r_x ≠ 0 jump LCons                              ; Is r_x empty?
4          r_R := r_acc
5          return
6  LCons:  r_t := Mem[r_x]                            ; Load the first data
7          if even(r_t) jump LPair
8          r_t := r_t div 2
9          r_acc := r_acc + r_t
10         jump LTail
11 LPair:  r_s := Mem[r_t]                       ; Get the first pair element
12         r_acc := Mem[r_acc + r_s]
13         r_t := Mem[r_t + 4]                        ; and the second element
14         r_acc := r_acc + r_t
15 LTail:  r_x := Mem[r_x + 4]
16         jump Loop
```

Figure 5-4: Assembly code for the function in Figure 5-2

cell. Also, the safety policy can allow the agent to assume that the value read from the first word of a list cell is either odd or otherwise a pointer to a `Pair` cell. Similarly, the agent can assume that the value it reads from the second word of a cell is either null or else a pointer to a list cell. We formalize this safety policy in the next section.

5.2 Formalizing the Safety Policy

At the core of a safety policy is a list of instructions whose execution may violate safety. The safety policy specifies, for each one, what is the verification condition that guarantees its safe execution. In the variant of PCC described here, the instructions that are handled specially are the memory operations along with the function calls and returns. This choice is hard coded in the verification-condition generator. However, the specific verification condition for each of these instructions is customizable. In such an implementation, we can control very precisely what memory locations can be read, what memory locations can be written and what can be written into them, what functions we call and in what context, and in what context we return from a function. This turns out to be sufficient for a very large class of safety policies. We shall explore in Section 5.7 a safety policy for which this is not sufficient and for which we must change the verification condition generator.

The customizable elements of the safety policy are the following:

- A language of symbolic expressions and formulas that can be used to express verification conditions.
- A set of function preconditions and postconditions for all functions that form the interface between the host and the agent.
- A set of proof rules for verification conditions.

In the rest of this section we describe in turn these elements.

The Syntax of the Logic

For this presentation we use a first-order language of symbolic expressions and formulas, as shown below:

$$\begin{array}{llll} \text{Formulas} & F & ::= & \texttt{true} \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid F_1 \Rightarrow F_2 \mid \forall x.F \mid \exists x.F \\ & & & \mid \texttt{addr}\ E_a \mid E_1 = E_2 \mid E_1 \neq E_2 \mid f\ E_1 \ldots E_n \\ \text{Expressions} & E & ::= & x \mid \texttt{sel}\ E_m\ E_a \mid \texttt{upd}\ E_m\ E_a\ E_v \mid f\ E_1 \ldots E_n \end{array}$$

We consider here only the subset of logical connectives that we need for examples. In practice, a full complement of connectives can be used. The formula ($\texttt{addr}\ E_a$) is produced by VCGen as a verification condition for a memory read or a memory write to address E_a. This formula holds whenever E_a denotes a valid address. Formulas can also be constructed using a set of formula constructors that are specific to each safety policy.

The language of expressions contains variables and a number of constructors that includes integer numerals and arithmetic operators, and can also be extended by safety policies. A notable expression construct is ($\texttt{sel}\ E_m\ E_a$) that is used to denote the contents of memory address denoted by E_a in memory state E_m. The construct ($\texttt{upd}\ E_m\ E_a\ E_v$) denotes a new memory state that is obtained by storing the value E_v at address E_a in memory state E_m. For example, the contents of the address c in a memory that is obtained from memory state m after writing the value 1 at address a followed by writing of value 2, can be written:

$$\texttt{sel}\ (\texttt{upd}\ (\texttt{upd}\ m\ a\ 1)\ b\ 2)\ c$$

A safety policy extends the syntax of the logic by defining new expression and formula constructors. In particular, for our example agent we add constructors for encoding types and a predicate constructor for encoding the typing judgment:

$$\begin{array}{llll} \text{Word types} & W & ::= & \texttt{int} \mid \texttt{ptr}\ \{S\} \mid \texttt{list}\ W \mid \{x \mid F(x)\} \\ \text{Structure types} & S & ::= & W \mid W;S \\ \text{Formulas} & F & ::= & \ldots \mid E : W \mid \texttt{listinv}\ E_m \end{array}$$

We distinguish among types the *word types*, whose values fit in a machine register or memory word. Pointers can point to an area of memory containing a sequence of words. The word type $\{x \mid F(x)\}$ contains all those values for which the formula F is true (this type is sometimes called a comprehension or set type). The typing formula constructor ":" is written in infix notation. We also add the `listinv` formula constructor that will be used to state that the contents of the memory satisfies the representation invariant for lists of pairs. The precise definition of the typing and the `listinv` formulas will be given on page 200, with respect to a predetermined mapping of values and memory addresses to types. Informally, we say that `listinv` M holds when each memory address that is assigned a pointer type contains values that are assigned appropriate types.

Using these constructors we can write the low-level version of the ML typing judgment x : `maybepair list` as

$$x : \texttt{list}\ \{y \mid (\texttt{even}\ y) \Rightarrow y : \texttt{ptr}\ \{\texttt{int}; \texttt{int}\}\}$$

In the rest of this section we use the abbreviation `mp_list` for the type `maybepair list`. Notice that we have built-in the recursive type of lists in our logic, in order to avoid the need for recursion at the level of the logic, but we choose to express the union and tuple types explicitly.

5.2.1 EXERCISE [RECOMMENDED, ⋆]: The singleton type is a type populated by a single value. Write a formula in the above logic corresponding to the assertion that x has type singleton for the value v. Show also how you can write using our language of types the singleton type for the value v. □

5.2.2 EXERCISE [RECOMMENDED, ⋆]: Consider an alternative representation for the `maybepair` type. A value of this type is a pointer to a tagged memory area containing a tag word followed either by another word encoding an `Int` if the tag value if 0, or by two words encoding a `Pair` if the tag value is 1. Write the formula corresponding to the assertion that x has this representation. □

The Preconditions and Postconditions

A PCC safety policy contains preconditions and postconditions for the functions that form the interface between the agent and the host. These are either functions defined by the agent and invoked by the host or library functions exported by the host for use by the agent. These preconditions and postconditions are expressed as logic formulas that use a number of formula and expression constructors specific to the safety policy.

Function preconditions and postconditions at the level of the assembly language are expressed in terms of argument and return registers, and thus

specify also the calling convention. For verification purposes, we model the memory as a pseudo-register $\mathtt{r}_M$.

The function precondition and postcondition for our agent are:

$$\begin{array}{lcl} \mathtt{Pre}_{\mathtt{sum}} & = & \mathtt{r}_x : \mathtt{mp\_list} \wedge \mathtt{listinv}\, \mathtt{r}_M \\ \mathtt{Post}_{\mathtt{sum}} & = & \mathtt{listinv}\, \mathtt{r}_M \end{array}$$

The safety policy requires that the memory state be well-typed after the agent returns and allows the agent to assume that the memory state is well-typed when the host invokes it. Notice that we do not specify constraints on the integer arguments and results. This reflects our decision that any value whatsoever can be used as an integer.

Technically, the preconditions and postconditions are not well-formed formulas in our logic because they use register names as expressions. However, we can obtain valid formulas from them once we have a substitution of register names with expressions. We shall see later how this works out in detail.

5.2.3 EXERCISE [RECOMMENDED, ⋆]: Write the precondition and postcondition of a function of OCaml type `(int * int) * int list → int list`. The first argument is represented as a pointer to a sequence of two integers; the second is a list. Consider that the return value is placed in register $\mathtt{r}_R$. □

5.2.4 EXERCISE [RECOMMENDED, ⋆]: Consider a function that takes in register $\mathtt{r}_1$ a pointer to a sequence of integer lists. The length of the sequence is passed in register $\mathtt{r}_2$. The function does not return anything. Write the precondition and postcondition for this function. □

The Proof Rules

The last part of the safety policy is a set of proof rules that can be used to reason about formulas in our logic in general and about verification conditions in particular. In Figure 5-5 we show, in natural deduction form, a selection of the derivation rules for the first-order logical connectives. We show the conjunction introduction (ANDI) and the two conjunction eliminations (ANDEL, ANDER), and the similar rules for implication. We also have two rules (MEM0 and MEM1) that allow us to reason about the `sel` and `upd` constructors for memory expressions. These two rules should not be necessary for most type-based safety policies because in those cases all we care to know about the contents of a memory location is its type, not its value.

Note that in this set of base proof rules we do not yet specify when we can prove that `addr` holds. This is the prerogative of the safety-policy specific rules that we describe next.

$$\frac{F_1 \qquad F_2}{F_1 \wedge F_2}\ (\textsc{andi}) \qquad \frac{F_1 \wedge F_2}{F_1}\ (\textsc{andel}) \qquad \frac{F_1 \wedge F_2}{F_2}\ (\textsc{ander})$$

$$\frac{\begin{array}{c}F_1\\ \vdots\\ F_2\end{array}}{F_1 \Rightarrow F_2}\ (\textsc{impi}) \qquad \frac{F_1 \Rightarrow F_2 \qquad F_1}{F_2}\ (\textsc{impe})$$

$$\frac{A = A'}{\texttt{sel}\ (\texttt{upd}\ M\ A\ V)\ A' = V}\ (\textsc{mem0}) \qquad \frac{A \neq A'}{\texttt{sel}\ (\texttt{upd}\ M\ A\ V)\ A' = \texttt{sel}\ M\ A'}\ (\textsc{mem1})$$

Figure 5-5: Built-in proof rules

Each safety policy can extend the built-in proof rules with new rules. In fact, this is necessary if the safety policy uses formula constructors beyond the built-in ones. The rules specific to our safety policy are shown in Figure 5-6. We have rules for reasoning about the type constructors: lists (NIL, CONS), set types (SET) and pointers to sequences (THIS and NEXT). These are similar to corresponding rules from type systems with recursive types and tuples. Next come two rules for reasoning about the typing properties of reading and writing from pointers. The rule SEL says that the location referenced by a pointer to a word type in a well-typed memory state has the given word type. The rule UPD is used to prove that a well-typed write preserves well-typedness of memory. These rules are similar to corresponding rules for reference types.

Notice that these proof rules are exposing more concrete implementation details than the corresponding source-level rules. For example, the CONS rule specifies that a list cell is represented as a pointer to a pair of words, of which the first one stores the data and the second the tail of the list.

Finally, the PTRADDR rule relates the safety-policy specific formula constructors with the built-in `addr` memory safety formula constructor. This rule says that addresses that can be proved to have pointer type in our type system are valid addresses. And since this is the only rule whose conclusion uses the `addr` constructor, the safety policy is essentially restricting memory accesses to such addresses.

5.2.5 EXERCISE [RECOMMENDED, ⋆⋆]: Add a new `array` type constructor to our safety policy and write the proof rules for its usage. An array is represented

$$\frac{}{0 : \texttt{list}\ W}\ (\text{NIL}) \qquad \frac{E : \texttt{list}\ W \qquad E \neq 0}{E : \texttt{ptr}\ \{W; \texttt{list}\ W\}}\ (\text{CONS})$$

$$\frac{E : \{y \mid F(y)\}}{F(E)}\ (\text{SET}) \qquad \frac{E : \texttt{ptr}\ \{W; S\}}{E : \texttt{ptr}\ \{W\}}\ (\text{THIS})$$

$$\frac{E : \texttt{ptr}\ \{W; S\}}{E + 4 : \texttt{ptr}\ \{S\}}\ (\text{NEXT}) \qquad \frac{A : \texttt{ptr}\ \{W\} \qquad \texttt{listinv}\ M}{(\texttt{sel}\ M\ A) : W}\ (\text{SEL})$$

$$\frac{\texttt{listinv}\ M \qquad A : \texttt{ptr}\ \{W\} \qquad V : W}{\texttt{listinv}\ (\texttt{upd}\ M\ A\ V)}\ (\text{UPD}) \qquad \frac{A : \texttt{ptr}\ \{W\}}{\texttt{addr}\ A}\ (\text{PTRADDR})$$

Figure 5-6: Proof rules specific to the example safety policy

as a pointer to a memory area that contains the number of elements in the array in the first word and then the array elements in order. Consider first the case where each element is a word type (as in OCaml), and then the case when each element can be a structure (as in C). □

We have shown here just a few of the rules for a simple safety policy. A safety policy for a full language can easily have hundreds of proof rules. For example, the Touchstone implementation of PCC for the Java type system (Colby et al., 2000) has about 150 proof rules.

5.3 Verification-Condition Generation

So far, we have shown how to set up the safety policy; now we need to describe a method for enforcing it. An analogous situation in the realm of high-level type systems is when we have setup a type system, with a language of types and a set of typing rules, and we need to design a type checker for it. A type checker must scan the code and must know what typing rule to apply at each point in the code. In fact, some type checkers work by explicitly collecting typing constraints that are solved in a separate module. Our PCC infrastructure accomplishes a similar task, and separates the scanning of the code from the decision of what safety policy proof rules to apply. The scanning is done by the verification-condition generator, which also identifies *what* must be checked for each instruction. *How* the check is performed is decided by the Checker module, with considerable help from the proof that accompanies the code. In a regular type checker, there is no pressing need to separate code scanning from the construction of the typing derivation,

since the scanning process is often simple and the structure of the typing derivation closely follows that of the code. This is not true for low-level type checking. In fact, programs written in assembly language may have very little structure.

To illustrate some of the difficulties of type checking low-level code, consider the following fragment of code written in ML, where x is a variable of type T `list` and the variable t occurs in the expression e also with type T `list`:

```
match x with
  _ :: t → e
```

A type checker for ML parses this code, constructs an abstract syntax tree (AST) and then it verifies its well-typedness in a relatively simple manner by traversing the AST. This is possible because the `match` expression packages in one construction all the elements that are needed for type checking: the expression to be matched, the patterns with the variables they define, and the bodies of the cases.

Consider now one particular compilation of this code fragment:

$\mathtt{r}_t := \mathtt{r}_x$
$\mathtt{r}_t := \mathtt{r}_t + 4$
`if` $\mathtt{r}_x = 0$ `jump LNil`
$\mathtt{r}_t := \mathtt{Mem}[\mathtt{r}_t]$
...

We assume that the variable x is allocated to register $\mathtt{r}_x$ and that the expression e is compiled with the assumption that, on entry, the variable t is allocated to the register $\mathtt{r}_t$. We observe that the code for compiling the `match` construct is spread over several non-contiguous instructions mixed with the instructions that implement the cases themselves. This is due to the intrinsically sequential nature of assembly language. It would be hard to implement a type checker for assembly language that identifies the code for the `match` by recognizing patterns, as a source-level type checker does. Also, such a type checker would be sensitive to code generation and optimization choices.

Another difficulty is that some high-level operations are split into several small operations. For example the extraction of the tail of the list is separated into the computation of an address in register $\mathtt{r}_t$ and a memory load. We cannot check one of the two instructions in isolation of the other because they both can be used in other contexts as well. Furthermore, it is not sufficient to type check the addition $\mathtt{r}_t + 4$ as we would do in a high-level language (i.e., verify that both operands have compatible arithmetic types). Instead we need to remember that we added the constant 4 to the contents of register $\mathtt{r}_x$, so that when we reach the load instruction, we can determine that we

are loading the second word from a list cell. Additionally, our type-checking algorithm has to be flow sensitive and path sensitive because the outcomes of conditional expressions sometimes determine the type of values. In our example, if the conditional falls through then we know that r_x points to a list cell and therefore that r_t points to the second element in a list cell. If, however, the conditional jumps to `LNil`, then we cannot even assign a type to r_t after the addition.

Yet another complication with assembly language is that, unlike in high-level languages, we cannot count on a variable having a single type throughout its scope. In assembly language the registers play the role of variables and since there is a finite number of them, compilers reuse them aggressively to hold different data at different program points. In our example, the register r_t is used before the load to hold both a pointer to a memory location containing a list and after the load instruction to hold a list. We must thus keep different types for registers for different program points. Chapter 4 discusses these problems extensively.

There are a number of approaches for overcoming these difficulties. All of them do maintain different types for registers at different program points but differ on how they handle the dependency on conditionals and the splitting of high-level operations into several instructions. At one extreme is the Java Bytecode Verifier (Lindholm and Yellin, 1997), which typechecks programs written in the Java Virtual Machine Language (JVML). The JVML is relatively high-level and maintains complicated operations bundled in high-level constructs. For instance, in JVML you cannot separate the address computation from the memory access itself. In the context of our example, this means that the addition and the load instruction would be expressed as one bytecode instruction. The JVML is designed such that the outcome of conditionals does not matter for type checking. For example, array-bounds checks and pointer null-checks are bundled with the memory access in high-level bytecode instructions. This approach simplifies the type-checking problem but has the disadvantage that the agent producer cannot really do much optimization. Also this approach puts more burden on the code receiver for compiling and optimizing the code.

Another approach is Typed Assembly Language (TAL), described in Chapter 4, where a more sophisticated type system is used to keep track of the intermediate result of unbundled instructions. But even in TAL some low-level instructions are treated as bundles for the purpose of verification. Examples are memory allocation and array accesses.

Here we are going to describe a type checking method that can overcome the difficulties described above. The method is based on *symbolic evaluation*, and it was originally used in the context of program verification. The method

is powerful enough to verify full correctness of a program, not just its well-typedness, which will come in handy when we consider safety policies beyond type safety.

Symbolic Evaluation

In order to introduce symbolic evaluation, consider the code fragment from above but without the conditional.

$$\mathsf{r}_t := \mathsf{r}_x$$
$$\mathsf{r}_t := \mathsf{r}_t + 4$$
$$\mathsf{r}_t := \mathtt{Mem}[\mathsf{r}_t]$$

This fragment exhibits the problems due to reuse of registers with different types and the splitting of high-level operations into low-level instructions. We have already observed that it is more important to remember the effect of the addition instruction than it is to type check it immediately as we see it. In fact, we are going to postpone all checking as much as possible and are going to focus on "remembering" the effect of instructions instead. Observe that if we allow arbitrary complex operands in our instructions, we can rewrite the above code sequence as follows:

$$\mathsf{r}_t := \mathtt{Mem}[\mathsf{r}_x + 4]$$

In this variant, the address computation is bundled with the memory access, and we can actually perform the usual pattern matching to recognize what typing rule to apply. Symbolic evaluation is a technique that has the effect of collecting the results of intermediate computations to create the final result as a complex expression whose meaning is equivalent to the entire computation. A symbolic evaluator is an interpreter that maintains for each register a symbolic expression. We will use the symbol σ to range over *symbolic states*, which are mappings from register names to symbolic expressions. The symbolic state is initialized with a distinct fresh variable for each register, to model the lack of information about the initial values of the registers. For our example the initial symbolic state is:

$$\sigma_0 = \{\mathsf{r}_t = t, \mathsf{r}_x = x, \mathsf{r}_M = m\}$$

where t and x are distinct fresh variables. Technically, this symbolic state says that at the given program point the following invariant holds:

$$\exists t.\exists x.\exists m.\mathsf{r}_t = t \wedge \mathsf{r}_x = x \wedge \mathsf{r}_M = m$$

The symbolic evaluator proceeds forward to interpret the instructions and modifies the symbolic state as specified by the instruction. We show below the sequence of symbolic states during symbolic evaluation.

$$\sigma = \{\mathtt{r}_t = t, \mathtt{r}_x = x, \mathtt{r}_M = m\}$$

$\mathtt{r}_t := \mathtt{r}_x$

$$\sigma = \{\mathtt{r}_t = x, \mathtt{r}_x = x, \mathtt{r}_M = m\}$$

$\mathtt{r}_t := \mathtt{r}_t + 4$

$$\sigma = \{\mathtt{r}_t = x + 4, \mathtt{r}_x = x, \mathtt{r}_M = m\}$$

$\mathtt{r}_t := \mathtt{Mem}[\mathtt{r}_t]$

$$\sigma = \{\mathtt{r}_t = (\mathtt{sel}\ m\ (x + 4)), \mathtt{r}_x = x, \mathtt{r}_M = m\}$$

When the instruction "$\mathtt{r}_t := \mathtt{r}_x$" is processed, the symbolic evaluator looks up the value of $\mathtt{r}_x$ in the current symbolic state and then sets $\mathtt{r}_t$ to that value. Notice how at the time the load instruction is processed, the symbolic evaluator can figure out that the address being accessed is $x + 4$.

In order to handle memory reads and writes we use a pseudo-register $\mathtt{r}_M$ and the `sel` and `upd` constructors introduced in Section 5.2. For memory loads and writes, the symbolic evaluator also emits the required verification conditions using the `addr` constructor. For example, the verification condition for the load instruction would be $(\mathtt{addr}\ (x + 4))$.

Another element of interest is the handling of conditionals. In order to allow for path sensitive checking the symbolic evaluator maintains, in addition to the symbolic state, a list of assumptions about the state. These assumptions are simply formulas involving the same existentially quantified variables that the symbolic state uses. As the symbolic evaluator follows the branches of a conditional, it extends the list of assumptions with formulas that capture the outcome of the conditional expression.

If we now add back the conditional instruction in our example, the symbolic state and the set of assumptions (initially A) at each point are shown below:

$$\sigma = \{\mathtt{r}_t = t, \mathtt{r}_x = x, \mathtt{r}_M = m\}, A$$

$\mathtt{r}_t := \mathtt{r}_x$

$$\sigma = \{\mathtt{r}_t = x, \mathtt{r}_x = x, \mathtt{r}_M = m\}, A$$

$\mathtt{r}_t := \mathtt{r}_t + 4$

$$\sigma = \{\mathtt{r}_t = x + 4, \mathtt{r}_x = x, \mathtt{r}_M = m\}, A$$

$\mathtt{if}\ \mathtt{r}_x = 0\ \mathtt{jump}\ \mathtt{LNil}$

$$\sigma = \{\mathtt{r}_t = x + 4, \mathtt{r}_x = x, \mathtt{r}_M = m\}, A \wedge x \neq 0$$

$\mathtt{r}_t := \mathtt{Mem}[\mathtt{r}_t]$

$$\sigma = \{\mathtt{r}_t = \mathtt{sel}\ m\ (x + 4), \mathtt{r}_x = x, \mathtt{r}_M = m\}, A \wedge x \neq 0$$

...

LNil: $$\sigma = \{\mathtt{r}_t = x + 4, \mathtt{r}_x = x, \mathtt{r}_M = m\}, A \wedge x = 0$$

The symbolic state immediately before the load instruction essentially states that the following invariant holds at that point:

$$\exists t.\exists x.\exists m.\mathtt{r}_t = x \wedge \mathtt{r}_x = x + 4 \wedge \mathtt{r}_M = m \wedge A \wedge x \neq 0$$

This means that the Checker module would have to check the following verification condition for the load instruction:

$$\forall t.\forall x.\forall m.(\mathsf{r}_t = x \wedge \mathsf{r}_x = x + 4 \wedge \mathsf{r}_M = m \wedge A \wedge x \neq 0) \Rightarrow \mathtt{addr}\ (x + 4)$$

Symbolic evaluation has many applications in program analysis. In the following two exercises you can explore how one can use symbolic evaluation to verify easily the correctness of some code transformations.

5.3.1 EXERCISE [RECOMMENDED, ⋆]: Consider the following two code fragments. The one on the right has been obtained from the one on the left by performing a few simple local optimizations. First, we did register allocations, by renaming register r_a, r_b, r_c, and r_d to r_1, r_2, r_3 and r_4 respectively. Then we removed the dead instruction from line 1. We performed copy propagation followed by common subexpression elimination in line 5. Finally, we performed instruction scheduling by moving the instruction from line 3 to be the last in the block.

1	$\mathsf{r}_a := 2$	$\mathsf{r}_1 := \mathsf{r}_2 + 1$
2	$\mathsf{r}_a := \mathsf{r}_b + 1$	$\mathsf{r}_4 := \mathsf{r}_1$
3	$\mathsf{r}_c := \mathsf{r}_a + 2$	$\mathsf{r}_3 := \mathsf{r}_1 + 2$
4	$\mathsf{r}_d := 1$	
5	$\mathsf{r}_d := \mathsf{r}_b + \mathsf{r}_d$	

Show that the result of symbolic evaluation for the registers live at the end of the two basic blocks is identical if you start with symbolic states $\{\mathsf{r}_b = b\}$ and $\{\mathsf{r}_2 = b\}$ respectively. This suggests that symbolic evaluation is insensitive to some common optimizations. □

5.3.2 EXERCISE [RECOMMENDED, ⋆]: Now consider the first code fragment shown in Exercise 5.3.1 and add the instruction "$\mathsf{r}_a := 3$" immediately before line 5. In this case it is not correct to perform common-subexpression elimination. Show now that the result of symbolic evaluation is different for the modified code fragment and the transformed code from Exercise 5.3.1. This suggests that symbolic evaluation can be use to verify the result of compiler optimizations. This technique is in fact so powerful that it can be used to verify most optimizations that the GNU C compiler performs (Necula, 2000). □

Before we can give a complete formal definition of the VCGen, we must consider what happens in the cases when the symbolic evaluator should not follow directly the control-flow of the program. Two such cases are for loops (when following the control-flow would make VCGen loop forever) and for functions (when it is desirable to scan the body of a function only once). In order to handle those cases, VCGen needs some assistance from the agent producer, in the form of code annotations.

The Role of Program Annotations

The VCGen module attempts to execute the untrusted program symbolically in order to signal all potentially unsafe operations. To make this execution possible in finite time and without the need for conservative approximations on the part of VCGen, we require that the program be annotated with invariant predicates. At least one such invariant must be specified for each cycle in the program's control-flow graph. An easy but conservative way to enforce such a constraint is to require an invariant annotation for every backward branch target.

The agent code shown in Figure 5-4 has one loop whose body starts at the label `Loop`. There must be one invariant annotation somewhere in that loop. Let us say that the agent producer places the following invariant at label `Loop`:

$$\texttt{Loop: INV} = r_x : \texttt{mp\_list} \wedge \texttt{listinv}\ \texttt{r}_M$$

The invariant annotation says that whenever the execution reaches the label `Loop` the contents of register $\texttt{r}_x$ is a list. It also says that the contents of the memory satisfies the representation invariants of lists. Just like the preconditions and postconditions, the invariants can refer to register names.

A valid question at this point is who discovers this annotation and how. There are several possibilities. First, annotations can be inserted by hand by the programmer. This is the only alternative when the agent code is programmed directly in assembly language or when the programmer wants to hand-optimize the output of a compiler. It is true that this method does not scale well, but it is nevertheless a feature of PCC that the code receiver does not care whether the code is produced by a trusted compiler, and will gladly accept code that was written or optimized by hand.

Another possibility is that the annotations can be produced automatically by a certifying compiler. For our simple type safety policy the only annotations that are necessary consist of type declarations for the live registers at that point. See Necula (1998) for more details.

Finally, note that the invariant annotations are required but cannot be trusted to be correct as they originate from the same possibly untrusted source as the code itself. Nevertheless, VCGen can still use them safely, as described in the next section.

The Verification-Condition Generator

Now we have all the elements necessary to describe the verification-condition generator for the case of one function whose precondition and postcondition are specified by the safety policy. We will assume that each invariant annota-

tion occupies one instruction slot, even though in practice they are stored in a separate region of the agent. Let *Inv* be the partial mapping from program counters to invariant predicates. If $i \in Dom(Inv)$, then there is an invariant Inv_i at program counter i. Next, for a more uniform treatment of functions and loops, we will consider that the first instruction in each agent function is an invariant annotation with the precondition predicate. In our example, this means that $Inv_1 = \mathtt{r}_x : \mathtt{mp\_list} \wedge \mathtt{listinv}\ \mathtt{r}_M$. This, along with the loop invariant (with the same predicate) at index 2 are all the loop invariants in the example. Thus, $Dom(Inv) = \{1, 2\}$, and the first few lines of our agent example are modified as follows:

```
1 sum:   INV r_x : mp_list ∧ listinv r_M
2 Loop:  INV r_x : mp_list ∧ listinv r_M
3        if r_x ≠ 0 jump LCons                    ; list is empty
```

Given a symbolic state σ and an expression e that contains references to register names, we write $(\sigma\ e)$ to denote the result of substituting the register names in e with the expressions given by σ. We extend this notation to formulas F that refer to register names (e.g., function preconditions or postconditions, or loop invariants). We also write $\sigma[r \leftarrow e]$ to denote a symbolic state that is the same as σ but with register r mapped to e.

We write Π_i for the instruction (or annotation) at the program counter i.

The core of the verification-condition generator is a symbolic evaluation function *SE* that given a value i for the program counter and a symbolic state σ, produces a formula that captures all of the verification conditions from the given program counter until the next return instruction or invariant. The definition of the *SE* function is shown below:

$$SE(i,\sigma) = \begin{cases} SE(i+1, \sigma[r \leftarrow \sigma\ e]) & \text{if } \Pi_i = r := e \\ (\sigma\ e) \Rightarrow SE(L,\sigma) \wedge & \text{if } \Pi_i = \mathtt{if}\ e\ \mathtt{jump}\ \mathtt{L} \\ \quad (\mathtt{not}\ (\sigma\ e)) \Rightarrow SE(i+1,\sigma) & \\ \mathtt{addr}\ (\sigma\ a) \wedge & \text{if } \Pi_i = r := \mathtt{Mem}[a] \\ \quad SE(i+1, \sigma[r \leftarrow (\sigma\ (\mathtt{sel}\ \mathtt{r}_M\ a))]) & \\ \mathtt{addr}\ (\sigma\ a) \wedge & \text{if } \Pi_i = \mathtt{Mem}[a] := e \\ \quad SE(i+1, \sigma[r_M \leftarrow (\sigma\ (\mathtt{upd}\ r_M\ a\ e))] & \\ \sigma\ Post & \text{if } \Pi_i = \mathtt{return} \\ \sigma\ I & \text{if } \Pi_i = \mathtt{INV}\ I \end{cases}$$

Symbolic evaluation is defined by case analysis of the instruction contained at a given program counter. Symbolic evaluation is undefined for values of the program counter that do not contain a valid instruction. In the case of a `set` instruction, the symbolic evaluator substitutes the current symbolic state into the right-hand side of the instruction and then uses the result as the new

value of the destination register. Then the symbolic evaluator continues with the following instruction. For a conditional, the symbolic evaluator adds the proper assumption about the outcome of the conditional expression. Memory operations are handled like assignments but with the generation of additional verification conditions.

When either the return instruction or an invariant is encountered, the symbolic evaluator stops with a predicate obtained by substituting the current symbolic state into the postcondition or the invariant formula. The symbolic evaluator also ensures (using a simple check not shown here) that each loop in the code has at least one invariant annotation. This ensures the termination of the *SE* function.

What remains to be shown is how the verification-condition generator uses the *SE* function. For each invariant in the code, VCGen starts a symbolic evaluation with a symbolic state initialized with distinct variables. Assuming that the set of registers is $\{\mathtt{r}_1, \ldots, \mathtt{r}_n\}$, we define the *global verification condition* *VC* as follows:

$$\begin{array}{lcl} VC & = & \bigwedge_{i \in Dom(Inv)} \forall x_1 \ldots x_n. \quad \sigma_0\ Inv_i \Rightarrow SE(i+1, \sigma_0) \\ & & \text{where } \sigma_0 = \{\mathtt{r}_1 = x_1, \ldots, \mathtt{r}_n = x_n\} \end{array}$$

Essentially the VCGen evaluates symbolically every path in the program that connects two invariants or an invariant and a return instruction. In Figure 5-7 we show the operation of the VCGen algorithm on the agent code from Figure 5-4 (after we have added the invariant annotations for the precondition and the loop, as explained at the beginning of this section). We show on the left the program points and a brief description of each action. Some actions result in extending the stack of assumptions that the Checker is allowed to make. These assumptions are shown underlined and with an indentation level that encodes the position in the stack of each assumption. Thus an assumption at a given indentation level implicitly discards all previously occurring assumptions at the same or larger indentation level. Finally, we show right-justified and boxed the checking goals submitted to the Checker.

There are two invariants (in lines 1 and 2) and for each one we generate fresh new variables for registers, we assume that the invariant holds, and then we start the symbolic evaluator. For the first invariant, the symbolic evaluator when starting in line 2 encounters an invariant and terminates.

Every boxed formula shown flushed right in Figure 5-7 is a verification condition that VCGen produces and the Checker module has to verify for some arbitrary values of the initial variables.

Notice that the invariant formulas are used both as assumptions and as verification conditions. There is a strong similarity between the role of invariants and that of predicates in a proof by induction. In the latter case the

1: Generate fresh values $\mathtt{r}_M = m_0$, $\mathtt{r}_R = r_0$, $\mathtt{r}_x = x_0$, $\mathtt{r}_{acc} = acc_0$,
$\mathtt{r}_t = t_0$ and $\mathtt{r}_s = s_0$

1: Assume Invariant $\underline{x_0 : \mathtt{mp\_list}}$
$\underline{\mathtt{listinv}\ m_0}$

2: Invariant $\boxed{x_0 : \mathtt{mp\_list}}$
$\boxed{\mathtt{listinv}\ m_0}$

2: Generate fresh values $\mathtt{r}_M = m_1$, $\mathtt{r}_R = r_1$, $\mathtt{r}_x = x_1$, $\mathtt{r}_{acc} = acc_1$,
$\mathtt{r}_t = t_1$ and $\mathtt{r}_s = s_1$

2: Assume Invariant $\underline{x_1 : \mathtt{mp\_list}}$
$\underline{\mathtt{listinv}\ m_1}$

3: Branch 3 taken $\underline{x_1 \neq 0}$

6: Check load $\boxed{\mathtt{addr}\ x_1}$

7: Branch 7 taken $\underline{\mathtt{even}\ (\mathtt{sel}\ m_1\ x_1)}$

11: Check load $\boxed{\mathtt{addr}\ (\mathtt{sel}\ m_1\ x_1)}$

13: Check load $\boxed{\mathtt{addr}\ ((\mathtt{sel}\ m_1\ x_1) + 4)}$

15: Check load $\boxed{\mathtt{addr}\ (x_1 + 4)}$

16: Goto Loop

2: Invariant $\boxed{(\mathtt{sel}\ m_1\ (x_1 + 4)) : \mathtt{mp\_list}}$
$\boxed{\mathtt{listinv}\ m_1}$

7: Branch 7 not taken $\underline{\mathtt{odd}\ (\mathtt{sel}\ m_1\ x_1)}$

10: Goto LTail

15: Check load $\boxed{\mathtt{addr}\ (x_1 + 4)}$

16: Goto Loop

2: Invariant $\boxed{(\mathtt{sel}\ m_1\ (x_1 + 4)) : \mathtt{mp\_list}}$
$\boxed{\mathtt{listinv}\ m_1}$

3: Branch 3 not taken $\underline{x_1 = 0}$

5: Return $\boxed{\mathtt{listinv}\ m_1}$

Figure 5-7: The sequence of actions taken by VCGen

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x_1 : \texttt{mp\_list} \quad x_1 \neq 0}}{x_1 : \texttt{ptr}\,\{\texttt{maybepair};\texttt{mp\_list}\}}\,\text{CONS}}{x_1 : \texttt{ptr}\,\{\texttt{maybepair}\}}\,\text{THIS}}{(\texttt{sel}\ m_1\ x_1) : \texttt{maybepair}}\,\text{SEL}}{\texttt{even}\,(\texttt{sel}\ m_1\ x_1) \Rightarrow (\texttt{sel}\ m_1\ x_1) : \texttt{ptr}\,\{\texttt{int};\texttt{int}\}}\,\text{SET} \qquad \texttt{even}\,(\texttt{sel}\ m_1\ x_1)}{(\texttt{sel}\ m_1\ x_1) : \texttt{ptr}\,\{\texttt{int};\texttt{int}\}}\,\text{IMPE}}{(\texttt{sel}\ m_1\ x_1) : \texttt{ptr}\,\{\texttt{int}\}}\,\text{THIS}}{\texttt{addr}\,(\texttt{sel}\ m_1\ x_1)}\,\text{PTRADDR}$$

$\texttt{listinv}\ m_1$

Figure 5-8: Proof of a verification condition

predicate is assumed to hold and with this assumption we must prove that it holds for a larger value in a well founded order. This effectively ensures that the invariant formulas are preserved through an arbitrary execution from one invariant point to another.

Let us consider now how one proves the verification conditions. The first interesting one is the `addr` from line 6. Let

$$\texttt{maybepair} \stackrel{def}{=} \{y \mid \texttt{even}(y) \Rightarrow y : \texttt{ptr}\,\{\texttt{int};\ \texttt{int}\}\}$$

To construct its proof, we first derive $x_1 : \texttt{ptr}\,\{\texttt{maybepair};\texttt{mp\_list}\}$ using the rule CONS with the assumptions $x_1 : \texttt{mp\_list}$ and $x_1 \neq 0$. Then we can derive $\texttt{addr}\ x_1$ using the rule PTRADDR.

A more interesting case is that of proving $\texttt{addr}\ (\texttt{sel}\ m_1\ x_1)$ from the assumptions $x_1 : \texttt{mp\_list}$, $\texttt{listinv}\ m_1$, $x_1 \neq 0$, and $\texttt{even}\,(\texttt{sel}\ m_1\ x_1)$. This proof is shown in Figure 5-8.

5.3.3 EXERCISE [⋆⋆, ↛]: Construct the the proof of the verification condition corresponding to the loop invariant from line 2. You must prove that $\texttt{sel}\ m_1\ (x_1 + 4) : \texttt{mp\_list}$ from the assumptions $x_1 : \texttt{mp\_list}$, $\texttt{listinv}\ m_1$, $x_1 \neq 0$, and $\texttt{even}\,(\texttt{sel}\ m_1\ x_1)$. □

5.3.4 EXERCISE [⋆⋆⋆]: Notice that we have to prove that $\texttt{sel}\ m_1\ x_1 : \texttt{ptr}\,\{\texttt{int}\}$ several times as a step in proving those verification conditions from Figure 5-7 that refer to $(\texttt{sel}\ m_1\ x_1)$. Show how you can add an invariant to the program to achieve the effect of proving this fact only once. □

We have been arguing that symbolic evaluation is just an alternative method for type checking, with additional benefits for checking more complex safety policies. Since there is a simple type checker at the source level for our type system, it seems reasonable to wonder whether we could hope to build automatically the proofs of these verification conditions. This is indeed possible for such type-based safety policies. Consider for instance how the proof shown in Figure 5-8 could be constructed through a goal-directed manner. The goal is an `addr` formula, and we observe that only the PTRADDR among our rules (shown in Figure 5-6) has a conclusion that matches the goal. The subgoal now is $(\texttt{sel}\ m_1\ x_1) : \texttt{ptr}\ \{\texttt{int}\}$. In order to prove that the result of reading from a memory location has a certain type, we must prove that the memory is well-typed and the address has some pointer type. When we try to prove that x_1 has a pointer type, we find among the assumptions that $x_1 : \texttt{mp\_list}$. The remaining steps can be easily constructed by a theorem prover that knows the details of the type system. This general strategy was used successfully to construct a simple theorem prover that can build automatically and efficiently proofs of verification conditions for the entire Java type safety policy (Colby et al., 2000).

5.3.5 EXERCISE [⋆⋆⋆]: Extend the verification-condition generator approach shown here to handle a function call instruction `call L`, where L is a label that is considered the start of a function. For each such function there is a precondition and a postcondition. Make the simplifying assumption that the `call` instruction saves the return address and a set of callee-save registers on a special stack that cannot be manipulated directly by the program. A `ret` instruction always returns to the last return address saved on the stack and also restores the callee-save registers. □

5.3.6 EXERCISE [⋆⋆]: It is sometimes useful to use more kinds of annotations in addition to the loop invariants. For example, the agent producer might know that a certain point in the code is not reachable, as is the case for the label `L1` in the code fragment shown below:

```
      call exit
L1:   UNREACHABLE
      ...
```

In such a case it is useful to add an annotation `UNREACHABLE` to signal to the symbolic evaluator that it can stop the evaluation at that point. Show how you can change the symbolic evaluator to handle these annotation without allowing the agent producer to "lie" about reachability of code. □

5.3.7 EXERCISE [⋆⋆]: Extend the symbolic evaluator to handle the indirect jump instruction `jump at` e, where e must evaluate to a valid program counter.

Indirect jumps are often used to implement efficiently `switch` statements, in which case the destination address is one of a statically-known set of labels. Assume that immediately after the indirect jump instruction there is an annotation of the form `JUMPDEST(L1, L2)` to declare that the destination address is one of `L1` or `L2`. □

5.3.8 EXERCISE [★★★★, ↛]: Extend the symbolic evaluator to handle stack frames. The basic idea is that there is a dedicated stack pointer register r_{SP} that always points at the last used stack word. This register can only be incremented or decremented by a constant amount. You can ignore stack overflow issues. The stack frame for a function has a fixed size that is declared with an annotation. The only accesses to it are through the stack pointer register along with a constant offset. The key insight is that since there is no aliasing to the stack frame slots they can be treated as pseudo registers. Make sure you handle properly the overlapping of stack frames at function calls. □

This completes our simplified account of the operation of VCGen. Note that the VCGen defined here constructs a global verification condition that it then passes to the Checker module. This approach, while natural and easy to describe, turns out to be too wasteful. For large examples on the order of millions of instructions it is quite common for this monolithic formula to require hundreds of megabytes for storage, slowing down the checking process considerably. A high-level type checker that would construct an explicit typing derivation would be just as wasteful. A more efficient VCGen architecture passes to the Checker module each verification condition as it is produced. After the checker validates it, the verification condition is discarded and the symbolic evaluation resumes. This optimization might not seem interesting from a scientific point of view, but it is illustrative of a number of engineering details that must be addressed to make PCC scalable.

5.4 Soundness Proof

In this section we prove that the type checking technique presented so far is sound, in the sense that "well-typed programs cannot go wrong." More precisely, we prove that if the global verification condition for a program is provable using the proof rules given by the safety policy, then the program is guaranteed to execute without violating memory safety. The method we use is similar to those used for type systems for high-level languages. We define formally the operational semantics of the assembly language, along with the notion of "going wrong." It is a bit more difficult to formalize the notion of well-typed programs. In high-level type systems there is a direct

connection between the typing derivations and the abstract-syntax tree of the program. In our case, the connection is indirect: we first use a verification-condition generator and then we exhibit a derivation of the global verification condition using the safety policy proof rules. In order to reflect this staging in the operation of our type checker, we split the soundness proof into a proof of soundness of the set of safety policy rules and a proof of soundness of the VCGen algorithm.

Soundness of the Safety Policy

The ultimate goal of our safety policy is to provide memory safety. In order to prove that our typing rules enforce memory safety, we must first define the semantics of the expression and formula constructors that we have defined.

The semantic domain for the expressions is the set of integers,[1] except for the memory expressions that we model using partial maps from integers to integers.

Next we observe that the typing formulas involving pointer types and the `listinv` formulas have a well-defined meaning only in a given context that assigns types to addresses. The necessary context is a mapping $\mathcal{M}$ from a *valid* address to the word type of the value stored at that address. Since we do not consider allocation or deallocation, our type system ensures that the mapping $\mathcal{M}$ remains constant throughout the execution of the program.

We write $\models_{\mathcal{M}} F$ when the formula F holds in the memory typing $\mathcal{M}$. A few of the most interesting cases from the definition of $\models_{\mathcal{M}}$ are shown below:

$$
\begin{array}{lll}
\models_{\mathcal{M}} F_1 \wedge F_2 & \texttt{iff} & \models_{\mathcal{M}} F_1 \text{ and } \models_{\mathcal{M}} F_2 \\
\models_{\mathcal{M}} F_1 \Rightarrow F_2 & \texttt{iff} & \text{whenever } \models_{\mathcal{M}} F_1 \text{ then } \models_{\mathcal{M}} F_2 \\
\models_{\mathcal{M}} \forall x.F(x) & \texttt{iff} & \forall e \in \mathbb{Z}.\ \models_{\mathcal{M}} F(e) \\
 & & \\
\models_{\mathcal{M}} a : \texttt{int} & \texttt{iff} & a \in \mathbb{Z} \\
\models_{\mathcal{M}} a : \texttt{list}\ W & \texttt{iff} & a = 0 \vee (\mathcal{M}(a) = W \wedge \mathcal{M}(a+4) = \texttt{list}\ W) \\
\models_{\mathcal{M}} a : \texttt{ptr}\ \{S\} & \texttt{iff} & \forall i.0 \leq i < |S| \Rightarrow \mathcal{M}(a + 4 * i) = S_i \\
\models_{\mathcal{M}} a : \{y \mid F(y)\} & \texttt{iff} & \models_{\mathcal{M}} F(a) \\
 & & \\
\models_{\mathcal{M}} \texttt{listinv}\ m & \texttt{iff} & \forall a \in Dom(\mathcal{M}).a \in Dom(m) \text{ and } \models_{\mathcal{M}} m\ a : \mathcal{M}(a) \\
\models_{\mathcal{M}} \texttt{addr}\ a & \texttt{iff} & a \in Dom(\mathcal{M})
\end{array}
$$

In the above definition we used the notation $|S|$ for the length of a sequence of word types S, and S_i for the i^{th} element of the sequence.

1. A more accurate model would use integers modulo 2^{32} in order to reflect the finite range of integers that are representable as machine words.

With these definitions we can now start to prove the soundness of the derivation rules. Given a rule with variables $x_1, \ldots, x_n$, premises $H_1, \ldots, H_m$ and conclusion C, we must prove

$$\models_{\mathcal{M}} \forall x_1. \forall x_2. \ldots \forall x_n. (H_1 \wedge \cdots \wedge H_m) \Rightarrow C$$

For example, the soundness of rule SEL requires proving the following fact:

$$\models_{\mathcal{M}} \forall a. \forall W. \forall m. (a : \mathtt{ptr}\ \{W\}) \wedge (\mathtt{listinv}\ m) \Rightarrow (\mathtt{sel}\ m\ a) : W$$

From the first assumption we derive that $\mathcal{M}(a) = W$. From the second assumption we derive that $\models_{\mathcal{M}} m\ a : W$ and since $\models_{\mathcal{M}} (\mathtt{sel}\ m\ a) = m\ a$ we obtain the desired conclusion.

5.4.1 EXERCISE [RECOMMENDED, ⋆]: Prove that CONS and NEXT are sound. □

5.4.2 EXERCISE [⋆⋆, ↛]: Prove the soundness of the remaining rules shown in Figure 5-6. □

An Operational Semantics for Assembly Language

Next we formalize an operational semantics for the assembly language. We model the execution state as a mapping ρ from register names to values. Just like in the previous chapter, the domain of values is $\mathbb{Z}$, except for the $\mathtt{r}_M$ register, which takes as values partial mappings from $\mathbb{Z}$ to $\mathbb{Z}$. Since we do not consider allocation and deallocation, the domain of the memory mapping does not change. Let $\mathcal{A}ddr$ be that domain. The operational semantics is defined only for programs whose memory accesses are only to addresses in the $\mathcal{A}ddr$ domain.

We write $(\rho\ e)$ for the result of evaluating in the register state ρ the expression e, which can refer to register names. We write $\rho[\mathtt{r}_r \leftarrow v]$ for the new register state obtained after setting register $\mathtt{r}_r$ to value v in state ρ.

The operational semantics is defined in Figure 5-9 in the form of a small-step transition relation $(i, \rho) \rightsquigarrow (i, \rho')$ from a given program counter and register state to another such pair. Notice that the transition relation is defined for memory operations only if the referenced addresses are valid.

We follow the usual convention and leave the transition relation undefined for those states where the execution is considered unsafe. For instance, the transition relation is not defined if the program counter is outside the code area or if it points to an unrecognized instruction. More importantly, the transition relation is not defined if a memory access is attempted at an invalid address.

$$(i,\rho) \rightsquigarrow \begin{cases} (i+1, \rho[\mathtt{r}_d \leftarrow \rho\ e]), & \textit{if} \quad \Pi_i = \mathtt{set}\ \mathtt{r}_d\ \mathtt{to}\ e \\ (i+1, \rho[\mathtt{r}_d \leftarrow \rho\ (\mathtt{sel}\ \mathtt{r}_M\ e)]), & \textit{if} \quad \Pi_i = \mathtt{load}\ \mathtt{r}_d\ \mathtt{from}\ e \\ & \quad \textit{and} \quad \rho\ e \in \mathcal{A}ddr \\ (i+1, \rho[\mathtt{r}_M \leftarrow \rho\ (\mathtt{upd}\ \mathtt{r}_M\ e_2\ e_1)]), & \textit{if} \quad \Pi_i = \mathtt{write}\ e_1\ \mathtt{to}\ e_2 \\ & \quad \textit{and} \quad \rho\ e_2 \in \mathcal{A}ddr \\ (L, \rho), & \textit{if} \quad \Pi_i = \mathtt{if}\ e\ \mathtt{goto}\ L \\ & \quad \textit{and} \quad \rho\ e \\ (i+1, \rho), & \textit{if} \quad \Pi_i = \mathtt{if}\ e\ \mathtt{goto}\ L \\ & \quad \textit{and} \quad \rho\ (\mathtt{not}\ e) \\ (i+1, \rho), & \textit{if} \quad \Pi_i = \mathtt{INV}\ I \end{cases}$$

Figure 5-9: The abstract machine for the soundness proof

Soundness of Verification-Condition Generation

The soundness theorem for VCGen states that if the verification condition holds, and all addresses that are in $Dom(\mathcal{M})$ are valid addresses (i.e., they belong to $\mathcal{A}ddr$), then the execution starting at the beginning of the agent in a state that satisfies the precondition will make progress either forever or until it reaches a return instruction in a state that satisfies the postcondition. What this theorem rules out is the possibility that the execution gets stuck either because it tries to execute an instruction at an invalid program counter, or it tries to dereference an invalid address. The formal statement of the theorem is the following:

5.4.3 THEOREM [SOUNDNESS OF VCGEN]: Let ρ_1 be a state such that $\models_{\mathcal{M}} \rho_1\ \mathit{Pre}$. If $Dom(\mathcal{M}) \subseteq \mathcal{A}ddr$ and if $\models_{\mathcal{M}} VC$ then the execution starting at $(1, \rho_1)$ can make progress either forever, or until it reaches a `return` instruction in state ρ, in which case $\models_{\mathcal{M}} \rho\ \mathit{Post}$. □

We prove by induction on the number of execution steps that either we have reached the return instruction, or else we can make further progress. As in all proofs by induction, the most delicate issue is the choice of the induction hypothesis. Informally, our induction hypothesis is that for each execution state there is a "corresponding" state of the symbolic evaluator.

In order to express the notion of correspondence, we must consider the differences between the concrete execution states ρ (mapping register names to

values) and the symbolic evaluation states σ (mapping register names to symbolic expressions that use expression constructors and variables). To bridge these two notions of states we need a mapping ϕ from variables that appear in σ, to values. For a symbolic expression e that contains variables, we write $(\phi\ e)$ for the result of replacing the variables in e as specified by ϕ and evaluating the result. Consequently, we write $\phi \circ \sigma$ for a mapping from register names to values that maps each register name $\mathtt{r}_i$ to the value $\phi\ (\sigma\ \mathtt{r}_i)$. Thus $\phi \circ \sigma$ is a concrete execution state.

The main relationship that we impose between ρ and σ is that there exists a mapping ϕ such that $\rho = \phi \circ \sigma$. The full induction hypothesis relates these states with the program counter i, and is defined as follows:

$$IH(i, \rho, \sigma, \phi) \stackrel{\text{def}}{=} \rho = \phi \circ \sigma \text{ and } \models_{\mathcal{M}} \phi\ (SE(i, \sigma))$$

The core of the soundness proof is the following lemma:

5.4.4 THEOREM [PROGRESS]: Let Π be a program such that $\models_{\mathcal{M}} VC$ and $Dom(\mathcal{M}) \subseteq \mathcal{A}ddr$. For any execution state (i, ρ) and σ and ϕ such that $IH(i, \rho, \sigma, \phi)$ then either:

- $\Pi_i = \mathtt{return}$, and $\models_{\mathcal{M}} \rho\ Post$, or
- there exist new states ρ', σ' and a mapping ϕ' such that $(i, \rho) \rightarrow (i', \rho')$ and $IH(i', \rho', \sigma', \phi')$. □

Proof: The proof is by case analysis on the current instruction. Since we have that $\models_{\mathcal{M}} \phi\ SE(i, \sigma)$ we know that $SE(i, \sigma)$ is defined, hence the program counter is valid and Π_i is a valid instruction. We show here only the most interesting cases.

Case: $\Pi_i = \mathtt{return}$. In this case $SE(i, \sigma) = \sigma\ Post$ and from $\models_{\mathcal{M}} \phi\ SE(i, \sigma)$ along with $\rho = \phi \circ \sigma$ we can infer that $\models_{\mathcal{M}} \rho\ Post$.

Case: $\Pi_i = \mathtt{load}\ \mathtt{r}_d\ \mathtt{from}\ e$. In this case $SE(i, \sigma) = \mathtt{addr}\ (\sigma\ e) \wedge SE(i + 1, \sigma[\mathtt{r}_d \leftarrow \sigma\ (\mathtt{sel}\ \mathtt{r}_M\ e)])$. Let $\sigma' = \sigma[\mathtt{r}_d \leftarrow \sigma\ (\mathtt{sel}\ \mathtt{r}_M\ e)]$, $\rho' = \rho[\mathtt{r}_d \leftarrow \rho\ (\mathtt{sel}\ \mathtt{r}_M\ e)]$, $i' = i + 1$ and $\phi' = \phi$. In order to prove progress, we must prove $\rho\ e \in \mathcal{A}ddr$. The induction hypothesis $IH(i, \rho, \sigma, \phi)$ ensures that $\models_{\mathcal{M}} (\mathtt{addr}\ (\phi(\sigma\ e)))$, which in turn means that $(\rho\ e) \in Dom(\mathcal{M})$. Since we require that the memory typing be defined only on valid addresses we obtain the progress condition.

Next we have to prove that the induction hypothesis is preserved. The only interesting part of this proof is that $\phi' \circ \sigma' = \rho'$, which in turn requires proving that $\phi(\sigma\ (\mathtt{sel}\ \mathtt{r}_M\ e)) = \rho\ (\mathtt{sel}\ \mathtt{r}_M\ e)$. This follows from $\phi \circ \sigma = \rho$.

Case: $\Pi_i = \mathtt{INV}\ I$. In this case $SE(i, \sigma) = \sigma\ I$. We know that $\models_{\mathcal{M}} \phi(\sigma\ I)$ and therefore $\models_{\mathcal{M}} \rho\ I$. The execution can always make progress for an invariant

instruction and we must choose $i' = i + 1$ and $\rho' = \rho$. We know that $\models_{\mathcal{M}} VC$ and hence

$$\models_{\mathcal{M}} \forall x_1 \ldots . \forall x_n . \sigma_0\, I \Rightarrow SE(i+1, \sigma_0)$$

where $\sigma_0 = \{\mathsf{r}_1 = x_1, \ldots, \mathsf{r}_n = x_n\}$. We choose $\sigma' = \sigma_0$ and ϕ' as follows:

$$\phi' = \{x_1 = \rho\ \mathsf{r}_1, \ldots, x_n = \rho\ \mathsf{r}_n\}$$

This ensures that $\rho = \phi' \circ \sigma'$ and also that $\models_{\mathcal{M}} \phi'\ SE(i+1, \sigma')$, which completes this case of the proof. □

5.4.5 EXERCISE [RECOMMENDED, ⋆, ↛]: Finish the proof of Theorem 5.4.4 by proving the remaining cases (assignment, conditional branch and memory write). □

The progress theorem constitutes the inductive case of the proof of the soundness theorem 5.4.3.

5.4.6 EXERCISE [⋆]: Prove Theorem 5.4.3. □

5.5 The Representation and Checking of Proofs

In previous sections, we showed how verification-condition generation can be used to verify certain properties of low-level code. The soundness theorem states that VCGen constructs a valid verification condition for an agent program only if the agent meets the safety policy. One way to verify the validity of the verification condition is to witness a derivation using a sound system of proof rules. In PCC such a derivation must be attached to the untrusted code so that the Checker module can find and check it. For this to work properly in practice, we need a framework for encoding proofs of logical formulas so that they are relatively compact and easy to check. We would like to have a framework and not just one proof checker for a given logic because we want to be able to change the set of axioms and inference rules as we adapt PCC to different safety policies. We would like to be able to adapt proof checking to other safety policies with as few changes to the infrastructure as possible. In this section we present a logical framework derived from the Edinburgh Logical Framework (Harper, Honsell, and Plotkin, 1993), along with associated proof representation and proof checking algorithms, that have the following desirable properties:

- The framework can be used to encode judgments and derivations from a wide variety of logics, including first-order and higher-order logics.
- The implementation of the proof checker is parameterized by a high-level description of the logic. This allows a unique implementation of the proof checker to be used with many logics and safety policies.

- The proof checker performs a directed, one-pass inspection of the proof object, without having to perform search. This leads to a simple implementation of the proof checker that is easy to trust and install in existing extensible systems.
- Even though the proof representation is detailed, it is also compact.

The above desiderata are important not only for proof-carrying code but for any application where proofs are represented and manipulated explicitly. One such application is a proof-generating theorem prover. A theorem prover that generates an explicit proof object for each successfully proved predicate enables a distrustful user to verify the validity of the proved theorem by checking the proof object. This effectively eliminates the need to trust the soundness of the theorem prover at the relatively small expense of having to trust a much simpler proof checker.

The first impulse when designing efficient proof representation and validation algorithms is to specialize them to a given logic or a class of related logics. For example, we might define the representation and validation algorithms by cases, with one case for each proof rule in the logic. This approach has the major disadvantage that new algorithms must be designed and implemented for each logic. To make matters worse, the size of such proof checking implementations grow with the number of proof rules in the logic. We would prefer instead to use general algorithms parameterized by a high-level description of the particular logic of interest.

We choose the Edinburgh Logical Framework (LF) as the starting point in our quest for efficient proof manipulation algorithms because it scores very high on the first three of the four desirable properties listed above. Edinburgh LF is a simple variant of λ-calculus with the property that, if a predicate is represented as an LF type then any LF expression of that type is a proof of that predicate. Thus, the simple logic-independent LF type-checking algorithm can be used for checking proofs.

The Edinburgh Logical Framework

The Edinburgh Logical Framework (also referred to as LF) has been introduced by Harper, Honsell, and Plotkin (1993) as a metalanguage for high-level specification of logics. LF provides natural support for the management of binding operators and of the hypothetical and schematic judgments through LF bound variables. Consider for example, the usual formulation of the implication introduction rule IMPI in first-order logic, shown in Figure 5-5. This rule is hypothetical because the proof of the right-hand side of the implication can use the assumption that the left-hand side holds. However, there is

a side condition requiring that this assumption not be used elsewhere in the proof. As we shall see below, LF can represent this side condition in a natural way by representing the assumption as a local variable bound in the proof of the right side of the implication. The fact that these techniques are supported directly by the logical framework is a crucial factor for the succinct formalization of proofs.

The LF type theory is a language with entities at three levels: objects, types and kinds, whose abstract syntax is shown below:

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

Types are used to classify objects and similarly, kinds are used to classify types. The type $\Pi x{:}A.B$ is a dependent function type with x bound in B. In the special case when x does not occur in B, we use the more familiar notation $A \to B$. Also, `Type` is the base kind, a is a type constant and c is an object constant. Dependent types are covered in detail in Chapter 2.

The encoding of a logic in LF is described by an *LF signature* Σ that contains declarations for a set of LF type constants and object constants corresponding to the syntactic formula constructors and to the proof rules. For a more concrete discussion, I describe in this section the LF representation of the safety policy that we have developed for our example agent.

The syntax of the logic is described in Figure 5-10. This signature defines an LF type constant for each kind of syntactic entity in the logic: expressions (ι), formulas (o), word types (w), and structure types (s). Then, there is an LF constant declaration for each syntactic constructor, whose LF type describes the arity of the constructor and the types of the arguments and constructed value. Two of these are worth explaining. The `settype` constructor, used to represent word types of the form $\{y \mid F(y)\}$, has one argument, the function F from expressions to formulas; similarly, the `all` constructor encodes universally quantified formulas. In both of these cases, we are representing a binding in the object logic (i.e., the logic that is being represented) with a binding in LF. The major advantage of this representation is that α-equivalence and β-reduction in the object logic are supported implicitly by the similar mechanisms in LF. This *higher-order representation* strategy is essential for a concise representation of logics with binding constructs.

The LF representation function $\ulcorner\cdot\urcorner$ is defined inductively on the structure of expressions, types and formulas. For example:

$$\begin{array}{rcl} \ulcorner P \Rightarrow (P \wedge P) \urcorner & = & \mathtt{imp}\ \ulcorner P \urcorner\ (\mathtt{and}\ \ulcorner P \urcorner\ \ulcorner P \urcorner) \\ \ulcorner \forall x.\mathtt{addr}\ x \urcorner & = & \mathtt{all}\ (\lambda x : \iota.\mathtt{addr}\ x) \end{array}$$

ι	:	Type			
o	:	Type			
w	:	Type			
s	:	Type			
zero	:	ι			
sel	:	$\iota \to \iota \to \iota$			
upd	:	$\iota \to \iota \to \iota \to \iota$			
int	:	w			
list	:	$w \to w$			
seq1	:	$w \to s$			
seq2	:	$w \to s \to s$			
ptr	:	$s \to w$			
settype	:	$(\iota \to o) \to w$			

(a)

true	:	o
and	:	$o \to o \to o$
impl	:	$o \to o \to o$
all	:	$(\iota \to o) \to o$
eq	:	$\iota \to \iota \to o$
neq	:	$\iota \to \iota \to o$
addr	:	$\iota \to o$
hastype	:	$\iota \to w \to o$
ge	:	$\iota \to \iota \to o$

(b)

Figure 5-10: LF signature for the syntax of first-order predicate logic with equality and subscripted variables, showing expression (a) and predicate (b) constructors

5.5.1 EXERCISE [⋆]: Write the LF representation of the predicate $\forall a.a : \texttt{ptr}\ \{\texttt{int}\} \Rightarrow \texttt{addr}\ a$. □

The strategy for representing proofs in LF is to define a type family "pf" indexed by representation of formulas. Then, we represent the proof of "F" as an LF expression having type "pf F." This representation strategy is called "judgments as types and derivations as objects" and was first used in the work of Harper, Honsell, and Plotkin (1993). Note that the dependent types of LF allow us to encode not only that an expression encodes a proof but also which formula it proves.

One can view the axioms and inference rules as proof constructors. This justifies representing the axioms and inference rules in a manner similar to the syntactic constructors, by means of LF constants. The signature shown in Figure 5-11 contains a fragment of the proof constructors required for the proof rules shown in Figure 5-5 (for first-order logic) and Figure 5-6 (for our safety policy). Note how the dependent types of LF can define precisely the meaning of each rule. For example, the declaration of the constant "andi"

$$
\begin{array}{lcl}
\mathtt{pf} & : & o \to \mathtt{Type} \\
 & & \\
\mathtt{truei} & : & \mathtt{pf}\ \mathtt{true} \\
\mathtt{andi} & : & \Pi p{:}o.\Pi r{:}o.\mathtt{pf}\ p \to \mathtt{pf}\ r \to \mathtt{pf}\ (\mathtt{and}\ p\ \ r) \\
\mathtt{andel} & : & \Pi p{:}o.\Pi r{:}o.\mathtt{pf}\ (\mathtt{and}\ p\ \ r) \to \mathtt{pf}\ p \\
\mathtt{ander} & : & \Pi p{:}o.\Pi r{:}o.\mathtt{pf}\ (\mathtt{and}\ p\ \ r) \to \mathtt{pf}\ r \\
\mathtt{impi} & : & \Pi p{:}o.\Pi r{:}o.(\mathtt{pf}\ p \to \mathtt{pf}\ r) \to \mathtt{pf}\ (\mathtt{impl}\ p\ \ r) \\
\mathtt{impe} & : & \Pi p{:}o.\Pi r{:}o.\mathtt{pf}\ (\mathtt{impl}\ p\ \ r) \to \mathtt{pf}\ p \to \mathtt{pf}\ r \\
\mathtt{alli} & : & \Pi p{:}\iota \to o.(\Pi v{:}\iota.\mathtt{pf}\ (p\ v)) \to \mathtt{pf}\ (\mathtt{all}\ \ p) \\
\mathtt{alle} & : & \Pi p{:}\iota \to o.\Pi e{:}\iota.\mathtt{pf}\ (\mathtt{all}\ \ p) \to \mathtt{pf}\ (p\ e) \\
\mathtt{mem0} & : & \Pi m{:}\iota.\Pi a{:}\iota.\Pi v{:}\iota.\Pi a'{:}\iota.\mathtt{pf}\ (\mathtt{eq}\ a\ a') \to \mathtt{pf}\ (\mathtt{eq}\ (\mathtt{sel}\ (\mathtt{upd}\ m\ a\ v)\ a')\ v) \\
\mathtt{mem1} & : & \Pi m{:}\iota.\Pi a{:}\iota.\Pi v{:}\iota.\Pi a'{:}\iota. \\
 & & \qquad \mathtt{pf}\ (\mathtt{neq}\ a\ a') \to \mathtt{pf}\ (\mathtt{eq}\ (\mathtt{sel}\ (\mathtt{upd}\ m\ a\ v)\ a')\ (\mathtt{sel}\ m\ a')) \\
 & & \\
\mathtt{cons} & : & \Pi E{:}\iota.\Pi W{:}w. \\
 & & \mathtt{pf}\ (\mathtt{hastype}\ E\ (\mathtt{list}\ W)) \to \mathtt{pf}\ (\mathtt{neq}\ E\ \mathtt{zero}) \to \\
 & & \mathtt{pf}\ (\mathtt{hastype}\ E\ (\mathtt{ptr}\ (\mathtt{seq2}\ W\ (\mathtt{seq1}\ (\mathtt{list}\ W))))). \\
\mathtt{set} & : & \Pi E{:}\iota.\Pi F{:}\iota \to o.\mathtt{pf}\ (\mathtt{hastype}\ E\ (\mathtt{settype}\ F)) \to \mathtt{pf}\ (F\ E).
\end{array}
$$

Figure 5-11: LF signature for safety policy proof rules

says that, in order to construct the proof of a conjunction of two predicates, one can apply the constant "andi" to four arguments, the first two being the two conjuncts and the other two being the representations of proofs of the conjuncts respectively.

The LF representation function $\ulcorner\cdot\urcorner$ is extended to derivations and is defined recursively on the derivation, as shown in the following examples (the letters $\mathcal{D}$ are used to name sub-derivations):

$$
\ulcorner\ \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ F_1 & F_2\end{array}}{F1 \wedge F_2}\ \urcorner = \mathtt{andi}\ \ulcorner F_1\urcorner\ \ulcorner F_2\urcorner\ \ulcorner\mathcal{D}_1\urcorner\ \ulcorner\mathcal{D}_2\urcorner
$$

$$
\ulcorner\ \dfrac{\begin{array}{c}[F_1] \\ \vdots\ \mathcal{D}^u \\ F_2\end{array}}{F_1 \Rightarrow F_2}\ \urcorner = \mathtt{impi}\ \ulcorner F_1\urcorner\ \ulcorner F_2\urcorner\ (\lambda u{:}\mathtt{pf}\ \ulcorner F_1\urcorner.\ulcorner\mathcal{D}^u\urcorner)
$$

$$M \;=\; \texttt{impi}\ \ulcorner F \urcorner\ (\texttt{and}\ \ulcorner F \urcorner\ \ulcorner F \urcorner)\ (\lambda x{:}\texttt{pf}\ \ulcorner F \urcorner.\texttt{andi}\ \ulcorner F \urcorner\ \ulcorner F \urcorner\ x\ x)$$

Figure 5-12: LF representation of a proof of $F \Rightarrow (F \wedge F)$

In the representation of the implication introduction proof rule, the letter u is the name of the assumption that F_1 holds. Note how the representation encodes the constraint that this assumptions must be local to the proof of F_2.

To conclude the presentation of the LF representation, consider the proof of the formula "$F \Rightarrow (F \wedge F)$." The LF representation of this proof is shown in Figure 5-12.

5.5.2 EXERCISE [$\star$]: Write the LF representation of the proof of the formula $\forall a.a : \texttt{ptr}\ \{\texttt{int}\} \Rightarrow \texttt{addr}\ a$, using the proof rules from our safety policy. □

The LF Type System

The main advantage of using LF for proof representation is that proof validity can be checked by a simple type-checking algorithm. That is, to check that the LF object M is the representation of a valid proof of the predicate F we use the LF typing rules (to be presented below) to verify that M has type $\texttt{pf}\ \ulcorner F \urcorner$ in the context of the signature Σ declaring the valid proof rules.

Type checking in the LF type system is defined by means of four judgments described below:

$\Gamma \overset{LF}{\vdash} A : K$	A is a valid type of kind K
$\Gamma \overset{LF}{\vdash} M : A$	M is a valid object of type A
$A \equiv_{\beta\eta} B$	type A is $\beta\eta$-equivalent to type B
$M \equiv_{\beta\eta} N$	object M is $\beta\eta$-equivalent to object N

where Γ is a typing context assigning types to LF variables. These typing judgment are with respect to a given signature Σ.

The derivation rules for the LF typing judgments are shown in Figure 5-13. For the $\beta\eta$-equivalence judgments we omit the rules that define it to be an equivalence and a congruence.

As an example of how LF type checking is used to perform proof checking, consider LF term M shown in Figure 5-12, representing a proof of the predicate $F \Rightarrow (F \wedge F$ by implication introduction followed by conjunction introduction. It is easy to verify, given the LF typing rules and the declaration of the

Types $\boxed{\Gamma \stackrel{LF}{\vdash} A : K}$

$$\frac{\Sigma(a) = K}{\Gamma \stackrel{LF}{\vdash} a : K}$$

$$\frac{\Gamma \stackrel{LF}{\vdash} A : \Pi x{:}B.K \qquad \Gamma \stackrel{LF}{\vdash} M : B}{\Gamma \stackrel{LF}{\vdash} A\ M : [M/x]K}$$

$$\frac{\Gamma \stackrel{LF}{\vdash} A : \mathsf{Type} \qquad \Gamma, x : A \stackrel{LF}{\vdash} B : \mathsf{Type}}{\Gamma \stackrel{LF}{\vdash} \Pi x{:}A.B : \mathsf{Type}}$$

Objects $\boxed{\Gamma \stackrel{LF}{\vdash} M : A}$

$$\frac{\Sigma(c) = A}{\Gamma \stackrel{LF}{\vdash} c : A}$$

$$\frac{\Gamma(x) = A}{\Gamma \stackrel{LF}{\vdash} x : A}$$

$$\frac{\Gamma, x : A \stackrel{LF}{\vdash} M : B}{\Gamma \stackrel{LF}{\vdash} \lambda x{:}A.M : \Pi x{:}A.B}$$

$$\frac{\Gamma \stackrel{LF}{\vdash} M : \Pi x{:}A.B \qquad \Gamma \stackrel{LF}{\vdash} N : A}{\Gamma \stackrel{LF}{\vdash} MN : [N/x]B}$$

$$\frac{\Gamma \stackrel{LF}{\vdash} M : A \qquad A \equiv_{\beta\eta} B}{\Gamma \stackrel{LF}{\vdash} M : B}$$

Equivalence $\boxed{M \equiv_{\beta\eta} N}$

$$(\lambda x{:}A.M)N \equiv_{\beta\eta} [N/x]M$$

Figure 5-13: The LF type system

constants involved, that this proof has the LF type "$\mathtt{pf}\ (\mathtt{imp}\ulcorner F\urcorner\ (\mathtt{and}\ulcorner F\urcorner\ulcorner F\urcorner))$." The adequacy of LF type checking for proof checking in the logic under consideration is stated formally in the Theorems 5.5.3 and 5.5.4 below. These theorems follow immediately from lemmas proved in Harper, Honsell, and Plotkin (1993). They continue to hold if the logic is extended with new expression and predicate constructors.

5.5.3 THEOREM [ADEQUACY OF SYNTAX REPRESENTATION]:

1. If E is a closed expression, then $\cdot \stackrel{LF}{\vdash} \ulcorner E\urcorner : \iota$. If M is a closed LF object such that $\cdot \stackrel{LF}{\vdash} M : \iota$, then there exists an expression E such that $\ulcorner E\urcorner \equiv_{\beta\eta} M$.

2. If W is a word-type, then $\cdot \stackrel{LF}{\vdash} \ulcorner W\urcorner : w$. If M is a closed LF object such that $\cdot \stackrel{LF}{\vdash} M : w$, then there exists a word type W such that $\ulcorner W\urcorner \equiv_{\beta\eta} M$.

3. If S is a structured type, then $\cdot \stackrel{LF}{\vdash} \ulcorner S\urcorner : s$. If M is a closed LF object such that $\cdot \stackrel{LF}{\vdash} M : s$, then there exists a structured type S such that $\ulcorner S\urcorner \equiv_{\beta\eta} M$.

4. If F is a closed formula, then $\cdot \stackrel{LF}{\vdash} \ulcorner F\urcorner : o$. If M is a closed LF object such that $\cdot \stackrel{LF}{\vdash} M : o$, then there exists a formula F such that $\ulcorner F\urcorner \equiv_{\beta\eta} M$. □

5.5.4 THEOREM [ADEQUACY OF DERIVATION REPRESENTATION]:

1. If $\mathcal{D}$ is a derivation of F then $\cdot \stackrel{LF}{\vdash} \ulcorner \mathcal{D}\urcorner : \mathtt{pf}\ \ulcorner F\urcorner$.

2. If M is a closed LF object such that $\cdot \stackrel{LF}{\vdash} M : \mathtt{pf}\ \ulcorner F \urcorner$, then there exists a derivation $\mathcal{D}$ of F such that $\ulcorner \mathcal{D} \urcorner \equiv_{\beta\eta} M$. □

In the context of PCC, Theorem 5.5.4(2) says that if the agent producer can exhibit an LF object having the type "$\mathtt{pf}\ \ulcorner VC \urcorner$" then there is a derivation of the verification condition within the logic, which in turn means that the verification condition is valid and the agent code satisfies the safety policy.

Owing to the simplicity of the LF type system, the implementation of the type checker is simple and easy to trust. Furthermore, because all of the dependencies on the particular object logic are separated in the signature, the implementation of the type checker can be reused directly for proof checking in various first-order or higher-order logics. The only logic-dependent component of the proof checker is the signature, which is usually easy to verify by visual inspection.

Unfortunately, the above-mentioned advantages of LF representation of proofs come at a high price. The typical LF representation of a proof is large, due to a significant amount of redundancy. This fact can already be seen in the proof representation shown in Figure 5-12, where there are six copies of F as opposed to only three in the predicate to be proved. The effect of redundancy observed in practice increases non-linearly with the size of the proofs. Consider for example, the representation of the proof of the n-way conjunction $F \wedge \ldots \wedge F$. Depending on how balanced is the binary tree representing this predicate, the number of copies of F in the proof representation ranges from an expected value of $n \log n$ (when the tree is perfectly balanced) to a worse case value of $n^2/2$ (when the tree degenerates into a list). The redundancy of representation is not only a space problem but also leads to inefficient proof checking, because all of the redundant copies have to be type checked and then checked for equivalence with instances of F from the predicate to be proved.

The proof representation and checking framework presented in the next section is based on the observation that it is possible to retain only the skeleton of an LF representation of a proof and to use a modified LF type-checking algorithm to reconstruct on the fly the missing parts. The resulting *implicit LF* (or LF_i) representation inherits the advantages of the LF representation (i.e., small and logic-independent implementation of the proof checker) without the disadvantages (i.e., large proof sizes and slow proof checking).

Implicit LF

The solution to the redundancy problem is to eliminate the redundant subterms from the proof. In most cases we can eliminate all copies of a given

subterm from the proof and rely instead on the copy that exists within the predicate to be proved, which is constructed by the VCGen and is trusted to be well formed. But now the code receiver will be receiving proofs with missing subterms. One possible strategy is for the code receiver to reconstruct the original form of the proof and then to use the simple LF type checking algorithm to validate it. But this does not save proof-checking time and requires significantly more working memory than the size of the incoming LF_i proof. Instead, we modify the LF type-checking algorithm to reconstruct the missing subterms while it performs type checking. One major advantage of this strategy is that terms that are reconstructed based on copies from the verification condition do not need to be type checked themselves.

We will not show the formal details of the type reconstruction algorithm but will show instead how it operates on a simple example. For expository purposes, the missing proof subterms are marked with placeholders, written as $*$. Consider now the proof of the predicate $F \Rightarrow (F \wedge F)$ of Figure 5-12. If we replace all copies of "F" with placeholders we get the following LF_i object:

$$\texttt{impi}\ *_1\ *_2\ (\lambda u{:}*_3.\texttt{andi}\ *_4\ *_5\ u\ u)$$

This implicit proof captures the structure of the proof without any redundant information. The subterms marked with placeholders can be recovered while verifying that the term has type "$\texttt{pf}\ (\texttt{impl}\ulcorner F\urcorner\ (\texttt{and}\ulcorner F\urcorner\ \ulcorner F\urcorner))$," as described below.

Reconstruction starts by recognizing the top-level constructor `impi`. The expected type of the entire term, "$\texttt{pf}\ (\texttt{impl}\ \ulcorner F\urcorner\ (\texttt{and}\ \ulcorner F\urcorner\ \ulcorner F\urcorner))$," is "matched" against the result type of the `impi` constant, as given by the signature Σ. The result of this matching is an instantiation for placeholders 1 and 2 and a residual type-checking constraint for the explicit argument of `impi`, as follows:

$$\begin{array}{lllll} & *_1 & \equiv & \ulcorner F\urcorner \\ & *_2 & \equiv & \texttt{and}\ \ulcorner F\urcorner\ \ulcorner F\urcorner \\ \vdash & (\lambda u{:}*_3.\texttt{andi}\ *_4\ *_5\ u\ u) & : & \texttt{pf}\ \ulcorner F\urcorner \to \texttt{pf}\ (\texttt{and}\ \ulcorner F\urcorner\ \ulcorner F\urcorner) \end{array}$$

Reconstruction continues with the remaining type-checking constraint. From its type we can recover the value of placeholder 3 and a typing constraint for the body:

$$\begin{array}{llll} & *_3 & \equiv & \texttt{pf}\ \ulcorner F\urcorner \\ u : \texttt{pf}\ \ulcorner F\urcorner \vdash & \texttt{andi}\ *_4\ *_5\ u\ u & : & \texttt{pf}\ (\texttt{and}\ \ulcorner F\urcorner\ \ulcorner F\urcorner) \end{array}$$

Now `andi` is the top-level constant and by matching its result type as declared in the signature with the goal type of the constraint we get the instantiation

for placeholders 4 and 5 and two residual typing constraints:

$$
\begin{array}{rlcl}
 & *_4 & \equiv & \ulcorner F \urcorner \\
 & *_5 & \equiv & \ulcorner F \urcorner \\
u : \mathtt{pf}\ \ulcorner F \urcorner \vdash & u & : & \mathtt{pf}\ \ulcorner F \urcorner \\
u : \mathtt{pf}\ \ulcorner F \urcorner \vdash & u & : & \mathtt{pf}\ \ulcorner F \urcorner
\end{array}
$$

The remaining two constraints are solved by the variable typing rule. Note that this step involves verifying the equivalence of the objects $\ulcorner F \urcorner$ from the assumption and the goal. This concludes the reconstruction and checking of the entire proof. We reconstructed the full representation of the proof by instantiating all placeholders with well-typed LF objects. We know that these instantiations are well-typed because they are ultimately extracted from the original constraint type, which is assumed to contain only well-typed subterms.

The formalization of the reconstruction algorithm described informally above is in two stages. First, we show a variant of the LF type system, called implicit LF or LF_i, that extends LF with placeholders. This type system has the property that all well-typed LF_i terms can be reconstructed to well-typed LF terms. However, unlike the original LF type system, the LF_i type system is not amenable to a direct implementation of deterministic type checking. Instead, we use a separate reconstruction algorithm.

An object M is fully reconstructed, or fully explicit, when it is placeholder free. We write $\mathrm{PF}(M)$ to denote this property. We extend this notation to type environments and write $\mathrm{PF}(\Gamma)$ to denote that all types assigned in Γ to variables are placeholder free.

The LF_i typing rules are an extension of the LF typing rules with two new typing rules for dealing with implicit abstraction and placeholders, and one new β-equivalence rule dealing with implicit abstraction. These additions are shown in Figure 5-14. The LF_i typing judgment is written $\Gamma \stackrel{i}{\vdash} M : A$.

Note that according to the LF_i type system placeholders cannot occur on a function position, but only as arguments in an application. This restriction allows us to simplify the reconstruction algorithm by avoiding higher-order unification. Note also that several LF_i rules require that the types involved do not contain placeholders. This restriction simplifies greatly the proofs of soundness of the reconstruction algorithms and does not seem to diminish the effectiveness of the LF_i representation.

A quick analysis of the LF_i typing rules reveals that they are not directly useful for type checking or type inference. The main reason is that type checking an application involves "guessing" appropriate A and N. The type A can sometimes be recovered from the type of the application head, but the term

Objects $\boxed{\Gamma \stackrel{i}{\vdash} M : A}$

$$\frac{\Gamma \stackrel{i}{\vdash} M : A \qquad A \equiv_{\beta\eta} B \qquad \mathrm{PF}(A)}{\Gamma \stackrel{i}{\vdash} M : B}$$

$$\frac{\Gamma, x : A \stackrel{i}{\vdash} M : B}{\Gamma \stackrel{i}{\vdash} \lambda x{:}*.M : \Pi x{:}A.B}$$

$$\frac{\Gamma \stackrel{i}{\vdash} M : \Pi x{:}A.B \qquad \Gamma \stackrel{i}{\vdash} N : A \qquad \mathrm{PF}(A)}{\Gamma \stackrel{i}{\vdash} M\ N : [N/x]B}$$

$$\frac{\Gamma \stackrel{i}{\vdash} M : \Pi x{:}A.B \qquad \Gamma \stackrel{i}{\vdash} N : A \qquad \mathrm{PF}(A)}{\Gamma \stackrel{i}{\vdash} M\ * : [N/x]B}$$

Equivalence $\boxed{M \equiv_{\beta\eta} N}$

$$(\lambda x{:}*.M)N \equiv_{\beta\eta} [N/x]M$$

Figure 5-14: The rules that are new in the LF_i type system

N in an application to a placeholder cannot be found easily in general. This is not a problem for us because we need the LF_i type-system only as a step in proving the correctness of the type-reconstruction algorithm, and not as the basis for an implementation of a type-checking algorithm.

The only property of interest of the LF_i type system is that once we have a typing derivation we can reconstruct the object involved and a corresponding LF typing derivation for it. To make this more precise we introduce the notation $M \nearrow M'$ to denote that M' is a fully-reconstructed version of the implicit object M (i.e., $\mathrm{PF}(M')$). This means that M' can be obtained from M by replacing all of its placeholders with fully-explicit LF objects. Note that the reconstruction relation is not a function as there might be several reconstructions of a given implicit object or type.

5.5.5 THEOREM [SOUNDNESS OF LF_i TYPING]: If $\Gamma \stackrel{i}{\vdash} M : A$ and $\mathrm{PF}(\Gamma)$, $\mathrm{PF}(A)$, then there exists M' such that $M \nearrow M'$ and $\Gamma \stackrel{\mathrm{LF}}{\vdash} M' : A$. □

5.5.6 EXERCISE [⋆⋆, ↛]: Prove Theorem 5.5.5 □

5.6 Proof Generation

We have seen that a successfully checked proof of the verification condition guarantees that the verification condition is valid, which in turn guarantees that the code adheres to the safety policy. The PCC infrastructure is simple, easy-to-trust and automatic. But this is only because all the difficult tasks have been delegated to the code and proof producers. The first difficult task, besides writing code that is indeed safe, is to generate the code annotations consisting of loop invariants for all loops and of function specifications for all

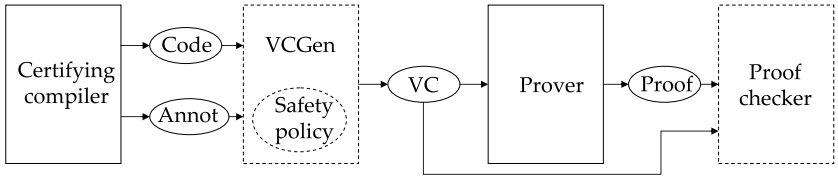

Figure 5-15: Interaction between untrusted PCC tools (continuous lines) and trusted PCC infrastructure (interrupted lines)

local functions. The other difficult task is to prove the verification condition produced by the verification-condition generator.

Fortunately there are important situations when both the generation of the annotations and of the proof can be automated. Consider the situation in which there exists a high-level language, perhaps a domain-specific one, in which the safety policy is guaranteed to be satisfied by a combination of static and run-time checks. For example, if the safety policy is memory safety then any memory-safe high-level language can be used. The key insight is that in these systems the safety policy is guaranteed to hold by the design of the static and run-time checks. In essence, the high-level type checker acts as a theorem prover. All we have to do is to show that a sufficient number and kind of static and run-time checks have been performed.

Figure 5-15 shows the interaction between the untrusted PCC tools used by the code producer and the trusted PCC infrastructure used by the code receiver. The annotations are generated automatically by a *certifying compiler* from high-level language to assembly language. For safety polices that follow closely the high-level type system, it is surprisingly easy for a compiler to produce the loop invariants, which are essentially conjunctions of type declarations for the live registers at the given program point. This is information that the compiler can easily maintain and emit.

Before it can generate the required proofs, the code producer must pass the annotated code to a local copy of VCGen. The proof itself is generated by a theorem prover customized for the specific safety policy. As discussed in Section 5.3, such a theorem prover is little more than a type checker. However, unlike a regular type checker or theorem prover, the PCC theorem prover must generate explicit representation of the proofs. The architecture shown in Figure 5-15 is described in detail in Necula (1998).

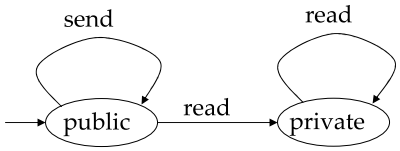

Figure 5-16: A privacy policy

5.7 PCC beyond Types

The presentation of PCC so far has focused on type-based safety policies. We have shown that verification condition generation followed by theorem proving can overcome many of the difficulties of type checking programs written in low-level languages. It should be obvious that we can take the example that we used so far and change the type system by simply changing the proof rules, with no changes to the infrastructure itself. But the machinery we have constructed in the process can be used to enforce more complex safety polices than are usually associated with types. And we can do this with very few changes, thanks both to the modular design of the infrastructure and to the choice of using the lower-level mechanism of logic rather than committing to a high-level type system. However, everytime the set of proof rules is changed, one must redo the proof of soundness. In this section, we explore one example of a safety policy that goes beyond types.

Consider a safety policy that allows access to two host services: *read* the contents of a local file and *send* data over the network. The host wishes to enforce the policy that the agent cannot send data after it has read local files. This is a conservative way to ensure that no local file contents will be leaked. This example is taken from Schneider (2000).

This safety policy can be described using the state machine shown in Figure 5-16. Initially the agent is in the `public` state in which it can use both the `send` and the `read` services. However, once it uses the `read` service the agent transitions in the `private` state, in which it can use only the `read` service.

In order to enforce such a safety policy, it is sufficient to check that the `send` service cannot be used after the `read` service has been used. At the level of assembly language, these services would be most likely implemented as function calls. In that case the privacy safety policy can be implemented as a precondition on the `send` function. Instead of introducing a general mechanism for handling function calls (see Exercise 5.3.5), we use a special-purpose handling of the instructions `call read` and `call send`.

In the presentation of PCC from previous sections, there is no element of the state of the computation that reflects whether a certain function has been invoked or not. One way to address this issue is to require that the agent code keep track of its own public/private state at run-time, presumably in a register or a memory location. Then the postcondition of `read` would require that this state element reflect the `private` state and the precondition of `send` would require that the state element reflect the `public` state. This strategy is appropriate when the producer of the agent code wishes to use run-time checking to enforce the safety policy, in which case it would have to prove that the appropriate checks have been inserted. This strategy also has the benefit of not requiring any changes in the PCC infrastructure.

We will pursue another alternative. We will modify VCGen and the symbolic evaluator to keep track of the public/private state. And since we prefer to extend the PCC infrastructure with a general-purpose mechanism rather than a specific policy, the VCGen extension should be able to record any information about the *history* of the execution, not just its public/private state.

For this purpose we extend the symbolic evaluation state with another pseudo-register, called $\mathtt{r}_H$ to store a sequence of interesting events in the past of the computation. The set of symbolic expressions that this register can have are shown below:

$$\begin{array}{lllll} \text{Histories} & H & ::= & x \mid \mathtt{event}\ V\ H \\ \text{Events} & V & ::= & \mathtt{init} \mid \mathtt{read} \mid \mathtt{send} \end{array}$$

Additionally we add a number of formulas that we can use for stating properties of the history of execution:

$$\text{Formulas} \quad F \quad ::= \quad \ldots \mid \mathtt{publicState}\ H \mid \mathtt{privateState}\ H$$

As usual when we extend the language of formulas we must also extend the proof rules. For our safety policy we add the following three proof rules:

$$\mathtt{publicState}\ (\mathtt{event}\ \mathtt{init}\ H) \qquad \text{(INIT)}$$

$$\frac{\mathtt{publicState}\ H}{\mathtt{publicState}\ (\mathtt{event}\ \mathtt{send}\ H)} \qquad \text{(SEND)}$$

$$\mathtt{privateState}\ (\mathtt{event}\ \mathtt{read}\ H) \qquad \text{(READ)}$$

The definition of the VCGen and the symbolic evaluator can remain unchanged for the instructions considered so far, except that the $\mathtt{r}_H$ register can be used in loop invariants and function preconditions and postconditions. In particular, for the privacy safety policy the invocations of the `read`

and `send` services can be handled in the symbolic evaluator as follows:

$$SE(i,\sigma) = \begin{cases} \ldots & \\ SE(i+1, \sigma[\mathsf{r}_H \leftarrow (\sigma\ (\texttt{event read}\ \mathsf{r}_H))]) & \text{if } \Pi_i = \texttt{call read} \\ \texttt{publicState}\ (\sigma\ \mathsf{r}_H) \wedge & \text{if } \Pi_i = \texttt{call send} \\ SE(i+1, \sigma[\mathsf{r}_H \leftarrow (\sigma\ (\texttt{event send}\ \mathsf{r}_H))]) & \end{cases}$$

The symbolic evaluator extends the history state with information about the services that were used. Additionally, the `send` call requires through its precondition that the history of the computation be consistent with the `public` state of the safety policy. A realistic symbolic evaluator would support a general-purpose function call instruction, in which case the effect of the `read` and `send` functions could be achieved by appropriate function preconditions and postconditions.

5.7.1 EXERCISE [⋆⋆, ↛]: Add two actions `lock` e and `unlock` e that can be used to acquire and release a lock that is denoted by the expression e. Define a PCC safety policy (extensions to the logic, new proof rules and changes to the symbolic evaluator) that requires correct use of locks: a lock cannot be acquired or released twice in a row, and the agent must release all locks upon return. □

5.7.2 EXERCISE [⋆⋆, ↛]: The verification-condition generator that we described in Section 5.3 cannot enforce a safety policy that allows the agent to "probe" the accessibility of a memory page by attempting a read from an address within that page. This is a common way to check for stack overflow in many systems. Show how you can change the symbolic evaluator to use the history register for the purpose of specifying such a safety policy. □

This example shows how to use PCC for safety policies that go beyond type checking. In fact, PCC is extremely powerful in this sense. Any safety policy that could be enforced by an interpreter using run-time checking could in principle be enforced by PCC. A major advantage of PCC over interpreters is that it can check properties that would be very expensive to check at run time. Consider, for example, how complicated it would be to write an interpreter that enforces at run-time a fine grained memory safety policy. Each memory word would have to be instrumented with information whether it is accessible or not. By comparison, we can use PCC along with a strong type system to achieve the same effect, with no run-time penalty.

5.8 Conclusion

Below is a list of the most important ways in which PCC improves over other existing techniques for enforcing safe execution of untrusted code:

- PCC operates at **load time** before the agent code is installed in the host system. This is in contrast with techniques that enforce the safety policy by relying on extensive run-time checking or even interpretation. As a result PCC agents run at native-code speed, which can be ten times faster than interpreted agents (written for example using Java bytecode) or 30% faster than agents whose memory operations are checked at run time.

  Additionally, by doing the checking at load time it becomes possible to enforce certain safety policies that are hard or impossible to enforce at run time. For example, by examining the code of the agent and the associated "explanation" PCC can verify that a certain interrupt routine terminates within a given number of instructions executed or that a video frame rendering agent can keep up with a given frame rate. Run-time enforcement of timing properties of such fine granularity is hard.

- The trusted computing base in PCC is **small**. PCC is simple and small because it has to do a relatively simple task. In particular, PCC does not have to discover on its own whether and why the agent meets the safety policy.

- For the same reason, PCC can operate even on agents expressed in **native-code** form. And because PCC can verify the code after compilation and optimization, the checked code is ready to run without needing an additional interpreter or compiler on the host. This has serious software engineering advantages since it reduces the amount of security critical code and it is also a benefit when the host environment is too small to contain an interpreter or a compiler, such as is the case for many embedded software systems.

- PCC is **general.** All PCC has to do is to verify safety explanations and to match them with the code and the safety policy. By standardizing a language for expressing the explanations and a formalism for expressing the safety policies, it is possible to implement a single algorithm that can perform the required check, for any agent code, any valid explanation and a large class of safety policies. In this sense a single implementation of PCC can be used for checking a variety of safety policies.

The PCC infrastructure is designed to complement a cryptographic authentication infrastructure. While cryptographic techniques such as digital signatures can be used by the host to verify external properties of the agent

program, such as freshness and authenticity, or the author's identity, the PCC infrastructure checks internal semantic properties of the code such as what the code does and what it does not do. This enables the host to prevent safety breaches due to either malicious intent (for agents originating from untrusted sources) or due to programming errors (for agents originating from trusted sources).

However, proof-carrying code is not without costs. The most notable challenge to using PCC is the difficulty of producing code annotations and proofs. In some cases, these can be produced automatically based on some high-level language invariants. But in general a human is required to be involved and the more complex the safety policy the more onerous the burden of proof can be expected to be. All that PCC offers in this direction is a way to shift this burden from the code received to the code producer who can be expected to have more computational power, and especially more knowledge of why the code satisfies the safety policy.

Proof-carrying code is a witness to the fact that programming language technology and type theory are the basis of valuable techniques for solving practical engineering problems. However, in the process of applying these techniques for the design of a PCC infrastructure, it became necessary to adapt the off-the-shelf techniques in non-trivial ways to the particular application domain. Some of that adaptation can be carried out in a theoretical setting, such as the extension of Edinburgh LF to implicit LF, while other parts involve real engineering.