Fundamenta Informaticae 77 (2007) 1–28
IOS Press

Tutorial Examples of the Semantic Approach to Foundational Proof-Carrying Code

Amy P. Felty
School of Information Technology and Engineering
University of Ottawa, Canada
*afelty@site.uottawa.ca**

Abstract. Proof-carrying code provides a mechanism for insuring that a host, or code consumer, can safely run code delivered by a code producer. The host specifies a safety policy as a set of axioms and inference rules. In addition to a compiled program, the code producer delivers a formal proof of safety expressed in terms of those rules that can be easily checked. Foundational proof-carrying code (FPCC) provides increased security and greater flexibility in the construction of proofs of safety. Proofs of safety are constructed from the smallest possible set of axioms and inference rules. For example, typing rules are not included. In our semantic approach to FPCC, we encode a semantics of types from first principles and the typing rules are proved as lemmas. In addition, we start from a semantic definition of machine instructions and safety is defined directly from this semantics. Since FPCC starts from basic axioms and low-level definitions, it is necessary to build up a library of lemmas and definitions so that reasoning about particular programs can be carried out at a higher level, and ideally, also be automated. We describe a high-level organization that involves Hoare-style reasoning about machine code programs. This organization is presented using two running examples. The examples, as well as illustrating the above mentioned approach to organizing proofs, is designed to provide a tutorial introduction to a variety of facets of our FPCC approach. For example, it illustrates how to prove safety of programs that traverse input data structures as well as allocate new ones.

1. Introduction

In our first presentation of the semantic approach to foundational proof-carrying code (FPCC) [2], we encoded a semantics of types and proved typing rules as lemmas from the basic definitions. We also gave a direct encoding of machine semantics from which we built several layers of definitions so that

*Address for correspondence: SITE, University of Ottawa, 800 King Edward Avenue, Ottawa, Ontario K1N 6N5, Canada

reasoning about programs was similar to reasoning using Hoare-style program verification rules. This work extended the original proof-carrying code (PCC) work [17] which stated typing rules as axioms and generated a safety theorem using a verification condition generator (VCG). Both the axioms and the VCG were parts of the system that had to be trusted.

In FPCC, much progress has been made in a variety of directions since our original work. Type systems that are currently handled are more sophisticated and include contravariant recursive types [3] and mutable references [1]. Also, larger machine instructions sets have been encoded [13]. In addition, foundational versions of typed assembly languages (TAL) [14] have been developed for use in FPCC systems (e.g. [7, 20, 21]). An alternative syntactic approach has also been explored [10].

Although we presented an example in our first account [2], it was not large enough to illustrate the structure of proofs of safety in general, or demonstrate the style of reasoning that is used to build such proofs. This paper attempts to fill this gap. Although the examples are larger, we keep the semantics simple. We only require a simple semantics of types and a simple set of machine instructions as in our first account. Because any FPCC system is built in layers so that reasoning about particular programs is done at a fairly high level, this example could be carried over fairly directly to current FPCC systems which use machine instruction sets for real machines and more complex type systems. Like our previous work, we adopt the semantic approach to FPCC here.

The first example program we present is a program which takes a list of integers as input and extends it by adding a new element at the head. It is simple, but complex enough to require recursive data types and the allocation of new memory. In addition, the program uses all of the instructions available in our simple instruction set except for jumps. We later introduce a second example — a machine language program to reverse a list of integers. The computation includes traversing the input list as well as building a new one. The main extension over the first example is the introduction of a loop.

We present four versions of the first example, starting simple, and adding complexity at each step. In the first two versions, we start with a version of PCC that can be viewed as a modified presentation of Necula's original work [17, 18], one which uses Hoare-style program verification rules for machine instructions instead of an explicit VCG. Using such rules is fairly similar to the use of a VCG, but our version provides a slightly higher level of security. In original PCC, the safety proof is a proof of the formula output by the VCG; the VCG program must be trusted. Here, the proof steps which apply the Hoare-style rules are encoded as part of the safety proof. We must trust these rules because they are a part of our basic safety policy, but this should be simpler than trusting a VCG program. Roughly, using the Hoare rules corresponds to recording the primitive steps of the VCG in the proof so that they can be later checked.

In Section 2, we start with a simple safety policy, simpler than that considered by Necula, and we prove safety of the example program with respect to the Hoare rules. Section 3 extends the safety policy to include memory safety, which is essentially the same policy considered by Necula. Presenting the example in two steps allows us to start simple, separating out the issues arising from considering a more complex safety policy. In particular, including memory safety involves adding new preconditions to the Hoare rules for load and store instructions. Although we still have a relatively simple policy, complications arise, mainly from the fact that proving that the new preconditions hold requires reasoning about types. We present a set of type rules for this purpose. At this stage, we still do not introduce foundational aspects, but we do extend Necula's typing rules. His rules could be used to prove safety of programs that traversed recursive data structures such as lists, but not programs that allocated new ones. Our presentation here can be viewed as a non-foundational version of our original presentation [2],

where an allocation predicate expressing which memory locations are allocated is added explicitly to typing judgments. We conclude this section with a complete proof of memory safety of the example program, and discuss in detail the aspects of the proof that differ from those in the first proof, mainly due to the new form of the Hoare rules for the load and store instructions.

In the rest of the paper, we turn to FPCC. As a first step, in Section 4, we give definitions for types and prove the typing rules of the previous section as lemmas that follow from these definitions. The foundational versions of the safety proofs for our examples have been fully formalized in Coq [6, 4], so at this stage, we also begin discussing this formalization.

The second step in presenting the foundational approach is the low-level encoding of machine instruction semantics. In Section 5, we first present a version of our encoding for the simple safety policy. In this and the next section, we follow the approach of Appel and Michael [13]. We start with a direct encoding of machine instructions as a step relation relating one machine state to another, and we prove a theorem stating that safety follows from "progress" and "preservation" lemmas. In the formal proof, we began by adopting and modifying some of the basic definitions in the Coq libraries used in Hamid et. al.'s syntactic approach to FPCC [10]. We present a proof of safety of our example program in the new context. We structure the proof and provide enough detail so that we can see a correspondence between reasoning using Hoare rules and reasoning using the progress and preservation lemmas. For instance, readers familiar with tools for formal verification of programs, but not necessarily familiar with progress and preservation lemmas for programming languages, should gain a better understanding of how to build and apply such tools in the PCC setting. Although most of the formal proof presented here was done interactively, we attempt to provide some insight into how to automate such proofs.

Section 6 incorporates memory safety into the direct encoding of the machine semantics, and discusses the resulting changes in the safety proof of the example program. In Section 7, we discuss the second example program and its safety proof. This example illustrates how conditional jump instructions are added to the semantics, and how programs with loops are proved safe. Finally, Section 8 discusses automating proofs of safety and other general issues.

This paper extends an earlier one [8] where we present the example for reversing a list and discuss the proof in two stages, first using Hoare-style rules and then using progress and preservation lemmas. Here, we go into more detail by starting with a simpler example and a simpler safety policy, and introduce increasing complexity in four stages instead of two. When we present the reverse program here, we give a simpler safety proof than our earlier one and describe only the aspects that are not covered by the first example program. We also include an extended discussion of a variety of issues.

2. Machine Semantics as Hoare-Style Rules

We introduce the set of machine instructions used in our first example program by presenting a set of Hoare-style rules for reasoning about them, given in Figure 1. These can be viewed as a modified form of Necula and Lee's VCG [17, 19] expressing a simplified version of safety. Note that there is one rule for each machine instruction and that these rules are axioms. The mov rule is a version of the usual assignment rule where the precondition is obtained by starting with the postcondition and replacing occurrences of the destination register r_d with constant c. The two rules for addition are similar to the mov rule. In the first, the destination register is replaced by a constant c added to the contents of a source register r_s, and in the second, there are two source registers whose contents are added together. In the

$$\frac{}{\{I[c/r_d]\}\ \texttt{MOV}\ \ r_d := c\ \{I\}}\ \mathsf{mov}$$

$$\frac{}{\{I[r_s + c/r_d]\}\ \texttt{ADDC}\ r_d := r_s + c\ \{I\}}\ \mathsf{addc}$$

$$\frac{}{\{I[r_{s_1} + r_{s_2}/r_d]\}\ \texttt{ADD}\ r_d := r_{s_1} + r_{s_2}\ \{I\}}\ \mathsf{add}$$

$$\frac{}{\{I[m(r_s + c)/r_d]\}\ \texttt{LD}\ r_d := m(r_s + c)\ \{I\}}\ \mathsf{ld}$$

$$\frac{}{\{I[m[r_d + c \mapsto r_s]/m]\}\ \texttt{ST}\ m(r_d + c) := r_s\ \{I\}}\ \mathsf{st}$$

$$\frac{\{A\}S_1\{C\} \qquad \{C\}S_2\{B\}}{\{A\}S_1; S_2\{B\}}\ \mathsf{sequence}$$

$$\frac{A \Rightarrow A' \qquad \{A'\}S\{B'\} \qquad B' \Rightarrow B}{\{A\}S\{B\}}\ \mathsf{Implied}$$

Figure 1. Hoare-style Rules for Machine Instructions

ld rule, $r_s + c$ is an address, and the value at that address in memory is the value that replaces r_d. In the st rule, it is the entire memory in the postcondition that is replaced by an expression representing a new memory, to obtain the precondition. This new memory is the same as the old except for an update at one address (address $r_d + c$). In general, we write $m[a_1 \mapsto w_1, \ldots, a_n \mapsto w_n]$ to represent a memory such that for $i = 1, \ldots, n$, address a_i has value w_i, and for all other addresses, the corresponding value is obtained from m. When we write this expression, we assume $a_1, \ldots, a_n$ are distinct.

The rules in Figure 1 can be proven sound using usual techniques for showing the soundness of Hoare logic for simple high-level languages. (See [12] for example.) In particular, all of the instructions considered here can be considered versions of the assignment statement where register names can be viewed as variable names and memory can be viewed as an array. The formulas in the preconditions and postconditions are first-order formulas built on a term language that includes arithmetic and array-update expressions. Thus soundness of these rules is easily established. The remaining rules are the same as those that appear in high-level languages; thus their soundness is immediate. Proving the implications in the Implied rule requires reasoning in first-order logic using standard logical rules. In our examples, it is mainly simple reasoning about integer arithmetic that is required. We have not yet included any rules for jump instructions, which would clearly complicate the proof of soundness. We do not consider such issues here, since the Hoare rules are used for illustration purposes only. In particular, we will see how the information embodied in these rules is used in the formalization of progress and preservation presented later, providing insight into the correspondence between these two styles of reasoning.

We will assume programs have the form $(S_1; \ldots; S_n; \texttt{JMP}\ r)$ where r is a return address. For now, statements $S_1; \ldots; S_n$ include only instructions for which there are rules in the figure. There is also an instruction ILLEGAL, which causes a program to loop indefinitely. There is no rule for this instruction,

which means that no program containing this instruction can be proved safe. The predicate $safe\_exit$ is introduced to express that it is safe to transfer control to the return address upon completion of execution. The program's precondition Pre must include some assumptions about this predicate providing information to show that whatever value r has at the end of the computation is safe. The omission of a rule for `ILLEGAL` and the safe exit condition are all that is required to express our first simple safety policy. In particular, we define a program to be safe if we can prove $\{Pre\}S_1; \ldots; S_n\{safe\_exit(r)\}$, which we denote as $safe(Pre, (S_1; \ldots; S_n; \texttt{JMP}\ r))$. We can prove that programs satisfy other properties beyond safety by including additional postconditions.

A proof of safety can be built by repeated application of the sequence rule with applications of rules for particular instructions at the leaves of the proof tree. As usual in such proofs, we can start with postcondition $safe\_exit(r)$ and apply the rule corresponding to statement S_n to obtain some formula I_n. Then I_n is used as the postcondition of statement S_{n-1} to compute I_{n-1}, and so on. In the simplest form of proof, we have leaves of the form $\{I_1\}S_1\{I_2\}, \{I_2\}S_2\{I_3\}, \ldots, \{I_n\}S_n\{safe\_exit(r)\}$ plus an application of the Implied rule requiring proof of $Pre \Rightarrow I_1$.

We abbreviate proofs of this form here by writing

$$\{Pre\}\{I_1\}S_1\{I_2\}S_2\{I_3\}\cdots\{I_n\}S_n\{safe\_exit(r)\}.$$

In particular, we use a linear form eliminating duplicate copies of formulas which occur as both pre- and post-conditions. An extra formula appearing before a Hoare triple abbreviates an application of the Implied rule. Here $\{Pre\}\{I_1\}S_1\{I_2\}$ denotes the subproof

$$\frac{Pre \Rightarrow I_1 \qquad \{I_1\}S_1\{I_2\} \qquad I_2 \Rightarrow I_2}{\{Pre\}S_1\{I_2\}}\ \textsf{Implied}$$

Proofs of the implications in the premises of applications of Implied are left implicit. Our examples will also include other applications of Implied, usually used to simplify proofs. For example, if we write

$$\{Pre\}\{I_1\}S_1\{I_2'\}\{I_2\}S_2\{I_3\}\cdots\{I_n\}S_n\{safe\_exit(r)\}$$

the addition of $\{I_2'\}$ indicates an application of Implied in the same form as above with leftmost premise $I_2' \Rightarrow I_2$ and conclusion $\{I_2'\}S_2\{I_3\}$.

In our sample programs, we assume a representation of integer lists where the empty list uses one memory location and is just a tag whose value is 0. If the list is non-empty, then three consecutive memory locations are used. The first contains the tag value 1. The second contains an integer, and the third contains a pointer to the rest of the list. We assume there are 32 registers, denoted r_0 to r_{31}. A program that extends a list with one element at the head is given in Figure 2. We assume that register r_0 has an input integer value, that register r_2 contains an input list of integers, and that register r_7 contains a designated return address. We also assume that there is a set of consecutive memory locations (unbounded) that are unallocated, and the first location in this set is given by the value of an *allocation pointer* whose value is stored in r_8. The program allocates new memory and increases the value of the allocation pointer as needed. This program builds a new list by storing values at three consecutive memory locations starting at address r_8, and returning a pointer to the new list in r_1. The first line puts the tag value 1 in register r_3 and the second line stores this value in memory at address r_8. The `ADD` instruction computes the value of the new head of the list by adding 1 to the input value in r_0, and the following line stores this value in the memory location following the tag value. Next, the construction of

MOV $r_3 := 1$	r_3 gets value 1
ST $m(r_8 + 0) := r_3$	store this value in $m(r_8)$
ADD $r_3 := r_0 + r_3$	r_3 gets value $r_0 + 1$
ST $m(r_8 + 1) := r_3$	store this value in $m(r_8 + 1)$
ST $m(r_8 + 2) := r_2$	store list r_2 in $m(r_8 + 2)$
ADDC $r_1 := r_8 + 0$	store r_8's current value in output register r_1
ADDC $r_8 := r_8 + 3$	update allocation pointer r_8 by 3
LD $r_0 := m(r_1 + 1)$	load head of list r_1 into r_0
ADDC $r_6 := r_7 + 0$	move return address into r_6
JMP r_6	return

Figure 2. A Program which Extends a List with a New Value at the Head

the new list is completed by putting a pointer to the input list at the address following the new tag and new list head. Next, in the first ADDC instruction, r_1 is set to point to the new list. In the following line, the allocation pointer is increased by 3. The LD instruction sets the value of r_0 to the value of the new head of the list. Finally, the return address is moved into r_6 and control returns there. The program does some unnecessary computation, but this is done for illustration purposes.

Assuming that the initial designated return address r_7 is safe, proofs of safety for this example are very simple. To complicate things slightly, we add a postcondition $r_8 = r_1 + 3$ to show the relation between the final value of the allocation pointer r_8 and the address of the new output list r_1. Figure 3 contains a proof in the linear form described earlier, where for each line of code the axiom for the instruction at that line is applied. We can modify the proof by adding, for example, $safe\_exit(r_7)$ as I'_6 just above I_6, introducing the new proof obligation

$$safe\_exit(r_7) \Rightarrow (safe\_exit(r_7 + 0) \wedge r_8 + 3 = r_8 + 0 + 3)$$

via the Implied rule. Then $I_1, \ldots, I_5$ become simply $safe\_exit(r_7)$, simplifying the details of the proof above the new I'_6.

In the PCC setting, we want to automate proof construction as much as possible, so in general we will not have arbitrary intermediate conditions such as I'_6. We will, however, have *hints*, which are formulas representing preconditions for particular lines of code that are generated automatically by a certifying compiler [17]. Often, such hints are preconditions of targets of direct jumps and conditional jumps, which represent loop invariants. In this paper, we add "hints" such as I'_6 when convenient for illustration purposes.

3. Memory Safety

We now extend the safety policy to insure that load and store instructions occur only at allowable memory locations. We present this new safety policy as an extension to the Hoare rules of Figure 1, modifying the ld and st rules to add conditions stating that loads can only occur at readable memory locations and stores

$$
\begin{array}{rl}
Pre: & \{safe\_exit(r_7)\} \\
I_1: & \{safe\_exit(r_7+0) \wedge r_8+3 = r_8+0+3\} \\
 & \texttt{MOV } r_3 := 1 \\
I_2: & \{safe\_exit(r_7+0) \wedge r_8+3 = r_8+0+3\} \\
 & \texttt{ST } m(r_8+0) := r_3 \\
I_3: & \{safe\_exit(r_7+0) \wedge r_8+3 = r_8+0+3\} \\
 & \texttt{ADD } r_3 := r_0 + r_3 \\
I_4: & \{safe\_exit(r_7+0) \wedge r_8+3 = r_8+0+3\} \\
 & \texttt{ST } m(r_8+1) := r_3 \\
I_5: & \{safe\_exit(r_7+0) \wedge r_8+3 = r_8+0+3\} \\
 & \texttt{ST } m(r_8+2) := r_2 \\
I_6: & \{safe\_exit(r_7+0) \wedge r_8+3 = r_8+0+3\} \\
 & \texttt{ADDC } r_1 := r_8+0 \\
I_7: & \{safe\_exit(r_7+0) \wedge r_8+3 = r_1+3\} \\
 & \texttt{ADDC } r_8 := r_8+3 \\
I_8: & \{safe\_exit(r_7+0) \wedge r_8 = r_1+3\} \\
 & \texttt{LD } r_0 := m(r_1+1) \\
I_9: & \{safe\_exit(r_7+0) \wedge r_8 = r_1+3\} \\
 & \texttt{ADDC } r_6 := r_7+0 \\
I_{10}: & \{safe\_exit(r_6) \wedge r_8 = r_1+3\}
\end{array}
$$

$$Pre \Rightarrow I_1: \quad safe\_exit(r_7) \Rightarrow (safe\_exit(r_7+0) \wedge r_8+3 = r_8+0+3)$$

Figure 3. Safety Proof for Program in Figure 2

$$\frac{}{\{I[m(r_s+c)/r_d] \wedge readable(r_s+c)\}\ \texttt{LD}\ r_d := m(r_s+c)\ \{I\}}\ \textsf{ld}$$

$$\frac{}{\{I[m[r_d+c \mapsto r_s]/m] \wedge writable(r_d+c)\}\ \texttt{ST}\ m(r_d+c) := r_s\ \{I\}}\ \textsf{st}$$

Figure 4. Hoare-Style Rules for New Safety Policy

can only occur at writable memory locations. (See Figure 4.) The new preconditions are essentially the same as those in Necula and Lee's VCG presentation [17, 19].

To prove programs safe, we need a policy on readable and writable addresses. The allocation pointer is used to specify this policy. In particular, we will say that all addresses beginning with the initial value of r_8 are writable initially. We only handle non-mutable data, so once a location is written and the allocation pointer advanced past it, it will no longer be writable. We will allow all addresses starting at some initial address, denoted $start$, to be read. In addition, we want to be able to read all memory locations containing parts of the input data as well as intermediate data that is created during execution of the program. For example, when the input is a list, we want to read all locations that are part of the list.

For our examples, we define the following policy, which is included in the program preconditions.

$$Policy := r_8 \geq start \wedge (\forall w.w \geq r_8 \Rightarrow writable(w)) \wedge (\forall w.w \geq start \Rightarrow readable(w)).$$

In order to show that all memory locations that the program reads from are indeed readable, we need to include information about the types of the input data in the precondition of the program. Typing judgments depend on the contents of memory and the set of currently allocated locations; in the case of lists, all memory locations used to represent a list must be allocated. An *allocation predicate* is used to specify a particular set of allocated addresses. A typing judgment stating that w has type τ in memory m with allocation set A is written $w :_{m,A} \tau$. In our example, the set of addresses assumed to be allocated before execution starts is defined as $A(w) := (start \leq w < r_8)$. We add the necessary typing judgments to the program precondition, obtaining:

$$Pre := safe\_exit(r_7) \wedge (r_0 :_{m,A} int) \wedge (r_2 :_{m,A} intlist) \wedge Policy. \quad (1)$$

Proving safety now requires showing that various type constraints are met. We present here the typing rules we use in our example. First, we define a valid type to be any type τ for which typing is preserved when unallocated memory locations are modified and also when the set of allocated memory locations is increased. More formally:

Definition 1. A type τ is *valid* if for any word w, memory m, and allocation predicate A, whenever $w :_{m,A} \tau$ holds, then:

1. for any words u and v, if $\neg A(v)$ holds, then $w :_{m[v \mapsto u],A} \tau$ holds; and
2. for any allocation predicate A' such that $\forall x.A(x) \Rightarrow A'(x)$ holds, then $w :_{m,A'} \tau$ holds.

Rules expressing this definition, as well as all remaining typing rules we use in proving safety of the example program are given in Figure 5. Those for lists are stated in terms of lists of arbitrary type τ. In addition, integers and integer lists are assumed to be valid types. These rules can be proven sound with respect to a particular definition of types [2], which we omit for now. We return to this issue when we discuss the formalization of these rules in Section 4.

Figure 6 contains a new safety proof of our example program. This proof is of safety only; we have removed the additional postcondition used in the previous section. In this proof, we have added hint I'_8. Note that the instruction at line 8 loads the head of the output list r_1. The fact that r_1 is a non-empty list of integers is used to prove that memory location $r_1 + 1$ is indeed readable. Note that we obtain I_7 from

$$\frac{w :_{m,A} \tau \qquad valid(\tau) \qquad \neg A(v)}{w :_{m[v \mapsto u],A} \tau}\ \text{valid\_mem\_update}$$

$$\frac{w :_{m,A} \tau \qquad valid(\tau) \qquad \forall x.A(x) \Rightarrow A'(x)}{w :_{m,A'} \tau}\ \text{valid\_allocset}$$

$$\frac{w :_{m,A} int}{w+1 :_{m,A} int}\ \text{int\_succ} \qquad \frac{w :_{m,A} list(\tau) \qquad valid(\tau)}{readable(w)}\ \text{list\_readable}$$

$$\frac{w :_{m,A} list(\tau) \qquad valid(\tau) \qquad m(w) \neq 0}{readable(w+1)}\ \text{list\_readable1}$$

$$\frac{w :_{m,A} list(\tau) \qquad valid(\tau) \qquad m(w) \neq 0}{readable(w+2)}\ \text{list\_readable2}$$

$$\frac{w :_{m,A} list(\tau) \qquad valid(\tau) \qquad m(w) \neq 0}{m(w+1) :_{m,A} \tau}\ \text{list\_field1}$$

$$\frac{w :_{m,A} list(\tau) \qquad valid(\tau) \qquad m(w) \neq 0}{m(w+2) :_{m,A} list(\tau)}\ \text{list\_field2}$$

$$\frac{valid(\tau) \qquad m(w) = 0 \qquad A(w) \qquad readable(w)}{m(w) :_{m,A} list(\tau)}\ \text{empty\_list}$$

$$\frac{\begin{array}{lll} & A(w) & readable(w) \\ m(w+1) :_{m,A} \tau \qquad m(w) = 1 & A(w+1) & readable(w+1) \\ m(w+2) :_{m,A} list(\tau) \qquad valid(\tau) & A(w+2) & readable(w+2) \end{array}}{m(w) :_{m,A} list(\tau)}\ \text{non\_empty\_list}$$

Figure 5. Typing Rules for Integers and Lists

$$
\begin{array}{rl}
Pre: & safe\_exit(r_7) \wedge (r_0 :_{m,A} int) \wedge (r_2 :_{m,A} intlist) \wedge Policy \\
I_1: & \{safe\_exit(r_7) \wedge (r_8+0 :_{m[r_8+0\mapsto 1, r_8+1\mapsto r_0+1, r_8+2\mapsto r_2], A'} intlist) \wedge \\
& \ m[r_8+0\mapsto 1, r_8+1\mapsto r_0+1, r_8+2\mapsto r_2](r_8+0) \neq 0 \wedge \\
& \ writable(r_8+2) \wedge writable(r_8+1) \wedge writable(r_8+0)\} \\
& \texttt{MOV}\ r_3 := 1 \\
I_2: & \{safe\_exit(r_7) \wedge (r_8+0 :_{m[r_8+0\mapsto r_3, r_8+1\mapsto r_0+r_3, r_8+2\mapsto r_2], A'} intlist) \wedge \\
& \ m[r_8+0\mapsto r_3, r_8+1\mapsto r_0+r_3, r_8+2\mapsto r_2](r_8+0) \neq 0 \wedge \\
& \ writable(r_8+2) \wedge writable(r_8+1) \wedge writable(r_8+0)\} \\
& \texttt{ST}\ m(r_8+0) := r_3 \\
I_3: & \{safe\_exit(r_7) \wedge (r_8+0 :_{m[r_8+1\mapsto r_0+r_3, r_8+2\mapsto r_2], A'} intlist) \wedge \\
& \ m[r_8+1\mapsto r_0+r_3, r_8+2\mapsto r_2](r_8+0) \neq 0 \wedge \\
& \ writable(r_8+2) \wedge writable(r_8+1)\} \\
& \texttt{ADD}\ r_3 := r_0 + r_3 \\
I_4: & \{safe\_exit(r_7) \wedge (r_8+0 :_{m[r_8+1\mapsto r_3, r_8+2\mapsto r_2], A'} intlist) \wedge \\
& \ m[r_8+1\mapsto r_3, r_8+2\mapsto r_2](r_8+0) \neq 0 \wedge \\
& \ writable(r_8+2) \wedge writable(r_8+1)\} \\
& \texttt{ST}\ m(r_8+1) := r_3 \\
I_5: & \{safe\_exit(r_7) \wedge (r_8+0 :_{m[r_8+2\mapsto r_2], A'} intlist) \wedge m[r_8+2\mapsto r_2](r_8+0) \neq 0 \wedge \\
& \ writable(r_8+2)\} \\
& \texttt{ST}\ m(r_8+2) := r_2 \\
I_6: & \{safe\_exit(r_7) \wedge (r_8+0 :_{m,A'} intlist) \wedge m(r_8+0) \neq 0\} \\
& \texttt{ADDC}\ r_1 := r_8 + 0 \\
I_7: & \{safe\_exit(r_7) \wedge (r_1 :_{m,A'} intlist) \wedge m(r_1) \neq 0\} \\
& \texttt{ADDC}\ r_8 := r_8 + 3 \\
I'_8: & \{safe\_exit(r_7) \wedge (r_1 :_{m,A} intlist) \wedge m(r_1) \neq 0\} \\
I_8: & \{safe\_exit(r_7+0) \wedge readable(r_1+1)\} \\
& \texttt{LD}\ r_0 := m(r_1+1) \\
I_9: & \{safe\_exit(r_7+0)\} \\
& \texttt{ADDC}\ r_6 := r_7 + 0 \\
I_{10}: & \{safe\_exit(r_6)\}
\end{array}
$$

Proof Obligations from the Implied rule: $Pre \Rightarrow I_1$ $I'_8 \Rightarrow I_8$

Figure 6. Safety Proof for Program in Figure 2 under Extended Safety Policy

I'_8 by replacing all occurrences of r_8 by $r_8 + 3$, which changes the definition of the allocation set. We abbreviate the new set as A' where $A'(w) := (start \leq w < r_8 + 3)$. We can see that although the same set of rules are applied as in the proof that used the simpler safety policy, the formulas are considerably more complex.

The last line of the figure shows the remaining two proof obligations. Their proofs, which we consider now, will complete the safety proof. First consider $Pre \Rightarrow I_1$, which expands to:

$$\begin{array}{c}[safe\_exit(r_7) \wedge (r_0 :_{m,A} int) \wedge (r_2 :_{m,A} intlist) \wedge r_8 \geq start \wedge \\ (\forall w.w \geq r_8 \Rightarrow writable(w)) \wedge (\forall w.w \geq start \Rightarrow readable(w))] \Rightarrow \\ [safe\_exit(r_7) \wedge (r_8 + 0 :_{m[r_8+0\mapsto 1, r_8+1\mapsto r_0+1, r_8+2\mapsto r_2],A'} intlist) \wedge \\ m[r_8 + 0 \mapsto 1, r_8 + 1 \mapsto r_0 + 1, r_8 + 2 \mapsto r_2](r_8 + 0) \neq 0 \wedge \\ writable(r_8 + 2) \wedge writable(r_8 + 1) \wedge writable(r_8 + 0)].\end{array}$$

The $writable$ subformulas clearly follow from the policy on writable addresses. The subformula above the $writable$ subformulas reduces to $1 \neq 0$. The typing judgment is the only remaining non-trivial subformula. Note that r_8, $r_8 + 1$, and $r_8 + 2$ are not allocated according to the allocation predicate A, but are allocated according to A'. From $(r_2 :_{m,A} intlist)$, we can apply valid_mem_update three times to obtain: $(r_2 :_{m[r_8+0\mapsto 1, r_8+1\mapsto r_0+1, r_8+2\mapsto r_2],A} intlist)$. Now, applying valid_allocset, we conclude:

$$(r_2 :_{m[r_8+0\mapsto 1, r_8+1\mapsto r_0+1, r_8+2\mapsto r_2],A'} intlist). \tag{2}$$

Let m' denote the memory $m[r_8 + 0 \mapsto 1, r_8 + 1 \mapsto r_0 + 1, r_8 + 2 \mapsto r_2]$. Formula (2) is equivalent to:

$$(m'(r_8 + 2) :_{m',A'} intlist). \tag{3}$$

By similar reasoning, we can conclude $(r_0 :_{m[r_8+0\mapsto 1, r_8+1\mapsto r_0+1, r_8+2\mapsto r_2],A'} int)$, and then by int_succ, we get $(r_0 + 1 :_{m[r_8+0\mapsto 1, r_8+1\mapsto r_0+1, r_8+2\mapsto r_2],A'} int)$, which is equivalent to:

$$(m'(r_8 + 1) :_{m',A'} int). \tag{4}$$

By definition of memory update, we know that

$$m'(r_8 + 0) = 1 \tag{5}$$

Using (3), (4), and (5) we can now apply non_empty_list. As already noted the addresses r_8, $r_8 + 1$, and $r_8 + 2$ are in the allocated set A', so the three allocation premises of this rule hold. The $readable$ premises are provable from the safety policy.

The remaining proof obligation in Figure 6, $I'_8 \Rightarrow I_8$, expands to:

$$[safe\_exit(r_7) \wedge (r_1 :_{m,A} intlist) \wedge m(r_1) \neq 0] \Rightarrow [safe\_exit(r_7 + 0) \wedge readable(r_1 + 1)].$$

The proof is simple. The $readable$ subgoal follows directly from an application of list_readable1. This completes the safety proof.

A general advantage of the PCC approach to software safety is that the trusted computing base (TCB) is small. In a PCC system that implements the safety policy and set of rules discussed so far, we must trust the set of inference rules for basic logic, the rules of Figure 1 which describe the machine semantics,

and the rules of Figure 5 for types. If the implementation uses a logical framework such as LF [11] as many PCC systems do, including the early ones [17, 2], then we must trust the encoding of such rules in the framework. Finally, we must trust the implementation of the proof checker for the logic or logical framework. When we turn to FPCC in the next sections, we will simplify the TCB first by replacing typing rules with formally proved typing lemmas, and second by replacing the Hoare rules with a lower-level encoding of machine semantics.

4. Typing Definitions and Lemmas

We now turn to FPCC, and in this section, we begin by giving a foundational approach to typing, which means that we give definitions for types and formally prove the typing rules of Figure 5 as lemmas. At the same time, we describe our formalization of FPCC in Coq. We use a modified Coq syntax in presenting the definitions. In the rest of the paper, when we discuss formal proofs of safety of example programs, these proofs will use the formal versions of the typing lemmas discussed here.

We use a direct Coq formalization where we give Coq types to represent data such as words and memories, and we use Coq's logic to represent logical expressions such as allocation predicates, preconditions, postconditions, and invariants. $Word$ is defined to be the set of natural numbers. For simplicity, we do not build in fixed-size words, though this can and has been done in various PCC systems (for example [13]). We write Mem to represent the function type $(Word \rightarrow Word)$. In particular, memory is modeled as a function from machine addresses to machine values, usually written with uppercase M. An allocation predicate, usually denoted A, is any Coq predicate of type $Word \rightarrow Prop$. (Coq's $Prop$ denotes the type of logical formulas.)

The definitions and lemmas from which the rules in Figure 5 follow were mostly presented in [2]. We repeat the definitions in Figure 7 to highlight the fact that all types are defined objects. A type is defined to be a predicate of three arguments: an allocation predicate, a memory, and a value. The second definition illustrates that while $(w :_{M,A} \tau)$ is the typing abbreviation, τ is the predicate, and the other three parameters (A, M, and w) to the typing judgment are in fact arguments to τ. The int type contains all machine integers, which includes all elements of $Word$. The definitions starting with con and ending with $record_3$ all define type constructors providing various ways to build new types from existing types. Generalizing from $record_1$, $record_2$, and $record_3$, it is easy to see how to build records of any size. Next, subtyping is defined using logical implication, followed by the definition of the rec operator. Using this operator, recursive types are defined to be all types $(rec\ F)$ for which the least fixed-point of the argument F is $(rec\ F)$, i.e., all types with the property that for all machine states (A, M, w), $(rec\ F)(A, M, w) \Leftrightarrow (F\ (rec\ F))(A, M, w)$. The last two definitions illustrate its use in defining the list type constructor.

The typing rules in Figure 5 are proven fairly directly from these definitions. We omit the proofs here. More specifically, we prove a series of lemmas about each of the type constructors in Figure 7. For each property expressed and proved for all of the type constructors, we can use the resulting set of lemmas directly to prove automatically that the corresponding property holds of any type built up from these constructors. For example, the types library includes a set of lemmas stating that the type constructors preserve validity. For instance, if types τ_1 and τ_2 are valid, then $(record_2\ \tau_1\ \tau_2)$ is valid. These lemmas can be used to prove that any type built from the type constructors is valid. The class of types that can be built from these constructors includes a large class of covariant recursive types.

$$
\begin{array}{rcl}
Ty & := & (Word \to Prop) \to Mem \to Word \to Prop \\
w :_{M,A} \tau & := & \tau(A, M, w) \\
int(A, M, w) & := & True \\
(con\ c)(A, M, w) & := & c = w \\
(ref\ \tau)(A, M, w) & := & readable(w) \wedge A(w) \wedge \tau(A, M, M(w)) \\
(offset\ n\ \tau)(A, M, w) & := & \tau(A, M, w + n) \\
(\tau \cup \tau')(A, M, w) & := & \tau(A, M, w) \vee \tau'(A, M, w) \\
(\tau \cap \tau')(A, M, w) & := & \tau(A, M, w) \wedge \tau'(A, M, w) \\
(field\ i\ \tau)(A, M, w) & := & (offset\ i\ (ref\ \tau))(A, M, w) \\
(record_1\ \tau)(A, M, w) & := & (field\ 0\ \tau)(A, M, w) \\
(record_2\ \tau_0\ \tau_1)(A, M, w) & := & ((field\ 0\ \tau_0) \cap (field\ 1\ \tau_1))(A, M, w) \\
(record_3\ \tau_0\ \tau_1\ \tau_2)(A, M, w) & := & ((field\ 0\ \tau_0) \cap (field\ 1\ \tau_1) \cap (field\ 2\ \tau_2))(A, M, w) \\
(\tau \sqsubseteq \tau') & := & \forall A, M, w. \tau(A, M, w) \Rightarrow \tau'(A, M, w) \\
(rec\ F)(A, M, w) & := & \forall \tau. valid(\tau) \Rightarrow (F(\tau) \sqsubseteq \tau) \Rightarrow \tau(A, M, w) \\
(listcon\ \tau)\ \tau' & := & (record_1\ (con\ 0)) \cup (record_3\ (con\ 1)\ \tau\ \tau') \\
(list(\tau))(A, M, w) & := & rec\ (listcon\ \tau)(A, M, w)
\end{array}
$$

Figure 7. Definitions for Basic Types and Type Constructors

5. Encoding Machine Semantics Directly

We now return to the example program in Figure 2 and discuss its safety proof in the FPCC setting. We again proceed in two steps, starting with the simple safety policy in this section, and adding memory safety in the next section. As a first step, in this section we provide a more foundational specification of machine semantics by starting with a low-level machine model. At the same time, we continue describing our formalization of FPCC in Coq. At a high level, this and the next section can be viewed as a formalization of the Hoare logic in Sections 2 and 3, though we are not formalizing the rules directly. We start at a more primitive level, formalizing the notion of state, which in Hoare logic is left implicit. (State is of course important, however, when proving properties of Hoare logic such as its soundness.) In addition, in this version we include jump instructions.

We define the type Reg to be the type of the set of 32 registers r_0 to r_{31}. We define machine instructions as an inductive type $Instr$; instructions and their types are given in Figure 8. The figure includes a jump instruction, jmp, and two conditional branch instructions, bgt and beq, that we have not yet seen.

Register banks are represented as functions from registers to values; $RegFile$ denotes the type $(Reg \to Word)$. We define a machine state to be a triple of the form (R, M, pc) where R is a register bank, M is a memory, and pc is a $Word$ representing the program counter. In particular $State$ denotes the type $RegFile \times Mem \times Word$. We define a step relation that relates two machine states, one before execution and one after execution of a particular instruction. We write $(R, M, pc \mapsto R', M', pc')$ to denote this relation, and $(R, M, pc \mapsto^* R', M', pc')$ to denote its reflexive transitive closure.

$$
\begin{array}{rcl}
mov & : & Reg \rightarrow Word \rightarrow Instr \\
addc & : & Reg \rightarrow Reg \rightarrow Word \rightarrow Instr \\
add & : & Reg \rightarrow Reg \rightarrow Reg \rightarrow Instr \\
ld & : & Reg \rightarrow Reg \rightarrow Word \rightarrow Instr \\
st & : & Reg \rightarrow Word \rightarrow Reg \rightarrow Instr \\
jmp & : & Reg \rightarrow Instr \\
bgt & : & Reg \rightarrow Reg \rightarrow Word \rightarrow Instr \\
beq & : & Reg \rightarrow Reg \rightarrow Word \rightarrow Instr \\
ill & : & Instr
\end{array}
$$

Figure 8. Constants Representing Machine Instructions and their Types

Machine instructions are encoded as 32-bit machine integers. These integers are decoded into machine instructions by extracting information from specific bits. The step relation is defined by extracting the instruction at line pc in M, decoding it, and changing the machine state according to the semantics of the particular instruction. The decode operation is written Dc and has type $Word \rightarrow Instr$. We omit the details of its definition. We also define two update operations, one on registers and one on memory. In particular, we write $R[d \mapsto v]$, where R is a register bank, d is a register, and v is a word, to represent the function which maps d to v and all other registers r to $(R\ r)$. We often write R_i to abbreviate register bank application $(R\ r_i)$. Similarly, we write $M[a \mapsto v]$, where M is a memory, a is a word representing an address, and v is a word, to denote a new function which is the same as M except that the new function maps address a to value v. These update relations and the step relation are defined in Figure 9. Their definitions are essentially the same as those found in our earlier work [2] and in the Coq formalization of syntactic FPCC [10]. Note that the update relations are Coq functions, while the step relation is a Coq predicate, i.e., $(R, M, pc \mapsto R', M', pc')$ has Coq type $Prop$. Note that the `ILLEGAL` instruction maps to the false proposition. Thus, any machine state where the program counter points to an `ILLEGAL` instruction will not be related to any other state by the step relation.

Following Michael and Appel [13], we define $safe$, $Progress$, and $Preservation$ predicates as in Figure 10, and prove the safety rule, also in the figure. The $safe$ predicate expresses the fact that execution of a safe program doesn't get stuck. With the simple safety policy, the only way to get stuck is by executing an `ILLEGAL` instruction. Note that $safe$ is a predicate on a machine state (R, M, pc). The code is in M and pc points to the first instruction. In the definitions of $Progress$ and $Preservation$, Inv is another predicate which takes a machine state as an argument. Our formalization also includes a definition for $safe\_exit$.

Returning to the example program, we encode it in Coq using a predicate $listextend(M)$ stating which addresses in memory M contain the instructions, and what instructions are in these memory locations. For simplicity, we assume that the 10-line program in Figure 2 is in addresses 1–10 of memory, where address 10 contains the jump to the return location. In particular, the definition of $listextend(M)$

$$
\begin{array}{rl}
(R[d \mapsto v]\ r) := & \mathit{if}\ r = d\ \mathit{then}\ v\ \mathit{else}\ (R\ r) \\
(M[a \mapsto v]\ w) := & \mathit{if}\ w = a\ \mathit{then}\ v\ \mathit{else}\ (M\ w) \\
(R, M, pc \mapsto R', M', pc') := & \mathit{match}\ (Dc\ (M\ pc))\ \mathit{with} \\
(mov\ r_d\ w) & \Rightarrow R' = R[r_d \mapsto w] \wedge M' = M \wedge pc' = pc + 1 \\
(addc\ r_d\ r_s\ w) & \Rightarrow R' = R[r_d \mapsto R_s + w] \wedge M' = M \wedge pc' = pc + 1 \\
(add\ r_d\ r_{s_1}\ r_{s_2}) & \Rightarrow R' = R[r_d \mapsto R_{s_1} + R_{s_2}] \wedge M' = M \wedge pc' = pc + 1 \\
(ld\ r_d\ r_s\ w) & \Rightarrow R' = R[r_d \mapsto (M(R_s + w))] \wedge M' = M \wedge pc' = pc + 1 \\
(st\ r_d\ w\ r_s) & \Rightarrow R' = R \wedge M' = M[R_d + w \mapsto R_s] \wedge pc' = pc + 1 \\
(jmp\ r) & \Rightarrow R' = R \wedge M' = M \wedge pc' = (R\ r) \\
(bgt\ r_{s_1}\ r_{s_2}\ w) & \Rightarrow R' = R \wedge M' = M \wedge pc' = (\mathit{if}\ R_{s_1} > R_{s_2}\ \mathit{then}\ w\ \mathit{else}\ pc + 1) \\
(beq\ r_{s_1}\ r_{s_2}\ w) & \Rightarrow R' = R \wedge M' = M \wedge pc' = (\mathit{if}\ R_{s_1} = R_{s_2}\ \mathit{then}\ w\ \mathit{else}\ pc + 1) \\
\mathit{ill} & \Rightarrow \mathit{False}
\end{array}
$$

Figure 9. The Step Relation

$$
\begin{array}{rcl}
safe(R, M, pc) & := & \forall R', M', pc'.[(R, M, pc \mapsto^* R', M', pc') \Rightarrow \\
 & & \quad \exists R'', M'', pc''.(R', M', pc' \mapsto R'', M'', pc'')] \\
Progress(Inv) & := & \forall R, M, pc.[Inv(R, M, pc) \Rightarrow \\
 & & \quad \exists R', M', pc'.(R, M, pc \mapsto R', M', pc')] \\
Preservation(Inv) & := & \forall R, M, pc, R', M', pc'.Inv(R, M, pc) \Rightarrow \\
 & & \quad (R, M, pc \mapsto R', M', pc') \Rightarrow Inv(R', M', pc')
\end{array}
$$

$$
\frac{Inv(R, M, pc) \qquad Progress(Inv) \qquad Preservation(Inv)}{safe(R, M, pc)}\ \mathsf{safety}
$$

$$
safe\_exit(w) := \forall R, M, pc.(pc = w \Rightarrow safe(R, M, pc))
$$

Figure 10. Definitions and Lemmas for Proving Safety

has the form:

$$\begin{aligned} listextend(M) := \quad & (Dc\ (M\ 1)) = (mov\ r_3\ 1) \wedge \\ & (Dc\ (M\ 2)) = (st\ r_8\ 0\ r_3) \wedge \\ & \quad \vdots \\ & (Dc\ (M\ 9)) = (addc\ r_6\ r_7\ 0) \wedge \\ & (Dc\ (M\ 10)) = (jmp\ r_6). \end{aligned}$$

Since we model memory as a single function which does not differentiate between code and data locations, we have the additional proof obligation of showing that our program does not include self-modifying code. If there was overlap between the code and data parts of memory, we would likely not be able to prove safety. In the Hoare rules of Section 2, there is an implicit separation of code from data in memory because there is no connection between the statement part of judgments and the memory. In the Coq formalization, we add a parameter $start$ and a hypothesis that states that $start$ is greater than the address where the last line of the program occurs. The formalization is valid for any value of $start$ that satisfies this assumption. The safety proof requires showing that all store instructions executed by the program will occur at addresses greater than or equal to $start$. In particular, the precondition assumes the initial value of allocation pointer r_8 is not smaller than $start$, and we show that this condition is an invariant of the program.

We started in this section with a fairly low-level encoding of the machine semantics and built up to with the high-level derived rule safety. Reasoning using this rule corresponds closely to reasoning using the Hoare-style rules of Section 2. In the new setting, we prove a formula of the form

$$\forall R \forall M \forall pc\ (Pre(R, M, pc) \Rightarrow safe(R, M, pc))$$

where Pre is now a predicate over a state. For our example program, this predicate will include the precondition used in the proof in Section 2, but also includes the initial value of the program counter, the starting assumption about r_8, as well as the formula stating where the program lies in memory:

$$Pre(R, M, pc) := safe\_exit(R_7) \wedge pc = 1 \wedge R_8 \geq start \wedge listextend(M).$$

To use the safety rule, we need a predicate Inv which expresses a program invariant. Inv will have one clause for every line of the program stating what is true at the point when that line is executed. For our example program, we obtain these formulas fairly directly from the Hoare proof in Figure 3. Here, for line 6 we will use I'_6 in the simplified proof discussed in Section 2 instead of I_6 in the figure, as well as the simpler versions of $I_1, \ldots, I_5$. Here, these formulas become predicates over a state. In this example, we only need the register bank argument from the state, so we omit the others. We also add the invariant $R_8 \geq start$. These formulas are shown in Figure 11. The full predicate Inv has the following form:[1]

$$\begin{aligned} Inv(R, M, pc) := \quad & [listextend(M) \wedge \\ & ((pc = 1 \wedge I_1(R)) \vee \cdots \vee (pc = 10 \wedge I_{10}(R)))] \vee \\ & safe(R, M, pc) \end{aligned}$$

[1] We have added the condition $R_8 \geq start$ to each clause of the invariant, but note that we could have instead put it outside the disjunction.

$$
\begin{aligned}
I_1(R) = \cdots = I_6(R) &:= safe\_exit(R_7) \wedge R_8 \geq start \\
I_7(R) &:= safe\_exit(R_7 + 0) \wedge R_8 + 3 = R_1 + 3 \wedge R_8 \geq start \\
I_8(R) = I_9(R) &:= safe\_exit(R_7 + 0) \wedge R_8 = R_1 + 3 \wedge R_8 \geq start \\
I_{10}(R) &:= safe\_exit(R_6) \wedge R_8 = R_1 + 3 \wedge R_8 \geq start
\end{aligned}
$$

Figure 11. Clauses of the Invariant for Safety Proof of the Program in Figure 2

As we will see shortly, the second clause of Inv's top-level disjunction, $safe(R, M, pc)$, is used for the case when the program counter has the value of the return address.

Although we must provide Inv in full in order to apply the safety rule, most of it can be computed automatically. In this example, we can view I'_6 and the postcondition from Section 2 as the input used to compute all the formulas in Figure 11. I_6 and I_{10}, respectively, are taken from these inputs. The remaining clauses in Figure 11 are computed directly using the Hoare rules of Figure 1, even though these rules are not part of the new encoding. We can view this as using a kind of a VCG, but one that does not have to be trusted, because if the preconditions are computed incorrectly, we will not be able to prove the premises of the safety rule. We do, however, treat the new clause $R_8 \geq start$ specially. It is an invariant that remains unchanged even as the value stored in r_8 changes. In particular, in this example, when computing $I_7(R)$ from $I_8(R)$ using the addc rule, notice that R_8 in the second conjunct of $I_8(R)$ is replaced by $R_8 + 3$ in $I_7(R)$ as the rule says it should, but R_8 in the third conjunct remains unchanged.

We now have all the ingredients we need to prove the three premises of the safety rule. Proving the first premise involves showing that the invariant holds in the initial state, i.e., the invariant follows from the precondition where $pc = 1$. The proof is simple, since from the precondition, it follows immediately that $listextend(M)$ and that $pc = 1 \wedge I_1(R)$.

Both the progress and preservation proofs have the invariant as a hypothesis, which is used to break the proofs into cases. In the case when the second disjunct holds, $safe(R, M, pc)$, both progress and preservation follow directly from the definition of $safe$. When the first disjunct holds, we know $listextend(M)$, and the proof proceeds by cases on the inner disjuncts, which give a specific value for the program counter and an assumption corresponding to the precondition of the corresponding line of code. For progress, we must show that no matter which line we are at in the program, there is a next step. This is immediate for all programs that do not use the `ILLEGAL` instruction.

Proving $Preservation(Inv)$ is where Hoare-style reasoning takes place. We have a case for each line of the program; for $pc = 10$, it is straightforward to prove $safe(R', M', pc')$ from the hypothesis $safe\_exit(R_6)$; for $pc = 1, \ldots, 9$, under the assumption that $I_{pc}(R)$ and $(R, M, pc \mapsto R', M', pc')$ hold, we show that $I_{pc'}(R')$ holds. For the cases where we calculated I_{pc} from $I_{pc'}$ by a straightforward application of one of the Hoare-style axioms, the proof is immediate. The step relation encodes the same information as the corresponding Hoare rule, so all the work was done when we applied the rule by hand to determine the right I_{pc} to include in Inv. In addition, we must show that each step does not modify the code. This is only relevant for store instructions, since they are the only ones that change the contents of memory. For example, consider the following subgoal of the preservation proof:

$$(R, M, pc \mapsto R', M', pc'), listextend(M), pc = 4, I_4(R) \vdash Inv(R', M', pc').$$

From the step relation and the clause for line 4 in the definition of $listextend(M)$, which contains a

store instruction, we can deduce: $R' = R$, $M' = M[R_8 + 1 \mapsto R_3]$, and $pc' = 5$. Thus, we must show $Inv(R, M[R_8{+}1 \mapsto R_3], 5)$. By definition of Inv, this reduces to showing that $listextend(M[R_8{+}1 \mapsto R_3])$ and that $I_5(R)$ holds. Since $I_4(R)$ and $I_5(R)$ are the same formula, the latter is immediate. For the former, note that the new memory has a new value at a location greater than R_8, which is greater or equal to $start$. From this fact, it follows fairly directly that the code is not modified.

Consider another example subgoal:

$$(R, M, pc \mapsto R', M', pc'), listextend(M), pc = 9, I_9(R) \vdash Inv(R', M', pc').$$

Line 9 is an `ADDC` instruction, which gives $R' = R[r_6 \mapsto R_7 + 0]$, $M' = M$, and $pc' = 10$. In this case, we must show $I_{10}(R[r_6 \mapsto R_7 + 0])$ from $I_9(R)$. Expanding definitions, we must show:

$$\begin{array}{l} safe\_exit(R_7 + 0) \wedge R_8 = R_1 + 3 \wedge R_8 \geq start \vdash \\ \quad safe\_exit(R[r_6 \mapsto R_7 + 0]\ r_6) \wedge (R[r_6 \mapsto R_7 + 0]\ r_8) = (R[r_6 \mapsto R_7 + 0]\ r_1) + 3 \wedge \\ \quad\quad (R[r_6 \mapsto R_7 + 0]\ r_8) \geq start. \end{array}$$

This follows directly from the definition of register update since $(R[r_6 \mapsto R_7 + 0]\ r_6)$ is $R_7 + 0$, $(R[r_6 \mapsto R_7 + 0]\ r_1)$ is R_1, and $(R[r_6 \mapsto R_7 + 0]\ r_8)$ is R_8.

More reasoning is needed for the cases when I_{pc} comes from a hint, and this reasoning corresponds to what is needed to prove the corresponding Implied rule in the Hoare version of the proof. Consider the subgoal:

$$(R, M, pc \mapsto R', M', pc'), listextend(M), pc = 6, I_6(R) \vdash Inv(R', M', pc').$$

Line 6 is another `ADDC` instruction, which gives $R' = R[r_1 \mapsto R_8 + 0]$, $M' = M$, and $pc' = 7$. In this case, we must show $I_7(R[r_1 \mapsto R_8 + 0])$ from $I_6(R)$. Expanding definitions, we must show:

$$\begin{array}{l} safe\_exit(R_7) \wedge R_8 \geq start \vdash \\ \quad safe\_exit((R[r_1 \mapsto R_8 + 0]\ r_7) + 0) \wedge (R[r_1 \mapsto R_8 + 0]\ r_8) + 3 = (R[r_1 \mapsto R_8 + 0]\ r_1) + 3 \wedge \\ \quad\quad (R[r_1 \mapsto R_8 + 0]\ r_8) \geq start. \end{array}$$

Expanding the register update definitions, this reduces to showing:

$$safe\_exit(R_7) \wedge R_8 \geq start \vdash safe\_exit(R_7 + 0) \wedge R_8 + 3 = R_8 + 0 + 3 \wedge R_8 \geq start.$$

The proof requires a few more steps than the others, though it is still simple because of the simple safety policy. We will see in the next section that extending the safety policy complicates such proofs considerably.

Finally, we consider the subgoal:

$$(R, M, pc \mapsto R', M', pc'), listextend(M), pc = 7, I_7(R) \vdash Inv(R', M', pc').$$

In this case, we must show $I_8(R[r_8 \mapsto R_8 + 3])$ from $I_7(R)$. Line 7 modifies the value of allocation pointer r_8, but when applying the Hoare rule for `ADDC` to obtain $I_7(R)$ from $I_8(R)$, we did not modify R_8

$$
\begin{array}{l}
(R, M, pc \mapsto R', M', pc') := match\ (Dc\ (M\ pc))\ with \\
\quad \vdots \\
\quad (ld\ r_d\ r_s\ w) \Rightarrow readable(R_s + w) \wedge R' = R[r_d \mapsto (M(R_s + w))] \wedge M' = M \wedge pc' = pc + 1 \\
\quad (st\ r_d\ w\ r_s) \Rightarrow writable(R_d + w) \wedge R' = R \wedge M' = M[R_d + w \mapsto R_s] \wedge pc' = pc + 1 \\
\quad \vdots
\end{array}
$$

Figure 12. Building Memory Safety into the Step Relation

in the subformula $R_8 \geq start$, although we did modify it in the rest of formula. Expanding definitions in the above goal, we must show:

$$
\begin{array}{l}
safe\_exit(R_7 + 0) \wedge R_8 + 3 = R_1 + 3 \wedge R_8 \geq start \vdash \\
\quad safe\_exit((R[r_8 \mapsto R_8 + 3]\ r_7) + 0) \wedge (R[r_8 \mapsto R_8 + 3]\ r_8) = (R[r_8 \mapsto R_8 + 3]\ r_1) + 3 \wedge \\
\qquad ([r_8 \mapsto R_8 + 3]\ r_8) \geq start.
\end{array}
$$

Expanding the register update definitions, this reduces to showing:

$$
\begin{array}{l}
safe\_exit(R_7 + 0) \wedge R_8 + 3 = R_1 + 3 \wedge R_8 \geq start \vdash \\
\quad safe\_exit(R_7 + 0) \wedge R_8 + 3 = R_1 + 3 \wedge R_8 + 3 \geq start.
\end{array}
$$

The third conjunct is provable using some simple reasoning about arithmetic. In general, all such subgoals will be provable as long as the allocation pointer is only increased and never decreased.

6. Integrating Memory Safety into Proofs of Progress and Preservation

Continuing the discussion of FPCC, we now take the second step, adding memory safety to our safety policy. This requires, first of all, integrating memory safety into the machine model. In addition, recall that type information was important in proving safety of the example in Section 3. The proofs here will use the formalization discussed in Section 4. Again, we continue in the context of our Coq formalization.

To integrate memory safety into our low-level machine model, we build the constraints on memory accesses directly into the step relation. At this stage, $readable$ and $writable$ are introduced into the formalization with no definitions. Their meaning is given later when specifying the read/write policy for a particular program. Only the clauses for the `LD` and `ST` instructions change. Their new definitions are shown in Figure 12. Note that the presence of the `ILLEGAL` instruction is no longer the only way for computation to get stuck. Any machine state such that the program counter points to a `LD` instruction which attempts to read from an unreadable location, or a `ST` instruction which attempts to write to an unwritable location, will not be related to any other state by the step relation.

The definitions of $safe$, $safe\_exit$, $Progress$, and $Preservation$ as well as the statements and proof of the safety rule in Section 5 are not affected by the new safety policy. When proving safety for individual programs, the new proof obligations to show that `LD` and `ST` instructions use only readable or writable locations appear as part of the proof of the $Progress$ premise of the safety rule.

We now formalize the proof in Figure 6. In the new setting, the precondition of the example program must include the clauses capturing the typing judgments and safety policy expressed in the precondition of the Hoare proof in Section 3. It must now also include the assumptions about the initial value of the program counter and the location and contents of the code in memory, needed for showing that the code is never modified during program execution. We write the definition of the initial allocation predicate as $A_R(w) := (start \leq w < R_8)$ where argument R is written as a subscript to A, showing explicitly its dependence on the contents of the register bank. Using this definition, the precondition, which in the previous proof is given as formula (1) is now defined formally below.

$$\begin{aligned} Pre(R, M, pc) \quad := \quad & safe\_exit(R_7) \wedge (R_0 :_{M,A_R} int) \wedge (R_2 :_{M,A_R} intlist) \wedge R_8 \geq start \wedge \\ & \forall w.(w \geq R_8 \Rightarrow writable(w)) \wedge \forall w.(w \geq start \Rightarrow readable(w)) \wedge \\ & pc = 1 \wedge listextend(M). \end{aligned}$$

Note that this formal version of the read/write policy shows its dependence on argument R (because of the value R_8).

As before, we compute the clauses of Inv, this time starting with hint I'_8 and postcondition I_{10} in Figure 6 which give us clauses $I_8(R, M)$ and $I_{10}(R, M)$ in Figure 13. Now, these clauses depend both on the register bank and memory. As before, we add $R_8 \geq start$. Also, as before, we use the Hoare-style rules to obtain the remaining clauses of Inv. Figure 13 contains the complete set of clauses. In these clauses, we write A'_R for the allocation predicate defined as $A'_R(w) := (start \leq w < R_8 + 3)$. We again treat $R_8 \geq start$ specially, leaving it unchanged in all invariant clauses, even though r_8 changes. Note that line 7 is the only line that modifies r_8's value. Thus in computing $I_7(R, M)$ from $I_8(R, M)$, we leave the subformula $R_8 \geq start$ unchanged, though we do replace R_8 with $R_8 + 3$ in the rest of the formula. (In particular, this replacement only occurs in one place — when A_R becomes A'_R.) Although much of the information in this figure is the same as in Figure 6, we repeat it here because this is the version of the proof that has been formalized in Coq.

The full predicate Inv has the same form as in the proof discussed in Section 5:

$$\begin{aligned} Inv(R, M, pc) := \quad & [listextend(M) \wedge \\ & ((pc = 1 \wedge I_1(R, M)) \vee \cdots \vee (pc = 10 \wedge I_{10}(R, M)))] \vee \\ & safe(R, M, pc). \end{aligned}$$

The proof also is similar to proofs that have already been presented. In particular, in Section 3, we showed the Hoare-style proof for this program under the extended safety policy, and in Section 5, we discussed how the preservation theorem corresponds to Hoare-style reasoning. Here, we show two subgoals, illustrating the $readable$ and $writable$ subgoals, which illustrate that with the new policy, some of the Hoare-style reasoning is now carried out within the progress subproof. Line 5 contains one of the ST instructions. For this case, we have the following subgoal in the progress subproof:

$$listextend(M), pc = 5, I_5(R, M) \vdash \exists R', M', pc'.(R, M, 5 \mapsto R', M', pc').$$

Expanding definitions, we must show:

$$\begin{aligned} & listextend(M), pc = 5, safe\_exit(R_7) \wedge R_8 \geq start \wedge (R_8 + 0 :_{M[R_8+2 \mapsto R_2], A'_R} intlist) \wedge \\ & \quad M[R_8 + 2 \mapsto R_2](R_8 + 0) \neq 0 \wedge writable(R_8 + 2) \vdash \\ & \quad \exists R', M', pc'.(writable(R_8 + 2) \wedge R' = R \wedge M' = M[R_8 + 2 \mapsto R_2] \wedge pc' = 6). \end{aligned}$$

$$
\begin{aligned}
I_1(R,M) &:= safe\_exit(R_7) \wedge R_8 \geq start \wedge \\
&\quad (R_8 + 0 :_{M[R_8+0\mapsto 1, R_8+1\mapsto R_0+1, R_8+2\mapsto R_2], A'_R} intlist) \wedge \\
&\quad M[R_8 + 0 \mapsto 1, R_8 + 1 \mapsto R_0 + 1, R_8 + 2 \mapsto R_2](R_8 + 0) \neq 0 \wedge \\
&\quad writable(R_8 + 2) \wedge writable(R_8 + 1) \wedge writable(R_8 + 0) \\
I_2(R,M) &:= safe\_exit(R_7) \wedge R_8 \geq start \wedge \\
&\quad (R_8 + 0 :_{M[R_8+0\mapsto R_3, R_8+1\mapsto R_0+R_3, R_8+2\mapsto R_2], A'_R} intlist) \wedge \\
&\quad M[R_8 + 0 \mapsto R_3, R_8 + 1 \mapsto R_0 + R_3, R_8 + 2 \mapsto R_2](R_8 + 0) \neq 0 \wedge \\
&\quad writable(R_8 + 2) \wedge writable(R_8 + 1) \wedge writable(R_8 + 0) \\
I_3(R,M) &:= safe\_exit(R_7) \wedge R_8 \geq start \wedge \\
&\quad (R_8 + 0 :_{M[R_8+1\mapsto R_0+R_3, R_8+2\mapsto R_2], A'_R} intlist) \wedge \\
&\quad M[R_8 + 1 \mapsto R_0 + R_3, R_8 + 2 \mapsto R_2](R_8 + 0) \neq 0 \wedge \\
&\quad writable(R_8 + 2) \wedge writable(R_8 + 1) \\
I_4(R,M) &:= safe\_exit(R_7) \wedge R_8 \geq start \wedge \\
&\quad (R_8 + 0 :_{M[R_8+1\mapsto R_3, R_8+2\mapsto R_2], A'_R} intlist) \wedge \\
&\quad M[R_8 + 1 \mapsto R_3, R_8 + 2 \mapsto R_2](R_8 + 0) \neq 0 \wedge \\
&\quad writable(R_8 + 2) \wedge writable(R_8 + 1) \\
I_5(R,M) &:= safe\_exit(R_7) \wedge R_8 \geq start \wedge (R_8 + 0 :_{M[R_8+2\mapsto R_2], A'_R} intlist) \wedge \\
&\quad M[R_8 + 2 \mapsto R_2](R_8 + 0) \neq 0 \wedge writable(R_8 + 2) \\
I_6(R,M) &:= safe\_exit(R_7) \wedge R_8 \geq start \wedge (R_8 + 0 :_{M,A'_R} intlist) \wedge M(R_8 + 0) \neq 0 \\
I_7(R,M) &:= safe\_exit(R_7) \wedge R_8 \geq start \wedge (R_1 :_{M,A'_R} intlist) \wedge M(R_1) \neq 0 \\
I_8(R,M) &:= safe\_exit(R_7) \wedge R_8 \geq start \wedge (R_1 :_{M,A_R} intlist) \wedge M(R_1) \neq 0 \\
I_9(R,M) &:= safe\_exit(R_7 + 0) \wedge R_8 \geq start \\
I_{10}(R,M) &:= safe\_exit(R_6) \wedge R_8 \geq start
\end{aligned}
$$

Figure 13. Clauses of the Invariant for Program in Figure 2 under Extended Safety Policy

99	MOV $r_0 := 0$	store 0 in r_0
100	ST $m(r_8 + 0) := r_0$	store 0 at $m(r_8)$
101	ADDC $r_2 := r_8 + 0$	store r_8's value in r_2
102	ADDC $r_8 := r_8 + 1$	increase r_8 by 1
103	LD $r_5 := m(r_1 + 0)$	load tag of list r_1 into r_5
104	BEQ $(r_5 = r_0)$ 114	jump to point after loop end
105	LD $r_3 := m(r_1 + 1)$	load head of list r_1 into r_3
106	LD $r_1 := m(r_1 + 2)$	load tail of list r_1 into r_1
107	ADDC $r_4 := (r_0 + 1)$	r_4 gets value 1
108	ST $m(r_8 + 0) := r_4$	store this value in $m(r_8)$
109	ST $m(r_8 + 1) := r_3$	store head in $m(r_8 + 1)$
110	ST $m(r_8 + 2) := r_2$	store r_2 (new tail) in $m(r_8 + 2)$
111	ADDC $r_2 := r_8 + 0$	store r_8's current value in r_2
112	ADDC $r_8 := r_8 + 3$	update allocation pointer r_8 by 3
113	JMP r_6	jump back to loop start
114	ADDC $r_1 := (r_2 + 0)$	r_1 gets value of r_2
115	JMP r_7	return

Figure 14. A Program for Reversing a List

It is easy to instantiate R', M', and pc' so that the equalities in this subgoal are provable. The $writable$ conjunct is directly provable from the hypotheses.

Line 8 contains the only LD instruction. The subgoal for this line in the progress lemma is:

$$listextend(M), pc = 8, I_8(R, M) \vdash \exists R', M', pc'.(R, M, 8 \mapsto R', M', pc').$$

Again, we expand definitions to obtain:

$$listextend(M), pc = 8, safe\_exit(R_7) \wedge R_8 \geq start \wedge (R_1 :_{M,A} intlist) \wedge M(R_1) \neq 0 \vdash$$
$$\exists R', M', pc'.(readable(R_1 + 1) \wedge R' = R[r_0 \mapsto M(R_1 + 1)] \wedge M' = M \wedge pc' = 9)$$

Again, we instantiate R', M', and pc' to prove the equalities. As in the proof in Section 3, the $readable$ subgoal is proven using the list_readable1 rule.

7. Loop Invariants in Safety Proofs

Figure 14 contains a second example program, which takes a list of integers as input in r_1 and returns the list in reverse order in the same register. Here, we give line numbers, which will be used in the formal proof. As before r_8 is the allocation pointer. We make the same assumptions about it as before, and we use the same policy for readable and writable addresses. The first four lines of the program

$$
\begin{aligned}
Pre(R, M, pc) &:= safe\_exit(R_7) \wedge R_6 = 103 \wedge (R_1 :_{M,A_R} intlist) \wedge R_8 \geq start \wedge \\
&\quad \forall w.(w \geq R_8 \Rightarrow writable(w)) \wedge \forall w.(w \geq start \Rightarrow readable(w)) \wedge \\
&\quad pc = 99 \wedge listrev(M) \\
I_{103}(R, M) &:= safe\_exit(R_7) \wedge R_6 = 103 \wedge (R_1 :_{M,A_R} intlist) \wedge R_0 = 0 \wedge \\
&\quad \forall w.(w \geq R_8 \Rightarrow writable(w)) \wedge R_8 \geq start \\
I_{115}(R, M) &:= safe\_exit(R_7) \wedge R_8 \geq start
\end{aligned}
$$

Figure 15. Precondition and Predefined Clauses of the Invariant

perform the initialization steps; value 0 representing an empty list is stored in r_0, and this empty list is stored at the memory location pointed to by the allocation pointer. Register r_2 stores the reversed list as it is built, and is initialized to point to the new empty list. Lines 103–113 contain the main loop of the program. First, the tag of the next location in the input list is loaded into r_5 and checked. If it is 0, then the program jumps to the point after the loop (line 114), puts the result in r_1, and jumps to some designated return point stored in r_7. Otherwise the body of the loop is executed. In this case, the next three memory locations starting at the allocation pointer are used to store the new list. The tail of the new list is assigned to the value of r_2, which is a pointer to the reversed list as constructed so far, and r_2 is updated to point to the new beginning of the reversed list. Finally, the allocation pointer is increased by 3, and control returns to the beginning of the loop. Here, we assume that r_6 contains the value of the loop start, in this case 103.

To prove safety, we must provide a loop invariant as a precondition to the first line of the loop. We assume, as usual, that this invariant is provided as a hint by a certifying compiler. Also, as usual we must provide a precondition of the program as a whole and a postcondition for safe exit. For the safety proof using the simple safety policy, we can use the following formula for all these conditions as well as for the invariant clause of every line of code: $safe\_exit(R_7) \wedge R_6 = 103 \wedge R_8 \geq start$.

Safety under the policy which includes memory safety is of course more complicated. Figure 15 contains the precondition of the program, the loop invariant, and postcondition that we use in the proof. The precondition and the loop invariant contain clauses for values that never change, in this case the values of r_6 and r_7. As before, the precondition includes a clause stating where in memory the code is and what the exact instructions are; in this case denoted by $listrev(M)$. The precondition also contains the type information for the input r_1. This typing subformula also appears in the loop invariant, though the value of r_1 does change each time through the loop because it gets reset to point to the tail of the list. Also, to handle allocation correctly, we need to add the policy for writable addresses to the loop invariant: $\forall w.(w \geq R_8 \Rightarrow writable(w))$. In particular, we need to show that this statement is invariant as the value of r_8 in R changes each time through the loop. This extra invariant was not needed in a straight-line program since the the value of r_8 only increased as execution proceeded through the list of instructions. This extra information is required whenever there is the possibility for control to jump back to an earlier instruction. This formula is also used to prove that the store instructions inside the loop are to writable locations. The postcondition is the same as for the previous example, stating only that the return address is safe and that the allocation pointer invariant continues to hold. In the program precondition and the loop invariant, we use A_R as we did in Section 6. In this section, we also use two

$$
\begin{array}{rcl}
I_{99}(R,M) & := & safe\_exit(R_7) \wedge R_6 = 103 \wedge (R_1 :_{M[R_8+0 \mapsto 0], A^1_R} intlist) \wedge 0 = 0 \wedge \\
 & & \forall w.(w \geq R_8 \Rightarrow writable(w)) \wedge R_8 \geq start \wedge writable(R_8 + 0) \\
I_{100}(R,M) & := & safe\_exit(R_7) \wedge R_6 = 103 \wedge (R_1 :_{M[R_8+0 \mapsto R_0], A^1_R} intlist) \wedge R_0 = 0 \wedge \\
 & & \forall w.(w \geq R_8 \Rightarrow writable(w)) \wedge R_8 \geq start \wedge writable(R_8 + 0) \\
I_{101}(R,M) & := & I_{102}(R,M) \\
I_{102}(R,M) & := & safe\_exit(R_7) \wedge R_6 = 103 \wedge (R_1 :_{M, A^1_R} intlist) \wedge R_0 = 0 \wedge \\
 & & \forall w.(w \geq R_8 \Rightarrow writable(w)) \wedge R_8 \geq start \\
I_{104}(R,M) & := & (R_5 = R_0 \Rightarrow I_{114}(R,M)) \wedge (R_5 \neq R_0 \Rightarrow I_{105}(R,M)) \\
I_{105}(R,M) & := & safe\_exit(R_7) \wedge R_6 = 103 \wedge \\
 & & (M(R_1 + 2) :_{M[R_8+0 \mapsto R_0+1, R_8+1 \mapsto M(R_1+1), R_8+2 \mapsto R_2], A^3_R} intlist) \wedge \\
 & & R_0 = 0 \wedge \forall w.(w \geq R_8 \Rightarrow writable(w)) \wedge R_8 \geq start \wedge \\
 & & writable(\{R_8 + 2, R_8 + 1, R_8 + 0\}) \wedge readable(\{R_1 + 2, R_1 + 1\} \\
 & & \vdots \\
I_{110}(R,M) & := & safe\_exit(R_7) \wedge R_6 = 103 \wedge (R_1 :_{M[R_8+2 \mapsto R_2], A^3_R} intlist) \wedge R_0 = 0 \wedge \\
 & & \forall w.(w \geq R_8 \Rightarrow writable(w)) \wedge R_8 \geq start \wedge writable(R_8 + 2) \\
I_{111}(R,M) & := & I_{112}(R,M) \\
I_{112}(R,M) & := & safe\_exit(R_7) \wedge R_6 = 103 \wedge (R_1 :_{M, A^3_R} intlist) \wedge R_0 = 0 \wedge \\
 & & \forall w.(w \geq R_8 \Rightarrow writable(w)) \wedge R_8 \geq start \\
I_{113}(R,M) & := & safe\_exit(R_7) \wedge R_6 = 103 \wedge (R_1 :_{M, A_R} intlist) \wedge R_0 = 0 \wedge \\
 & & \forall w.(w \geq R_8 \Rightarrow writable(w)) \wedge R_8 \geq start \\
I_{114}(R,M) & := & safe\_exit(R_7) \wedge R_8 \geq start
\end{array}
$$

Figure 16. More Clauses of the Invariant

other abbreviations:

$$
\begin{array}{rcl}
A_R(w) & := & (start \leq w < R_8) \\
A^1_R(w) & := & (start \leq w < R_8 + 1) \\
A^3_R(w) & := & (start \leq w < R_8 + 3)
\end{array}
$$

As in the previous example, the Hoare rules for instructions are used to compute the remaining clauses of Inv. Here, we treat both the subformula $R_8 \geq start$ and the new subformula $\forall w.(w \geq R_8 \Rightarrow writable(w))$ specially; that is, they are left unchanged even in the case when applying a Hoare rule would change them. Some of these remaining clauses are given in Figure 16. Here, we write $readable(\{w_1, \ldots, w_n\})$ to abbreviate $readable(w_1) \wedge \cdots \wedge readable(w_n)$, and similarly for $writable$. Note that line 113 of the program is a jump to line 103 (back to the beginning of the loop). We define the precondition of the jump, $I_{113}(R, M)$ to be the same as the precondition for the jump target $I_{103}(R, M)$. Line 104 is a conditional jump and $I_{104}(R, M)$ is defined using 2 conjuncts, one for each possible next

value of the program counter.

Once the invariant is in place, the proof proceeds as in the previous example. We again consider some subgoals from the preservation proof, this time showing subgoals that involve the new aspects of this example such as reasoning from the loop invariant and handling conditional branch instructions. Although these aspects are new, the kind of reasoning and the level of difficulty is no different from the previous example. First, we consider line 103, whose precondition is the loop invariant:

$$(R, M, pc \mapsto R', M', pc'), listrev(M), pc = 103, I_{103}(R, M) \vdash Inv(R', M', pc').$$

Line 103 is a LD instruction, which gives $R' = R[r_5 \mapsto M(R_1 + 0)]$, $M' = M$, and $pc' = 104$. In this case, we must show $I_{104}(R[r_5 \mapsto M(R_1 + 0)], M)$ from $I_{103}(R, M)$. Expanding definitions, we must show:

$$\begin{array}{l} safe\_exit(R_7) \wedge R_6 = 103 \wedge (R_1 :_{M,A_R} intlist) \wedge R_0 = 0 \wedge \\ \quad \forall w.(w \geq R_8 \Rightarrow writable(w)) \wedge R_8 \geq start \vdash \\ \quad\quad (M(R_1 + 0) = R_0 \Rightarrow safe\_exit(R_7) \wedge R_8 \geq start) \wedge \\ \quad\quad\quad (M(R_1 + 0) \neq R_0 \Rightarrow \\ \quad\quad\quad\quad safe\_exit(R_7) \wedge R_6 = 103 \wedge \\ \quad\quad\quad\quad (M(R_1 + 2) :_{M[R_8+0 \mapsto R_0+1, R_8+1 \mapsto M(R_1+1), R_8+2 \mapsto R_2], A_R^3} intlist) \wedge \\ \quad\quad\quad\quad R_0 = 0 \wedge \forall w.(w \geq R_8 \Rightarrow writable(w)) \wedge R_8 \geq start \wedge \\ \quad\quad\quad\quad writable(\{R_8 + 2, R_8 + 1, R_8 + 0\}) \wedge readable(\{R_1 + 2, R_1 + 1\})) \end{array}$$

The reasoning required here is mainly reasoning from the typing rules, which we do not describe in detail. In the progress proof for this line of code, we must prove $readable(R_1 + 0)$. This follows from $(R_1 :_{M,A_R} intlist)$ by the list_readable rule.

Next consider line 104, which contains the branching instruction.

$$(R, M, pc \mapsto R', M', pc'), listrev(M), pc = 104, I_{104}(R, M) \vdash Inv(R', M', pc').$$

Expanding $I_{104}(R, M)$, we get:

$$(R_5 = R_0 \Rightarrow I_{114}(R, M)), (R_5 \neq R_0 \Rightarrow I_{105}(R, M)) \vdash Inv(R', M', pc').$$

The proof proceeds by cases on the branch test, reducing to two trivial subproofs:

$$\begin{array}{l} R_5 = R_0, (R_5 = R_0 \Rightarrow I_{114}(R, M)) \vdash I_{114}(R, M) \\ R_5 \neq R_0, (R_5 \neq R_0 \Rightarrow I_{105}(R, M)) \vdash I_{105}(R, M) \end{array}$$

As a last example, consider line 112, which is one of the lines that modifies r_8. After several steps, most subgoals are easily solved. The only interesting subformulas are those that contain R_8:

$$\forall w.(w \geq R_8 \Rightarrow writable(w)) \wedge R_8 \geq start \vdash \forall w.(w \geq R_8 + 3 \Rightarrow writable(w)) \wedge R_8 + 3 \geq start.$$

This subgoal is easily provable by simple arithmetic.

8. Discussion

The complete proof for the first example program and the simple safety policy is approximately 2000 lines of Coq script. Roughly 350 lines is the part that is specific to proving safety of the program, and the rest is the foundational part. Moving to the more complex memory safety policy required about 900 additional lines. Most of those, about 750, included the foundational encoding of types. The rest was due to the extra proof obligations required to show safety of the program under the extended safety policy. As discussed along the way, various parts of the proof are amenable to automation, particularly proofs of safety for specific programs, since they follow a specific pattern. For example, given a set of hints representing preconditions for particular lines of code, the remaining clauses of the invariant can be automatically generated, and most cases of the progress and preservation lemmas are simple. In addition, the library of lemmas built for the type constructors in Section 4 allow automatic proof of a variety of properties of any type built up from the type constructors.

In fact, our first prototype system gave us some experience with fully automating proofs. In this system, we used the typing rules in Section 4 and the Hoare-style rules in Section 2 as axioms. Thus the system was not yet foundational, but instead concentrated on handling allocation of data structures correctly. This prototype was implemented in λProlog [15, 16], and proofs of safety of a variety of examples, including the list reverse program presented here, were constructed fully automatically. Since the typing rules have since been derived, and since reasoning using the safety rule corresponds to reasoning using the Hoare-style rules, the proof we generated automatically is similar to the proof done by hand in Coq. In fact, our motivation for doing the Coq proof was to study the similarities and differences in the two styles of reasoning to gain an understanding of how to automate proofs using only the foundational rules. Most of the non-trivial proof search involves determining which typing rules to apply and fairly straightforward reasoning about arithmetic equalities and inequalities, which can easily be handled by a system with simple but efficient rewriting capabilities. Proving that $listextend(M)$ and $listrev(M)$ are invariants (i.e., the code is not modified by execution of the program) was not part of our original automated proof, but also follows by simple reasoning and arithmetic.

As we discussed, our approach allows the integration of proofs of other properties into safety proofs. (For example, the proof in Figure 3 includes an additional postcondition.) In the short version of this paper [8], in addition to safety of the reverse program, we proved that the output register r_1 containing the reversed list does indeed have type $intlist$. To prove this property, we also needed to include the type of integer list r_2 as a postcondition.

Another approach to structuring safety proofs, which would lead to more modular reasoning, would be to derive the Hoare rules of Section 2 from the direct step-relation encoding. Hamid and Shao [9], in fact, derive a version of Hoare-style rules in the context of reasoning using TAL in a syntactic FPCC system. Perhaps their approach could be carried over to our setting. Such rules are more complex than those presented here, so it is not a matter of simply stating and proving such rules. For example, when starting with the step-relation encoding various aspects of the state become explicit in such rules.

Chang et. al. [5] argue that because there exist a variety of code verification strategies, it is best to use a verifier that is best suited to the code verification strategy. Most examples of safety policies, including the ones considered here, have been simple. The setting described here is general and flexible and may be a good starting point for handling a variety of strategies. This is a subject of future work.

Acknowledgments

The author acknowledges the support of the Natural Sciences and Engineering Research Council of Canada.

References

[1] Ahmed, A. J., Appel, A. W., Virga, R.: A Stratified Semantics of General References Embeddable in Higher-Order Logic, *Proc. Seventeenth Annual IEEE Symposium on Logic in Computer Science*, IEEE Computer Society, 2002.

[2] Appel, A. W., Felty, A. P.: A Semantic Model of Types and Machine Instructions for Proof-Carrying Code, *Proc. 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM Press, 2000.

[3] Appel, A. W., McAllester, D.: An Indexed Model of Recursive Types for Foundational Proof-Carrying Code, *ACM Transactions on Programming Languages and Systems*, **13**(5), September 2001, 657–683.

[4] Bertot, Y., Castéran, P.: *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*, Springer, 2004.

[5] Chang, B.-Y. E., Chlipala, A., Necula, G. C., Schneck, R. R.: The Open Verifier Framework for Foundational Verifiers, *Proc. ACM SIGPLAN International Workshop on Types in Language Design and Implementation*, ACM Press, 2005.

[6] Coq Development Team, LogiCal Project: *The Coq Proof Assistant Reference Manual: Version 8.0*, Technical report, INRIA, 2006.

[7] Crary, K.: Toward a Foundational Typed Assembly Language, *Proc. 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM Press, 2003.

[8] Felty, A. P.: A Tutorial Example of the Semantic Approach to Foundational Proof-Carrying Code, *Proc. Sixteenth International Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science 3465, Springer-Verlag, 2005.

[9] Hamid, N. A., Shao, Z.: Interfacing Hoare Logic and Type Systems for Foundational Proof-Carrying Code, *Proc. Seventeenth International Conference on Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science 3223, Springer-Verlag, 2004.

[10] Hamid, N. A., Shao, Z., Trifonov, V., Monnier, S., Ni, Z.: A Syntactic Approach to Foundational Proof-Carrying Code, *Journal of Automated Reasoning*, **31**(3–4), November 2003, 191–229.

[11] Harper, R., Honsell, F., Plotkin, G.: A Framework for Defining Logics, *Journal of the ACM*, **40**(1), January 1993, 143–184.

[12] Huth, M. R. A., Ryan, M. D.: *Logic in Computer Science: Modelling and Reasoning about Systems*, Second edition, Cambridge University Press, 2004.

[13] Michael, N. G., Appel, A. W.: Machine Instruction Syntax and Semantics in Higher Order Logic, *Proc. Seventeenth International Conference on Automated Deduction*, Lecture Notes in Computer Science 1831, Springer-Verlag, 2000.

[14] Morrisett, G., Walker, D., Crary, K., Glew, N.: From System F to Typed Assembly Language, *ACM Transactions on Programming Languages and Systems*, **21**(3), May 1999, 527–568.

[15] Nadathur, G., Miller, D.: An Overview of λProlog, *Proc. Fifth International Conference and Symposium on Logic Programming*, MIT Press, 1988.

[16] Nadathur, G., Mitchell, D. J.: System Description: Teyjus — A Compiler and Abstract Machine Based Implementation of λProlog, *Proc. Sixteenth International Conference on Automated Deduction*, Lecture Notes in Computer Science 1632, Springer-Verlag, 1999.

[17] Necula, G.: Proof-carrying Code, *Proc. 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM Press, 1997.

[18] Necula, G. C.: *Compiling with Proofs*, Ph.D. Thesis, Carnegie Mellon University, Pittsburgh, PA, 1998.

[19] Necula, G. C., Lee, P.: *Proof-Carrying Code*, Technical Report CMU-CS-96-165, Carnegie Mellon University, November 1996.

[20] Swadi, K. N.: *Typed Machine Language*, Ph.D. Thesis, Princeton University, Princeton, NJ, 2003.

[21] Tan, G., Appel, A. W., Swadi, K. N., Wu, D.: Construction of a Semantic Model for a Typed Assembly Language, *Proc. Fifth International Conference on Verification, Model Checking and Abstract Interpretation*, Lecture Notes in Computer Science 2937, Springer-Verlag, 2004.