A Tutorial Example of the Semantic Approach to Foundational Proof-Carrying Code *

Amy P. Felty

School of Information Science and Technology
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 a detailed example. The example, 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 reasoning about programs was similar to reasoning using Hoare-style program verification rules. This work extended the original proof-carry code (PCC) work [13] 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 *Proceedings of the 16th International Conference on Rewriting Techniques and Applications, April 2005,* ©*Springer-Verlag.*

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 [9]. In addition, foundational versions of typed assembly languages (TAL) [10] have been developed for use in FPCC systems (e.g. [6, 14, 15]). Also, an alternative syntactic approach has been explored [8].

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 example is larger, we keep the semantics simple. We only require the simple semantics of types and the same 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. A more detailed comparison to syntactic approaches is future work.

The example presented here is a machine language program which reverses a list of integers. This example is complex enough to require recursive data types. It takes a list as input, and the computation includes traversing this input list as well as building a new one. The latter operation requires allocating new memory along the way. In addition, the program uses most of the instructions available in our simple instruction set.

After presenting the example program in Sect. 2, we present the typing lemmas in Sect. 3, followed by the encoding of machine instruction semantics. We present the machine semantics in two steps. As a first step, we prove the safety of our example program with respect to a set of Hoare-style program verification rules for machine instructions given in Section 4. Using such rules is fairly similar to the use of a VCG in the original PCC framework [13], but 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. After presenting these rules, we discuss the safety proof of our example program.

Sect. 5 presents the second step in encoding machine instruction semantics. Here, we follow the approach of Appel and Michael [9]. We start with a direct encoding of machine instructions as a step relation relating one machine state to another, and we prove a theorems stating that safety follows from "progress" and "preservation" lemmas. We do not derive the Hoare rules of Sect 4, but we build up a library of lemmas which provides reasoning similar in style to using such rules. Describing both approaches here allows us to compare them. In particular, our example, discussed again in Sect. 6 provides enough detail to illustrate how the two styles of reasoning correspond. It would be interesting to take this work a step further and derive Hoare-style rules from the direct step-relation encoding. Hamid and Shao [7], 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.

The proof discussed in Sect. 6 has been fully formalized in Coq [5]. 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 [8]. Most of the proof was done interactively, but we discuss its automation in Sect. 7, where we also discuss other issues and related work.

2 Example

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}. We introduce our set of machine instructions by directly presenting the reverse program in Fig. 1. We assume that register r_0 has value 0 and

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	BEQ $(r_0 = r_0)$ 103	jump back to loop start
114	ADDC $r_1 := (r_2 + 0)$	r_1 gets value of r_2
115	JMP r_7	return

Fig. 1. A Program for Reversing a List

that input register r_1 contains a list of integers. 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. The first 3 lines of the program perform the initialization steps; an 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 3 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. In addition to the instructions used in this program, our simple programming language also includes a `MOV` instruction, and another branching instruction `BGT` which compares two values and branches if the first is greater than the second.

To prove safety, the precondition of this program must include our assumptions $r_0 = 0$ and that r_1 contains a list of integers. We write this latter assumption as the typing judgment $(r_1 :_{m,r_8} intlist)$. Typing judgments depend on the contents of memory and the set of currently allocated locations; in particular all memory locations used to represent a list must be allocated. We leave the exact specification of this set unspecified here, but assume that it is a subset of all memory locations occurring before the allocation pointer r_8. To indicate this dependence, memory m and allocation pointer r_8 are given explicitly as subscripts to the typing judgment.

The precondition of this program must include additional information that is part of the loop invariant needed to prove safety of the program. For instance, the policy on readable memory locations is needed. We assume that all memory locations after a particular location $start$ are readable, expressed as the formula $Policy$ and defined as follows:

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

We assume that the memory locations that we are permitted to write to are a subset of the readable locations. In particular, we assume they are all locations starting at the allocation pointer r_8 and that the allocation pointer r_8 is greater than $start$:

$$(r_8 \geq start) \wedge \forall w.(w \geq r_8 \Rightarrow writable(w)).$$

The loop invariant also includes $safe\_exit(r_7)$ which states that the return location is indeed safe. The complete precondition is stated as the precondition of the first line of code (line 100), defined as formula I_{100} in Fig. 2.

$$\begin{array}{ll}
I_{100}: & Policy \wedge (r_8 \geq start) \wedge \forall w.(w \geq r_8 \Rightarrow writable(w)) \wedge \\
& \wedge safe_exit(r_7) \wedge r_0 = 0 \wedge (r_1 :_{m,r_8} intlist) \\
I_{102}: & allocptr\ r_8\ 1 \\
I_{103}: & Policy \wedge (r_8 - 1 \geq start) \wedge \forall w.(w \geq r_8 \Rightarrow writable(w)) \wedge \\
& safe_exit(r_7) \wedge r_0 = 0 \wedge (r_1 :_{m,r_8} intlist) \wedge \\
& (r_2 :_{m,r_8} intlist) \wedge readable(r_1) \\
I_{112}: & allocptr\ r_8\ 3 \\
I_{114}: & safe_exit(r_7) \wedge (r_1 :_{m,r_8} intlist) \wedge (r_2 :_{m,r_8} intlist) \\
I_{115}: & safe_exit(r_7) \wedge (r_1 :_{m,r_8} intlist)
\end{array}$$

Fig. 2. Preconditions for Selected Lines of Code

Fig. 2 also includes preconditions of some other lines of code in the program. In general, we write I_c to denote the precondition of line c of the code. In addition to the precondition of the entire program, we must have preconditions of all of the jump points, in this case lines 103 and 114. The precondition of line 103 is the loop invariant. When executing the loop body and when exiting it, we must know the type of register r_2, which stores the intermediate results, i.e., the reversed list as it is being constructed. This typing judgment appears in both I_{103} and I_{114}. Because line 103 is a load instruction, I_{103} contains the requirement that the load is from a readable location. Everything else in I_{103} comes from the precondition and remains invariant when executing the loop body. In this example, we also include a precondition for the last line of the program I_{115}. Much of the information provided in Fig. 2, including the typing information, can be generated automatically by a certifying compiler [13]. We call such compiler-generated formulas *hints* to distinguish from those we calculate later. To handle allocation correctly, we also need to know which lines in the program modify the allocation pointer. Lines I_{102} and I_{112} provide this information; in particular, the register serving as the allocation pointer and the amount it is increased at a given line is stated.

3 Types

We present the typing rules that are used in the proof of safety of the example program. We leave out the definitions and lemmas needed to prove these rules. We simply note that we require most of the definitions of types and type constructors and lemmas about them that were presented in our earlier work [2].

We define a *valid* type to be any type τ for which the following two rules hold.

$$\frac{w :_{m,A} \tau \qquad \neg A(v)}{w :_{m[v \mapsto u],A} \tau} \qquad\qquad \frac{w :_{m,A} \tau \qquad \forall x.A(x) \Rightarrow A'(x)}{w :_{m,A'} \tau}$$

In these rules A is an arbitrary *allocation predicate* specifying the set of allocated addresses. In our example, $A(w) := (start \leq w < r_8)$. The expression $m[v \mapsto u]$ denotes the memory m modified so that location v has value u. Integers and integer lists are both valid types [2].

The remaining typing rules we use in proving safety of our program are given in Fig. 3. They are stated in terms of lists of arbitrary type τ.

4 Machine Semantics as Hoare-Style Rules

Fig. 4 contains a set of Hoare-style rules for our machine instructions. Unlike the typing rules in the previous section, we take these rules as axioms. As mentioned earlier, although we must trust them, they provide more security than a VGC. The first rule is used to prove safety of a program with respect to a precondition Pre. We assume the program is a sequence of machine instructions ending with a `JMP` to a safe return point. Note that there is one rule for each machine instruction and that these rules are axioms.

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

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

$$\frac{w :_{m,A} list(\tau) \qquad valid(\tau)}{readable(w)}$$

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

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

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

$$\frac{\begin{array}{cccc} & & A(w) & readable(w) \\ m(w+1) :_{m,A} \tau & m(w) = 1 & A(w+1) & readable(w+1) \\ m(w+2) :_{m,A} list(\tau) & valid(\tau) & A(w+2) & readable(w+2) \end{array}}{m(w) :_{m,A} list(\tau)}$$

Fig. 3. Typing rules for integer lists

$$\frac{Pre \Rightarrow I_1 \quad \{I_1\}S_1\{I_2\} \quad \ldots \quad \{I_n\}S_n\{I_{n+1}\} \quad I_{n+1} \Rightarrow safe\_exit(r)}{safe(Pre, (S_1; \ldots; S_n; \texttt{JMP}\ r))}\ \text{safety}$$

$$\frac{}{\{I[r_s + c/r_d]\}\ \texttt{ADDC}\ \ r_d := r_s + c\ \{I\}}\ \text{addc} \qquad \frac{}{\{I[c/r_d]\}\ \texttt{MOV}\ \ r_d := c\ \{I\}}\ \text{mov}$$

$$\frac{}{\{I[r_{s_1} + r_{s_2}/r_d]\}\ \texttt{ADD}\ \ r_d := r_{s_1} + r_{s_2}\ \{I\}}\ \text{add} \qquad \frac{}{\{I_{m(r)}\}\ \texttt{JMP}\ \ r\ \{I\}}\ \text{jmp}$$

$$\frac{}{\{(r_{s_1} > r_{s_2} \rightarrow I_c) \wedge (\neg(r_{s_1} > r_{s_2}) \rightarrow I)\}\ \texttt{BGT}\ \ (r_{s_1} > r_{s_2})\ c\ \{I\}}\ \text{bgt}$$

$$\frac{}{\{(r_{s_1} = r_{s_2} \rightarrow I_c) \wedge (r_{s_1} \neq r_{s_2} \rightarrow I)\}\ \texttt{BEQ}\ \ (r_{s_1} = r_{s_2})\ c\ \{I\}}\ \text{beq}$$

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

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

$$\frac{A \rightarrow A' \qquad \{A'\}S\{B'\} \qquad B' \rightarrow B}{\{A\}S\{B\}}\ \text{Implied}$$

Fig. 4. Hoare-style rules for machine instructions

A proof of safety is built by starting with the postcondition I_{n+1} and applying the rule corresponding to statement S_n to obtain I_n. Then I_n is used as the postcondition of statement S_{n-1} to compute I_{n-1}, etc. For any statement S_k ($1 \leq k \leq n$), if there is an associated hint I_k, this hint is used as the postcondition of statement S_{k-1}. Let I'_k be the formula obtained by applying the axiom for statement S_k using postcondition I_{k+1}. At this point the Implied rule is used, resulting in proof obligation $I_k \Rightarrow I'_k$. The formula I_c in rules bgt and beq is the precondition of the statement at location c. Requiring hints for all jump points insures that such a formula always exists when applying the proof strategy just described.

The $allocptr$ hints also generate proof obligations. The hint I_k tells us how to modify the postcondition of S_k. If I_{k+1} is the postcondition computed by applying the appropriate axiom, and I'_{k+1} is obtained from I_{k+1} because of the $allocptr$ hint, the we have proof obligation $I'_{k+1} \Rightarrow I_{k+1}$. We will see how to use the $allocptr$ hints to modify postconditions for our example program in Sect. 5.

Finally, we also have the proof obligations that appear as the first and last premise in the safety rule.

We can modify the safety rule so that the program includes a postcondition $Post$ and the final premise states $I_n \Rightarrow (safe\_exit(r) \wedge Post)$. We use this version of the safety rule in our proof, so that in addition to safety, we prove that the output reversed list does indeed have type $intlist$.

5 Encoding Machine Semantics Directly

We define the type Reg to be the type of the set of 32 registers r_0 to r_{31}. $Word$ 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 [9]). We write Mem to represent the function type ($Word \rightarrow Word$). In particular, memory is modelled as a function from machine addresses to machine values. Similarly register banks are functions from registers to values; $RegFile$ denotes the function type ($Reg \rightarrow Word$).

We define a machine state to be a triple of the form (R, M, pc) where R is a register bank (of type $RegFile$), M is a memory (of type Mem), and pc is a $Word$. 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 the reflexive transitive closure of this operation.

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.

We leave out the details, which can be found in our earlier work [2]. We note that what we have described so far is the part of our formalization in Coq where we have adopted and modified some basic definitions from Hamid et. al. [8].

Following Michael and Appel [9], we define $safe$, $Progress$, and $Preservation$ predicates as follows, and prove the safe* rule below.

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

$$\frac{Inv(R,M,pc) \qquad Progress(Inv) \qquad Preservation(Inv)}{safe(R,M,pc)}\ \text{safe}^*$$

The $safe$ predicate expresses the fact that execution of safe programs don't get stuck, for example, trying to execute a load from a non-readable location or a store to a non-writable location. Note that $safe$ is now a predicate on a machine state. The code is in M and pc points to the first instruction. In the definitions of $Progress$ and $Preservation$, Inv is a predicate which takes a machine state (R,M,pc) as an argument.

Finally, we define $safe\_exit$ as follows:

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

6 Example Revisited

It can be seen from the formalization discussed in the previous section that we start from a fairly low-level encoding of the machine semantics and end with the high-level derived rule safe*. Reasoning using this rule corresponds closely to reasoning using the Hoare-style rules of Sect. 4. In the new setting, a proof of safety starts by applying safe*. To do so, 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 the lines for which we already have hints, we use those hints fairly directly. We modify them to become predicates over a register bank and memory. If R is the function representing a register bank, we abbreviate $R(r_i)$ as R_i. We write M for memory functions. Using this encoding, we modify the formulas given in Fig. 2 to obtain the predicates in Fig. 5.

We must actually make one more modification. We must add information to the invariant so that we can show that this program does not include self-modifying code. This information is not needed in the proof using the Hoare-style rules of Sect. 4. In these rules, 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 our new encoding, we define the program using a predicate $listrev(M)$ which states that decoding the instruction at line 100 gives instruction 100 as defined in Fig. 1, and similarly for all the lines of code in the program. If there was overlap between the code and data parts of memory, we would not be able to prove safety. To make explicit that there is no overlap, we add the formula $start > 1000$ to I_{100} and I_{103}. It then becomes part of the loop

$$
\begin{aligned}
I_{100}(R, M) &:= Policy \wedge (R_8 \geq start) \wedge \forall w.(w \geq R_8 \Rightarrow writable(w)) \wedge \\
&\quad \wedge safe\_exit(R_7) \wedge R_0 = 0 \wedge (R_1 :_{M,R_8} intlist) \\
I_{103}(R, M) &:= Policy \wedge (R_8 - 1 \geq start) \wedge \forall w.(w \geq R_8 \Rightarrow writable(w)) \wedge \\
&\quad safe\_exit(R_7) \wedge R_0 = 0 \wedge (R_1 :_{M,R_8} intlist) \wedge \\
&\quad (R_2 :_{M,R_8} intlist) \wedge readable(R_1) \\
I_{114}(R, M) &:= safe\_exit(R_7) \wedge (R_1 :_{M,R_8} intlist) \wedge (R_2 :_{M,R_8} intlist) \\
I_{115}(R, M) &:= safe\_exit(R_7) \wedge (R_1 :_{M,R_8} intlist)
\end{aligned}
$$

Fig. 5. Clauses of the invariant that come from hints

invariant, and since all of the store instructions are inside the loop, we can prove that $listrev(M)$ remains invariant even while M is changing. The formula $start > 1000$ does not appear in Fig. 5, and we continue to leave it out of the invariant clauses that we present below. Although it is important to the proof, it is not important to the rest of the presentation, and it is easy to prove that it remains a constant at each step.

The full predicate Inv has the following form:

$$
\begin{aligned}
Inv(R, M, pc) := [&(listrev\ M) \wedge \\
&(pc = 100 \wedge I_{100}(R, M)) \vee \cdots (pc = 115 \wedge I_{115}(R, M))] \vee \\
&safe(R, M, pc)
\end{aligned}
$$

The second clause of Inv's top-level disjunction is used when the program counter gets the value of r_7. The definition of $safe\_exit$ is used directly to prove this case. The clauses for lines of code that are not defined in Fig. 5 can be automatically calculated by simply applying the rules in Sect. 4. As we have stated, we do not prove the Hoare rules as lemmas from our new encoding of machine semantics. Instead, we apply them by hand to get Inv. It would be easy and much better to write a program to automatically generate them. Note that such a program would not be part of the trusted code; if an invariant is incorrect, it would not be possible to prove the program safe. To illustrate, some of these remaining clauses of Inv are given in Fig. 6. Given a memory function M, we write $M[a_1 \mapsto w_1, \ldots, a_n \mapsto w_n]$ to denote a new function which is the same as M except that for $i = 1, \ldots, n$, the new function maps address a_i to value w_i. We write $readable(\{w_1, \ldots, w_n\})$ to abbreviate $readable(w_1) \wedge \cdots \wedge readable(w_n)$, and similarly for $writable$.

First, consider I_{113} which is the precondition for the statement $(\mathsf{BEQ}\ \ (r_0 = r_0)\ 103)$. We applied the beq rule to obtain I_{113} from I_{114}. Note that the precondition in this rule has a true and a false case. We only need the true case here, so we can take as a precondition simply I_c, which in our case is I_{103}. In addition, we need to consider the fact that the statement at line 113 follows a line which increased the allocation pointer r_8 by 3. We must modify I_{103} to account for this increase. Our signal to do so comes from the hint I_{112} in Fig. 2, which we repeat (for documentation purposes only) in Fig. 6 just before the definition of I_{112}. In particular, we must subtract 3 from the expression $R_8 - 1$ in I_{103} to obtain $(R_8 - 4 \geq start)$ in I_{113}.

To obtain I_{112}, we simply apply the addc rule to I_{113}, replacing R_8 with $R_8 + 3$. Most invariant clauses are obtained from such simple rule applications. I_{111} is also obtained by a simple application of addc.

$$
\begin{aligned}
I_{101}(R,M) := {} & Policy \wedge (R_8 \geq start) \wedge \forall w.(w \geq R_8 + 1 \Rightarrow writable(w)) \wedge \\
& safe\_exit(R_7) \wedge R_0 = 0 \wedge (R_1 :_{M,R_8} intlist) \wedge \\
& (R_8 :_{M,R_8} intlist) \wedge readable(R_1) \\
I_{102}(R,M) := {} & allocptr\ R_8\ 1 : \\
& Policy \wedge (R_8 \geq start) \wedge \forall w.(w \geq R_8 + 1 \Rightarrow writable(w)) \wedge \\
& safe\_exit(R_7) \wedge R_0 = 0 \wedge (R_1 :_{M,R_8} intlist) \wedge \\
& (R_2 :_{M,R_8} intlist) \wedge readable(R_1) \\
I_{104}(R,M) := {} & (R_5 = R_0 \Rightarrow I_{114}) \wedge (R_5 \neq R_0 \Rightarrow I_{105}) \\
I_{105}(R,M) := {} & Policy \wedge (R_8 - 1 \geq start) \wedge \forall w.(w \geq R_8 + 3 \Rightarrow writable(w)) \wedge \\
& \wedge safe\_exit(R_7) \wedge R_0 = 0 \wedge \\
& (R_1 :_{M[R_8 \mapsto R_0+1, R_8+1 \mapsto M(R_1+1), R_8+2 \mapsto R_2], R_8} intlist) \wedge \\
& (R_8 :_{M[R_8 \mapsto R_0+1, R_8+1 \mapsto M(R_1+1), R_8+2 \mapsto R_2], R_8} intlist) \wedge \\
& readable(\{M(R_1+2), R_1+2, R_1+1\}) \wedge \\
& writable(\{R_8, R_8+1, R_8+2\}) \\
& \vdots \\
I_{110}(R,M) := {} & Policy \wedge (R_8 - 1 \geq start) \wedge \forall w.(w \geq R_8 + 3 \Rightarrow writable(w)) \wedge \\
& \wedge safe\_exit(R_7) \wedge R_0 = 0 \wedge (R_1 :_{M[R_8+2 \mapsto R_2], R_8} intlist) \wedge \\
& (R_8 :_{M[R_8+2 \mapsto R_2], R_8} intlist) \wedge readable(R_1) \wedge writable(R_8+2) \\
I_{111}(R,M) := {} & Policy \wedge (R_8 - 1 \geq start) \wedge \forall w.(w \geq R_8 + 3 \Rightarrow writable(w)) \wedge \\
& \wedge safe\_exit(R_7) \wedge R_0 = 0 \wedge (R_1 :_{M,R_8} intlist) \wedge \\
& (R_8 :_{M,R_8} intlist) \wedge readable(R_1) \\
I_{112}(R,M) := {} & allocptr\ R_8\ 3 : \\
& Policy \wedge (R_8 - 1 \geq start) \wedge \forall w.(w \geq R_8 + 3 \Rightarrow writable(w)) \wedge \\
& \wedge safe\_exit(R_7) \wedge R_0 = 0 \wedge (R_1 :_{M,R_8} intlist) \wedge \\
& (R_2 :_{M,R_8} intlist) \wedge readable(R_1) \\
I_{113}(R,M) := {} & Policy \wedge (R_8 - 4 \geq start) \wedge \forall w.(w \geq R_8 \Rightarrow writable(w)) \wedge \\
& \wedge safe\_exit(R_7) \wedge R_0 = 0 \wedge (R_1 :_{M,R_8} intlist) \wedge \\
& (R_2 :_{M,R_8} intlist) \wedge readable(R_1)
\end{aligned}
$$

Fig. 6. More clauses of the invariant.

Next, consider I_{110} which is the precondition for ($\mathsf{ST}\ \ m(r_8+2) := r_2$). We obtain I_{110} by first replacing the memory expression M which appears as a subscript to the typing judgments by $M[R_8+2 \mapsto R_2]$, and then adding a new $writable$ conjunct.

Working backward, I_{109} back through I_{105} are obtained by straightforward applications of the appropriate rules. We omit the details, showing only the last in the series, I_{105}. We then obtain I_{104} by applying the beq rule. This brings us back to I_{103} which was already given in Fig. 5. Note that at the point I_{103} was generated as a hint, the allocation pointer had to be taken into account; in this case, the increment by 1 at line 102 means we decremented R_8 by 1 to obtain $(R_8 - 1 \geq start)$ in I_{103}. Finally, I_{102} and I_{101} are also obtained by straightforward rule applications.

We show that our example program is safe whenever precondition I_{100} holds for the initial register bank and memory. We must add the fact that the program counter starts at line 100 and that $listrev(M_0)$ holds. Thus, the safety theorem is stated:

$$\forall R_0, M_0, pc_0.(pc_0 = 100 \wedge listrev(M) \wedge I_{100}(R_0, M_0)) \Rightarrow safe(R_0, M_0, pc_0).$$

To prove this theorem, we apply the safe^* rule, which means we must show that $Inv(R_0, M_0, pc_0)$, $Progress(Inv)$, and $Preservation(Inv)$ hold under the assumptions $pc_0 = 100$, $listrev(M_0)$, and $I_{100}(R_0, M_0)$. $Inv(R_0, M_0, pc_0)$ is a disjunction, and we prove the first disjunct, and the proof is immediate. Since $pc_0 = 100$, showing $Inv(R_0, M_0, pc_0)$ reduces to showing that $listrev(M_0)$ and that $I_{100}(R_0, M_0)$. To show progress, we must show that no matter which line we are at in the program, there is a next step. This is straightforward, and includes proving $readable$ and $writable$ subgoals for load and store instructions. These follow immediately from the fact that the preconditions of all such instructions contain the necessary $readable$ and $writable$ facts.

Proving $Preservation(Inv)$ is where Hoare-style reasoning takes place. We have a case for each line of the program; for $pc = 100, \ldots, 114$, under the assumption that $Inv_i(R, M)$ and $(R, M, i \mapsto R', M', pc')$ hold, we show that $Inv_{i+1}(R', M', pc')$ holds. For the cases where we calculated Inv_i from Inv_{i+1} 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 Inv_i to include in Inv. More reasoning is needed for the cases when Inv_i comes from a hint. The subgoals we must prove correspond to the proof obligations that were described earlier. Consider the formulas in Fig. 7. Formula I'_k is obtained from formula I_{k+1} by an application of the Hoare axiom for the statement at line k. Note that there is one such clause in Fig. 7 for every line of code for which we started out with a hint. The proof obligations we are left with are to show that $I_{100} \Rightarrow I'_{100}$, $I_{103} \Rightarrow I'_{103}$, and $I_{114} \Rightarrow I'_{114}$. The third one is straightforward. The second one is the most complex. The first and second together require all of the typing rules in Fig. 3. This reasoning corresponds to applications of the $\mathsf{Implied}$ rule in the proof using the Hoare-style rules.

7 Discussion

The complete proof is approximately 3000 lines of Coq script. Roughly half of that is the foundational part and the other half is the proof of safety of the example program.

$$
\begin{aligned}
I'_{100}(R, M) := \; & allocptr\ R_8\ 1 : \\
& Policy \wedge (R_8 \geq start) \wedge \forall w.(w \geq R_8 + 1 \Rightarrow writable(w)) \wedge \\
& safe\_exit(R_7) \wedge R_0 = 0 \wedge (R_1 :_{M[R_8 \mapsto R_0], R_8} intlist) \wedge \\
& (R_8 :_{M[R_8 \mapsto R_0], R_8} intlist) \wedge readable(R_1) \wedge writable(R_8) \\
I'_{103}(R, M) := \; & (M(R_1) = R_0 \Rightarrow I_{114}) \wedge (M(R_1) \neq R_0 \Rightarrow I_{105}) \wedge readable(R_1) \\
I'_{114}(R, M) := \; & safe\_exit(R_7) \wedge (R_2 :_{M, R_8} intlist)
\end{aligned}
$$

Fig. 7. Clauses that form proof obligations

The latter part could be fully automated. In fact, in our first prototype system, we used the typing rules in Sect. 3 and the Hoare-style rules in Sect 4 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 [11, 12], 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 safe^* 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 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 $listrev(M)$ is an invariant, which was not part of our original automated proof, involves simple but numerous subgoals which followed from simple arithmetic rules.

Our example program is one representative from a large class of programs that could be proved safe with the same kind of automated proof search. Although we did not include the basic definitions, our $intlist$ type was defined using a library of definitions for a wide variety of type constructors. Any programs manipulating data structures built from such type constructors fit into this class.

PCC systems that use foundational versions of TAL go even further in the direction of easily automated safety proofs. They essentially reduce such proofs to type checking. Safety in such a setting is limited to what is expressible in TAL. Chang et. al. [4] argue that because there exist a variety of code verification strategies, it is better to use a verifier that is best suited to the code verification strategy. Most examples of safety policies have been simple. In fact, our example does not use a safety policy any more sophisticated than what can be expressed in TAL. But when extending such policies to include more complex properties, other strategies besides TAL may become important. Our approach to automating proofs should provide more flexibility in handling a variety of strategies. This is another subject of future work.

Acknowledgments

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

References

1. Amal J. Ahmed, Andrew W. Appel, and Roberto Virga. A stratified semantics of general references embeddable in higher-order logic. In *Seventeenth Annual IEEE Symposium on Logic in Computer Science*, pages 75–86, July 2002.
2. Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 243–253, 2000.
3. Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems*, 13(5):657–683, September 2001.
4. Bor-Yuh Evan Chang, Adam Chlipala, George C. Necula, and Robert R. Schneck. The open verifier framework for foundational verifiers. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation*, January 2005.
5. Coq Development Team. The Coq Proof Assistant reference manual: Version 7.4. Technical report, INRIA, 2003.
6. Karl Crary and Susmit Sarkar. Toward a foundational typed assembly language. In *Thirtieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 198–212, 2003.
7. Nadeem A. Hamid and Zhong Shao. Interfacing hoare logic and type systems for foundational proof-carrying code. In *Seventeenth International Conference on the Applications of Higher Order Logic Theorem Proving*, volume 3223 of *Lecture Notes in Computer Science*, pages 118–135. Springer-Verlag, September 2004.
8. Nadeem A. Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. A syntactic approach to foundational proof-carrying code (extended version). *Journal of Automated Reasoning*, 31(3–4):191–229, 2003.
9. Neophytos G. Michael and Andrew W. Appel. Machine instruction syntax and semantics in higher order logic. In *Seventeenth International Conference on Automated Deduction*, Lecture Notes in Computer Science, pages 7–24. Springer-Verlag, June 2000.
10. Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.
11. Gopalan Nadathur and Dale Miller. An overview of λProlog. In K. Bowen and R. Kowalski, editors, *Fifth International Conference and Symposium on Logic Programming*. MIT Press, 1988.
12. Gopalan Nadathur and Dustin. J. Mitchell. System description: Teyjus — a compiler and abstract machine based implementation of λProlog. In *The Sixteenth International Conference on Automated Deduction*, pages 287–291. Springer-Verlag, July 1999.
13. George Necula. Proof-carrying code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119. ACM Press, January 1997.
14. Kedar N. Swadi and Andrew W. Appel. Foundational semantics for TAL syntactic rules via typed machine language. http://www.cs.princeton.edu/~kswadi/papers/tml.ps, March 2002.

15. Gang Tan, Andrew W. Appel, Kedar N. Swadi, and Dinghao Wu. Construction of a semantic model for a typed assembly language. In *Fifth International Conference on Verification, Model Checking and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 30–43. Springer-Verlag, January 2004.