Machine Instruction Syntax and Semantics in Higher Order Logic

Neophytos G. Michael and Andrew W. Appel

Computer Science Department, Princeton University, 35 Olden Street,
Princeton, NJ 08544, USA
`nmichael@cs.princeton.edu`
`appel@cs.princeton.edu`

Abstract. Proof-carrying code and other applications in computer security require machine-checkable proofs of properties of machine-language programs. These in turn require axioms about the opcode/operand encoding of machine instructions and the semantics of the encoded instructions. We show how to specify instruction encodings and semantics in higher-order logic, in a way that preserves the factoring of similar instructions in real machine architectures. We show how to automatically generate proofs of instruction decodings, global invariants from local invariants, Floyd-Hoare rules and predicate transformers, all from the specification of the instruction semantics. Our work is implemented in ML and Twelf, and all the theorems are checked in Twelf. A version of this paper is to appear at the 17th International Conference on Automated Deduction to be held between June 17-20, 2000 in Pittsburgh, Pennsylvania.

1 Introduction

The security problem for mobile code or for component software is this: an untrusted program (or program fragment) is to execute in a host environment (the *code consumer*), and we want to ensure that it will do no harm. Proof Carrying Code (PCC) [1] is a framework for solving this problem by providing such assurances to the host. In the PCC framework the code consumer advertises a *safety policy* which specifies the logic in which it will accept proofs, the regions of readable or writable addresses, and so on. The code producer must construct a proof that the machine-language program satisfies the safety policy; the proof might be generated using hints from the compiler that generated the code. This proof along with the code is communicated to the host environment and the host verifies it before executing the code. PCC has significant advantages over other approaches that address the same problem (such as software fault isolation [6] or byte code interpretation [7]): no performance penalty is taken since the code is run at native speeds, and the proofs are performed on native machine code so no unsoundness can be introduced in the translation (or compilation) from the proved program to the one that will actually execute. For well-chosen safety policies, the proofs can be generated completely automatically.

In Appel and Felty [5] we gave an overview of our PCC system and described how it differs from the approach taken by Necula [2]. Instead of building type-inference rules into the safety policy, we model types as defined predicates using the primitives of ordinary logic; we prove typing rules as lemmas, and show how to model a wide variety of type constructors. This way the PCC safety policy is independent of the code producer's programming language and type system. The machine description semantics are moved from the verification-condition generator to the safety policy. More specifically our safety policy consists of the following:

1. The logic: a fairly standard higher order logic[1] ($\mathcal{L}$) consisting of eight inference rules for the logic and twenty-nine for arithmetic (with addition and multiplication taken as primitives).
2. The machine code syntax and semantics: this is encoded as the definition of the `step` relation ($\mapsto$) that describes the syntax and semantics of the machine. `Step` formally captures the notion of a single instruction execution. These axioms also define the `decode` relation that completely specifies instruction opcodes and operands (machine syntax) for all legal machine-code instructions.
3. Safety constraints: these are statements[2] in $\mathcal{L}$ that describe general properties of the runtime system (such as readable and safe-to-jump memory locations). They may also contain typing judgments for the initial contents of the register bank.

The small size of the logic is one of the major advantages of our approach. It contains no inference rules on types and no Hoare-logic rules for instructions (thus avoiding all complications due to substitution). Since it is so small, the proof checker can be likewise small. Thus the trusted computing base (TCB) can be verified easily (either by hand or through other means). A small TCB is the essence of PCC.

To simplify the presentation of the following sections we will use the *toy* machine (from [5]), a word-addressed 16-bit CPU. Its instruction set is presented in figure 1. Our system currently works with two other machine architectures (Sparc and Mips) and when appropriate we will also use examples from these.

[1] Our logic $\mathcal{L}$, is a sublogic of the Calculus of Constructions [11] and of the logic used in the HOL theorem prover [12], so our proofs can be checked in either Coq or HOL. Our current implementation uses Twelf [4].

[2] We offer a brief introduction to the syntax of our object logic: A metalogic (Twelf) type is a `type`, and an object-logic type is a `tp`. Object-logic types are constructed from `num` (the type of rationals), `form` (the type of formulas) and the `arrow` constructor. Object-level terms of type T have type (`tm` T) in the metalogic. Terms of type (`pf` A) are terms representing proofs of object formula A. The term `lam` $[x]\, F(x)$ is the object-logic function that maps x to $F(x)$ and `@` is the application operator for λ-terms. See Appel and Felty [5] for more details.

Instruction	Fields				Effect
add	0	d	$s1$	$s2$	$r_d := r_{s1} + r_{s2}$
addi	1	d	s	c	$r_d := r_s + \texttt{sign_ext}(c)$
load	2	d	s	c	$r_d := m[r_s + \texttt{sign_ext}(c)]$
store	3	$s1$	$s2$	c	$m[r_{s2} + \texttt{sign_ext}(c)] := r_{s1}$
jump	4	d	s	c	$r_d := r_{pc}\,;\, r_{pc} := r_s + \texttt{sign_ext}(c)$
bgt	5	$s1$	$s2$	c	if $r_{s1} > r_{s2}$ then $r_{pc} := r_{pc} + \texttt{sign_ext}(c)$
beq	6	$s1$	$s2$	c	if $r_{s1} = r_{s2}$ then $r_{pc} := r_{pc} + \texttt{sign_ext}(c)$

Fig. 1. The toy machine instruction set.

2 Overview

Our focus in this paper is twofold: concise axioms modeling machine architectures, and efficient proofs using those axioms.

We will describe in detail our `step` relation and show how it succinctly captures the syntax and semantics of real machines. Since it is by far the largest piece of our safety policy we are of course concerned about its correctness. To this end we will show how parts of it can be automatically generated from existing systems. Here we tackle the syntax of machine instructions using machine descriptions from the New Jersey Machine Code Toolkit [8]. We also show how to automatically generate proofs of correspondence between machine code integers and statements involving the `decode` relation.

We will describe the engineering aspects of generating small proofs of safety. Program safety is proved using a coinduction theorem based on progress and preservation of an invariant. We construct invariant expressions whose size is linear in the number of program instructions, and structure the progress and preservation proofs so that – modulo the parts that will have to be built by our tactical theorem prover – they are linear in size. In building these invariants we need to use the weakest preconditions of instructions and we will show how to automatically generate lemmas for a Hoare logic of machine language from the `step` relation. Our safety proofs will be linear-sized trees of applications of these Hoare lemmas.

Figure 2 shows our system operating on a small program that computes the sum of a linked list of integers. The goal of the system is to prove that the initial machine configuration (`IMC`) is `safe`, in symbols the following theorem:

$$\texttt{IMC}(r_0, m_0) \to \texttt{safe}(r_0, m_0)$$

where

$$\texttt{IMC}(r, m) := m(100) = 8976 \wedge \cdots \wedge m(105) = 24859$$
$$\texttt{safe}(r, m) := \forall r', m'\ (r, m \mapsto^* r', m') \to \exists r'', m''(r', m' \mapsto r'', m'').$$

The `IMC` describes parts of memory at the moment the program will run (in this case only the part containing the program itself). The `step` relation $r, m \mapsto r', m'$

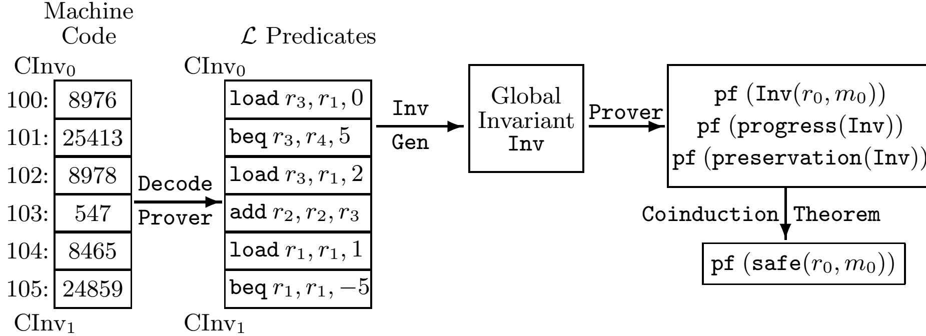

Fig. 2. Generating safety proofs.

formally describes a single instruction execution, i.e. given a machine at state (r, m), after execution of the instruction found at $r(\mathtt{pc})$, the machine will be at state (r', m'). The `safe` property states that no matter how far the execution proceeds, it never gets stuck, i.e. executes an illegal instruction or performs an illegal fetch.

The PCC system is presented with a list of machine code instructions (i.e. integers). The instruction stream is fed through the `decode-prover` whose job is to discover the instruction each integer represents, and to produce the symbolic representation of each instruction – which is a predicate that describes the instruction's semantics. The `decode-prover` also produces proofs of this correspondence. Following this, the predicates are fed into the `invariant-generator` which builds the global invariant to be used in the coinduction proof. Constructing invariants is not computable in general, so the prover requires hints in the form of local loop invariants decorating the targets of backward branches. Once the global invariant is built we must prove the three preconditions of the coinduction theorem[3] (see figure 3) in order to apply it. This is done by the `prover` and given the three proofs we apply the rule to finally establish $\mathtt{safe}(r_0, m_0)$.

3 Machine Semantics

In this section we show that the semantics of machine instructions can be easily and concisely expressed in higher order logic. We begin by explaining the idea using the toy machine, and then explore the problems in defining a semantic description of a real CPU.

Each instruction defines a relation between the machine state (registers, memory) before and after its execution. We treat both the memory and register bank as functions from integers to integers. Each instruction then becomes

[3] A note on notation: in the interest of brevity we will sometimes use mathematical notation when presenting `Twelf` terms.

$$\texttt{progress}(\texttt{Inv}) := \forall r, m\ \texttt{Inv}(r,m) \rightarrow \exists\, r', m'\ (r, m \mapsto r', m')$$

$$\texttt{preservation}(\texttt{Inv}) := \forall r, m, r', m'\ \texttt{Inv}(r,m) \wedge (r, m \mapsto r', m') \rightarrow \texttt{Inv}(r', m')$$

$$\frac{\texttt{Inv}(r,m) \qquad \texttt{progress}(\texttt{Inv}) \qquad \texttt{preservation}(\texttt{Inv})}{\texttt{safe}(r,m)}$$

Fig. 3. The coinduction theorem.

$$\begin{aligned}
\texttt{upd}(f,d,x,f') &:= \forall z \text{ if } (z = d) \text{ then } f'(z) = x \text{ else } f'(z) = f(z)\\
\texttt{i_add}(d,s_1,s_2)(r,m,r',m') &:= \exists \mathit{sum}\ \texttt{plus_mod16}(r(s_1), r(s_2), \mathit{sum}) \wedge\\
&\qquad\qquad \texttt{upd}(r,d,\mathit{sum},r') \wedge \texttt{no_mem_change}(m,m')\\
\texttt{i_load}(d,s_1,c)(r,m,r',m') &:= \exists \mathit{cext}, \mathit{addr}\\
&\qquad \texttt{sign_ext}(3,c,\mathit{cext}) \wedge\\
&\qquad \texttt{plus_mod16}(r(s1),\mathit{cext},\mathit{addr}) \wedge\\
&\qquad \texttt{upd}(r,d,m(\mathit{addr}),r') \wedge\\
&\qquad \texttt{readable}(\mathit{addr}) \wedge \texttt{no_mem_change}(m,m')
\end{aligned}$$

Fig. 4. Semantics of the `add` and `load` instruction of the toy machine.

a predicate which takes (r, m, r', m') as input, and holds when the instruction can safely take state (r, m) to (r', m'). In figure 4 we show the terms expressing the semantics for the "add" and "load" instructions of the toy machine. The `Twelf` term `i_add` (what we will call a *constructor* in section 4) expects three arguments (d, s_1, s_2) and returns a predicate of type `instr`, defined as:

$$\texttt{instr} = \texttt{regs} \rightarrow \texttt{mem} \rightarrow \texttt{regs} \rightarrow \texttt{mem} \rightarrow \texttt{form}.$$

It is this predicate that we view as the semantics of the instruction. Thus for the add instruction, $\texttt{i_add}(d, s_1, s_2)$ holds when for some integer *sum*, the following three equations hold:

$$\begin{array}{l}
\mathit{sum} = (r(s_1) + r(s_2)) \bmod 2^{16}\\
\forall x \text{ if } (x = d)\ \text{ then }\ r'(x) = \mathit{sum}\ \text{ else }\ r'(x) = r(x)\\
\forall x\ m(x) = m'(x).
\end{array}$$

The situation is similar for the semantics of the "load" instruction. But we wish to consider a program safe only if all of its memory accesses are within a specified region. Therefore our step relation admits only a subset of executable load instructions: those that load from `readable` addresses. The designer of the

safety policy must provide axioms that define the `readable` predicate. In general the semantics of each instruction must enforce the proper conditions under which the instruction can be executed. For "add" there are no such conditions; we can always add two numbers.

Real hardware can be a lot more complex than our simplistic toy machine. On a modern CPU one has to deal with the issues of delayed branches, address alignment, stores and loads of different sizes, condition registers, sign extension, instructions with multiple effects, and ALU operations not directly expressible in our arithmetic, to mention just a few. We claim that all of these can be handled relatively easily with the right set of abstractions and definitions. Space restrictions only allow us to deal with a representative subset here. We will use the Sparc CPU in the presentation.

- Condition Registers: We model condition registers exactly as we model physical registers. We assign a number to each of them that is outside the range of representable register numbers and refer to them exactly the same way we refer to regular registers. Instructions that need to modify individual bits do so by the use of appropriate definitions (see the `bits` predicate below).
- Delayed Branches: In order to keep their deep pipelines filled, some modern CPUs have introduced the notion of a delayed branch. On such CPUs one (or more) of the instructions following a branch will be executed even if the branch is taken, before the CPU starts executing instructions from the target address. We will assume a single instruction delay slot (the solution can be easily generalized to a delay slot of n instructions). We introduce another register called the next program counter[4] (`npc`) which holds the address at which the `pc` will be next. In the semantics of a branch instruction, if the branch is to be taken we simply set $r(\texttt{npc}) = \texttt{target}$ and the `step` relation takes care of updating $r(\texttt{pc})$ to $r(\texttt{npc})$ at the appropriate time.
- Address Alignment: Machine addresses have to be properly aligned depending on the instruction that uses them. Using the $\texttt{bits}(r, l, v, w)$ predicate (which holds when the value in the binary representation of w between bits r and l equals v; in symbols $\texttt{bits}(r, l, v, w) \Leftrightarrow v = \left\lfloor \frac{w}{2^r} \right\rfloor \bmod 2^{r-l+1}$) we can easily express such constraints. In a load-word instruction for the Sparc for instance we would insist that $\texttt{bits}(0, 1, 0, \textit{address})$ holds.
- Stores/Loads: We chose to model memory by a function $m : \texttt{num} \to \texttt{num}$ that we define only on word-aligned addresses. This way we avoid the complications of modifying individual bytes in a word. When we wish to store a byte quantity, the entire word must be fetched from memory, the byte spliced into it, and then stored back in memory. For load we have a similar situation. With the appropriate definitions all these operations can be specified painlessly. With careful selection of predicates most of them can be shared between the load and store instructions. One such example is the predicate `form_address` below. It computes a word aligned address and offset from an

[4] This is in fact how the hardware manages delayed branches. Some machines make the `npc` register explicit in the specifications [10].

unaligned one, and ensures that the original address was well aligned with respect to the size of the value we are trying to load/store.

$$\begin{aligned}&\texttt{form_address}(u_addr, alignment_bit, addr, offset, size) :=\\&\quad \texttt{bits}(0, alignment_bit, offset, u_addr) \wedge\\&\quad \texttt{minus_mod32}(u_addr, offset, addr) \wedge \texttt{modulo}(offset, size, 0)\end{aligned}$$

– Arithmetic Operations: Some of the arithmetic operations performed by modern CPUs are not directly expressible as functions in our logic. We cannot, for example, write the function that computes the bitwise "exclusive or" of two integers since our arithmetic primitives include only addition and multiplication and we have no recursion at the object level. Such operations are however, trivially expressed as relations (predicates). Here is for instance the `xor` predicate:

$$\begin{aligned}\texttt{xor}(a, b, c) = \forall i\, \exists x, y, r\ &\texttt{bits}(i, i, x, a) \wedge \texttt{bits}(i, i, y, b) \wedge\\&(\texttt{if}\ x = y\ \texttt{then}\ r = 0\ \texttt{else}\ r = 1) \wedge \texttt{bits}(i, i, r, c)\end{aligned}$$

Factoring via Higher-order Predicates. Machine instruction sets are highly factored, both in syntax and semantics. Consider for instance the ALU operations of any modern RISC chip. The ALU takes its input from two registers (or a register and a constant) and produces the result in another. The only difference between instructions is the operation performed. Our use of higher order logic allows us to exploit such factoring very effectively. We find the commonalities in families of instructions (even between families as in the load/store case above), factor those out and reuse well-chosen definitions. Here is an example from the Sparc. The definition of `i_aluxcc` is reused to define 23 different instructions. Argument `with_carry` specifies whether the instruction operates with a "carry", `modifies_icc` specifies whether it modifies the integer condition codes, and `func` is the predicate describing the operation performed by the instruction.

```
alu_fun = num arrow num arrow num arrow form.

i_aluxcc : tm (form arrow form arrow alu_fun arrow alu_typ) =
 lam3 [with_carry : tm form][modifies_icc : tm form][func : tm alu_fun]
  lam3 [rs1][reg_imm][rd]
   lam4 [r][m][r'][m']
    (exists3 [v][v'][r'']
     (load_reg_imm @ r @ reg_imm @ v) and
     (compute_with_carry @ with_carry @ func @ r @ rs1 @ v @ v') and
     (compute_cc @ modifies_icc @ r @ r'' @ v') and
     (upd_reg @ r'' @ rd @ v' @ r') and
     (no_memory_change m m')).

i_AND    = i_aluxcc @ false @ false @ and_oper.
i_ANDcc  = i_aluxcc @ false @ true  @ and_oper.
  "      "    "     "    "     "             -- 21 cases omitted.
```

Moreover we exploit commonality between machines. Many of our definitions that deal with the mechanics of splicing values into words, sign extension, and

arithmetic operations, are shared between semantic descriptions of different machines. Higher-order predicates are useful in expressing this kind of sharing; note that the `i_aluxcc` predicate above is higher order.

4 The Decode Relation

On a von Neumann machine, each instruction is represented in memory by an integer. The `decode` relation makes this notion precise. It is a predicate of four arguments (m, w, i, s) stating that address w in memory m contains the encoding of instruction i that has size s. Modern microprocessors have hundreds of instructions and to construct this relation manually would be a daunting task. The observation that the information we wish to encode is very similar to the information used by an assembler/disassembler led us to look for an automatic way to generate the relation.

The New Jersey Machine Code Toolkit [8] helps programmers write applications that process machine code – assemblers, disassemblers, code generators, and so on. The toolkit lets programmers encode and decode machine instructions symbolically. It transforms symbolic manipulations into bit manipulations, guided by a specification that defines mappings between symbolic and binary representations of instructions. Of interest to us here is the specification language (called SLED) for encoding and decoding assembly-language representations of machine instructions [9]. It is a concise, elegant, and semantically well-founded language, a fact that has made the translation into logic fairly painless. In fact our translation into $\mathcal{L}$ can be viewed as a semantics for the language.

Before describing our encoding of SLED into $\mathcal{L}$ we offer a brief introduction to the language. In order to accommodate machines with non-uniform instruction sizes the toolkit works with streams of *tokens* instead of instructions. Each instruction consists of one or more tokens. Tokens are further partitioned into *fields* which are sequences of contiguous bits within a token. *Patterns* in SLED serve two purposes: firstly they are used to constrain the division of streams into tokens, and secondly to constrain the values of fields in those tokens. Patterns can be combined with various operators to produce new patterns. The toolkit is concerned with two representations of machine instructions: machine code and assembly language. *Constructors* are used to connect the two representations.

Figure 5 presents a SLED specification of the toy machine architecture. The first two lines specify the 16-bit token `instr` and its fields: `op` which occupies bits 12 to 15, `rd` which occupies bits 8 to 11, and so on. The next line specifies a list of patterns (`add`, . . . , `beq`,) and for each one, it constrains the `op` field to have the value 0, . . . , 6 respectively. Finally the `constructors` clause specifies the toy machine instructions. A special toolkit shortcut is used here: if no pattern is specified in the constructor definition then all the names used in the constructor must be either patterns or fields and their conjunction is taken to be the pattern that will be generated by the constructor. In the next subsections we show how to map fields, patterns, and constructors into higher-order logic.

```
fields of instr (16)
  op 12:15  rd 8:11  rs1 4:7  rs2 0:3  c 0:3

patterns [add addi load store jump bgt beq] is op = 0 to 6

constructors  add    rd, rs1, rs2
              addi   rd, rs1, c
              load   rd, rs1, c
              store  rd, rs1, c
              jump   rd, rs1, c
              bgt    rd, rs1, c
              beq    rd, rs1, c
```

Fig. 5. The SLED specification for the toy machine.

4.1 Mapping fields into $\mathcal{L}$

The definition of the `bits` predicate (from section 3) makes it straightforward to map fields into $\mathcal{L}$. All that it takes is to supply the right and left bit specifiers of each field to this predicate. Since our definitions are curried, defining fields in $\mathcal{L}$ becomes very convenient and almost as terse as it is in SLED. For the toy machine the first two fields are translated as follows:

```
op  = bits @ (const 12) @ (const 15).
rd  = bits @ (const 8)  @ (const 11).
```

The `op` predicate expects two integers as arguments (`v`, `word`), and it holds when `v` is equal to the integer between the 12th and 15th bit of `word`.

4.2 Mapping Patterns into $\mathcal{L}$

Patterns in SLED constrain both the division of streams into tokens and the values of the fields in those tokens. They are composed of constraints on fields. Patterns can be combined using various operators to form other patterns. The RISC machine descriptions we have considered so far contain only conjunction and disjunction operators, and those are the ones we currently translate. We expect no problems in translating the rest when we choose to deal with CISC machines. Conjunction is used to constrain multiple fields within a single token. When `p` and `q` are patterns, the pattern "`p & q`" matches if both `p` and `q` match. For example, in the SLED description for Sparc [8] we find:[5]

```
patterns
 [ TABLE_F2 CALL TABLE_F3 TABLE_F4 ] is op = {0 to 3}
```

[5] This is another example of the terseness of SLED. In the definitions of these patterns Ramsey [9] makes use of a SLED feature called *generating expressions*, which describe ranges of lists either explicitly or implicitly as shown in the example.

```
[ UNIMP Bicc SETHI FBfcc CBccc ]    is TABLE_F2  &  op2 = [0 2 4 6 7]
NOP                                 is SETHI  &  rd = 0  &  imm22 = 0
```

In the first line TABLE_F2 is defined as the pattern that wants the op field to equal zero, in the second line TABLE_F2 is used in the definition of SETHI which is defined as the conjunction of patterns TABLE_F2 and op2 $= 4$. Finally in the last line pattern SETHI is used in the definition of the NOP pattern.[6] Patterns of this kind are very easy to translate into $\mathcal{L}$. We make use of a higher level infix "and" operator defined as:

```
num_pred = num arrow form.
&& : tm num_pred -> tm num_pred -> tm num_pred =
 [p1][p2] lam [w] (p1 @ w) and (p2 @ w).
```

Given && it is now easy to deal with conjunctive patterns by simply "anding" together the different conjuncts after mapping each of them to an $\mathcal{L}$ predicate. The example above then becomes:

```
p_TABLE_F2 = op @ (const 0).
p_SETHI    = p_TABLE_F2 && (op2 @ (const 4)).
p_NOP      = p_SETHI && (rd @ (const 0)) && (imm22 @ (const 0)).
```

Disjunction in patterns is usually used to group patterns for related instructions. In the following example from the Sparc SLED we use disjunction to group the logical, shift, and arithmetic instructions into three groups, which are then disjunctively combined into a pattern that matches any ALU instruction.

```
patterns
  logical is AND | ANDcc | ANDN | ANDNcc | OR | ORcc | ORN | ORNcc | ...
  arith   is ADD | ADDcc | ADDX | ADDXcc | TADDcc | TADDccTV | ...
  shift   is SLL | SRL   | SRA
  alu     is logical | arith | shift
```

Disjunction patterns are mostly used as opcodes to constructors and we show how we deal with them in the next subsection.

4.3 Mapping Constructors into $\mathcal{L}$

A constructor maps a list of operands to a pattern which stands for the binary representation of an operand or an instruction. There are two kinds of constructors, typed and untyped. Typed constructors generate instruction operands and untyped constructors generate instructions. The following definition from the Sparc specification is an example of a typed constructor:

```
constructors  imode simm13! : reg_or_imm  is  i = 1 & simm13
              rmode rs2     : reg_or_imm  is  i = 0 & rs2
```

[6] A NOP on the Sparc is a SETHI on r_0 with value 0, and since r_0 is hardwired to zero it has no effect.

Each line in the definition of a constructor specifies the opcode, the operands, the constructor type, and matching pattern. Usually the opcode is the constructor's name (as in this case). Constructors generate disjoint sum types. In the above, `imode` : `num` → `reg_or_imm` is the canonical injection from `num` into the `reg_or_imm` type – likewise for `rmode` : `num` → `reg_or_imm`. The type is defined implicitly at first use. Each constructor is applicable when the pattern following the `is` keyword is satisfied.

The above constructor definition captures the following idiom: many Sparc instructions (such as `add r1, reg_or_imm, r2`) take either a register or a constant as one of their arguments. The hardware differentiates between the two instances by the value of bit 13 (field `i`) in the representation of the instruction. Depending on the value of `i`, either `imode` or `rmode` can be applied, giving in each case a `reg_or_imm`.

We translate a typed constructor into $\mathcal{L}$ as follows. We first create a new object-logic type for the constructor type. For each of the injective arrows (`imode` and `rmode` above) we create an injective `Twelf` term (`c_imode` and `c_rmode`), as well as a discriminator term (`p_imode` and `p_rmode`). Finally we generate a predicate that decides the type itself (`p_reg_or_imm`), i.e. a term that when given an object of that type and a word decides whether that word contains the given object. We show these terms for the example below:

$$\begin{array}{l}\texttt{reg_or_imm} : \texttt{tp}\\ \texttt{c_imode} : \texttt{num} \longrightarrow \texttt{reg_or_imm}\\ \texttt{c_rmode} : \texttt{num} \longrightarrow \texttt{reg_or_imm}\end{array}$$

$$\begin{array}{ll}\texttt{p_imode}(\mathit{simm}) & := \texttt{i}(1)\ \texttt{\&\&}\ \texttt{simm13}(\mathit{simm})\\ \texttt{p_rmode}(s_2) & := \texttt{i}(0)\ \texttt{\&\&}\ \texttt{rs2}(s_2)\end{array}$$

$$\begin{array}{l}\texttt{p_reg_or_imm}(\mathit{regimm}, \mathit{word}) :=\\ \quad (\exists \mathit{simm}\ \texttt{p_imode}(\mathit{simm}, \mathit{word}) \wedge \mathit{regimm} = \texttt{c_imode}(\mathit{simm})) \vee\\ \quad (\exists s_2\ \texttt{p_rmode}(s_2, \mathit{word}) \wedge \mathit{regimm} = \texttt{c_rmode}(s_2))\end{array}$$

Untyped constructors represent the instructions themselves. Their translation into $\mathcal{L}$ is not much different from the typed case so we omit it.

Factoring via Higher-order Predicates. The extensive factoring present in the SLED specifications (through the wide use of "or" patterns) carries over to the translated higher-order logic terms. When translating a constructor that uses an "or" pattern as an opcode, we do not generate a unique term for each instruction but instead build just a single term that describes all of them. This way we preserve SLED's economy of syntax. Here is an example for the ALU instructions of the Sparc shown earlier. The constructor in the spec is the following:

```
constructors  alu rs1, reg_or_imm, rd
```

and we generate the following two terms for it:

$\texttt{p_alu_aux}(p_i, i_cons, s_1, \mathit{regimm}, s_2, \mathit{word}, i) :=$

$$\begin{aligned}
\texttt{p_instr}(word, i) &:= (\texttt{p_add} \mathbin{||_2} \texttt{p_addi} \mathbin{||_2} \texttt{p_load} \mathbin{||_2} \texttt{p_store} \mathbin{||_2} \\
&\quad\ \texttt{p_jump} \mathbin{||_2} \texttt{p_bgt} \mathbin{||_2} \texttt{p_beq})(word, i) \\
\texttt{decode}(m, w, i, s) &:= (s = 1) \wedge \texttt{p_instr}(m(w), i) \\
\texttt{step}(r, m, r', m') &:= \exists\, i, r'', size\ \texttt{decode}(m, r(pc), i, size) \wedge \\
&\qquad\qquad\quad \texttt{upd}(r, pc, r(pc) + size, r'') \wedge \\
&\qquad\qquad\quad i(r'', m, r', m')
\end{aligned}$$

Fig. 6. The `decode` and `step` relations for the toy machine.

$$\begin{aligned}
&\quad (p_i \ \texttt{\&\&}\ \texttt{rs1}(s_1)\ \texttt{\&\&}\ \texttt{p_reg_or_imm}(regimm)\ \texttt{\&\&}\ \texttt{rs2}(s_2))(word) \wedge \\
&\quad i = i_cons(s_1, regimm, s_2) \\
\texttt{p_alu}(word, i) &:= \exists s_1, rimm, s_2\ (\texttt{p_alu_aux}(\texttt{p_AND}, \texttt{i_AND}) \quad ||_5 \\
&\qquad \texttt{p_alu_aux}(\texttt{p_ANDcc}, \texttt{i_ANDcc}) \ ||_5 \\
&\qquad \vdots \quad \vdots \quad \vdots \quad \vdots \quad \text{-- 35 cases omitted} \\
&\qquad \texttt{p_alu_aux}(\texttt{p_SRA}, \texttt{i_SRA}))(s_1, rimm, s_2, word, i)
\end{aligned}$$

where `p_AND` is the opcode pattern, `i_AND` is the instruction constructor and likewise for the rest of them. Here again we make use of a higher level "or" ($||_5$) operator to factor out the common arguments to the auxiliary predicate.

Our `decode-generator` is a 3200-line ML program that operates directly on SLED specifications. Since it generates a large portion of our safety policy it ought to be considered trusted code (along with the SLED specifications). We feel that this is a small enough program that can be thoroughly and convincingly debugged into correctness. Furthermore its output is human readable and only a constant factor bigger (between 2x and 3x) than the original SLED specification. Thus the output can easily be inspected and debugged directly. The program currently does not share any code with the New Jersey Machine Code Toolkit although the front-end code and some of the analysis that the two programs perform could be shared. We plan to investigate an integration of the two tools in the future.

4.4 The Decode and Step Relations

We are finally in a position to present the `decode` relation for the toy machine (see figure 6). After all the instruction predicates have been emitted, the `decode-generator` creates a predicate for the top-level token (i.e. `instr` in the case of the toy spec). This predicate is the disjunction of all the instruction predicates (modulo factoring as described above). `Decode` is then defined in terms of this predicate. Figure 6 also shows the `step` relation for the toy machine. It is a predicate mapping the machine state $(r, m) \mapsto (r', m')$ by requiring the existence

of an instruction i, a register bank r'', and an integer *size* such that location $r(pc)$ in memory m decodes to i, updating the register bank r with the next `pc` produces r'', and finally instruction i safely maps (r'', m) to (r', m'). `Step` models the meaning of a single instruction execution.

5 Machine Code Proofs

In this section we discuss some of the issues in generating the proofs used in the coinduction theorem (figure 3).

5.1 Hoare-logic predicates for local invariants

In the Floyd-Hoare logic one tries to establish statements of the form $P\,\{S\}\,Q$, where S is a program statement, and P, Q are logical formulae. $P\,\{S\}\,Q$ means that if P holds, and S executes to completion, then Q holds. The logic specifies a set of axioms and inference rules that allow the deduction of statements of this form. The assignment axiom for instance states: $\vdash P[E/V]\,\{V{:=}E\}\,P$. In our framework we have no such axioms or rules; nevertheless, our preservation statement (in figure 3) bears a striking resemblance to a Hoare judgment. What is stated there is in essence equivalent to:

$$Inv(r,m)\ \ \{(r,m) \mapsto (r',m')\}\ \ Inv(r',m') \tag{1}$$

i.e. if the invariant holds at (r,m), then it must hold at the new state (r',m') at which we were taken by the execution of some instruction (a single step). This similarity is of course no accident; we wish to exploit the well understood theory of Hoare logic in order to construct the weakest preconditions that will allow us to prove preservation.

Our invariant (as described in detail in previous work [5]) is in essence a disjunction of statements[7] of the form $r(pc) = n \wedge \mathtt{decode}(m,n,i,1) \wedge I_n(r,m)$ where i is the instruction found at $m(n)$ and I_n is the local invariant at n. To make the situation more concrete assume that at $r(pc)$ we find instruction $\mathtt{add}(r_1,r_2,r_3)$ $(r_1 := r_2 + r_3)$, and that after completion of this instruction, we wish predicate $Q(r,m)$ to hold at the new state. The question now is what should I_n be in order to be able to prove equation 1, or equivalently the statement:

$$\begin{array}{c} r(pc) = n \wedge \mathtt{decode}(m,n,i,1) \wedge i = \mathtt{add}(r_1,r_2,r_3) \wedge I_n(r,m) \wedge \\ (r,m \mapsto r',m') \rightarrow Q(r',m'). \end{array} \tag{2}$$

[7] The invariant presented in Appel and Felty [5] could grow exponentially large for certain kinds of programs. By the use of appropriate higher-order definitions we have remedied this problem and now produce invariants that are always linear in the number of program instructions and in the size of the compiler-inserted loop invariants (see subsection 5.2). The structure of the new invariant is beyond the scope of this paper. The discussion in this section is equally applicable to either kind of invariant.

It is not difficult to see that one such I_n is the following: $Q(r, m)[(r_2 + r_3)/r_1]$, i.e. the formula we get after applying the assignment axiom of Hoare logic to the postcondition $Q(r, m)$. In building the invariant though, we do not wish to perform substitution of terms for two main reasons. Firstly, if we are not careful during substitution the local invariants could grow exponentially large.[8] The goal is to end up with small proofs of safety; an exponentially large theorem is unlikely to have a small proof. Secondly, our logic does not contain axioms that express term substitution; such axioms would render the proof checker more complex and would defeat our efforts for a small TCB. Instead we view substitution as a relation between terms and express the notion concisely by higher-order definitions. These definitions allow us to express $I_n(r, m)$ in terms of $Q(r, m)$ in such a way that the size of local invariants stays constant, and substitution is completely avoided (at this stage). We define predicate `let_upd` in terms of `upd` (introduced in figure 4) as follows:

$$\texttt{let_upd}(r, a, v, f) := \forall r' \; \texttt{upd}(r, a, v, r') \to f(r').$$

Predicate `let_upd` specifies that for any function r' that updates r at a with value v, $f(r')$ must hold (we note that there is exactly one such r'; `upd` is deterministic).

Using this predicate we can succinctly express the weakest precondition for each of our instructions. Below we show the term for the add instruction; compare `hx_add` with the semantics of `add` shown in figure 4.

$$\begin{aligned}\texttt{hx_add}(d, s_1, s_2, post)(r, m) := \exists sum \; \texttt{plus_mod16}(r(s_1), r(s_2), sum) \wedge \\ \texttt{let_upd}(r, d, sum, \lambda r'.post(r', m))\end{aligned}$$

The last argument to `hx_add` is the postcondition, and the return value is a predicate on (r, m) expressing the weakest precondition for the add. Our system currently generates all the predicate transformers (such as `hx_add` above) automatically for each instruction from the step relation of each machine. The program performing the translation is not part of the TCB; if there is a bug in it then we will simply fail to prove preservation.

In proving preservation we will have to prove a statement very similar to that in equation 2 for each instruction in our program (but see section 5.2). Such statements can be proved once and for all as lemmas and applied each time the corresponding instruction is encountered. The extensive use of such lemmas will have a profound effect on the size of our safety proofs. We have currently proven such lemmas for all the instructions of the toy machine by hand. It is our intention to generate them and their proofs automatically from the `step` relation of each machine.

[8] Consider for example the program $(r_2 := r_1 + r_1; r_3 := r_2 + r_2; r_4 := r_3 + r_3)$ with postcondition $Q(r_4)$. Its weakest precondition is $Q(((r_1 + r_1) + (r_1 + r_1)) + ((r_1 + r_1) + (r_1 + r_1)))$. The size of the argument to Q grows by a factor of two for each assignment.

5.2 Domain Specific Proofs

Precondition strengthening (shown below) is another rule of Hoare logic.

$$\frac{P' \rightarrow P \qquad P\,\{S\}\,Q}{P'\,\{S\}\,Q} \tag{3}$$

It states that if $P\,\{S\}\,Q$ then one may replace P by a stronger predicate. This scenario occurs when we deal with program loops, as we explain next. Safety proofs for programs with loops require the use of loop invariants. Construction of loop invariants is not computable in general, so our theorem prover requires hints in the form of typing judgments at every location that is the target of a backward jump. At such locations though, our `invariant-generator` would have computed a local invariant I_n (this is the weakest precondition of the instruction – see subsection 5.1). We wish to replace I_n by H_n (the typing hint at that location) as the precondition of that instruction, but in order to be able to do that we must establish that $H_n \rightarrow I_n$. After that, a lemma application similar to rule 3 allows us to conclude $H_n\{S\}\,Q$. We are building a tactical theorem prover that understands the structure of types and is able to produce such proofs. The "linear size of proofs" discussed in this paper excludes the size of the strengthening proofs. These are not necessarily large but a description of their structure is beyond the scope of this paper.

5.3 Decode Proof-Generation

Proofs involving the `decode` relation can be hard to generate since the definition itself is quite involved. Our `decode-prover` (see figure 2) is a `Twelf` logic program that analyzes the machine-code stream and not only discovers which instruction each integer represents but also produces a proof of this fact. More concretely, if integer n represents instruction i, we get a proof of statement `instruction`(n, i) from which a proof of `decode`(m, w, i, s) follows trivially (given a proof that $n = m(w)$). The `decode-prover` for the toy machine is about 600 lines of `Twelf`, currently hand written. We plan to generate the `decode-prover` itself from the SLED specification of each machine. Note that the `decode-prover` is not part of the TCB; any bug in it will simply produce an invariant from which it will be impossible to show preservation.

6 Related Work

There has been a large amount of work in the area of proofs of machine language programs using both first order [14] and higher order logics [15][16]. Some of this work was focused on proving the correctness of the compiler or the code generator (see for instance [13]). For a historical survey see Calvert [18]. The practice of proving the Hoare rules as lemmas (see subsection 5.1 and 5.2) in an underlying logic is widespread among the program-verification community [15][16][17].

Two pieces of work are most related to ours: Wahab [15] is concerned with correctness (not just safety) of assembly language programs. He defines a flow-graph language expressive enough to describe sequential machine code programs (he deals with the Alpha AXP processor). Substitution is a primitive operator and the logic contains rules detailing term equality under substitution. He proves the Hoare-logic rules as theorems and uses abstraction in order to massage the code stream and get shorter correctness proofs. The translation from machine code to the flow-graph language does not go through a "decode" relation. Also the use of substitution as a primitive makes this approach unsuitable for our purposes since it complicates the TCB.

Boyer and Yu [14] formally specify a subset of the MC68020 microprocessor within the logic of the Boyer-Moore Theorem Prover [19], a quantifier-free first order logic with equality. Their specification of the step relation is similar to ours (they also include a decode relation) but in their approach these relations are functions. The theorem prover they use allows them to "run" the step function on concrete data (i.e. once the step function is specified they automatically have a simulator for the CPU). Their logic, albeit first-order, appears to be larger than ours mainly because of its wealth of arithmetic operators (decoding can be done directly from the specification). Also their machine descriptions are larger than ours; the subset of the 68020 machine description is about 128K bytes while our description of the Sparc is less than half that size. Admittedly, the Motorola chip is much more complex than the Sparc, but we suspect that most of the size difference is attributed to our extensive use of factoring facilitated by higher order logic.

7 Conclusion and Future Work

We have shown how higher-order logic can be used to succinctly describe the syntax and semantics of machine instructions, in a manner that preserves the natural factoring of each architecture. Our `step` relation formally captures the notion of a single instruction execution. It consists mainly of two pieces: (1) the `decode` relation that specifies the syntax of machine instructions, and (2) axioms describing the semantics of each instruction by predicates mapping machine states to machine states. The `decode` relation is generated automatically from existing compiler tools. Large parts of the safety proof involving `decode` can be generated completely automatically. We explained how to build Hoare-logic predicate transformers from our `step` relation in order to simplify the construction of the global invariant, and how lemmas can be used to minimize the size of safety proofs involving this invariant. The system is implemented in `Twelf` [4] and all theorems have been mechanically checked.

We are building a PCC system that will be used to generate safety proofs for many different architectures. Building all the pieces of figure 2 for each machine would be a daunting and unrewarding task. We instead intend to generate most of the prover components shown in figure 2 completely automatically. Since the `decode-prover` is in essence a machine-code disassembler, we intend to gener-

ate it directly from the `decode` relation of each machine or alternatively from each machine's SLED specification. Note that the `decode-prover` not only disassembles but also builds proofs involving `decode`. The `invariant-generator` is again machine-instruction dependent and can also be generated directly from `decode` (we already generate the predicate transformers expressing the weakest precondition for each instruction automatically from `step`). It is our intention to automatically generate the Hoare-logic lemmas (of subsection 5.1) along with their proofs from `step` since there will be a large number of them and their proofs tend to be rather long. The proof of `preservation` (see figure 3) requires an inversion lemma for `decode`. We have not proved this lemma for any machine yet, but we expect the proof to be mundane and long (linear in the size of the instruction set). Our plan is to generate these proofs from `decode`. Finally we are working on a tactical theorem prover that will fill in parts of the proofs involving compiler inserted invariants at locations of backward branches (see subsection 5.2).

References

1. George Necula. Proof Carrying Code. In *The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,* pages 106-119, New York, January 1997. ACM Press.
2. George Ciprian Necula. Compiling with Proofs. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, September 1998.
3. Frank Pfenning. Logic Programming in the LF logical framework. In Gérard and Gordon Plotkin, editors, Logical Frameworks, pages 149-181. Cambridge University Press, 1991.
4. Frank Pfenning and Carsten Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In *the 16th International Conference on Automated Deduction.* Springer-Verlag, July 1999.
5. Andrew Appel and Amy Felty. A Semantic Model For Types and Machine Instructions for Proof-Carrying Code. In *the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '00)*, January 2000.
6. R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proc. 14th ACM Symposium on Operating System Principles,* pages 203-216, New York, 1993. ACM Press.
7. Tim Lindholm and Frank Yellin. The Java Virtual Machine Specification. Addison Wesley, 1997.
8. Norman Ramsey, Mary Fernandez. The New Jersey Machine-Code Toolkit. In *Proceedings of the 1995 USENIX Technical Conference,* pages 289-302, New Orleans, LA, Han. 1995.
9. Norman Ramsey, Mary Fernandez. Specifying Representations of Machine Instructions. In *ACM Transactions on Programming Languages and Systems,* pages 492-524 Vol. 19, No. 3, May 1997.
10. SPARC International, Inc. The SPARC Architecture Manual v. 8, Prentice-Hall, Inc. 1992.
11. Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation,* 76(2/3), pages 95-120, February/March 1988.

12. M. J. C. Gordon and T. F. Melham (editors). Introduction to HOL: A theorem proving environment for higher order logic, Cambridge University Press, 1993.
13. R. Milner and R. Weyhrauch. Proving Compiler Correctness in a Mechanized Logic. In *Machine Intelligence,* 7:51-70, 1972.
14. Robert S. Boyer and Yuan Yu. Automated Correctness Proofs of Machine Code Programs for a Commercial Microprocessor. In *the 11th International Conference of Automated Deduction,* pages 416-430. Springer-Verlag, 1992.
15. M. Wahab. Verification and Abstraction of Flow-Graph Programs with Pointers and Computed Jumps. Technical Report, University of Warwick, Coventry, UK.
16. M. Gordon. A Mechanized Hoare Logic of State Transitions. In *A Classical Mind: Essays in Honour of C. A. R. Hoare,* pages 143-159. Edited by A. W. Roscoe (Prentice-Hall, 1994).
17. M. Gordon. Mechanizing Programming Logics in Higher Order Logic. In *Current Trends in Hardware Verification and Automated Theorem Proving,* pages 387-439. Edited by G. Birtwistle and P. A. Subrahmanyam (Springer-Verlag, 1989).
18. David William John Stringer-Calvert. Mechanical Verification of Compiler Correctness. Ph.D. thesis, University of York, 1998.
19. Robert S. Boyer and J Strother Moore. A Computational Logic Handbook. Academic Press 1988.