Hoare Logic for Realistically Modelled Machine Code

Magnus O. Myreen and Michael J.C. Gordon

Computer Laboratory, University of Cambridge, Cambridge, UK

Abstract. This paper presents a mechanised Hoare-style programming logic framework for assembly level programs. The framework has been designed to fit on top of operational semantics of realistically modelled machine code. Many *ad hoc* restrictions and features present in real machine-code are handled, including finite memory, data and code in the same memory space, the behavior of status registers and hazards of corrupting special purpose registers (e.g. the program counter, procedure return register and stack pointer). Despite accurately modeling such low level details, the approach yields concise specifications for machine-code programs without using common simplifying assumptions (like an unbounded state space). The framework is based on a flexible state representation in which functional and resource usage specifications are written in a style inspired by separation logic. The presented work has been formalised in higher-order logic, mechanised in the HOL4 system and is currently being used to verify ARM machine-code implementations of arithmetic and cryptographic operations.

1 Introduction

Computer programs execute on machines where stacks have limits, integers are bounded and programs are stored in the same memory as data. However, verification of computer programs is almost without exception done using highly simplified models, where stacks and memory are unbounded, integers are arbitrarily large and the compilers are trusted to keep code and data apart. Proving properties of programs with respect to realistic models is generally avoided, since it is tedious as many of the common simplifying assumptions made by high-level programming logics tend to fit badly with realities of accurate low-level models. In this paper we present a programming logic that has been designed to fit on top of accurate models of machine languages.

We present a Hoare logic that has been carefully designed to accommodate many of the *ad hoc* restrictions and features of machine code: finite memory, data and code in the same memory space, the behaviour of status register, hazards of corrupting special purpose registers and some details that arise from hardware implementations. As an example of a restriction imposed by the underlying hardware, consider the following two seemingly equivalent implementations of the factorial program in ARM assembly. The example uses the ARM instructions `"MOV b, #1"` (set register `b` to `1`), `"MUL c, a, b"` (put the product of

O. Grumberg and M. Huth (Eds.): TACAS 2007, LNCS 4424, pp. 568–582, 2007.

the contents of registers `a` and `b` into register `c`, but see restrictions discussed shortly), `"SUBS a, a, #1"` (subtract `1` from register `a` and update status bits so that status bit `Z` is assigned the boolean expression `a-1=0`) and `"BNE L"` (jump to `L` if status bit `Z` is `0`).

```
    MOV   b, #1              MOV   b, #1
L:  MUL   b, a, b        L:  MUL   b, b, a
    SUBS  a, a, #1           SUBS  a, a, #1
    BNE   L                  BNE   L
```

The first implementation terminates with the factorial of `a` (modulo 2^{32}) in `b`, while the other one has an unpredictable outcome, `"MUL b, b, a"` is specified as 'unpredictable' for ARM in order to accommodate hardware optimisations [15]. Thus `"MUL c, a, b"` cannot be modelled as $c := a \times b$ without a side condition.

The judgments of our framework are total-correctness specifications that state the functional behaviour and resource usage of machine-code programs. We use a separating conjunction, similar to that of separation logic [13], in order to write concise specifications about resource usage as well as to avoid unwanted aliasing between special purpose registers (and normal registers as motivated above). Our specifications allow multiple code segments and use positioning functions to enable reasoning about mixtures of position independent code and position dependent code. As a result, procedures and procedural recursion is readily handled (without assuming an unbounded stack).

The Hoare triples described in this paper have been defined in higher-order logic. Rules for reasoning about them have been derived from the formal definitions of the Hoare triples, using the HOL4 system [6] (thus the rules are sound). We can reason about ARM machine code by instantiation of our framework's Hoare triples to a high-fidelity model of the ARM machine language. The specialisation of our framework to ARM machine code is presented in a companion paper [10]. Here we concentrate on the core ideas of our approach.

This paper is not the first to address the problem of verifying realistically modelled machine code. Some early work was done by Maurer [9], Clutterbuck and Carré [5] and Bevier [3]. Boyer and Yu [4] did impressive pioneering work on verifying machine code written for a commercial processor: they verified programs using the bare operational semantics of a model of the Motorola MC68020. Projects on proof-carrying code (PCC) [11] and particularly foundation PCC [1] have ignited new interest in verification of low-level code. Of work on PCC, Tan and Appel's work [16] is particularly relevant to this paper: they use a Hoare logic to reason about a detailed model of the Sparc machine language. As for most work on PCC, their aim is to address safety properties that can be proved automatically (e.g. type safety). Tan and Appel's approach is hampered by the requirement of an extensive soundness proof. Hardin, Smith and Young [7] verify machine code for Rockwell Collins AAMP7G using a form of symbolic simulation. Work by Klein, Tuch and Norrish [8] has similar goal as ours, but they reason at a higher level about realistically modelled C programs.

The remainder of this paper is organised as follows. Section 2 gives an overview of how our specifications relate to those of standard Hoare-triples and motivates

our design decisions. Section 3 contains the bulk of the material: it defines a Hoare triple for machine code, presents an example and shows how rules can be derived for procedures and procedure calls. Section 4 demonstrates how the framework can be instantiated to a given operational semantics of a machine language. Section 5 concludes with a summary.

2 Approach

This section motivates some key design decisions informally and gives an overview of the main ideas. The detailed definitions are given in the next section.

2.1 Basic Specifications

Our framework supports code specifications with multiple entries, multiple exits and multiple code segments, but for simplicity we start by considering specifications having single entry, single exit and single code segment. The full generality is described in Section 3.

Consider the ARM implementation of the factorial function given in the introduction. In classical Hoare logic, its specification could be written as follows with a side-condition:

$$\{(a = x) \wedge (x \neq 0)\}$$
$$\texttt{FACTORIAL}$$
$$\{(a = 0) \wedge (b = x!)\}$$

Side condition: The registers associated with a and b are distinct.

This specification is not satisfactory because it leaves many aspects unspecified. For example, it does not say whether the code modifies the status bits or what happens to the program counter.

We require specifications to mention each component of the state that might be altered during execution. That way we can easily see what is changed and what is not. Our approach is similar to that of separation logic [13,12], which assigns a memory footprint to each assertion. We make a stricter requirement: every state component (e.g. register, memory location, status bit) must appear in the footprint of an assertion. In our framework, the factorial program has the following specification where, for now, informally read $R\ a\ x$ as "register a has value x", $S\ b$ as "the status bits have value b", underscore ($\_$) as "some value" and $P * Q$, following separation logic, as "P and Q are true for disjoint parts of the state" (precise definitions of these concepts are given later).

$$\{R\ a\ x\ *\ R\ b\ \_\ *\ S\ \_\ *\ \langle x \neq 0 \rangle\}$$
$$\texttt{FACTORIAL}$$
$$\{R\ a\ 0\ *\ R\ b\ x!\ *\ S\ \_\ \}^{+4}$$

The superscript $^{+4}$ specifies that `FACTORIAL` increments the program counter by 4. The separating conjunction $*$ avoids the need for the side-condition, since the side condition is implied by the occurrence of $*$ between $R\ a\ x$ and $R\ b\ \_$ in the precondition.

2.2 Heterogeneous Specifications

Machine-code programs depend on a variety of different resources. Even in a simple setting we encounter registers, special registers, memory locations and various status bits. For this reason we treat all types of resources uniformly. Consider for instance the specification of the instructions STR (store) and DSTR (decrement-and-store). Read $M\ x\ y$ as "memory location x has value y".

$$\begin{array}{cc} \{R\ a\ x * R\ b\ y * M\ y\ \_\} & \{R\ a\ x * R\ b\ y * M\ (y{-}1)\ \_\} \\ \text{STR}\ b\ a & \text{DSTR}\ b\ a \\ \{R\ a\ x * R\ b\ y * M\ y\ x\}^{+1} & \{R\ a\ x * R\ b\ (y{-}1) * M\ (y{-}1)\ x\}^{+1} \end{array}$$

These specifications have a similar form to that of the factorial program, even though they specify the behavior of different types of resources.

Hoare-style reasoning can be applied to specifications. For example, DSTR given above can implement a stack push for a descending stack. We can state this with a specification $stack(sp, xs, n)$ defined to assert that the stack pointer (taken to be register 13) has value sp, that xs is on the stack and that there are n unused slots on top of the stack. We will use the HOL list notation $[x_0; \ldots; x_m]$ and the cons function defined by $\mathsf{cons}\ x_0\ [x_1; \ldots; x_m] = [x_0; x_1; \ldots; x_m]$. In order to define $stack(sp, xs, n)$, recursively define $ms(a, [x_0; x_1; \ldots; x_m])$ to mean "$M\ a\ x_0 * M\ (a{+}1)\ x_1 * \cdots * M\ (a{+}m)\ x_m$" and similarly $blank(a, n)$ to mean "$M\ a\ \_ * M\ (a{-}1)\ \_ * \cdots * M\ (a{-}(n{-}1))\ \_$". The specification $stack(sp, xs, n)$ is then defined to be $R\ 13\ sp * ms(sp, xs) * blank(sp{-}1, n)$.

Using this specification of a stack segment we are able to derive a specification for stack push from the specification of DSTR:

$$\begin{array}{c} \{R\ a\ x * stack(sp, xs, n{+}1)\} \\ \text{DSTR}\ b\ a \\ \{R\ a\ x * stack(sp{-}1, \mathsf{cons}\ x\ xs, n)\}^{+1} \end{array}$$

2.3 Positioning Functions

We use positioning functions to make our Hoare triple general. These functions are written as superscripts in our notation: $\{P\}^f\ cs^g\ \{Q\}^h$. We omit superscripts that are the identity function ($\lambda x.x$). The positioning functions specify entry points, exit points and code placement with respect to a variable base address. More concretely, $\{P\}^f\ cs^g\ \{Q\}^h$ states the following: for any address p, if the program counter points at address $f(p)$, the code sequence cs is stored at address $g(p)$ and P holds, then some time later the program counter will reach address $h(p)$ in a state where Q holds.

The positioning functions can be used to make position-independent specifications, position dependent specifications and mixtures of the two. A specification is position independent if the positioning functions describe offsets: we use $+n$ to abbreviate $\lambda x.x{+}n$, $-k$ to abbreviate $\lambda x.x{-}k$ and write nothing to mean a null offset, i.e. $\lambda x.x$. A specification is position dependent if it ignores its argument: e.g. $\lambda x.5$ and $\lambda x.y$.

```
sum:   CMP    a,#0              ; test: a = 0
       MOVEQ  r15,r14           ; return, if a = 0
       STR    a,[r13,#-4]!      ; push a
       STR    r14,[r13,#-4]!    ; push link-register
       LDR    r14,[a]           ; temp := node value
       ADD    s,s,r14           ; s := s + temp
       LDR    a,[a,#4]          ; a := address of left
       BL     sum               ; s := s + sum of a
       LDR    a,[r13,#4]        ; a := initial a
       LDR    a,[a,#8]          ; a := address of right
       BL     sum               ; s := s + sum of a
       LDR    r15,[r13],#8      ; pop two and return
```

Fig. 1. BINARY_SUM: ARM code to sum the values at the nodes of a binary tree

These positioning functions are useful as they can capture some of the non-trivial control structures used in machine-code. For example, the control structure of a procedure is easy to define: procedures are given a return address to which they must jump on completion. If we suppose that register 14 holds the return address, then we have the following format for procedure specifications:

$$\{P * R\ 14\ y\}\ cs\ \{Q * R\ 14\ \_\}^{\lambda x.y}$$

The superscript $\lambda x.y$ specifies that the value of the program counter is y on exit from cs no matter what it was on entry to cs. Section 3.6 presents a derivation of a call rule that evaluates the effect of a call to such a procedure.

The call rule and stack assertion, from above, have been used in the verification of recursive procedures in ARM code. An example of such a procedure is the code called BINARY_SUM shown in Figure 1. BINARY_SUM calculates the sum of values attached to the nodes of a binary tree. The trees we consider have nodes consisting of a value and addresses of two subtrees. Address 0 refers to the empty subtree. A predicate stating that tree t is stored with root at address x:

$$\begin{aligned} tree(x, \mathsf{Leaf}) &= \langle x = 0 \rangle \\ tree(x, \mathsf{Node}(z, l, r)) &= \exists x_1\ x_2.\ M\ x\ z * M\ (x{+}1)\ x_1 * M\ (x{+}2)\ x_2 * \\ &\qquad tree(x_1, l) * tree(x_2, r) * \langle x \neq 0 \rangle \end{aligned}$$

The specification of BINARY_SUM states that BINARY_SUM adds to register s the sum of the nodes of a tree that is addressed by register a. The specification also states that no more than $2 \times depth(t)$ words of stack space is required during execution. ([ ] is the empty list and $stack(sp, [\,], n) = R\ 13\ sp * blank(sp - 1, n)$).

$$\begin{aligned} &\{R\ a\ x * R\ s\ z * S\ \_ * \\ &\ \ tree(x, t) * stack(sp, [\,], 2 \times depth(t)) * R\ 14\ y\} \\ &\qquad\qquad \texttt{BINARY\_SUM} \\ &\{R\ a\ \_ * R\ s\ (z + sum(t)) * S\ \_ * \\ &\ \ tree(x, t) * stack(sp, [\,], 2 \times depth(t)) * R\ 14\ \_\}^{\lambda x.y} \end{aligned}$$

The formal ARM specification of `BINARY_SUM` requires some of the entities to be aligned addresses. Such details appear as slight variations of predicates M and R, for details see the companion paper [10].

2.4 Excessive Separation

The separating conjunction $*$ is set up in such a way that an occurrence of $R\ a\ x * R\ b\ y$ in a precondition will always imply $a \neq b$. This is both a weakness and a strength of our approach. It is a weakness since we will need many specifications for what seems to be special cases of a single operation. For instance, binary operators are given 5 different specifications.

$$\frac{\{R\ a\ x * R\ b\ y * R\ c\ \_\}}{\{R\ a\ x * R\ b\ y * R\ c\ (x \times y)\}^{+1}}\ \text{MUL}\ c\,a\,b \qquad \frac{\{R\ a\ x\}}{\{R\ a\ (x \times x)\}^{+1}}\ \text{MUL}\ a\,a\,a$$

$$\frac{\{R\ a\ x * R\ b\ y\}}{\{R\ a\ x * R\ b\ (x \times y)\}^{+1}}\ \text{MUL}\ b\,a\,b \qquad \frac{\{R\ a\ x * R\ b\ \_\}}{\{R\ a\ x * R\ b\ (x \times x)\}^{+1}}\ \text{MUL}\ b\,a\,a \qquad \frac{\{R\ a\ x * R\ b\ y\}}{\{R\ a\ x * R\ b\ (y \times x)\}^{+1}}\ \text{MUL}\ b\,b\,a$$

What appears to be an excessive use of $*$ is actually often a benefit. As mentioned earlier, not all the specifications above are true in every case. Furthermore, and particularly important, the separating conjunction makes the mechanisation significantly easier, as technicalities concerning register name aliasing diminish.

3 Hoare Triple for Machine Code

This section defines a Hoare triple for machine code and formalises what was informally presented in the previous section. This section ends with an example of how proof rules can be derived for procedure calls.

3.1 State Representation

We assume that a state is represented as one large set of basic state elements, where each element is an assertion specifying the state of a particular resource. State sets are required to enumerate all the resources of the observable state. In this presentation concrete states are enumerations of the following form:

$$\begin{array}{l}\{\ \mathsf{Reg}\ 0\ 820\,,\ \mathsf{Reg}\ 1\ 540\,,\ \mathsf{Reg}\ 2\ 412\,,\ \cdots\,,\ \mathsf{Reg}\ 15\ 512\,,\\ \ \ \mathsf{Mem}\ 0\ 34\,,\ \mathsf{Mem}\ 1\ 82\,,\ \mathsf{Mem}\ 2\ 11\,,\ \cdots\,,\ \mathsf{Mem}\ (2^{32}-1)\ 40\,,\\ \ \ \mathsf{Status\ F}\ \}\end{array}$$

Such sets contain 16 register elements $\mathsf{Reg}\ r\ x$ (register r holds value x), 2^{32} memory elements $\mathsf{Mem}\ a\ y$ (memory address a holds value y) and one status bit $\mathsf{Status}\ b$ (the status bit is b). No state is allowed to duplicate a basic state element, e.g. register 3 must not occur, in any state, as both $\mathsf{Reg}\ 3\ 34$ and $\mathsf{Reg}\ 3\ 45$. We will denote the set of all well-formed states by Σ, thus members of Σ represent states. Issues regarding restrictions on Σ are discussed further in Section 4.

The basic assertions described informally in the previous section can now be defined as predicates on states.

$$\begin{aligned} R\ r\ x &= \lambda s.\ (s = \{\mathsf{Reg}\ r\ x\}) \\ M\ a\ y &= \lambda s.\ (s = \{\mathsf{Mem}\ a\ y\}) \\ S\ b &= \lambda s.\ (s = \{\mathsf{Status}\ b\}) \end{aligned}$$

Let $split\ s\ (u, v)$ mean that the pair of sets (u, v) partitions the set s, i.e. $split\ s\ (u, v) = (u \cup v = s) \wedge (u \cap v = \emptyset)$. Separating conjunction $(*)$ and the notion of "some value" (written as a postfixed operator $\_$) are then defined by:

$$\begin{aligned} P * Q &= \lambda s.\ \exists u\, v.\ split\ s\ (u, v) \wedge P\ u \wedge Q\ v \\ P\ \_ &= \lambda s.\ \exists x.\ P\ x\ s \end{aligned}$$

3.2 Execution Predicate

The judgments of our Hoare logic are based on assertions about processor executions. We define the execution assertion $P \rightsquigarrow Q$ to mean that execution starting from any state which has a part satisfying P, *will reach* a state where *only* the part initially satisfying P has been changed and satisfies Q. Note that this incorporates a 'frame assumption'. The formal definition assumes a next-state function $next : \Sigma \to \Sigma$ and then uses $run(s, n)$ to denote the state reached after n steps starting from s (i.e. run is defined recursively by $run(s, 0) = s$ and $run(s, n+1) = run(next(s), n)$).

$$P \rightsquigarrow Q \quad = \quad \forall s \in \Sigma.\ \forall F.\ (P * F)\ s \Rightarrow \exists k.\ (Q * F)\ (run(s, k))$$

The following frame-rule, similar to that of separation logic, easily follows.

$$\frac{P \rightsquigarrow Q}{\forall F.\ (P * F) \rightsquigarrow (Q * F)}$$

3.3 Code Assertion

The basic execution predicate determines how the underlying processor executes on a bare state. In order to specify how code executes we need first to specify how code is located in memory and what the value of the program counter has.

Asserting the value of the program-counter is generally simple, say $R\ 15\ p$ if register 15 is the program counter. Let $pc(p)$ be such an assertion. Making a general assertion about the code in memory is more difficult. The idea is to use a kind of assertion we call a *code-pool*, which asserts that a union of possibly overlapping code sequences are part of the memory. Our approach is similar to that of Saabas and Uustalu [14] and Tan and Appel [16].

The definition of code-pool assertions uses a set-based separating conjunction operator $\circledast$ expressing the $*$-combination of the elements of an arbitrary set. Informally: $\circledast\ \{P_1, \cdots, P_n\} = P_1 * \cdots * P_n$ (when $P_1 \cdots P_n$ are distinct). The formal

definition is based on a partial bijection between predicates P_i and partitions of the state set. The definition is straightforward, but has a few subtle details which are not particularly interesting. It is omitted due to lack of space.

A code pool is an assertion obtained by applying $\circledast$ to the union of sets of basic instruction assertions $M\ p\ c$, where $M\ p\ c$ specifies that instruction c is executed if the program counter has value p (this is a special case of the notion of basic instruction assertion that we actually use). If cs is a sequence of instructions, then $Mset(p, cs)$ denotes the set of assertions stating that the sequence starts at position p and runs consecutively from there.

$$Mset(p, cs) = \{\ M\ (p + k)\ (cs[k]) \mid k < length(cs)\ \}$$

A pair (cs, f) is a code sequence cs together with a specification f of where to position it relative to a base address (see Section 2.3 for a discussion of positioning functions). We use $\mathcal{C}$ to range over sets of such pairs, and then define:

$$mpool(p, \mathcal{C}) = \circledast\,(\ \bigcup \{\ Mset(f(p), cs) \mid (cs, f) \in \mathcal{C}\ \}\ )$$

The intuition is that $mpool(p, \{(cs_1, f_1), \cdots, (cs_n, f_n)\})$ is the same as the expansion of $ms(f_1(p), cs_1) * \cdots * ms(f_n(p), cs_n)$ with the duplicated M-assertions removed by the set union. The benefit of using such a code pool is that it allows code sequences to overlap and builds into the representation the removal of duplicate sequences. This benefit is particularly apparent in the rule for procedural recursion, Section 3.6.

At the end of a verification of concrete code one can of course not have distinct sequences of code that overlap. Such an arrangement makes the precondition(s) of the machine-code Hoare-triple (defined in the next section) false and hence the specification trivially true. The following two equivalences simplify a code-pool into a simple sequence assertion.[1] Note that in the equation below and later, $+length(cs)$ denotes the function that adds the length of cs, thus $+length(cs) \circ f$ is the function $\lambda n.\ length(cs) + f(n)$.

$$mpool(p, \{(cs, f)\}) = ms(f(p), cs)$$
$$mpool(p, \{(cs, f), (cs', +length(cs) \circ f)\} \cup \mathcal{C}) = mpool(p, \{(cs; cs', f)\} \cup \mathcal{C})$$

3.4 Hoare Triple

In Section 2 we discussed a Hoare triple $\{P\}^f\ cs^{\,g}\ \{Q\}^h$. We will shortly generalise this to have sets of preconditions, sets of code sequences and sets of postconditions, but first we give a formal semantics of the simple case.

$$\{P\}^f\ cs^{\,g}\ \{Q\}^h \quad = \quad \forall p.\ (P * ms(g(p), cs) * pc(f(p))) \rightsquigarrow (Q * ms(g(p), cs) * pc(h(p)))$$

We can read $\{P\}^f\ cs^{\,g}\ \{Q\}^h$ as asserting that if the processor is started from a state satisfying P and (for any p) if $f(p)$ is in the program counter and the

[1] The first of these equalities is only true under the assumption that the length of cs does not exceed the length of the address space.

code cs stored as a sequence from address $g(p)$ onwards, then it will reach a state satisfying Q. The specification also guarantees termination with the code unchanged and the program counter updated to $h(p)$. The functions f and g are frequently the identity function, in which case the program counter points at the first instruction in the sequence of instructions cs. Notice that the meaning of $*$ ensures that the precondition $P * ms(g(p), cs) * pc(f(p))$ only holds when P does not mention the program counter or any memory location where cs is stored.

We generalise the simple case to multiple preconditions, code segments and postconditions, each with positioning functions f_i, g_i and h_i, respectively:

$$\{P_1\}^{f_1} \cdots \{P_n\}^{f_n}\ cs_1^{\,g_1}\ \cdots\ cs_m^{\,g_m}\ \{Q_1\}^{h_1} \cdots \{Q_k\}^{h_k}$$

The intuition is the following: if all the code segments are present in memory, then whenever one of the preconditions $\{P_i\}^{f_i}$ is true, some time later (at least) one of the postconditions $\{Q_j\}^{h_j}$ will be true.

For the definition of the general Hoare-triple collect the preconditions, code segments and postconditions into respective sets $\mathcal{P} = \{(P_1, f_1), \cdots, (P_n, f_n)\}$, $\mathcal{C} = \{(cs_1, g_1), \cdots, (cs_m, g_m)\}$ and $\mathcal{Q} = \{(Q_1, h_1), \cdots, (Q_k, h_k)\}$. The machine-code Hoare-triple, which is written here as $\mathcal{P} \mid \mathcal{C} \mid \mathcal{Q}$, is defined using disjunction over as set of predicates $\bigvee$ (formally: $\bigvee \mathcal{X} = \lambda s.\ \exists P \in \mathcal{X}.\ P\ s$).

$$\begin{aligned}\mathcal{P} \mid \mathcal{C} \mid \mathcal{Q} = \forall p.\ (\bigvee \{\ P * mpool(p, \mathcal{C}) * pc(f(p)) \mid (P, f) \in \mathcal{P}\ \}) \rightsquigarrow \\ (\bigvee \{\ Q * mpool(p, \mathcal{C}) * pc(f(p)) \mid (Q, f) \in \mathcal{Q}\ \})\end{aligned}$$

A variety of rules have been derived from this definition of Hoare triple. Some of the rules are presented in Figure 2. The rules for frame, shift and compose are used when joining specifications (as illustrated in the next section). Strengthen, weaken and merge are used when specifications are simplified. Contraction, extension and loop elimination add/remove entry points, exit points and code segments. The rule for loop elimination removes any number of interconnected exit points that match some set of entry point for a decreasing variant. The equivalences are mainly used in derivations of new rules.

3.5 Example: Composition

The rule for composition given in Figure 2 is quite abstract. We demonstrate its use by composing a specification of a decrement instruction and a branch instruction (c.f. the instructions of the factorial program). The branch instruction has two exit points, thus two postconditions. We illustrate the three possible compositions below.

$$\begin{array}{c} \{R\ a\ x * S\ \_\} \\ \text{SUBS}\ a\ a\ 1 \\ \{R\ a\ (x{-}1) * S\ (x{-}1 = 0)\}^{+1} \end{array} \qquad \begin{array}{c} \{S\ b\} \\ \text{BNE}\ k \\ \{S\ \mathsf{T} * \langle b \rangle\}^{+1} \\ \{S\ \mathsf{F} * \langle \neg b \rangle\}^{+k} \end{array}$$

Composition is commonly done in three stages: first the scope of the specifications is extended so that the footprints match, then the positioning functions

Let ":" denote insertion into a set and "$\prec$" denote any well-found relation.
Let $\mathcal{P} \circledast F = \{\ (P * F, f) \mid (P, f) \in \mathcal{P}\ \}$ and $\mathcal{P} \circledcirc g = \{\ (P, f \circ g) \mid (P, f) \in \mathcal{P}\ \}$.
Let $\langle b \rangle = \lambda s.\ (s = \emptyset) \wedge b$.

Frame, shift and compose.

$$\frac{\mathcal{P} \mid \mathcal{C} \mid \mathcal{Q}}{\forall F.\ \ \mathcal{P} \circledast F \mid \mathcal{C} \mid \mathcal{Q} \circledast F} \qquad \frac{\mathcal{P} \mid \mathcal{C} \mid \mathcal{Q}}{\forall g.\ \ \mathcal{P} \circledcirc g \mid \mathcal{C} \circledcirc g \mid \mathcal{Q} \circledcirc g}$$

$$\frac{\mathcal{P} \mid \mathcal{C} \mid \mathcal{Q} \cup \mathcal{M} \qquad \mathcal{M} \cup \mathcal{P}' \mid \mathcal{C}' \mid \mathcal{Q}'}{\mathcal{P} \cup \mathcal{P}' \mid \mathcal{C} \cup \mathcal{C}' \mid \mathcal{Q} \cup \mathcal{Q}'}$$

Contract, extend, strengthen and weaken.

$$\frac{\mathcal{P} \cup \mathcal{P}' \mid \mathcal{C} \mid \mathcal{Q}}{\mathcal{P} \mid \mathcal{C} \mid \mathcal{Q}} \qquad \frac{\mathcal{P} \mid \mathcal{C} \mid \mathcal{Q}}{\mathcal{P} \mid \mathcal{C} \cup \mathcal{C}' \mid \mathcal{Q}} \qquad \frac{\mathcal{P} \mid \mathcal{C} \mid \mathcal{Q}}{\mathcal{P} \mid \mathcal{C} \mid \mathcal{Q} \cup \mathcal{Q}'}$$

$$\frac{P' \Rightarrow P \qquad (P, f) : \mathcal{P} \mid \mathcal{C} \mid \mathcal{Q}}{(P', f) : \mathcal{P} \mid \mathcal{C} \mid \mathcal{Q}} \qquad \frac{Q \Rightarrow Q' \qquad \mathcal{P} \mid \mathcal{C} \mid (Q, f) : \mathcal{Q}}{\mathcal{P} \mid \mathcal{C} \mid (Q', f) : \mathcal{Q}}$$

Merge rules.

$$\frac{(P, f) : (P', f) : \mathcal{P} \mid \mathcal{C} \mid \mathcal{Q}}{(P \vee P', f) : \mathcal{P} \mid \mathcal{C} \mid \mathcal{Q}} \qquad \frac{\mathcal{P} \mid \mathcal{C} \mid (Q, f) : (Q', f) : \mathcal{Q}}{\mathcal{P} \mid \mathcal{C} \mid (Q \vee Q', f) : \mathcal{Q}}$$

$$\frac{\mathcal{P} \mid (cs, f) : (cs', +length(cs) \circ f) : \mathcal{C} \mid \mathcal{Q}}{\mathcal{P} \mid (cs +\!\!+ cs', f) : \mathcal{C} \mid \mathcal{Q}}$$

Loop elimination.

$$\frac{\forall v.\ I(v) \cup \mathcal{P} \mid \mathcal{C} \mid \mathcal{Q} \cup \{\ i \mid i \in I(v') \wedge v' \prec v\ \}}{\forall v.\ I(v) \cup \mathcal{P} \mid \mathcal{C} \mid \mathcal{Q}}$$

Various equivalences.

$$\mathcal{P} \mid \mathcal{C} \mid (\exists x.\ Q(x) * \langle b(x) \rangle, f) : \mathcal{Q} = \mathcal{P} \mid \mathcal{C} \mid \mathcal{Q} \cup \{\ (Q(x), f) \mid b(x)\ \}$$

$$(\exists x.\ P(x) * \langle b(x) \rangle, f) : \mathcal{P} \mid \mathcal{C} \mid \mathcal{Q} = \{\ (P(x), f) \mid b(x)\ \} \cup \mathcal{P} \mid \mathcal{C} \mid \mathcal{Q}$$

$$\mathcal{P} \mid \mathcal{C} \mid \mathcal{Q} = \forall p.\ \mathcal{P} \circledcirc (\lambda x.p) \mid \mathcal{C} \circledcirc (\lambda x.p) \mid \mathcal{Q} \circledcirc (\lambda x.p)$$

Fig. 2. Rules for the machine-code Hoare triple

are made to match by a shift and finally the composition rule is applied followed by an application of a code merge if applicable.

We start by constructing a specification for "SUBS $a\,a\,1$; BNE k". The frame rule is used to extend the specification of BNE and b is instantiated:

$$\begin{array}{c}\{R\ a\ (x{-}1) * S\ (x{-}1 = 0)\}\\ \text{BNE } k\\ \{R\ a\ (x{-}1) * S\ \mathsf{T} * \langle x{-}1 = 0\rangle\}^{+1}\\ \{R\ a\ (x{-}1) * S\ \mathsf{F} * \langle x{-}1 \neq 0\rangle\}^{+k}\end{array}$$

A shift by $+1$ makes the precondition of BNE match the postcondition of SUBS:

$$\begin{array}{c}\{R\ a\ (x{-}1) * S\ (x{-}1 = 0)\}^{+1}\\ \text{BNE } k\ ^{+1}\\ \{R\ a\ (x{-}1) * S\ \mathsf{T} * \langle x{-}1 = 0\rangle\}^{+2}\\ \{R\ a\ (x{-}1) * S\ \mathsf{F} * \langle x{-}1 \neq 0\rangle\}^{+(k+1)}\end{array}$$

An application of the composition rule followed by a code merge yields:

$$\begin{array}{c}\{R\ a\ x * S\ \_\}\\ \text{SUBS } a\,a\,1;\ \text{BNE } k\\ \{R\ a\ (x{-}1) * S\ \mathsf{T} * \langle x{-}1 = 0\rangle\}^{+2}\\ \{R\ a\ (x{-}1) * S\ \mathsf{F} * \langle x{-}1 \neq 0\rangle\}^{+(k+1)}\end{array}$$

Alternatively, the specification for SUBS can be tacked onto either branch of BNE. The compositions are done with shifts $+1$ and $+k$, respectively. The composition with shift $+k$ results in a specification with two code segments.

$$\begin{array}{c}\{R\ a\ x * S\ b\}\\ \text{BNE } k;\ \text{SUBS } a\,a\,1\\ \{R\ a\ (x{-}1) * S\ (x{-}1 = 0) * \langle b\rangle\}^{+2}\\ \{R\ a\ x * S\ \mathsf{F} * \langle \neg b\rangle\}^{+k}\end{array}\qquad\qquad\begin{array}{c}\{R\ a\ x * S\ b\}\\ \text{SUBS } a\,a\,1\ ^{+k}\quad \text{BNE } k\\ \{R\ a\ x * S\ \mathsf{T} * \langle b\rangle\}^{+1}\\ \{R\ a\ (x{-}1) * S\ (x{-}1 = 0) * \langle \neg b\rangle\}^{+(k+1)}\end{array}$$

3.6 Example: Procedures and Procedural Recursion

This section illustrates how specifications for procedures and procedure calls fit into our framework. We define the control-flow contract of a procedure and a procedure call, derive a rule stating the effect of a procedure call and finally present a rule that we have found useful when proving recursive procedures.

The standard contract of a procedure can be captured easily within our framework. Commonly a procedure is given a return address to which it must jump upon completion. Given a resource, say, lr that holds the return address we can specify a reasonably general contract as follows:

$$\text{PROC}(f, P, \mathcal{C}, Q) = \forall p.\ \{P * lr\ p\}^{f}\ \mathcal{C}\ \{Q * lr\ \_\}^{\lambda x.p}$$

Specifying a general procedure call is slightly more involved in our framework. We define a call to be a jump that starts with the program counter set to $h(p)$, for any p, stores the address $g(p)$ in lr and jumps to address $f(p)$.

$$\text{CALL}(f, \mathcal{C}, h, g) = \forall p.\ \{lr\ \_\}^{\lambda x.h(p)}\ (\mathcal{C} \circ (\lambda x.p))\ \{lr(g(p))\}^{\lambda x.f(p)}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\textsc{call}(f,\mathcal{C},h,g)}{\forall p.\ \{(lr\_,\lambda x.h(p))\}\mid \mathcal{C}\,\bar{\circ}\,(\lambda x.p)\mid\{(lr(g(p)),\lambda x.f(p))\}}}{\{(lr\_,\lambda x.h(p))\}\mid \mathcal{C}\,\bar{\circ}\,(\lambda x.p)\mid\{(lr(g(p)),\lambda x.f(p))\}}}{\{(P*lr\_,\lambda x.h(p))\}\mid \mathcal{C}\,\bar{\circ}\,(\lambda x.p)\mid\{(P*lr(g(p)),\lambda x.f(p))\}}}{} \quad (1)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\textsc{proc}(f,P,\mathcal{C}',Q)}{\forall p.\ \{(P*lr(p),f)\}\mid \mathcal{C}'\mid\{(Q*lr\_,\lambda x.p)\}}}{\{(P*lr(g(p)),f)\}\mid \mathcal{C}'\mid\{(Q*lr\_,\lambda x.g(p))\}}}{\{(P*lr(g(p)),f\circ\lambda x.p)\}\mid \mathcal{C}'\,\bar{\circ}\,(\lambda x.p)\mid\{(Q*lr\_,\lambda x.g(p)\circ\lambda x.p)\}}}{\{(P*lr(g(p)),\lambda x.f(p))\}\mid \mathcal{C}'\,\bar{\circ}\,(\lambda x.p)\mid\{(Q*lr\_,\lambda x.g(p))\}} \quad (2)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{(1)\quad(2)}{\{(P*lr\_,\lambda x.h(p))\}\mid(\mathcal{C}\,\bar{\circ}\,(\lambda x.p))\cup(\mathcal{C}'\,\bar{\circ}\,(\lambda x.p))\mid\{(Q*lr\_,\lambda x.g(p))\}}}{\{(P*lr\_,h)\}\,\bar{\circ}\,(\lambda x.p)\mid(\mathcal{C}\cup\mathcal{C}')\,\bar{\circ}\,(\lambda x.p)\mid\{(Q*lr\_,g)\}\,\bar{\circ}\,(\lambda x.p)}}{\forall p.\ \{(P*lr\_,h)\}\,\bar{\circ}\,(\lambda x.p)\mid(\mathcal{C}\cup\mathcal{C}')\,\bar{\circ}\,(\lambda x.p)\mid\{(Q*lr\_,g)\}\,\bar{\circ}\,(\lambda x.p)}}{\{(P*lr\_,h)\}\mid\mathcal{C}\cup\mathcal{C}'\mid\{(Q*lr\_,g)\}}$$

Fig. 3. A derivation of the call rule

The ARM instruction for branch-and-link `BL` satisfies a specification that is essentially the same as $\textsc{call}(+k,\{(\texttt{BL}\ k,+0)\},+0,+1)$.

The effect of executing a call $\textsc{call}(f,\mathcal{C},h,g)$ to a procedure $\textsc{proc}(f,P,\mathcal{C}',Q)$ is described by the call rule, derived in Figure 3.

$$\dfrac{\textsc{call}(f,\mathcal{C},h,g)\qquad\textsc{proc}(f,P,\mathcal{C}',Q)}{\{P*lr\_\}^{h}\ \mathcal{C}\cup\mathcal{C}'\ \{Q*lr\_\}^{g}}$$

The call rule is quite general. It does not restrict the procedure body or the call statement to be position dependent or independent. This was achieved by the inclusion of positioning functions h, g and f. Of these functions f has an artificial role when the procedure is position independent. Why should the procedure specification have a positioning function in common with the call specification, if the procedure specification is position independent?

In order to remove this oddity a special rule can be proved for calls to procedures that have the positioning function set to $\lambda x.x$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\textsc{proc}(\lambda x.x,P,\mathcal{C}',Q)}{\forall p.\ \{(P*lr(p),\lambda x.x)\}\mid\mathcal{C}'\mid\{(Q*lr\_,\lambda x.p)\}}}{\forall p.\ \{(P*lr(p),(\lambda x.x)\circ f)\}\mid\mathcal{C}'\,\bar{\circ}\,f\mid\{(Q*lr\_,(\lambda x.p)\circ f)\}}}{\forall p.\ \{(P*lr(p),f)\}\mid\mathcal{C}'\,\bar{\circ}\,f\mid\{(Q*lr\_,\lambda x.p)\}}}{\textsc{proc}(f,P,\mathcal{C}'\,\bar{\circ}\,f,Q)} \quad (3)$$

$$\dfrac{\textsc{call}(f,\mathcal{C},h,g)\qquad(3)}{\{P*lr\_\}^{h}\ \mathcal{C}\cup(\mathcal{C}'\,\bar{\circ}\,f)\ \{Q*lr\_\}^{g}}$$

Informally this rule can be understood as follows: A call with jump function f executes a position-independent procedure with code $\mathcal{C}'$, if code $\mathcal{C}'$ is placed using function f.

Procedural recursion of one or more procedures is proved by induction over a bounded variant function that decreases strictly on each recursive call. The observation that each recursive call pushes at least one value (the return address) onto the stack[2], suggests that induction over the natural number is sufficient. The remaining stack space[3] is a natural number that decreases for each recursive call. We have found the following induction rule useful in proofs of recursive procedures. Let v be some variant function, $<$ be less-than over the natural numbers and ψ be any boolean-valued function.

$$\frac{\forall x\ \mathcal{C}'.\ (\forall y.\ v(y) < v(x) \Rightarrow \psi(y, \mathcal{C}')) \Rightarrow \psi(x, \mathcal{C} \cup \mathcal{C}')}{\forall x.\ \psi(x, \mathcal{C})}$$

The parameter $\mathcal{C}$ is intended to hold a set of code segments. Notice that $\mathcal{C}$ does not occur in the assumption of the premise. The absence of $\mathcal{C}$ makes the rule easier to use, as one does not need to assume the code one is constructing.

The definitions and theorems of this section were used in the verification of `BINARY_SUM`, Section 3.6. The verification of `BINARY_SUM` was done as a case analysis over the structure of the tree. The case of a leaf was trivial as it exits on the second instruction. The case of a branch required more work. For it we assumed that there is some code $\mathcal{C}'$ that performs the desired function for the subtrees. We used the second version of the call rule to extract specifications for the `BL` instructions that perform the recursive calls. The specifications for all twelve instructions were then composed and the cases (leaf and branch) were merged. The induction rule, from above, was specialised to trees by setting v to *depth* (depth of a binary tree) and then used to eliminate the assumed specifications and imaginary code $\mathcal{C}'$. The same induction was also used in proving a variant of `BINARY_SUM` that has the last call replaced by a tail-recursive call. The details of both proofs are given in [10].

4 Formalisation and Specialisation

Section 3.1 made restrictions on the format of the sets that are members of the set of valid states Σ. Restrictions are needed in order to ensure the intended meaning of separation for separating conjunction $*$. This section describes how we avoid such issues in our formalisation of the general case and also how we address them when the general theory is specialised and used.

The general theory, which consists of the definition of the machine-code Hoare triple and its rules, can be proved without any restrictions on the structure of the state sets[4]. The machine-code Hoare triple can be defined and all its rules proved for any set of state sets Σ, given a next-state function $next : \Sigma \to \Sigma$ [5],

[2] We consider tail-recursive-call as a loop, not as a call.

[3] We will not assume an infinite stack as we do not assume an infinite state space.

[4] In the HOL mechanisation the type of a state element is parametrised by a type variable. The type of a state set is "α `set`".

[5] Alternatively, one can use a next-state relation $next : \Sigma \times \Sigma$, for this redefine $\rightsquigarrow$.

a program-counter assertion $pc : \alpha \to \Sigma \to \mathbb{B}$ and a basic instruction assertion $inst : \alpha \times \beta \to \Sigma \to \mathbb{B}$, for some set α of instruction addresses and some set β of instructions. These abstractions ease the proof effort. All the definitions and rules are parametrised by a 6-tuple $(\Sigma, \alpha, \beta, next, pc, inst)$.

When the general theory is instantiated and one wants to prove basic specifications for the elementary operations of a specific language (examples of basic specification: Section 2.2, 2.4 and 3.5), then one has to restrict the shape of Σ so that $*$ has its intended meaning. We have found that a practical method for restricting the shape of the state sets is to have them produced by a function. We define Σ to be the range of a function tr, i.e. $\Sigma = \{\ tr(x) \mid \text{any } x\ \}$, for some function tr that produces state sets of a specific form.

The function tr can be a translation function from a different state representation. If this is the case and the translation is accurate enough to also have an inverse $\bar{tr}$ (i.e. $\forall x.\ \bar{tr}(tr(x)) = x$), then one can define the next-state function for the set-based representation ($next$) using a next-state function over the other state representation (say $next_{sem}$): $next(s) = tr(next_{sem}(\bar{tr}(s)))$. The benefit of defining $next$ according to a next-state function over a different state-representation is a practical one. The detailed semantics of a machine-code language might be more readily defined using a state-representation different from the set-based representation that our approach requires. This is the case in the application of our framework to the ARM processor: we generate members of Σ formally from the representations of states used by the ARM model.

5 Summary

This paper has presented a Hoare logic that has been carefully designed to fit on top of accurately modelled operational semantics of machine languages. Specifications are built on a separating conjunction, that allows concise resource usage specifications and also helps avoid unwanted aliasing. Multiple code segments and positioning functions make our specifications support control flow that allows specifications of procedures and procedure calls, as well as general control flow between position independent and position dependent code. We build on previous work on separation logic [13] and unstructured control-flow [2,16].

Our framework has been fully formalised in higher-order logic, mechanised using the HOL4 system and has been applied to ARM machine-code using an existing high-fidelity model of the ARM processor [10]. We have not yet applied our framework to other architectures nor large case studies, but we think we have a methodology and implemented tools that will scale. Demonstrating this is the next phase of our research.

Acknowledgments. We would like to thank Anthony Fox, Joe Hurd, Konrad Slind, Thomas Tuerk, Matthew Parkinson, Josh Berdine, Nick Benton and Richard Bornat for research discussions, comments and substantial constructive criticism. The first author is funded by Osk.Huttusen Säätiö and EPSRC.

References

1. Andrew W. Appel. Foundational proof-carrying code. In *Proc. 16th IEEE Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society, 2001.
2. Michael A. Arbib and Suad Alagic. Proof rules for gotos. *Acta Informatica*, 11:139–148, 1979.
3. William R. Bevier. *A verified operating system kernel*. PhD thesis, University of Texas at Austin, 1987.
4. Robert S. Boyer and Yuan Yu. Automated proofs of object code for a widely used microprocessor. *J. ACM*, 43(1):166–192, 1996.
5. D. L. Clutterbuck and B. A. Carré. The verification of low-level code. *Software Engineering Journal*, 3:97–111, 1988.
6. The HOL4 System (Description). http://hol.sourceforge.net/documentation.html.
7. David S. Hardin, Eric W. Smith, and William D. Young. A robust machine code proof framework for highly secure applications. In Panagiotis Manolios and Matthew Wilding, editors, *Sixth International Workshop on the ACL2 Theorem Prover and Its Applications*, 2006.
8. Gerwin Klein, Harvey Tuch, and Michael Norrish. Types, bytes, and separation logic. To appear in Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. Springer, 2007.
9. W. D. Maurer. Proving the correctness of a flight-director program for an airborne minicomputer. In *Proceedings of the ACM SIGMINI/SIGPLAN interface meeting on Programming systems in the small processor environment*. ACM Press, 1976.
10. Magnus O. Myreen, Anthony C. J. Fox, and Michael J. C. Gordon. Hoare logic for ARM machine code. To appear in *Proceedings of the IPM International Symposium on Fundamentals of Software Engineering (FSEN)*. Springer, 2007.
11. George C. Necula. Proof-carrying code. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1997.
12. Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Proceedings of Computer Science Logic*, 2001.
13. John Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of 17th IEEE Symposium on Logic in Computer Science (LICS)*, 2002.
14. Ando Saabas and Tarmo Uustalu. A compositional natural semantics and hoare logic for low-level languages. *Electronic Notes in Theoretical Computer Science*, 156(1):151–168, 2006.
15. David Seal. *ARM Architecture Reference Manual*. Addison-Wesley, 2000.
16. Gang Tan and Andrew W. Appel. A compositional logic for control flow. In *Proceedings of Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 3855 of *Lecture Notes in Computer Science*. Springer, 2006.