Hoare Logic for Realistically Modelled Machine Code

Magnus Myreen

October 24, 2006

Outline

Motivation

Why program logics based on simplified models fail to scale to realistically modelled machine code:

Motivation

Why program logics based on simplified models fail to scale to realistically modelled machine code:

1. Infinite state space (stacks)

Motivation

Why program logics based on simplified models fail to scale to realistically modelled machine code:

1. Infinite state space (stacks)
2. Separation of code and data (memory accesses)

Motivation

Why program logics based on simplified models fail to scale to realistically modelled machine code:

1. Infinite state space (stacks)
2. Separation of code and data (memory accesses)
3. Nontrivial to mechanise (detailed models)

Motivation

Why program logics based on simplified models fail to scale to realistically modelled machine code:

1. Infinite state space (stacks)
2. Separation of code and data (memory accesses)
3. Nontrivial to mechanise (detailed models)
4. Special purpose resources (special registers)

Motivation

Why program logics based on simplified models fail to scale to realistically modelled machine code:

1. Infinite state space (stacks)
2. Separation of code and data (memory accesses)
3. Nontrivial to mechanise (detailed models)
4. Special purpose resources (special registers)
5. Other restrictions (restricted instructions)

Motivation

Why program logics based on simplified models fail to scale to realistically modelled machine code:

1. Infinite state space (stacks)
2. Separation of code and data (memory accesses)
3. Nontrivial to mechanise (detailed models)
4. Special purpose resources (special registers)
5. Other restrictions (restricted instructions)

   For instance: The ARM instruction for multiplication `MUL` c, a, b is *unpredictable* if registers a and c are the same. Hence $x := y \cdot x$ is allowed, but $x := x \cdot y$ is not.

Example

An ARM program for calculating the factorial of a positive number:

```
     MOV   b, #1       ; b := 1
L:   MUL   b, a, b     ; b := a × b
     SUBS  a, a, #1    ; a := a - 1
     BNE   L           ; jump to L if a ≠ 0
```

A classical Hoare-style specification:

$$\begin{array}{c}\{(a = x) \land (x \neq 0)\} \\ \texttt{FACTORIAL} \\ \{(a = 0) \land (b = x!)\}\end{array}$$

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

Example

An ARM program for calculating the factorial of a positive number:

```
    MOV   b, #1      ; b := 1
L:  MUL   b, a, b    ; b := a × b
    SUBS  a, a, #1   ; a := a - 1
    BNE   L          ; jump to L if a ≠ 0
```

A classical Hoare-style specification:

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

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

What is left unchanged?

Example

Separation logic solves the frame problem for memory properties by assigning a footprint to each assertion.

We do the same, but require the footprint to include any type of state element (not just memory assertions).

Example

Separation logic solves the frame problem for memory properties by assigning a footprint to each assertion.

We do the same, but require the footprint to include any type of state element (not just memory assertions).

The specification of the factorial program:

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

Star ($*$) is a separating conjunction from Separation Logic.

Example

Specification for multiplication and decrement-by-one:

$$\{R\ a\ x * R\ b\ y\}$$
$$\texttt{MUL}\ b\texttt{,}a\texttt{,}b$$
$$\{R\ a\ x * R\ b\ (x \cdot y)\}^{+1}$$

$$\{R\ a\ x * S\ \_\}$$
$$\texttt{SUB}\ a\texttt{,}a\texttt{,\#1}$$
$$\{R\ a\ (x{-}1) * S\ (x{-}1{=}0)\}^{+1}$$

Example

Specification for multiplication and decrement-by-one:

$$\begin{array}{c}\{R\ a\ x * R\ b\ y\} \\ \texttt{MUL}\ \ b\texttt{,}a\texttt{,}b \\ \{R\ a\ x * R\ b\ (x \cdot y)\}^{+1}\end{array} \qquad \begin{array}{c}\{R\ a\ x * S\ \_\} \\ \texttt{SUB}\ \ a\texttt{,}a\texttt{,\#1} \\ \{R\ a\ (x{-}1) * S\ (x{-}1{=}0)\}^{+1}\end{array}$$

Extension:

$$\begin{array}{c}\forall P.\quad \{R\ a\ x * R\ b\ y * P\} \\ \texttt{MUL}\ \ b\texttt{,}a\texttt{,}b \\ \{R\ a\ x * R\ b\ (x \cdot y) * P\}^{+1}\end{array}$$

Example

Specification for multiplication and decrement-by-one:

$$\{R\ a\ x * R\ b\ y\}\quad \texttt{MUL}\ b,a,b \quad \{R\ a\ x * R\ b\ (x \cdot y)\}^{+1}$$

$$\{R\ a\ x * S\ \_\}\quad \texttt{SUB}\ a,a,\#1 \quad \{R\ a\ (x{-}1) * S\ (x{-}1{=}0)\}^{+1}$$

Extension:

$$\{R\ a\ x * R\ b\ y * S\ \_\}$$
$$\texttt{MUL}\ b,a,b$$
$$\{R\ a\ x * R\ b\ (x \cdot y) * S\ \_\}^{+1}$$

Example

Specification for multiplication and decrement-by-one:

$$\begin{array}{c}\{R\ a\ x * R\ b\ y\}\\ \texttt{MUL}\ \ b\texttt{,}a\texttt{,}b\\ \{R\ a\ x * R\ b\ (x \cdot y)\}^{+1}\end{array} \qquad \begin{array}{c}\{R\ a\ x * S\ \_\}\\ \texttt{SUB}\ \ a\texttt{,}a\texttt{,\#1}\\ \{R\ a\ (x{-}1) * S\ (x{-}1{=}0)\}^{+1}\end{array}$$

Extension:

$$\begin{array}{c}\{R\ a\ x * R\ b\ y * S\ \_\}\\ \texttt{MUL}\ \ b\texttt{,}a\texttt{,}b\\ \{R\ a\ x * R\ b\ (x \cdot y) * S\ \_\}^{+1}\end{array}$$

Extension:

$$\begin{array}{c}\{R\ a\ x * R\ b\ (x \cdot y) * S\ \_\}\\ \texttt{SUB}\ \ a\texttt{,}a\texttt{,\#1}\\ \{R\ a\ (x{-}1) * R\ b\ (x \cdot y) * S\ (x{-}1{=}0)\}^{+1}\end{array}$$

Example

Specification for multiplication and decrement-by-one:

$$\begin{array}{c}\{R\ a\ x * R\ b\ y\}\\ \texttt{MUL}\ \ b\texttt{,}a\texttt{,}b\\ \{R\ a\ x * R\ b\ (x \cdot y)\}^{+1}\end{array} \qquad \begin{array}{c}\{R\ a\ x * S\ \_\}\\ \texttt{SUB}\ \ a\texttt{,}a\texttt{,\#1}\\ \{R\ a\ (x{-}1) * S\ (x{-}1{=}0)\}^{+1}\end{array}$$

Extension:

$$\begin{array}{c}\{R\ a\ x * R\ b\ y * S\ \_\}\\ \texttt{MUL}\ \ b\texttt{,}a\texttt{,}b\\ \{R\ a\ x * R\ b\ (x \cdot y) * S\ \_\}^{+1}\end{array}$$

Composition:

$$\begin{array}{c}\{R\ a\ x * R\ b\ y * S\ \_\}\\ \texttt{MUL}\ \ b\texttt{,}a\texttt{,}b;\ \texttt{SUB}\ \ a\texttt{,}a\texttt{,\#1}\\ \{R\ a\ (x{-}1) * R\ b\ (x \cdot y) * S\ (x{-}1{=}0)\}^{+2}\end{array}$$

Example

Specification of a branch:

$$
\begin{array}{c}
\{S\ b\} \\
\texttt{BNE \#}k \\
\{S\ \mathsf{T} * \langle b \rangle\}^{+1} \\
\{S\ \mathsf{F} * \langle \neg b \rangle\}^{+(k+2)}
\end{array}
$$

Example

Specification of a branch:

$$\begin{array}{c}\{S\ b\}\\ \texttt{BNE \#}k\\ \{S\ \mathsf{T} * \langle b \rangle\}^{+1}\\ \{S\ \mathsf{F} * \langle \neg b \rangle\}^{+(k+2)}\end{array}$$

Composition:

$$\begin{array}{c}\{R\ a\ x * R\ b\ y * S\ \_\}\\ \texttt{MUL } b\texttt{,}a\texttt{,}b\texttt{; SUB } a\texttt{,}a\texttt{,\#1; BNE \#}{-4}\\ \{R\ a\ (x{-}1) * R\ b\ (x \cdot y) * S\ \mathsf{T} * \langle x{-}1{=}0 \rangle\}^{+3}\\ \{R\ a\ (x{-}1) * R\ b\ (x \cdot y) * S\ \mathsf{F} * \langle x{-}1{\neq}0 \rangle\}^{+0}\end{array}$$

Example

Specification of a branch:

$$
\begin{gathered}
\{S\ b\} \\
\texttt{BNE \#}k \\
\{S\ \mathsf{T} * \langle b \rangle\}^{+1} \\
\{S\ \mathsf{F} * \langle \neg b \rangle\}^{+(k+2)}
\end{gathered}
$$

Composition:

$$
\begin{gathered}
\{R\ a\ x * R\ b\ (n!/x!) * S\ \_ * \langle x \neq 0 \rangle * \langle x \leq n \rangle\} \\
\texttt{MUL } b\texttt{,}a\texttt{,}b\texttt{; SUB } a\texttt{,}a\texttt{,\#1; BNE \#}{-4} \\
\{R\ a\ 0 * R\ b\ n! * S\ \_\}^{+3} \\
\{R\ a\ (x{-}1) * R\ b\ (n!/(x{-}1)!) * S\ \_ * \langle x{-}1{\neq}0 \rangle * \langle x{-}1{\leq}n \rangle * \langle x{-}1 < x \rangle\}^{+0}
\end{gathered}
$$

Example

Specification of a branch:

$$\{S\ b\}$$
$$\texttt{BNE \#}k$$
$$\{S\ \mathsf{T} * \langle b \rangle\}^{+1}$$
$$\{S\ \mathsf{F} * \langle \neg b \rangle\}^{+(k+2)}$$

Composition:

$$\{R\ a\ x * R\ b\ (n!/x!) * S\ \_ * \langle x \neq 0 \rangle * \langle x \leq n \rangle\}$$
$$\texttt{MUL}\ b\texttt{,}a\texttt{,}b\texttt{; SUB}\ a\texttt{,}a\texttt{,\#1; BNE \#}{-4}$$
$$\{R\ a\ 0 * R\ b\ n! * S\ \_\}^{+3}$$

Example

Specification of a branch:

$$\{S\ b\}$$
$$\texttt{BNE \#}k$$
$$\{S\ \mathsf{T} * \langle b\rangle\}^{+1}$$
$$\{S\ \mathsf{F} * \langle\neg b\rangle\}^{+(k+2)}$$

Composition:

$$\{R\ a\ x * R\ b\ 1 * S\ \_ * \langle x \neq 0\rangle\}$$
$$\texttt{MUL}\ b\texttt{,}a\texttt{,}b\texttt{; SUB}\ a\texttt{,}a\texttt{,\#1; BNE \#}{-4}$$
$$\{R\ a\ 0 * R\ b\ x! * S\ \_\}^{+3}$$

Example

Specification of a branch:

$$\{S\ b\}$$
$$\texttt{BNE}\ \texttt{\#}k$$
$$\{S\ \mathsf{T} * \langle b\rangle\}^{+1}$$
$$\{S\ \mathsf{F} * \langle \neg b\rangle\}^{+(k+2)}$$

Composition:

$$\{R\ a\ x * R\ b\ \_ * S\ \_ * \langle x \neq 0\rangle\}$$
$$\texttt{MOV}\ b\texttt{,\#1;}\ \texttt{MUL}\ b\texttt{,}a\texttt{,}b\texttt{;}\ \texttt{SUB}\ a\texttt{,}a\texttt{,\#1;}\ \texttt{BNE}\ \texttt{\#}{-4}$$
$$\{R\ a\ 0 * R\ b\ x! * S\ \_\}^{+4}$$

State Representation

A state is a set enumerating state elements. A concrete state:

$$\{\ \text{Reg}\ 0\ 820\,,\ \text{Reg}\ 1\ 540\,,\ \cdots\,,\ \text{Reg}\ 15\ 512\,,$$
$$\text{Mem}\ 0\ 34\,,\ \text{Mem}\ 1\ 82\,,\ \cdots\,,\ \text{Mem}\ (2^{32}-1)\ 40\,,$$
$$\text{Status F}\ \}$$

State Representation

A state is a set enumerating state elements. A concrete state:

$$
\begin{aligned}
\{\ & \text{Reg}\ 0\ 820\,,\ \text{Reg}\ 1\ 540\,,\ \cdots,\ \text{Reg}\ 15\ 512\,, \\
& \text{Mem}\ 0\ 34\,,\ \text{Mem}\ 1\ 82\,,\ \cdots,\ \text{Mem}\ (2^{32}-1)\ 40\,, \\
& \text{Status F}\ \}
\end{aligned}
$$

Assertions on parts of states:

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

State Representation

A state is a set enumerating state elements. A concrete state:

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

Assertions on parts of states:

$$\begin{array}{rcl} R\ r\ x & = & \lambda s.\ (s = \{\text{Reg}\ r\ x\}) \\ M\ a\ x & = & \lambda s.\ (s = \{\text{Mem}\ a\ x\}) \\ S\ b & = & \lambda s.\ (s = \{\text{Status}\ b\}) \end{array}$$

$$\begin{array}{rcl} \mathit{split}\ s\ (u, v) & = & (u \cup v = s) \wedge (u \cap v = \emptyset) \\ P * Q & = & \lambda s.\ \exists u\, v.\ \mathit{split}\ s\ (u, v) \wedge P\ u \wedge Q\ v \end{array}$$

Memory Access

The set-based state representation handles all resources uniformly.
Specifications for move and store:

$$\{R\ a\ x * R\ b\ \_\}\quad \texttt{MOV}\ b,a\quad \{R\ a\ x * R\ b\ x\}^{+1}$$

$$\{R\ a\ x * R\ b\ y * M\ y\ \_\}\quad \texttt{STR}\ a,b\quad \{R\ a\ x * R\ b\ y * M\ y\ x\}^{+1}$$

Decrement-and-store:

$$\{R\ a\ x * R\ b\ y * M\ (y-1)\ \_\}$$
$$\texttt{STR}\ a,\texttt{[}b\texttt{,\#-4]!}$$
$$\{R\ a\ x * R\ b\ (y-1) * M\ (y-1)\ x\}^{+1}$$

Memory Access

Define $\mathit{stack}(sp, [x_0, x_1, \cdots, x_m], n)$ to be a stack segment:

$$
\begin{aligned}
& R\ 13\ sp\ * \\
& M\ (sp{+}m)\ x_m * \cdots * M\ (sp{+}1)\ x_1 * M\ sp\ x_0\ * \\
& M\ (sp{-}1)\ \_ * M\ (sp{-}2)\ \_ * \cdots * M\ (sp{-}n)\ \_
\end{aligned}
$$

Memory Access

Define $stack(sp, [x_0, x_1, \cdots, x_m], n)$ to be a stack segment:

$$
\begin{array}{l}
R\ 13\ sp\ * \\
M\ (sp{+}m)\ x_m * \cdots * M\ (sp{+}1)\ x_1 * M\ sp\ x_0\ * \\
M\ (sp{-}1)\ \_ * M\ (sp{-}2)\ \_ * \cdots * M\ (sp{-}n)\ \_
\end{array}
$$

We can transform the specification of decrement-and-store:

$$\{R\ a\ x * R\ 13\ y * M\ (y{-}1)\ \_\}$$

$$\texttt{STR}\ a\texttt{,[13,\#-4]!}$$

$$\{R\ a\ x * R\ 13\ (y{-}1) * M\ (y{-}1)\ x\}^{+1}$$

Memory Access

Define $stack(sp, [x_0, x_1, \cdots, x_m], n)$ to be a stack segment:

$$\begin{array}{l} R\ 13\ sp\ * \\ M\ (sp{+}m)\ x_m * \cdots * M\ (sp{+}1)\ x_1 * M\ sp\ x_0\ * \\ M\ (sp{-}1)\ \_ * M\ (sp{-}2)\ \_ * \cdots * M\ (sp{-}n)\ \_ \end{array}$$

We can transform the specification of decrement-and-store:

$$\{R\ a\ x * R\ 13\ y * M\ (y{-}1)\ \_ * P\}$$

$$\texttt{STR}\ a\texttt{,[13,\#-4]!}$$

$$\{R\ a\ x * R\ 13\ (y{-}1) * M\ (y{-}1)\ x * P\}^{+1}$$

Memory Access

Define $stack(sp, [x_0, x_1, \cdots, x_m], n)$ to be a stack segment:

$$
\begin{array}{l}
R\ 13\ sp\ * \\
M\ (sp{+}m)\ x_m * \cdots * M\ (sp{+}1)\ x_1 * M\ sp\ x_0\ * \\
M\ (sp{-}1)\ \_ * M\ (sp{-}2)\ \_ * \cdots * M\ (sp{-}n)\ \_
\end{array}
$$

We can transform the specification of decrement-and-store:

$$
\begin{array}{c}
\{R\ a\ x * stack(y, xs, n{+}1)\} \\
\texttt{STR}\ a\texttt{,[13,\#-4]!} \\
\{R\ a\ x * stack(y{-}1, cons\ x\ xs, n)\}^{+1}
\end{array}
$$

Positions

The specifications shown so far have been of the form:

$$\{P\}\ code\ \{Q\}^{+k}$$

Positions

The specifications shown so far have been of the form:

$$\{P\}\ \mathit{code}\ \{Q\}^{h}$$

Positions

The specifications shown so far have been of the form:

$$\{P\}^{f}\ \mathit{code}^{\,g}\ \{Q\}^{h}$$

Positions

The specifications shown so far have been of the form:

$$\{P\}^{f}\ code^{\,g}\ \{Q\}^{h}$$

The meaning is best described using a *different* syntax:

$$\forall p. \quad \{f(p) : P\}\ \ g(p) : code\ \ \{h(p) : Q\}$$

Positions

The specifications shown so far have been of the form:

$$\{P\}^{f}\ code^{g}\ \{Q\}^{h}$$

The meaning is best described using a *different* syntax:

$$\forall p. \quad \{f(p) : P\}\ \ g(p) : code\ \ \{h(p) : Q\}$$

For position independent code use $\lambda x.x$ and offsets

$$\forall p. \quad \{p : P\}\ \ p : code\ \ \{p{+}4 : Q\}$$

Positions

The specifications shown so far have been of the form:

$$\{P\}^{f}\ code^{\,g}\ \{Q\}^{h}$$

The meaning is best described using a *different* syntax:

$$\forall p. \quad \{f(p) : P\}\ \ g(p) : code\ \ \{h(p) : Q\}$$

For position independent code use $\lambda x.x$ and offsets

$$\forall p. \quad \{p : P\}\ \ p : code\ \ \{p{+}4 : Q\}$$

For position dependent code use e.g. $\lambda x.0$ and $\lambda x.4$

$$\{0 : P\}\ \ 0 : code\ \ \{4 : Q\}$$

Procedures

Procedures have this form:

$$\forall y.\ \{P * R\ 14\ y\}\ \mathit{code}\ \{Q * R\ 14\ \_\}^{\lambda x.y}$$

which can be understood as:

$$\forall y\ p.\quad \{p : P * R\ 14\ y\}\ \ p : \mathit{code}\ \ \{y : Q * R\ 14\ \_\}$$

Procedures

Procedures have this form:

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

which can be understood as:

$$\forall y\ p.\quad \{p : P * R\ 14\ y\}\ \ p : code\ \ \{y : Q * R\ 14\ \_\}$$

The framework supports procedures by:

1. a rule that calculates the effect of a call
   (derived from the general rule for composition)
2. an induction rule for proving recursive procedures
   (complete induction over the natural numbers)

Procedures

A verified specification for a procedure, which calculates the sum of the nodes in a binary tree:

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

Procedures

The code for BINARY_SUM:

```
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],#+0     ; r14 := node value
      ADD    s,s,r14         ; s := s + r14
      LDR    a,[a],#+4       ; a := address of left
      BL     sum             ; s := s + sum of a
      LDR    a,[r13],#+4     ; a := original a
      LDR    a,[a],#+8       ; a := address of right
      BL     sum             ; s := s + sum of a
      LDR    r15,[r13,#-8]   ; pop two and return
```

Semantics

Define the execution $\rightsquigarrow$ from P to Q as:

$$\forall s \in \Sigma.\ \forall R.\ (P * R)\ s \implies \exists k.\ (Q * R)\ (run(k, s))$$

The meaning of $\{P\}\ c_0; \cdots ; c_n\ \{Q\}^h$ is given by:

$$\begin{array}{ll} \forall p. & (P * M\ p\ c_0 * \cdots * M\ (p+n)\ c_n * R\ 15\ p) \ \rightsquigarrow \\ & (Q * M\ p\ c_0 * \cdots * M\ (p+n)\ c_n * R\ 15\ h(p)) \end{array}$$

The formalised theory generalises $\{P\}\ code\ \{Q\}^h$ to allow multiple entry points, multiple exit points and multiple code segments:

$$\{P_1\}^{f_1} \cdots \{P_n\}^{f_n}\ code_1^{g_1} \cdots\ code_m^{g_m}\ \{Q_1\}^{h_1} \cdots \{Q_k\}^{h_k}$$

Summary

Features:

1. concise and usable specifications
2. finite state space
3. position (in)dependent code
4. (mutually) recursive procedures
5. mechanised in HOL4
6. is being used to verify ARM programs
   (on top of Anthony Fox's ARM model)

For details see my webpage, or ask me now!

`www.cl.cam.ac.uk/~mom22/`

Acknowledgements: I would like to thank Mike Gordon, Anthony Fox, Joe Hurd, Konrad Slind, Thomas Türk, Matthew Parkinson, Josh Berdine, Nick Benton and Richard Bornat for comments and discussions.