Decompilation into Logic — Improved

Magnus O. Myreen, Michael J. C. Gordon
Computer Laboratory, University of Cambridge, UK

Konrad Slind
Rockwell Collins, USA

FMCAD'12 Cambridge

Machine Code

This talk is about machine-code verification.

```
 0: E3A00000
 4: E3510000
 8: 12800001
12: 15911000
16: 1AFFFFFB
```

Verification of Machine Code

Challenges:

machine code

Verification of Machine Code

Challenges:

machine code

correctness

{P} code {Q}

Verification of Machine Code

Challenges:

machine code

code

ARM/x86/PowerPC model
(1000...10,000 lines each)

correctness

{P} code {Q}

Verification of Machine Code

Challenges:

machine code

code

ARM/x86/PowerPC model
(1000...10,000 lines each)

correctness

{P} code {Q}

Contribution: tools/methods which

- expose as little as possible of the big models to the user
- makes non-automatic proofs independent of the models

Decompilation into Logic

Example: given some (hard-to-read) ARM machine code

```
 0: E3A00000       mov r0, #0
 4: E3510000   L:  cmp r1, #0
 8: 12800001       addne r0, r0, #1
12: 15911000       ldrne r1, [r1]
16: 1AFFFFFB       bne L
```

Decompilation into Logic

Example: given some (hard-to-read) ARM machine code

```
 0: E3A00000       mov r0, #0
 4: E3510000   L:  cmp r1, #0
 8: 12800001       addne r0, r0, #1
12: 15911000       ldrne r1, [r1]
16: 1AFFFFFB       bne L
```

our decompiler produces a readable function in HOL:

$$
\begin{aligned}
f(r_0, r_1, m) &= \text{let } r_0 = 0 \text{ in } g(r_0, r_1, m) \\
g(r_0, r_1, m) &= \text{if } r_1 = 0 \text{ then } (r_0, r_1, m) \text{ else} \\
&\qquad \text{let } r_0 = r_0{+}1 \text{ in} \\
&\qquad \text{let } r_1 = m(r_1) \text{ in} \\
&\qquad\quad g(r_0, r_1, m)
\end{aligned}
$$

Certificate Theorems

Decompiler automatically proves a certificate theorem, which states that f describes the effect of the ARM code:

$$f_{pre}(r_0, r_1, m) \Rightarrow$$
$$\{\ (\mathsf{R0}, \mathsf{R1}, \mathsf{M}) \text{ is } (r_0, r_1, m) * \mathsf{PC}\ p * \mathsf{S}\ \}$$
$$p : \texttt{E3A00000 E3510000 12800001 15911000 1AFFFFFB}$$
$$\{\ (\mathsf{R0}, \mathsf{R1}, \mathsf{M}) \text{ is } f(r_0, r_1, m) * \mathsf{PC}\ (p + 20) * \mathsf{S}\ \}$$

Read informally:

if initially reg 0, reg 1 and memory described by (r_0, r_1, m) , then the code terminates with reg 0, reg 1, memory as $f(r_0, r_1, m)$

Preconditions

Precondition f_{pre} keeps track of side conditions:

$$
\begin{aligned}
f\_pre(r_0, r_1, m) \;=\; & \text{let } r_0 = 0 \text{ in } g\_pre(r_0, r_1, m) \\
g\_pre(r_0, r_1, m) \;=\; & \text{if } r_1 = 0 \text{ then } \mathit{true} \text{ else} \\
& \quad \text{let } r_0 = r_0{+}1 \text{ in} \\
& \quad \text{let } \mathit{cond} = r_1 \in \text{domain } m \wedge \mathit{aligned}(r_1) \text{ in} \\
& \quad \text{let } r_1 = m(r_1) \text{ in} \\
& \qquad g\_pre(r_0, r_1, m) \wedge \mathit{cond}
\end{aligned}
$$

Decompilation into Logic

Strengths:

- ☑ separates definition of ISA model from program verification (program verification is done based on decompiler output)
- ☑ can implement proof-producing program synthesis from HOL based on decompilation (translation validation)
- ☑ has been shown to scale to large verification projects:
  - functional correctness of garbage collectors,
  - Lisp implementations (ARM, x86, PowerPC), and
  - the seL4 microkernel (approx. 12,000 lines of ARM)

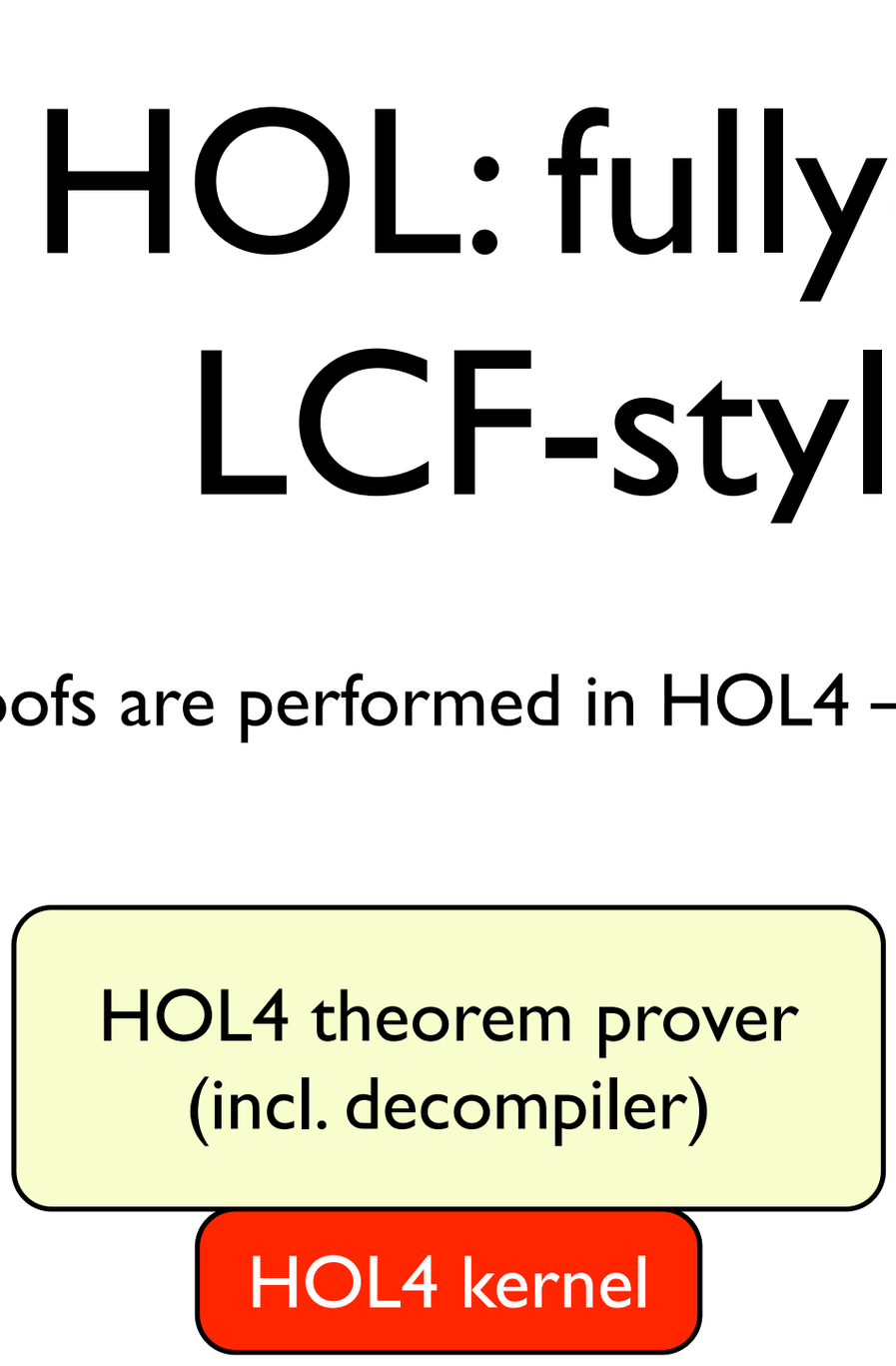
Performance Ignored...

Weaknesses:

- unnecessarily complicated (for historical reasons, uses ideas from separation logic that are irrelevant to decompilation)
- sometimes very slow (never optimised for speed, separation logic composition rule slow to execute in LCF-style prover)
  - decompilation of the seL4 microkernel (approx. 12,000 lines of ARM) takes 8 hours (for gcc -O2 output)
- not applicable to code with general-purpose code pointers, e.g. jump to code pointer.

HOL: fully-expansive LCF-style prover

Proofs are performed in HOL4 — a fully expansive theorem prover

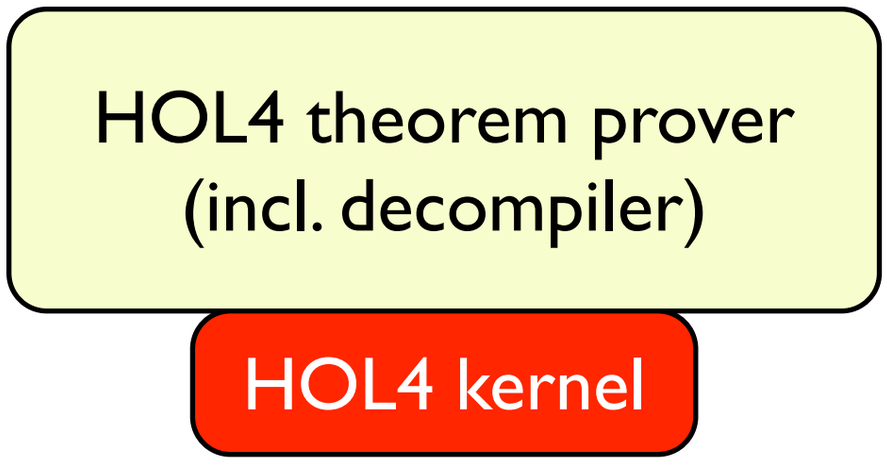


All proofs expand at runtime into primitive inferences in the HOL4 kernel.

The kernel implements the axioms and inference rules of higher-order logic.

HOL: fully-expansive LCF-style prover

Proofs are performed in HOL4 — a fully expansive theorem prover



All proofs expand at runtime into primitive inferences in the HOL4 kernel.

The kernel implements the axioms and inference rules of higher-order logic.

Example: proving 10+1=11 using the simplifier requires 85 primitive inferences (0.0003 seconds). No hope of producing a very fast tool...

This Talk:
Improving Decompilation

Weaknesses of old approach:

- unnecessarily complicated
- sometimes very slow
- cannot handle code pointers

This Talk:
Improving Decompilation

Weaknesses of old approach:

- unnecessarily complicated
- sometimes very slow
- cannot handle code pointers

Contribution:

- ☑ simpler Hoare logic
- ☑ revised approach for better speed
- ☑ support for code pointers

New Hoare triple

Decompiler performs proofs in terms Hoare triples:

$$\{\, pre \,\}\ code\ \{\, post \,\}$$

New Hoare triple

Decompiler performs proofs in terms Hoare triples:

$$\{\, pre \,\}\ code\ \{\, post \,\}$$

Semantics parametrised by $assert$ and $next$:

$$\begin{aligned} \forall state\ c.\ & assert\ (code \cup c, pre)\ state \implies \\ & \exists n.\ assert\ (code \cup c, post)\ (next^n(state)) \end{aligned}$$

New Hoare triple

Decompiler performs proofs in terms Hoare triples:

$$\{\, pre \,\}\ code\ \{\, post \,\}$$

Semantics parametrised by $assert$ and $next$:

$$\begin{aligned} \forall state\ c.\ assert\ (code \cup c, pre)\ state \implies \\ \exists n.\ assert\ (code \cup c, post)\ (next^n(state)) \end{aligned}$$

We instantiate these for each architecture (ARM, x86, PowerPC), e.g.

$$\begin{aligned} &\text{arm\_assert}\ (code, pc, r_0, r_1, \ldots, cond)\ state = \\ &\quad (cond \implies code \text{ is in memory of } state \text{ and} \\ &\qquad \text{the PC of } state \text{ is } pc \text{ and } \ldots) \end{aligned}$$

New Hoare triple

Decompiler performs proofs in terms Hoare triples:

$$\{\, pre \,\}\ code\ \{\, post \,\}$$

Semantics parametrised by $assert$ and $next$:

$$\begin{aligned}\forall state\ c.\ & assert\ (code \cup c, pre)\ state \implies \\ & \exists n.\ assert\ (code \cup c, post)\ (next^n(state))\end{aligned}$$

We instantiate these fo

pre/post are tuples, allows fast composition

$$\begin{aligned}&\text{arm\_assert}\ (code, pc, r_0, r_1, \ldots, cond)\ state = \\ &\quad (cond \implies code \text{ is in memory of } state \text{ and} \\ &\qquad \text{the PC of } state \text{ is } pc \text{ and } \ldots)\end{aligned}$$

New Hoare triple

Decompiler performs proofs in terms Hoare triples:

$$\{\, pre \,\}\ code\ \{\, post \,\}$$

Semantics parametrised by $assert$ and $next$:

$$\begin{aligned}\forall state\ c.\ & assert\ (code \cup c, pre)\ state \implies \\ & \exists n.\ assert\ (code \cup c, post)\ (next^n(state))\end{aligned}$$

We instantiate these fo

pre/post are tuples, allows fast composition

$$\begin{aligned}&\text{arm\_assert}\ (code, pc, r_0, r_1, \ldots, cond)\ state = \\ &\quad (cond \implies code \text{ is in memory of } state \text{ and} \\ &\qquad \text{the PC of } state \text{ is } pc \text{ and } \ldots)\end{aligned}$$

side-condition is accumulated in ‘postcondition’

New Hoare triple

Decompiler performs proofs in terms Hoare triples:

$$\{\, pre \,\}\ code\ \{\, post \,\}$$

Semantics parametrised by $assert$ and $next$:

$$\forall state\ c.\ assert\ (code \cup c, pre)\ state \implies$$

program counter value is explicitly part of pre/post

We instantiate these fo

p e/post are tuples, allows fast composition

$$\text{arm\_assert}\ (code, pc, r_0, r_1, \ldots, cond)\ state =$$
$$(cond \implies code \text{ is in memory of } state \text{ and}$$
$$\text{the PC of } state \text{ is } pc \text{ and } \ldots)$$

side-condition is accumulated in ‘postcondition’

Function extraction

Decompilation algorithm:

Step 1: evaluate underlying ISA model
(prove Hoare triples for each instruction)

Step 2: construct CFG and find 'decompilation rounds'
(usually one round per loop)

Step 3: for each round, compose a Hoare triple theorem:

$$\{pre[v_0 \ldots v_n]\}$$
$$code$$
$$\{\text{let } (v'_0 \ldots v'_n) = f(v_0 \ldots v_n) \text{ in } post[v'_0 \ldots v'_n]\}$$

if the code contains a loop, apply a loop rule (next slide...)

Function extraction

Decompilation algorithm:

Step 1: evaluate underlying ISA model
(prove Hoare triples for each instruction)

Step 2: construct CFG and find 'decompilation rounds'
(usually one round per loop)

Step 3: for each round, compose a Hoare triple theorem:

$$\{pre[v_0 \ldots v_n]\}$$
$$code$$
$$\{\text{let } (v'_0 \ldots v'_n) = f(v_0 \ldots v_n) \text{ in } post[v'_0 \ldots v'_n]\}$$

if the code

collects both update and side-conditions

Loop Rule

If there are loops in the code then apply:

$$(\forall x.\ \{pre\ x\}\, code\, \{\text{if } g\ x \text{ then } pre\ (f\ x) \text{ else } post\ (d\ x)\}$$
$$\Longrightarrow$$
$$(\forall x.\ \{pre\ x\}\, code\, \{post\ (\text{tailrec } g\ f\ d\ x))\}$$

where tailrec is a function format that satisfies:

$$\text{tailrec } g\ f\ d\ x = \text{if } g\ x \text{ then tailrec } g\ f\ d\ (f\ x) \text{ else } d\ x$$

Definition of tailrec is in the paper.

Example

Assembly code:

```
L0: ldr r1,[r2,r3]   ; load mem[r2+r3] into r1
L1: add r0,r1        ; add r1 to r0
L2: subs r3,#4       ; decrement r3 by 4
L3: bne L0           ; goto L0 if r3 ≠ 0
L4:
```

Example

Assembly code:

```
L0: ldr r1,[r2,r3]   ; load mem[r2+r3] into r1
L1: add r0,r1        ; add r1 to r0
L2: subs r3,#4       ; decrement r3 by 4
L3: bne L0           ; goto L0 if r3 ≠ 0
L4:
```

Extracted function:

$$\begin{aligned}
&\mathsf{sum}(cond, r_0, r_1, r_2, r_3, m) = \\
&\quad \text{let } cond = cond \wedge \mathsf{valid_address}\ (r_2 + r_3)\ m \text{ in} \\
&\quad \text{let } r_1 = m(r_2 + r_3) \text{ in} \\
&\quad \text{let } r_0 = r_0 + r_1 \text{ in} \\
&\quad \text{let } r_3 = r_3 - 4 \text{ in} \\
&\qquad \text{if } r_3 = 0 \text{ then } (cond, r_0, r_1, r_2, r_3, m) \\
&\qquad\qquad \text{else } \mathsf{sum}(cond, r_0, r_1, r_2, r_3, m)
\end{aligned}$$

Example

Assembly code:

```
L0: ldr r1,[r2,r3]   ; load mem[r2+r3] into r1
L1: add r0,r1        ; add r1 to r0
L2: subs r3,#4       ; decrement r3 by 4
L3: bne L0           ; goto L0 if r3 ≠ 0
L4:
```

Extracted function:

$$\text{sum}(cond, r_0, r_1, r_2, r_3, m) =$$
$$\text{let } cond = cond \wedge \text{valid\_address } (r_2 + r_3)\ m \text{ in}$$
$$\text{let } r_1 = m(r_2 + r_3) \text{ in}$$
$$\text{let } r_0 = r_0 + r_1 \text{ in}$$
$$\text{let } r_3 = r_3 - 4 \text{ in}$$
$$\text{if } r_3 = 0 \text{ then } (cond, r_0, r_1, r_2, r_3, m)$$
$$\text{else sum}(cond, r_0, r_1, r_2, r_3, m)$$

side-conditions part of function

Example

Assembly code:

```
L0: ldr r1,[r2,r3]   ; load mem[r2+r3] into r1
L1: add r0,r1        ; add r1 to r0
L2: subs r3,#4       ; decrement r3 by 4
L3: bne L0           ; goto L0 if r3 ≠ 0
L4:
```

Extracted function:

$$\begin{aligned}
&\mathsf{sum}(cond, r_0, r_1, r_2, r_3, m) = \\
&\quad \text{let } cond = cond \wedge \mathsf{valid_address}\ (r_2 + r_3)\ m \text{ in} \\
&\quad \text{let } r_1 = m(r_2 + r_3) \text{ in} \\
&\quad \text{let } r_0 = r_0 + r_1 \text{ in} \\
&\quad \text{let } r_3 = r_3 - 4 \text{ in} \\
&\qquad \text{if } r_3 = 0 \text{ then } (cond, r_0, r_1, r_2, r_3, m) \\
&\qquad\qquad \text{else } \mathsf{sum}(cond, r_0, r_1, r_2, r_3, m)
\end{aligned}$$

side-conditions part of function

Certificate theorem:

$$(\mathsf{sum}(c, r_0, r_1, r_2, r_3, m) = (\mathsf{true}, r_0', r_1', r_2', r_3', m')) \Longrightarrow$$

$$\{\ \text{ARM state holds } (r_0, r_1, r_2, r_3, m)\ \}$$

```
E7921003 E0800001 E2533004 1AFFFFFB
```

$$\{\ \text{ARM state holds } (r_0', r_1', r_2', r_3', m')\ \}$$

Example

Assembly code:

```
L0: ldr r1,[r2,r3]   ; load mem[r2+r3] into r1
L1: add r0,r1        ; add r1 to r0
L2: subs r3,#4       ; decrement r3 by 4
L3: bne L0           ; goto L0 if r3 ≠ 0
L4:
```

Extracted function:

$$\begin{aligned}&\mathsf{sum}(cond, r_0, r_1, r_2, r_3, m) = \\ &\text{let } cond = cond \wedge \mathsf{valid\_address}\ (r_2 + r_3)\ m \text{ in} \\ &\text{let } r_1 = m(r_2 + r_3) \text{ in} \\ &\text{let } r_0 = r_0 + r_1 \text{ in} \\ &\text{let } r_3 = r_3 - 4 \text{ in} \\ &\quad \text{if } r_3 = 0 \text{ then } (cond, r_0, r_1, r_2, r_3, m) \\ &\qquad\qquad \text{else } \mathsf{sum}(cond, r_0, r_1, r_2, r_3, m)\end{aligned}$$

side-conditions part of function

side-conditions must be true

Certificate theorem:

$$\begin{aligned}&(\mathsf{sum}(c, r_0, r_1, r_2, r_3, m) = (\mathsf{true}, r_0', r_1', r_2', r_3', m')) \Longrightarrow \\ &\{\ \text{ARM state holds } (r_0, r_1, r_2, r_3, m)\ \} \\ &\texttt{E7921003 E0800001 E2533004 1AFFFFFB} \\ &\{\ \text{ARM state holds } (r_0', r_1', r_2', r_3', m')\ \}\end{aligned}$$

Code pointers

Assembly code:

```
L0: ldr r4,[r5,r6]   ; load mem[r5+r6] into r4
L1: blx r4           ; call code-pointer r4
L2: subs r6,#4       ; decrement r6 by 4
L3: bne L0           ; goto L0 if r6 ≠ 0
L4:
```

Code pointers

Assembly code:

```
L0: ldr r4,[r5,r6]   ; load mem[r5+r6] into r4
L1: blx r4           ; call code-pointer r4
L2: subs r6,#4       ; decrement r6 by 4
L3: bne L0           ; goto L0 if r6 ≠ 0
L4:
```

call to code pointer

Code pointers

Assembly code:

```
L0: ldr r4,[r5,r6]   ; load mem[r5+r6] into r4
L1: blx r4           ; call code-pointer r4
L2: subs r6,#4       ; decrement r6 by 4
L3: bne L0           ; goto L0 if r6 ≠ 0
L4:
```

call to code pointer

Extracted function:

$$
\begin{aligned}
&\text{calls}(cond, pc, r_4, r_5, r_6, r_{14}, m) = \\
&\quad \text{if } pc = \texttt{L0} \text{ then} \\
&\qquad \text{let } cond = cond \wedge \text{valid\_address } (r_5 + r_6)\ m \text{ in} \\
&\qquad \text{let } r_4 = m(r_5 + r_6) \text{ in} \\
&\qquad \text{let } cond = cond \wedge \text{word\_aligned\_address } r_4 \text{ in} \\
&\qquad \text{let } (pc, r_{14}) = (r_4, \texttt{L2}) \text{ in} \\
&\qquad\quad (cond, pc, r_4, r_5, r_6, r_{14}, m) \\
&\quad \text{else if } pc = \texttt{L2} \text{ then} \\
&\qquad \text{let } r_6 = r_6 - 4 \text{ in} \\
&\qquad\quad \text{if } r_6 = 0 \text{ then } (cond, \texttt{L4}, r_4, r_5, r_6, r_{14}, m) \\
&\qquad\qquad\qquad \text{else } (cond, \texttt{L0}, r_4, r_5, r_6, r_{14}, m) \\
&\quad \text{else } (cond, pc, r_4, r_5, r_6, r_{14}, m)
\end{aligned}
$$

Code pointers

Assembly code:

```
L0: ldr r4,[r5,r6]   ; load mem[r5+r6] into r4
L1: blx r4           ; call code-pointer r4
L2: subs r6,#4       ; decrement r6 by 4
L3: bne L0           ; goto L0 if r6 ≠ 0
L4:
```

call to code pointer

PC is an input

Extracted function:

calls$(cond, pc, r_4, r_5, r_6, r_{14}, m)$ =

if $pc = \texttt{L0}$ then
  let $cond = cond \wedge$ valid_address $(r_5 + r_6)\ m$ in
  let $r_4 = m(r_5 + r_6)$ in
  let $cond = cond \wedge$ word_aligned_address r_4 in
  let $(pc, r_{14}) = (r_4, \texttt{L2})$ in
    $(cond, pc, r_4, r_5, r_6, r_{14}, m)$
else if $pc = \texttt{L2}$ then
  let $r_6 = r_6 - 4$ in
    if $r_6 = 0$ then $(cond, \texttt{L4}, r_4, r_5, r_6, r_{14}, m)$
      else $(cond, \texttt{L0}, r_4, r_5, r_6, r_{14}, m)$
else $(cond, pc, r_4, r_5, r_6, r_{14}, m)$

Code pointers

Assembly code:

```
L0: ldr r4,[r5,r6]   ; load mem[r5+r6] into r4
L1: blx r4           ; call code-pointer r4
L2: subs r6,#4       ; decrement r6 by 4
L3: bne L0           ; goto L0 if r6 ≠ 0
L4:
```

call to code pointer

PC is an input

Extracted function:

$\text{calls}(cond, pc, r_4, r_5, r_6, r_{14}, m) =$

test for value of PC

if $pc = \texttt{L0}$ then
  let $cond = cond \wedge \text{valid\_address } (r_5 + r_6)\ m$ in
  let $r_4 = m(r_5 + r_6)$ in
  let $cond = cond \wedge \text{word\_aligned\_address } r_4$ in
  let $(pc, r_{14}) = (r_4, \texttt{L2})$ in
    $(cond, pc, r_4, r_5, r_6, r_{14}, m)$
else if $pc = \texttt{L2}$ then
  let $r_6 = r_6 - 4$ in
    if $r_6 = 0$ then $(cond, \texttt{L4}, r_4, r_5, r_6, r_{14}, m)$
      else $(cond, \texttt{L0}, r_4, r_5, r_6, r_{14}, m)$
else $(cond, pc, r_4, r_5, r_6, r_{14}, m)$

Code pointers

Assembly code:

```
L0: ldr r4,[r5,r6]   ; load mem[r5+r6] into r4
L1: blx r4           ; call code-pointer r4
L2: subs r6,#4       ; decrement r6 by 4
L3: bne L0           ; goto L0 if r6 ≠ 0
L4:
```

call to code pointer

PC is an input

Extracted function:

$$
\begin{aligned}
&\text{calls}(cond, pc, r_4, r_5, r_6, r_{14}, m) = \\
&\quad \text{if } pc = \texttt{L0} \text{ then} \\
&\qquad \text{let } cond = cond \wedge \text{valid\_address } (r_5 + r_6)\ m \text{ in} \\
&\qquad \text{let } r_4 = m(r_5 + r_6) \text{ in} \\
&\qquad \text{let } cond = cond \wedge \text{word\_aligned\_address } r_4 \text{ in} \\
&\qquad \text{let } (pc, r_{14}) = (r_4, \texttt{L2}) \text{ in} \\
&\qquad\quad (cond, pc, r_4, r_5, r_6, r_{14}, m) \\
&\quad \text{else if } pc = \texttt{L2} \text{ then} \\
&\qquad \text{let } r_6 = r_6 - 4 \text{ in} \\
&\qquad\quad \text{if } r_6 = 0 \text{ then } (cond, \texttt{L4}, r_4, r_5, r_6, r_{14}, m) \\
&\qquad\qquad\quad \text{else } (cond, \texttt{L0}, r_4, r_5, r_6, r_{14}, m) \\
&\quad \text{else } (cond, pc, r_4, r_5, r_6, r_{14}, m)
\end{aligned}
$$

test for value of PC

PC updated

Code pointers

Assembly code:

```
L0: ldr r4,[r5,r6]   ; load mem[r5+r6] into r4
L1: blx r4           ; call code-pointer r4
L2: subs r6,#4       ; decrement r6 by 4
L3: bne L0           ; goto L0 if r6 ≠ 0
L4:
```

call to code pointer

PC is an input

Extracted function:

$$\begin{aligned}
&\text{calls}(cond, pc, r_4, r_5, r_6, r_{14}, m) = \\
&\quad \text{if } pc = \texttt{L0} \text{ then} \\
&\quad\quad \text{let } cond = cond \wedge \text{valid_address } (r_5 + r_6)\ m \text{ in} \\
&\quad\quad \text{let } r_4 = m(r_5 + r_6) \text{ in} \\
&\quad\quad \text{let } cond = cond \wedge \text{word_aligned_address } r_4 \text{ in} \\
&\quad\quad \text{let } (pc, r_{14}) = (r_4, \texttt{L2}) \text{ in} \\
&\quad\quad\quad (cond, pc, r_4, r_5, r_6, r_{14}, m) \\
&\quad \text{else if } pc = \texttt{L2} \text{ then} \\
&\quad\quad \ldots
\end{aligned}$$

test for value of PC

Certificate theorem:

$$\begin{aligned}
&(\text{calls}(c, pc, r_4, r_5, r_6, r_{14}, m) = (\text{true}, pc', r'_4, \ldots)) \Longrightarrow \\
&\{\ \text{ARM state holds } (r_4, r_5, r_6, r_{14}, m) \text{ and PC is } pc\ \} \\
&\texttt{E7954006 E12FFF34 E2566004 1AFFFFFB} \\
&\{\ \text{ARM state holds } (r'_4, r'_5, r'_6, r'_{14}, m') \text{ and PC is } pc'\ \}
\end{aligned}$$

Performance Numbers

Comparison between new and old approach.

Cost given in seconds (s) and HOL inference rules (i).

ARM machine code	instr.	time/cost of old	time/cost of new	reduction	model eval.
sum of array (Sec. I-A)	4	2.5 s (73,039 i)	0.3 s (4,019 i)	86 % (94 %)	7.8 s (1.5 Mi)
copying garbage collector [10]	89	50 s (1,526,281 i)	6.0 s (53,301 i)	88 % (97 %)	173 s (40 Mi)
1024-bit multiword addition	224	70 s (1,029,685 i)	1.2 s (10,802 i)	98 % (99 %)	37 s (8.9 Mi)
256-bit Skein hash function	1,352	5,366 s (21,432,926 i)	56 s (1,842,642 i)	99 % (91 %)	500 s (105 Mi)

Cost of evaluating the ISA model is separate as this cost is independent of decompilation approach.

Performance Numbers

Comparison between new and old approach.

Cost given in seconds (s) and HOL inference rules (i).

ARM machine code	instr.	time/cost of old	time/cost of new	reduction	model eval.
sum of array (Sec. I-A)	4	2.5 s (73,039 i)	0.3 s (4,019 i)	86 % (94 %)	7.8 s (1.5 Mi)
copying garbage collector [10]	89	50 s (1,526,281 i)	6.0 s (53,301 i)	88 % (97 %)	173 s (40 Mi)
1024-bit multiword addition	224	70 s (1,029,685 i)	1.2 s (10,802 i)	98 % (99 %)	37 s (8.9 Mi)
256-bit Skein hash function	1,352	5,366 s (21,432,926 i)	56 s (1,842,642 i)	99 % (91 %)	500 s (105 Mi)

Cost of evaluating the ISA model is separate as this cost is independent of decompilation approach.

Performance Numbers

Comparison between new and old approach.

Cost given in seconds (s) and HOL inference rules (i).

ARM machine code	instr.	time/cost of old	time/cost of new	reduction	model eval.
sum of array (Sec. I-A)	4	2.5 s (73,039 i)	0.3 s (4,019 i)	86 % (94 %)	7.8 s (1.5 Mi)
copying garbage collector [10]	89	50 s (1,526,281 i)	6.0 s (53,301 i)	88 % (97 %)	173 s (40 Mi)
1024-bit multiword addition	224	70 s (1,029,685 i)	1.2 s (10,802 i)	98 % (99 %)	37 s (8.9 Mi)
256-bit Skein hash function	1,352	5,366 s (21,432,926 i)	56 s (1,842,642 i)	99 % (91 %)	500 s (105 Mi)

Cost of evaluating the ISA model is separate as this cost is independent of decompilation approach.

More efficient ISA models...

Better, faster ISA models are in the pipeline...

ITP'12

Directions in ISA Specification

Anthony Fox

Computer Laboratory, University of Cambridge, UK

Abstract. This rough diamond presents a new domain-specific language (DSL) for producing detailed models of Instruction Set Architectures, such as ARM and x86. The language's design and methodology is discussed and we propose future plans for this work. Feedback is sought from the wider theorem proving community in helping establish future directions for this project. A parser and interpreter for the DSL has been developed in Standard ML, with an ARMv7 model used as a case study.

This paper describes recent work on developing a domain-specific language (DSL) for Instruction Set Architecture (ISA) specification. Various theorem

Summary

Decompilation:

- ☑ extracts functions from machine code (ARM, x86, PowerPC)
- ☑ proves that the extracted functions are faithful to the code
- ☑ useful in proof of full functional correctness (e.g. seL4, Lisp)

Summary

Decompilation:

- ☑ extracts functions from machine code (ARM, x86, PowerPC)
- ☑ proves that the extracted functions are faithful to the code
- ☑ useful in proof of full functional correctness (e.g. seL4, Lisp)

Improvements in this paper:

- ☑ simplified Hoare logic for easier mechanisation/automation
- ☑ significantly improved performance
- ☑ now more widely applicable (support for code pointers)

Summary

Decompilation:

- ☑ extracts functions from machine code (ARM, x86, PowerPC)
- ☑ proves that the extracted functions are faithful to the code
- ☑ useful in proof of full functional correctness (e.g. seL4, Lisp)

Improvements in this paper:

- ☑ simplified Hoare logic for easier mechanisation/automation
- ☑ significantly improved performance
- ☑ now more widely applicable (support for code pointers)

Questions?