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: 1AFFFFFFB
Verification of Machine Code

Challenges:

machine code

code
Verification of Machine Code

Challenges:

machine code

\{P\} code \{Q\}

correctness
Verification of Machine Code

Challenges:

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

- Correctness
  \{P\} code \{Q\}
Verification of Machine Code

Challenges:

Contribution: tools/methods which

- expose as little as possible of the big models to the user
- makes non-automatic proofs independent of the models
Decomposition 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: 1AFFFFFFFB  bne L
```
Decomposition 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:

\[
\begin{align*}
  f(r_0, r_1, m) &= \text{let } r_0 = 0 \text{ in } g(r_0, r_1, m) \\
  g(r_0, r_1, m) &= \text{if } r_1 = 0 \text{ then } (r_0, r_1, m) \text{ else } \\
                     &\quad \text{let } r_0 = r_0 + 1 \text{ in } \\
                     &\quad \text{let } r_1 = m(r_1) \text{ in } \\
                     &\quad g(r_0, r_1, m)
\end{align*}
\]
Certificate Theorems

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

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

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:

\[
\begin{align*}
  f_{pre}(r_0, r_1, m) &= \text{let } r_0 = 0 \text{ in } g_{pre}(r_0, r_1, m) \\
  g_{pre}(r_0, r_1, m) &= \text{if } r_1 = 0 \text{ then } \text{true} \text{ else} \\
  &\quad \text{let } r_0 = r_0 + 1 \text{ in} \\
  &\quad \text{let } \text{cond} = r_1 \in \text{domain } m \land \text{aligned}(r_1) \text{ in} \\
  &\quad \text{let } r_1 = m(r_1) \text{ in} \\
  &\quad g_{pre}(r_0, r_1, m) \land \text{cond}
\end{align*}
\]
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:

\{ \textit{pre} \} \textit{code} \{ \textit{post} \}
New Hoare triple

Decompiler performs proofs in terms Hoare triples:

\[ \{ \text{pre} \} \text{code} \{ \text{post} \} \]

Semantics parametrised by \textit{assert} and \textit{next}:

\[ \forall \text{state } c. \ \text{assert} \ (\text{code} \cup c, \text{pre}) \ \text{state} \implies \exists n. \ \text{assert} \ (\text{code} \cup c, \text{post}) \ (\text{next}^n(\text{state})) \]
New Hoare triple

Decompiler performs proofs in terms Hoare triples:

\{ \textit{pre} \} \textit{code} \{ \textit{post} \}

Semantics parametrised by \textit{assert} and \textit{next}:

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

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

\texttt{arm_assert} \ (\textit{code}, \textit{pc}, r_0, r_1, \ldots, \textit{cond}) \ \textit{state} =
(\textit{cond} \implies \textit{code} \text{ is in memory of } \textit{state} \text{ and}
\text{ the PC of } \textit{state} \text{ is } \textit{pc} \text{ and } \ldots)
New Hoare triple

Decompiler performs proofs in terms Hoare triples:

\[ \{ \text{pre} \} \text{code} \{ \text{post} \} \]

Semantics parametrised by \textit{assert} and \textit{next}:

\[
\forall \text{state } c. \ \text{assert} \ (\text{code} \cup c, \text{pre}) \ \text{state} \implies \\
\exists n. \ \text{assert} \ (\text{code} \cup c, \text{post}) \ (\text{next}^n(\text{state}))
\]

We instantiate these for:

\[
\text{arm_assert} \ (\text{code}, \text{pc}, r_0, r_1, \ldots, \text{cond}) \ \text{state} = \\
(\text{cond} \implies \text{code} \text{ is in memory of } \text{state} \text{ and} \\
\text{the PC of } \text{state} \text{ is } \text{pc} \text{ and } \ldots)
\]
New Hoare triple

Decompiler performs proofs in terms Hoare triples:

\{ \text{pre} \} \text{code} \{ \text{post} \}

Semantics parametrised by assert and next:

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

We instantiate these for:

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

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. \text{assert} (code \cup c, pre) state \implies

program counter value is explicitly part of pre/post

We instantiate these for pre/post are tuples, allows fast composition

\text{arm_assert} (code, pc, r_0, r_1, \ldots, cond) state =

\left( \sigma \implies code \text{ is in memory of } state \text{ and}
\right. 
\left. 
\text{the PC of } state \text{ is } pc \text{ and } \ldots \right)

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:

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

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

Decomposition 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:

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

if the code collects both update and side-conditions
Loop Rule

If there are loops in the code then apply:

\[(\forall x. \{\text{pre } x\} \text{ code } \{\text{if } g \ x \text{ then } \text{pre } (f \ x) \text{ else } \text{post } (d \ x)\}) \Rightarrow (\forall x. \{\text{pre } x\} \text{ code } \{\text{post } (\text{tailrec } g \ f \ d \ x)\})\]

where tailrec is a function format that satisfies:

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

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:

sum(\text{cond}, r_0, r_1, r_2, r_3, m) =
let \text{cond} = \text{cond} \land \text{valid_address} (r_2 + r_3) m \text{ in}
let r_1 = m(r_2 + r_3) \text{ in}
let r_0 = r_0 + r_1 \text{ in}
let r_3 = r_3 - 4 \text{ in}
if r_3 = 0 then (\text{cond}, r_0, r_1, r_2, r_3, m)
else sum(\text{cond}, 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 \neq 0
L4:

Extracted function:

\[
\text{sum}(\text{cond}, r_0, r_1, r_2, r_3, m) = \\
\text{let } \text{cond} = \text{cond} \land \text{valid_address (r_2 + r_3) m in} \\
\text{let } r_1 = m(r_2 + r_3) \text{ in} \\
\text{let } r_0 = r_0 + r_1 \text{ in} \\
\text{let } r_3 = r_3 - 4 \text{ in} \\
\text{if } r_3 = 0 \text{ then (cond, r_0, r_1, r_2, r_3, m) else sum (cond, r_0, r_1, r_2, r_3, m)}
\]

side-conditions part of function
The technique described in this paper has been implemented in the presence of code in ARM, x86 and PowerPC [10], [11], and decompilation also for implementation of proof-producing synthesis tools, as we have demonstrated that this decompilation technique can significantly ease verification of machine-code programs, which aids verification of machine-code programs. This technique permits reasoning directly in the underlying machine-code Hoare logic, even when the code captures the functional behaviour of the machine code.

In this paper, we focus on decompilation into logic. Given some concrete machine code from which it was extracted, the certificate for execution of this machine code it extracts a function which describes the side-condition must hold for correct execution. The side-condition is parametrised by the ISA model. The Hoare triple which is parametrised by the ISA model. The decompilation takes concrete machine code as input. From this machine code, it extracts a function which describes the C code above can be compiled to ARM assembly:

```
E7921003 E0800001 E2533004 1AFFFFFB
```

This function can be assembled into ARM machine 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:
```

We state these certificate theorems using a machine-code theorem prover [16] and applied to ARM machine code.

The improvements make the new approach faster, simpler and more generally applicable. In particular, the new technique allows the verifier to avoid tedious decompilation efforts which can significantly ease verification of machine-code programs. This new technique is an overview of MOD sponsored research and is released to inform projects that include high integrity machine code and a model of an instruction set architecture (ISA), this decompilation extracts functions (defined in logic). Given some concrete machine code the new function to the component.

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:**

```
sum(cond, r0, r1, r2, r3, m) = 
let cond = cond ∧ valid_address (r2 + r3) m in 
let r1 = m(r2 + r3) in 
let r0 = r0 + r1 in 
let r3 = r3 - 4 in 
if r3 = 0 then (cond, r0, r1, r2, r3, m) 
else sum(cond, r0, r1, r2, r3, m)
```

**Certificate theorem:**

```
(sum(c, r0, r1, r2, r3, m) = (true, r'_0, r'_1, r'_2, r'_3, m')) \implies 
\{ ARM state holds (r0, r1, r2, r3, m) \} 
E7921003 E0800001 E2533004 1AFFFFFB 
\{ ARM state holds (r'_0, r'_1, r'_2, r'_3, m') \}
```
Example

Assembly code:

L0: 1dr 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:

\[ \text{sum}(\text{cond}, r_0, r_1, r_2, r_3, m) = \]
\[ \text{let } \text{cond} = \text{cond} \land \text{valid_address } (r_2 + r_3) \text{ m in} \]
\[ \text{let } r_1 = m(r_2 + r_3) \text{ in} \]
\[ \text{let } r_0 = r_0 + r_1 \text{ in} \]
\[ \text{let } r_3 = r_3 - 4 \text{ in} \]
\[ \text{if } r_3 = 0 \text{ then } (\text{cond}, r_0, r_1, r_2, r_3, m) \]
\[ \text{else } \text{sum}(\text{cond}, r_0, r_1, r_2, r_3, m) \]

Certificate theorem:

\[ \text{(sum}(c, r_0, r_1, r_2, r_3, m) = (\text{true}, r'_0, r'_1, r'_2, r'_3, m')) \implies \]
\{ ARM state holds \( (r_0, r_1, r_2, r_3, m) \) \}
E7921003 E0800001 E2533004 1AFFFFFFB
\{ ARM state holds \( (r'_0, r'_1, r'_2, r'_3, m') \) \}
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:
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: 

Extracted function:

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

\[
\begin{align*}
\text{if } pc = L0 \text{ then} \\
\quad & \text{let } \text{cond} = \text{cond} \land \text{valid_address}(r_5 + r_6) \text{ m in} \\
\quad & \text{let } r_4 = m(r_5 + r_6) \text{ in} \\
\quad & \text{let } \text{cond} = \text{cond} \land \text{word_aligned_address } r_4 \text{ in} \\
\quad & \text{let } (pc, r_{14}) = (r_4, L2) \text{ in} \\
\quad & \quad (\text{cond}, \text{pc}, r_4, r_5, r_6, r_{14}, m) \\
\text{else if } pc = L2 \text{ then} \\
\quad & \text{let } r_6 = r_6 - 4 \text{ in} \\
\quad & \quad \text{if } r_6 = 0 \text{ then } (\text{cond}, L4, r_4, r_5, r_6, r_{14}, m) \\
\quad & \quad \quad \text{else } (\text{cond}, L0, r_4, r_5, r_6, r_{14}, m) \\
\text{else } (\text{cond}, \text{pc}, r_4, r_5, r_6, r_{14}, m)
\end{align*}
\]
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:

Extracted function:

calls(cond, pc, r4, r5, r6, r14, m) =
if pc = L0 then
  let cond = cond ∧ valid_address (r5 + r6) m in
  let r4 = m(r5 + r6) in
  let cond = cond ∧ word_aligned_address r4 in
  let (pc, r14) = (r4, L2) in
  (cond, pc, r4, r5, r6, r14, m)
else if pc = L2 then
  let r6 = r6 - 4 in
  if r6 = 0 then (cond, L4, r4, r5, r6, r14, m)
  else (cond, L0, r4, r5, r6, r14, m)
else (cond, pc, r4, r5, r6, r14, m)
Code pointers

Assembly code:

L0: 1dr 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 \neq 0
L4:                

call to code pointer

Extracted function:

calls(\text{cond, pc, } r_4, r_5, r_6, r_{14}, m) =
\begin{align*}
\text{if } pc = L0 \text{ then} \\
\quad \text{if } r_6 = 0 \text{ then } (\text{cond, L4, } r_4, r_5, r_6, r_{14}, m) \\
\quad \text{else } (\text{cond, L0, } r_4, r_5, r_6, r_{14}, m) \\
\text{else if } pc = L2 \text{ then} \\
\quad \text{let } r_6 = r_6 - 4 \text{ in} \\
\quad \text{if } r_6 = 0 \text{ then } (\text{cond, L4, } r_4, r_5, r_6, r_{14}, m) \\
\quad \text{else } (\text{cond, L0, } r_4, r_5, r_6, r_{14}, m) \\
\text{else } (\text{cond, pc, } r_4, r_5, r_6, r_{14}, m)
\end{align*}

test for value of PC
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:

Extracted function:

calls(\text{cond}, \text{pc}, r_4, r_5, r_6, r_{14}, m) =
\begin{align*}
&\text{if } \text{pc} = \text{L0} \text{ then} \\
&\text{let } \text{cond} = \text{cond} \land \text{valid_address} \ (r_5 + r_6) \ m \text{ in} \\
&\text{let } r_4 = m(r_5 + r_6) \text{ in} \\
&\text{let } \text{cond} = \text{cond} \land \text{wordAlignedAddress} \ r_4 \text{ in} \\
&\text{let } (\text{pc}, r_{14}) = (r_4, \text{L2}) \text{ in} \\
&\ (\text{cond}, \text{pc}, r_4, r_5, r_6, r_{14}, m) \\
&\text{else if } \text{pc} = \text{L2} \text{ then} \\
&\text{let } r_6 = r_6 - 4 \text{ in} \\
&\text{if } r_6 = 0 \text{ then } (\text{cond}, \text{L4}, r_4, r_5, r_6, r_{14}, m) \\
&\text{else } (\text{cond}, \text{L0}, r_4, r_5, r_6, r_{14}, m) \\
&\text{else } (\text{cond}, \text{pc}, r_4, r_5, r_6, r_{14}, m)
\end{align*}
The C code above can be compiled to ARM assembly:

```
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:                
```

The example below will illustrate what we mean.

```
Phase 1
(a[i]()) = (cond, pc, r4, r5, r6, r14, m)
    if pc = L0 then
    let cond = cond ∧ valid_address (r5 + r6) m in
    let r4 = m(r5 + r6) in
    let cond = cond ∧ word_aligned_address r4 in
    let (pc, r14) = (r4, L2) in
    (cond, pc, r4, r5, r6, r14, m)

Phase 2
(calls(c, pc, r4, r5, r6, r14, m) = (true, pc', r4', ...)) →
{ ARM state holds (r4, r5, r6, r14, m) and PC is pc }
E7954006 E12FFFF34 E2566004 1AFFFFFFFB
{ ARM state holds (r4', r5', r6', r14', m') and PC is pc' }
```

As mentioned above, this paper's contribution is to present
a new Hoare logic (Section III-A) that can be used for
extracting functions from ARM decompiled code. The new Hoare logic
does not require explicit mention of the program counter
word addresses and a return address is stored in
this case, a load is performed and a call is made to a code
extracted function has three parts; the first part describes the
function applies directly to the machine code via the certificate
extracted function, since any property proved of the extracted
function. Once the machine code has been decompiled, subsequent
theorems relate this property to the machine code.

The benefit of using decompilation in verification is that
often involving simplification through rewriting and calcu-
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:                
```

The original approach to decompilation was geared
towards automating proofs in a Hoare logic that was intended
for manual proofs — a complicated Hoare logic that was
arrays are stored in memory, it is easy to prove that
gained can seen in benchmarks listed in Figure 1.
Performance Numbers

Comparison between new and old approach.
Cost given in seconds (s) and HOL inference rules (i).

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

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).

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

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).

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

Cost of evaluating the ISA model is separate as this cost is independent of decompilation approach.
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 proving projects require ISA models; for example, for formalizing microprocessors, operating systems, compilers and machine code. As such, (often partial) ISA models exist for a number of architectures (e.g. x86, ARM and PowerPC) in a number of theorem provers (e.g. ACL2, PVS, HOL-Light, Isabelle/HOL, Coq and HOL4). These models differ in their presentation style, precise abstraction level (fidelity) and degrees of completeness. In part this reflects the nature of the projects for which the models have been originally developed, e.g. compiler verification and machine code verification. There are also differences based on the expressiveness and features of the theorem provers that are used. The ACL2 theorem prover has been used very successfully in this field for many years, where it has the advantage of providing very fast model evaluation. Recently, Warren Hunt has developed an ACL2-based specification of the Y86 processor, which implements a subset of the x86 architecture; see [3].

The main objective of the DSL is to make the task of modelling ISAs simpler, more reliable and less tedious. In particular, it should be possible for people who are not experts in HOL4 to readily read, develop and create ISA specifications for use in HOL4. Furthermore, it is also hoped that this work will help facilitate the dissemination of ISA models — enabling various concrete ISA models to be derived for different settings, tools and use cases.

Although various ISA DSLs currently exist, often these have been developed for writing compiler backends and binary code analysis tools, e.g. -RTL [9] and TSL [5]. The most closely related work is Lyrebird [1], which was developed as part of the seL4 project at NICTA. This tool supports fast simulation but it has not been successfully used in a theorem proving setting. Also of interest is the RockSalt project [6], which modelled x86 in Coq through the use of embedded DSLs. The aim of this work is to produce high-fidelity specifications that are inherently formal and yet prover/tool agnostic. The DSL and generated native prover specifications should be acceptable to both the engineering (computer architecture) and formal methods communities.
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

Decomposition:

- 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?