Machine-code verification for multiple architectures

— An application of decompilation into logic

Magnus O. Myreen, Konrad Slind, Michael J. C. Gordon

FMCAD 2008

Motivation

Formal verification of machine code:

machine code

code

Motivation

Formal verification of machine code:

machine code

code

correctness statement

$\{P\}\ code\ \{Q\}$

Motivation

Formal verification of machine code:

machine code

code

ARM/x86/PowerPC model

⋮

(7800/4500/2100 lines)

⋮

correctness statement

$\{P\}\ code\ \{Q\}$

Motivation

Formal verification of machine code:

ARM/x86/PowerPC model

machine code

code

⋮

(7800/4500/2100 lines)

⋮

correctness statement

$\{P\}\ code\ \{Q\}$

Contribution: a tool/method which

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

Motivation

Formal verification of machine code:

ARM/x86/PowerPC model

machine code

code

⋮

(7800/4500/2100 lines)

⋮

correctness statement

$\{P\}\ code\ \{Q\}$

Contribution: a tool/method which

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

Decompiler — extracts (with proof in HOL4) a function describing the effect of the code on the model.

Talk outline

1. what is decompilation into logic?
2. how to implement decompilation?

Basic idea

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

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

Basic idea

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
```

Basic idea

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
```

The decompiler produces a readable HOL4 function:

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

Decompilation, correct?

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

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

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

Decompilation, example

Precondition f_{pre} keeps track of side-conditions:

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

Decompilation, verification example

Decompiler automatically produced: f, f_{pre} and certificate.

Decompilation, verification example

Decompiler automatically produced: f, f_{pre} and certificate.

To verify functional correctness, formalise "linked-list in memory":

$$\begin{array}{rcl} \mathit{list}(\mathsf{nil}, a, m) & = & a = 0 \\ \mathit{list}(\mathsf{cons}\ x\ l, a, m) & = & \exists a'.\ m(a) = a' \land m(a{+}4) = x \land a \neq 0 \land \\ & & \quad \mathit{list}(l, a', m) \land \mathit{aligned}(a) \end{array}$$

Decompilation, verification example

Decompiler automatically produced: f, f_{pre} and certificate.

To verify functional correctness, formalise "linked-list in memory":

$$
\begin{array}{rcl}
\mathit{list}(\mathrm{nil}, a, m) & = & a = 0 \\
\mathit{list}(\mathrm{cons}\ x\ l, a, m) & = & \exists a'.\ m(a) = a' \wedge m(a{+}4) = x \wedge a \neq 0 \wedge \\
 & & \quad \mathit{list}(l, a', m) \wedge \mathit{aligned}(a)
\end{array}
$$

Manual part of verification proof (14 lines in HOL4):

$$
\begin{array}{rcl}
\forall x\ l\ a\ m.\ \mathit{list}(l, a, m) & \Rightarrow & f(x, a, m) = (\mathit{length}(l), 0, m) \\
\forall x\ l\ a\ m.\ \mathit{list}(l, a, m) & \Rightarrow & f_{pre}(x, a, m)
\end{array}
$$

Decompilation, verification example, cont.

Using the automatically proved certificate:

$$f_{pre}(r_0, r_1, m) \Rightarrow$$

$$\{\, (\mathsf{R0}, \mathsf{R1}, \mathsf{M}) \text{ is } (r_0, r_1, m) * \mathsf{PC}\ p * \mathsf{S} \,\}$$

$$p : \texttt{E3A00000 E3510000 12800001 15911000 1AFFFFFB}$$

$$\{\, (\mathsf{R0}, \mathsf{R1}, \mathsf{M}) \text{ is } f(r_0, r_1, m) * \mathsf{PC}\ (p + 20) * \mathsf{S} \,\}$$

Decompilation, verification example, cont.

Using the automatically proved certificate:

$$list(l, r_1, m) \Rightarrow$$

$$\{\, (\mathrm{R0}, \mathrm{R1}, \mathrm{M}) \text{ is } (r_0, r_1, m) * \mathrm{PC}\ p * \mathrm{S} \,\}$$

p : `E3A00000 E3510000 12800001 15911000 1AFFFFFB`

$$\{\, (\mathrm{R0}, \mathrm{R1}, \mathrm{M}) \text{ is } (length(l), 0, m) * \mathrm{PC}\ (p + 20) * \mathrm{S} \,\}$$

Decompilation, proof reuse

```
x86        0:  31C0              xor eax, eax
           2:  85F6          L1: test esi, esi
           4:  7405              jz L2
           6:  40                inc eax
           7:  8B36              mov esi, [esi]
           9:  EBF7              jmp L1
                             L2:

PowerPC    0:  38A00000          addi 5,0,0
           4:  2C140000      L1: cmpwi 20,0
           8:  40820010          bc 4,2,L2
          12:  7E80A02E          lwzx 20,0(20)
          16:  38A50001          addi 5,5,1
          20:  4BFFFFF0          b L1
                             L2:
```

Decompilation, proof reuse, cont.

Decompilation of x86 and PowerPC code:

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

$$
\begin{aligned}
f''(r_5, r_{20}, m) &= \text{let } r_5 = 0 \text{ in } g''(r_5, r_{20}, m) \\
g''(r_5, r_{20}, m) &= \text{if } r_{20} = 0 \text{ then } (r_5, r_{20}, m) \text{ else} \\
&\qquad \text{let } r_{20} = m(r_{20}) \text{ in} \\
&\qquad \text{let } r_5 = r_5{+}1 \text{ in} \\
&\qquad\quad g''(r_5, r_{20}, m)
\end{aligned}
$$

But in this case, easy to prove $f = f' = f''$ (3 lines in HOL4).

Decompilation, in a nut shell

Proof-producing decompilation:

- takes machine code, returns function and certificate
- keeps manual proofs independent of underlying model (possible proof reuse)

Talk outline

1. what is decompilation into logic?
2. how to implement decompilation?

Talk outline

1. what is decompilation into logic?
2. how to implement decompilation?
   - processor models
   - machine-code specifications: "$\{...\}$ *code* $\{...\}$"
   - tail-recursive functions

ISA models

Underlying ISA specifications:

- ARM – developed by Anthony Fox, verified against a register-transfer level model of an ARM processor;
- x86 – developed together with Susmit Sarkar, Peter Sewell, Scott Owens, etc, heavily tested against a real processor;
- PowerPC – a HOL4 translation of Xavier Leroy's PowerPC model, used in his proof of an optimising C compiler.

Large detailed models...

Machine code, x86

Even 'simple' instructions get complex definition.

Sequential op.sem. evaluated for instruction "40" (i.e. `inc eax`):

$$
\begin{array}{l}
\mathit{x86\_read\_reg}\ \mathrm{EAX}\ \mathit{state} = \mathit{eax}\ \wedge \\
\mathit{x86\_read\_eip}\ \mathit{state} = \mathit{eip}\ \wedge \\
\mathit{x86\_read\_mem}\ \mathit{eip}\ \mathit{state} = \mathrm{some}\ \mathtt{0x40} \Rightarrow \\
\quad \mathit{x86\_next}\ \mathit{state} = \\
\qquad \mathrm{some}\ (\mathit{x86\_write\_reg}\ \mathrm{EAX}\ (\mathit{eax} + 1) \\
\qquad\qquad (\mathit{x86\_write\_eip}\ (\mathit{eip} + 1) \\
\qquad\qquad (\mathit{x86\_write\_eflag}\ \mathrm{AF}\ \mathrm{none} \\
\qquad\qquad (\mathit{x86\_write\_eflag}\ \mathrm{SF}\ (\mathrm{some}\ (\mathit{sign\_of}\,(\mathit{eax} + 1))) \\
\qquad\qquad (\mathit{x86\_write\_eflag}\ \mathrm{ZF}\ (\mathrm{some}\ (\mathit{eax} + 1 = 0)) \\
\qquad\qquad (\mathit{x86\_write\_eflag}\ \mathrm{PF}\ (\mathrm{some}\ (\mathit{parity\_of}\,(\mathit{eax} + 1))) \\
\qquad\qquad (\mathit{x86\_write\_eflag}\ \mathrm{OF}\ \mathrm{none}\ \mathit{state})))))))
\end{array}
$$

Machine code, specifications

A machine-code specifications:

$$\{\,\mathsf{R\ EAX}\ a * \mathsf{EIP}\ p * \mathsf{S}\,\}$$
$$p : 40$$
$$\{\,\mathsf{R\ EAX}\ (a{+}1) * \mathsf{EIP}\ (p{+}1) * \mathsf{S}\,\}$$

where S existentially quantifies the status flags:

$$\mathsf{S} = \exists a\, s\, z\, p\, o.\ \mathsf{eflag\ AF}\ a * \mathsf{eflag\ SF}\ s * \mathsf{eflag\ ZF}\ z * \ldots.$$

Machine code, specifications

A machine-code specifications:

$$\forall P.\ \{\,\mathsf{R\ EAX}\ a * \mathsf{EIP}\ p * \mathsf{S} * P\,\}$$
$$p : 40$$
$$\{\,\mathsf{R\ EAX}\ (a{+}1) * \mathsf{EIP}\ (p{+}1) * \mathsf{S} * P\,\}$$

where S existentially quantifies the status flags:

$$\mathsf{S} = \exists a\, s\, z\, p\, o.\ \mathsf{eflag\ AF}\ a * \mathsf{eflag\ SF}\ s * \mathsf{eflag\ ZF}\ z * \ldots.$$

Machine code, specifications

A machine-code specifications:

$$\{\,\mathsf{R}\ \mathsf{EAX}\ a * \mathsf{EIP}\ p * \mathsf{S} * \mathsf{R}\ \mathsf{EBX}\ b\,\}$$
$$p : 40$$
$$\{\,\mathsf{R}\ \mathsf{EAX}\ (a{+}1) * \mathsf{EIP}\ (p{+}1) * \mathsf{S} * \mathsf{R}\ \mathsf{EBX}\ b\,\}$$

where S existentially quantifies the status flags:

$$\mathsf{S} = \exists a\, s\, z\, p\, o.\ \mathsf{eflag}\ \mathsf{AF}\ a * \mathsf{eflag}\ \mathsf{SF}\ s * \mathsf{eflag}\ \mathsf{ZF}\ z * \ldots.$$

Machine code, specifications

A machine-code specifications:

$$\{\,\mathsf{R\ EAX}\ a * \mathsf{EIP}\ p * \mathsf{S} * \mathsf{R\ EBX}\ b\,\}$$
$$p : \texttt{40}$$
$$\{\,\mathsf{R\ EAX}\ (a{+}1) * \mathsf{EIP}\ (p{+}1) * \mathsf{S} * \mathsf{R\ EBX}\ b\,\}$$

$$\{\,\mathsf{R\ EAX}\ a * \mathsf{EIP}\ p * \mathsf{S} * \mathsf{R\ EBX}\ b\,\}$$
$$p : \texttt{01D8}$$
$$\{\,\mathsf{R\ EAX}\ (a{+}b) * \mathsf{EIP}\ (p{+}2) * \mathsf{S} * \mathsf{R\ EBX}\ b\,\}$$

where S existentially quantifies the status flags:

$$\mathsf{S} = \exists a\, s\, z\, p\, o.\ \mathsf{eflag\ AF}\ a * \mathsf{eflag\ SF}\ s * \mathsf{eflag\ ZF}\ z * \ldots.$$

Machine code, specifications

A machine-code specifications:

$$\{\,\mathsf{R\ EAX}\ a * \mathsf{EIP}\ p * \mathsf{S} * \mathsf{R\ EBX}\ b\,\}$$
$$p : \texttt{40}$$
$$\{\,\mathsf{R\ EAX}\ (a{+}1) * \mathsf{EIP}\ (p{+}1) * \mathsf{S} * \mathsf{R\ EBX}\ b\,\}$$

$$\{\,\mathsf{R\ EAX}\ (a{+}1) * \mathsf{EIP}\ (p{+}1) * \mathsf{S} * \mathsf{R\ EBX}\ b\,\}$$
$$p{+}1 : \texttt{01D8}$$
$$\{\,\mathsf{R\ EAX}\ (a{+}1{+}b) * \mathsf{EIP}\ (p{+}3) * \mathsf{S} * \mathsf{R\ EBX}\ b\,\}$$

where S existentially quantifies the status flags:

$$\mathsf{S} = \exists a\, s\, z\, p\, o.\ \mathsf{eflag\ AF}\ a * \mathsf{eflag\ SF}\ s * \mathsf{eflag\ ZF}\ z * \ldots.$$

Machine code, specifications

A machine-code specifications:

$$\{\,\mathsf{R\ EAX}\ a * \mathsf{EIP}\ p * \mathsf{S} * \mathsf{R\ EBX}\ b\,\}$$
$$p : \texttt{4001D8}$$
$$\{\,\mathsf{R\ EAX}\ (a{+}1{+}b) * \mathsf{EIP}\ (p{+}3) * \mathsf{S} * \mathsf{R\ EBX}\ b\,\}$$

where S existentially quantifies the status flags:

$$\mathsf{S} = \exists a\, s\, z\, p\, o.\ \mathsf{eflag\ AF}\ a * \mathsf{eflag\ SF}\ s * \mathsf{eflag\ ZF}\ z * \ldots.$$

Tail-recursive functions

How to implement the proof-producing translation?

Key ideas:

1. define functions as instances of

$$tailrec(x) = \text{if } G(x) \text{ then } tailrec(F(x)) \text{ else } D(x)$$

Tail-recursive functions

How to implement the proof-producing translation?

Key ideas:

1. define functions as instances of

$$tailrec(x) = \text{if } G(x) \text{ then } tailrec(F(x)) \text{ else } D(x)$$

2. specify termination for value x as

$$pre(x) = \exists n.\ \neg(G(F^n(x)))$$

Tail-recursive functions

How to implement the proof-producing translation?

Key ideas:

1. define functions as instances of

$$tailrec(x) = \text{if } G(x) \text{ then } tailrec(F(x)) \text{ else } D(x)$$

2. specify termination for value x as

$$pre(x) = \exists n.\ \neg(G(F^n(x)))$$

3. but give the user

$$pre(x) = \text{ if } G(x) \text{ then } pre(F(x)) \text{ else true}$$

Tail-recursive functions

How to implement the proof-producing translation?

Key ideas:

1. define functions as instances of

$$tailrec(x) = \text{if } G(x) \text{ then } tailrec(F(x)) \text{ else } D(x)$$

2. specify termination for value x as

$$pre(x) = \exists n.\ \neg(G(F^n(x))) \wedge (\forall x.\ \ldots \Rightarrow H(x))$$

3. but give the user

$$pre(x) = (\text{if } G(x) \text{ then } pre(F(x)) \text{ else true}) \wedge H(x)$$

4. actually also insert side-condition $H(x)$

Tail-recursive functions, cont.

5. use the following loop rule, one loop at a time:

$$\forall P\ Q.$$

$$(\forall x.\ H(x) \land \quad G(x) \Rightarrow \{P(x)\}\ \mathit{code}\ \{P(F(x))\}) \Rightarrow$$

$$(\forall x.\ H(x) \land \neg G(x) \Rightarrow \{P(x)\}\ \mathit{code}\ \{Q(D(x))\}) \Rightarrow$$

$$(\forall x.\ \mathit{pre}(x) \Rightarrow \{P(x)\}\ \mathit{code}\ \{Q(\mathit{tailrec}(x))\})$$

Decompilation algorithm

Algorithm:

1. derive specifications for individual instructions;
2. find control flow;
3. compose specifications;
4. apply loop rule;
5. exit or go to step 3.

Details in paper...

Decompilation, restrictions

Restrictions:

1. heuristics used for control-flow discovery, cannot handle code-pointers (except subroutine call/return).
2. underlying ISA model must be deterministic (at least for the code which is decompiled).

Robust: heuristics only used for control-flow discovery.

Applications

Verification case studies done:

- copying garbage collectors, LISP primitives (car, cdr, cons, ...)

Used in proof-producing compiler:

- compiles HOL4 functions to ARM, x86, PowerPC code;
- compiler used to produce verified LISP interpreters.

Other applications?

- link to your favorite tool? (no need to trust C compiler...)

Summary

1. decompilation: given code, produces function and certificate;

Summary

1. decompilation: given code, produces function and certificate;
2. allows users to only deal with tail-recursive functions, instead of machine code;

Summary

1. decompilation: given code, produces function and certificate;
2. allows users to only deal with tail-recursive functions, instead of machine code;
3. separates manual proofs from definitions of processor state (thus scope for some proof-reuse);

Summary

1. decompilation: given code, produces function and certificate;
2. allows users to only deal with tail-recursive functions, instead of machine code;
3. separates manual proofs from definitions of processor state (thus scope for some proof-reuse);
4. easy to implement (only approx. 2000 lines of ML) .

Summary

1. decompilation: given code, produces function and certificate;
2. allows users to only deal with tail-recursive functions, instead of machine code;
3. separates manual proofs from definitions of processor state (thus scope for some proof-reuse);
4. easy to implement (only approx. 2000 lines of ML) .

Questions?