Trustworthy decompilation: Extracting models of machine code inside an ITP

Magnus O. Myreen
University of Cambridge

TEITP 2010

The GCD program in ARM machine code:

```
E1510002 B0422001 C0411002 01AFFFFB
```

Problems with machine code

Formal verification of machine code:

machine code

code

Problems with machine code

Formal verification of machine code:

machine code

code

correctness statement

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

Problems with machine code

Formal verification of machine code:

ARM/x86/PowerPC model

⋮

(12100/4500/2100 lines)

⋮

machine code

code

correctness statement

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

Problems with machine code

Formal verification of machine code:

machine code

code

ARM/x86/PowerPC model

⋮

(12100/4500/2100 lines)

⋮

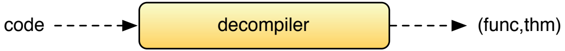
correctness statement

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

Contribution: tools/methods which

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

Proposed solution

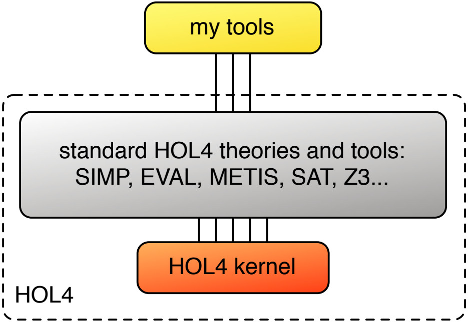


Decompiler:

- input: machine code
- output: function computed by code & certificate theorem

Trusted extension

My tools $=$ ML programs which steer HOL4 to a proof



Every proof passes the LCF-style logical kernel of HOL4.

This talk:

- explaining decompilation $\|$ demo
- pros/cons of HOL4

Models of machine languages

Formal verification of machine code:

ARM/x86/PowerPC model

machine code

code

⋮

(12100/4500/2100 lines)

⋮

correctness statement

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

Models of machine languages

Machine models borrowed from work by others:

ARM model, by Fox [ITP'10]

- covers practically all ARM instructions, for old and new ARMs
- extensively tested against real hardware

x86 model, by Sarkar et al. [POPL'09]

- covers all addressing modes in 32-bit mode x86
- includes approximately 30 instructions

PowerPC model, originally from Leroy [POPL'06]

- manual translation (Coq $\rightarrow$ HOL4) of Leroy's PowerPC model
- instruction decoder added

Hoare triple

Each model can be evaluated, e.g. ARM instruction `add r0,r0,r0` is described by theorem:

```
|- (ARM_READ_MEM ((31 >< 2) (ARM_READ_REG 15w state)) state =
    0xE0800000w) ∧ ¬state.undefined ⇒
   (NEXT_ARM_MMU cp state =
     ARM_WRITE_REG 15w (ARM_READ_REG 15w state + 4w)
     (ARM_WRITE_REG 0w
      (ARM_READ_REG 0w state + ARM_READ_REG 0w state) state))
```

Hoare triple

Each model can be evaluated, e.g. ARM instruction `add r0,r0,r0` is described by theorem:

```
|- (ARM_READ_MEM ((31 >< 2) (ARM_READ_REG 15w state)) state =
    0xE0800000w) ∧ ¬state.undefined ⇒
   (NEXT_ARM_MMU cp state =
     ARM_WRITE_REG 15w (ARM_READ_REG 15w state + 4w)
     (ARM_WRITE_REG 0w
      (ARM_READ_REG 0w state + ARM_READ_REG 0w state) state))
```

As a total-correctness machine-code Hoare triple:

```
|- SPEC ARM_MODEL
    (aR 0w x * aPC p)
    {(p,0xE0800000w)}
    (aR 0w (x+x) * aPC (p+4w))
```

Hoare triple

Each model can be evaluated, e.g. ARM instruction `add r0,r0,r0` is described by theorem:

```
|- (ARM_READ_MEM ((31 >< 2) (ARM_READ_REG 15w state)) state =
    0xE0800000w) ∧ ¬state.undefined ⇒
   (NEXT_ARM_MMU cp state =
     ARM_WRITE_REG 15w (ARM_READ_REG 15w state + 4w)
     (ARM_WRITE_REG 0w
      (ARM_READ_REG 0w state + ARM_READ_REG 0w state) state))
```

As a total-correctness machine-code Hoare triple:

```
|- SPEC ARM_MODEL
    (aR 0w x * aPC p)
    {(p,0xE0800000w)}
    (aR 0w (x+x) * aPC (p+4w))
```

Informal syntax for this talk:

$\{\,\mathrm{R0}\ x * \mathrm{PC}\ p\,\}$
p : E0800000
$\{\,\mathrm{R0}\ (x{+}x) * \mathrm{PC}\ (p{+}4)\,\}$

Demo.

Decompilation

Decompiler automates Hoare triple reasoning.

Example: Given some ARM machine code,

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

Decompilation

Decompiler automates Hoare triple reasoning.

Example: Given some 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

Decompiler automates Hoare triple reasoning.

Example: Given some 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 automatically extracts a readable function:

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

Decompilation, correct?

Decompiler automatically proves a certificate theorem:

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

which informally reads:

for any initially value (r_0, r_1, m) in reg 0, reg 1 and memory,
the code terminates with $f(r_0, r_1, m)$ in reg 0, reg 1 and memory.

Decompilation, verification example

To verify code: prove properties of function f,

$$\forall x\ l\ a\ m.\ \ list(l, a, m) \ \Rightarrow\ f(x, a, m) = (length(l), 0, m)$$
$$\forall x\ l\ a\ m.\ \ list(l, a, m) \ \Rightarrow\ f_{pre}(x, a, m)$$

since properties of f carry over to machine code via the certificate.

Decompilation, verification example

To verify code: prove properties of function f,

$$\forall x\, l\, a\, m.\;\; list(l,a,m) \;\Rightarrow\; f(x,a,m) = (length(l), 0, m)$$
$$\forall x\, l\, a\, m.\;\; list(l,a,m) \;\Rightarrow\; f_{pre}(x,a,m)$$

since properties of f carry over to machine code via the certificate.

Proof reuse: Given similar x86 and PowerPC code:

```
31C085F67405408B36EBF7
```

```
38A000002C140000408200107E80A02E38A500014BFFFFF0
```

which decompiles into f' and f'', respectively. Manual proofs above can be reused if $f = f' = f''$.

Demo.

Decompilation, algorithm

Algorithm:

1. derive a Hoare-triple for each instruction
2. find all paths through code
3. for each loop/sub-component:
   a. compose Hoare triples along each path
   b. merge resulting Hoare triples
   c. apply a loop rule, if necessary

The loop rule introduces a tail-recursive function, an instance of

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

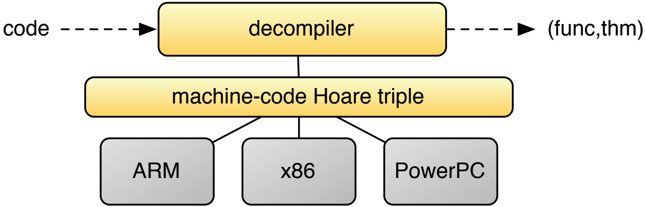
Decompiler, implementation

Implementation:

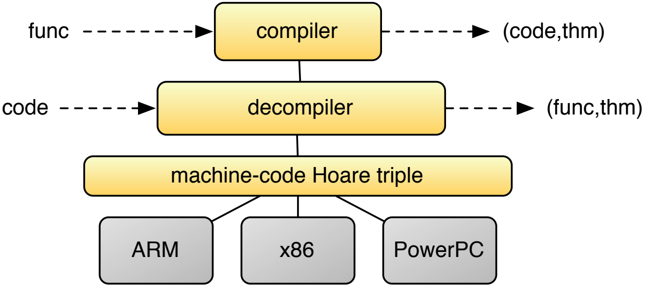
- ML program which fully-automatically performs forward proof,
- no heuristics and no dangling proof obligations,
- ‘smart’ tactics, e.g. `SIMP`, avoided to be robust.

Details in Myreen et al. [FMCAD’08].

Applications



Applications



Compiler

Synthesis often more practical. Given function f,

$$f(r_1) \;=\; \text{if } r_1 < 10 \text{ then } r_1 \text{ else let } r_1 = r_1 - 10 \text{ in } f(r_1)$$

our *compiler* generates ARM machine code:

```
E351000A     L:  cmp r1,#10
2241100A         subcs r1,r1,#10
2AFFFFFC         bcs L
```

Compiler

Synthesis often more practical. Given function f,

$$f(r_1) \;=\; \text{if } r_1 < 10 \text{ then } r_1 \text{ else let } r_1 = r_1 - 10 \text{ in } f(r_1)$$

our *compiler* generates ARM machine code:

```
E351000A     L:  cmp r1,#10
2241100A         subcs r1,r1,#10
2AFFFFFC         bcs L
```

and automatically proves a certificate HOL theorem:

$$\vdash \;\{\, \text{R1 } r_1 * \text{PC } p * \text{s} \,\}$$
$$p : \texttt{E351000A 2241100A 2AFFFFFC}$$
$$\{\, \text{R1 } f(r_1) * \text{PC } (p{+}12) * \text{s} \,\}$$

Compilation example, cont.

One can prove properties of f since it lives inside HOL:

$$\vdash \ \forall x.\ f(x) = x \bmod 10$$

Compilation example, cont.

One can prove properties of f since it lives inside HOL:

$$\vdash \ \forall x.\ f(x) = x \text{ mod } 10$$

Properties proved of f translate to properties of the machine code:

$$\vdash \ \{\text{R1}\ r_1 * \text{PC}\ p * \text{s}\}$$
$$p : \texttt{E351000A 2241100A 2AFFFFFC}$$
$$\{\text{R1}\ (r_1 \text{ mod } 10) * \text{PC}\ (p{+}12) * \text{s}\}$$

Compilation example, cont.

One can prove properties of f since it lives inside HOL:

$$\vdash \ \forall x.\ f(x) = x \text{ mod } 10$$

Properties proved of f translate to properties of the machine code:

$$\vdash \ \{\text{R1}\ r_1 * \text{PC}\ p * \text{s}\}$$
$$p : \texttt{E351000A 2241100A 2AFFFFFC}$$
$$\{\text{R1}\ (r_1 \text{ mod } 10) * \text{PC}\ (p{+}12) * \text{s}\}$$

Additional feature: the compiler can use the above theorem to extend its input language with: let $r_1 = r_1$ mod 10 in _

Additional feature: user-defined extensions

Using our theorem about mod, the compiler accepts:

$$
\begin{aligned}
g(r_1, r_2, r_3) \;=\; & \text{let } r_1 = r_1 + r_2 \text{ in} \\
& \text{let } r_1 = r_1 + r_3 \text{ in} \\
& \text{let } r_1 = r_1 \text{ mod } 10 \text{ in} \\
& \quad (r_1, r_2, r_3)
\end{aligned}
$$

Previously proved theorems can be used as building blocks for subsequent compilations.

Implementation

To compile function f:

1. generate, without proof, code from input f;
2. decompile, with proof, a function f' from generated code;
3. prove $f = f'$.

Implementation

To compile function f:

1. generate, without proof, code from input f;
2. decompile, with proof, a function f' from generated code;
3. prove $f = f'$.

Features:

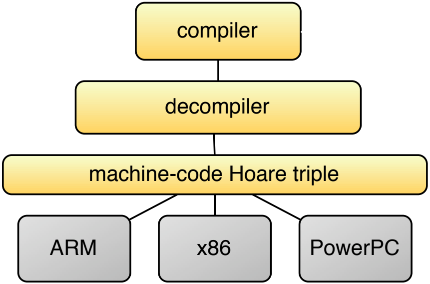
- code generation completely separate from proof
- supports many light-weight optimisations without any additional proof burden: instruction reordering, conditional execution, dead-code elimination, duplicate-tail elimination, ...
- allows for significant user-defined extensions

Details in Myreen et al. [CC'09]

Demo.

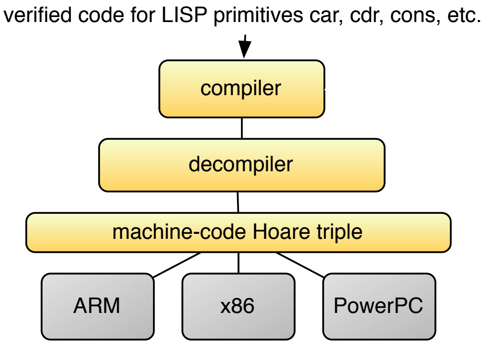
LISP case study

Verified LISP implementations via compilation.



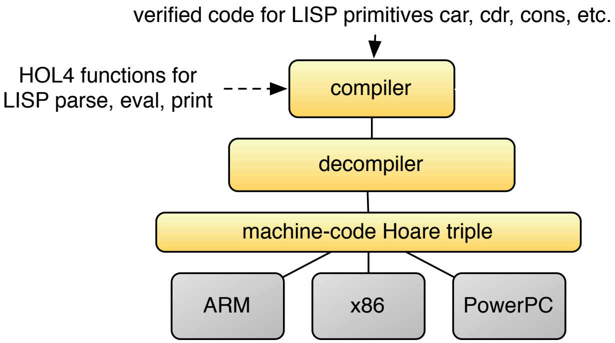
LISP case study

Verified LISP implementations via compilation.



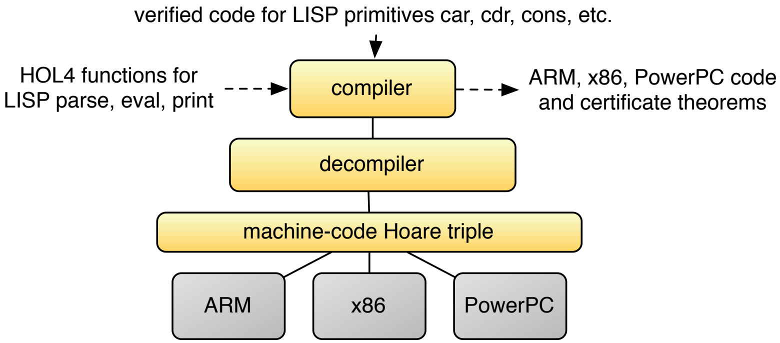
LISP case study

Verified LISP implementations via compilation.



LISP case study

Verified LISP implementations via compilation.



Demo.

Restrictions of decompilation

(De)compilation applicable only to programs where:

1. jumps are to fixed offsets or procedure returns,
2. code and data are kept separate, and
3. its semantics is deterministic.

Restrictions of decompilation

(De)compilation applicable only to programs where:

1. jumps are to fixed offsets or procedure returns,
2. code and data are kept separate, and
3. its semantics is deterministic.

Decompiler extensively used in proof of JIT compiler with:

1. code pointers,
2. self-modifying code, and
3. a non-deterministic ISA model.

Decompiler applied to ‘well-behaved’ sub-components.

This talk:

- explaining decompilation $\parallel$ demo
- pros/cons of HOL4

Pros/cons of HOL4

Pros:

- HOL4 is easily programmable
- lack of user interface — user at ML level
- easy to mix backwards/forwards reasoning
- conceptually simple

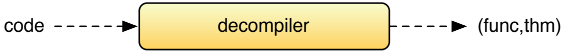
Cons:

- very space consuming, e.g. the term "$[1, 20, 3000]$" is represented by > 30 cons cells
- not automatic enough, not modular enough, ...

Talk summary

Decompilation:

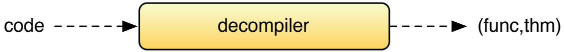
- automates Hoare triple reasoning,
- extracts function computed by code,
- useful for verification and code synthesis.



Talk summary

Decompilation:

- automates Hoare triple reasoning,
- extracts function computed by code,
- useful for verification and code synthesis.



Questions?

(I can demo the verified Lisp or JIT on request.)