Proof-producing decompilation and compilation

Magnus Myreen

May 2008

Introduction

This talk concerns verification of functional correctness of machine code for commercial processors (ARM, PowerPC, x86 ...).

Outline of talk:

- motivation for decompilation into logic
- implementing decompilation
- compilation

Verification

Current approaches for machine-code verification:

- direct reasoning about *next*-state function.
- annotating code with assertions:

```
    xor eax, eax

L1: test esi, esi
    jz L2

    inc eax
    mov esi, [esi]
    jmp L1
L2:
```

Verification

Current approaches for machine-code verification:

- direct reasoning about *next*-state function.
- annotating code with assertions:

```
     {...}
     xor eax, eax
     {...}
L1:  test esi, esi
     jz L2
     {...}
     inc eax
     mov esi, [esi]
     jmp L1
L2:  {...}
```

Verification

Current approaches for machine-code verification:

- direct reasoning about *next*-state function.
- annotating code with assertions:

```
    {...}
    xor eax, eax
    {...}
L1: test esi, esi
    jz L2
    {...}
    inc eax
    mov esi, [esi]
    jmp L1
L2: {...}
```

```
   mov r0, #0

L: cmp r1, #0
   ldrne r1, [r1]
   addne r0, r0, #1
   bne L
```

Verification

Current approaches for machine-code verification:

- direct reasoning about *next*-state function.
- annotating code with assertions:

```
     {...}
     xor eax, eax
     {...}
L1:  test esi, esi
     jz L2
     {...}
     inc eax
     mov esi, [esi]
     jmp L1
L2:  {...}
```

```
    {???}
    mov r0, #0
    {???}
L:  cmp r1, #0
    ldrne r1, [r1]
    addne r0, r0, #1
    bne L
    {???}
```

Proof reuse?

Our approach

Decompilation produces the following tail-recursive functions describing the effect of the code, f for x86 and f' for ARM:

$$
\begin{array}{l}
f(eax, esi, m) \ = \\
\quad \textbf{let } eax = eax \otimes eax \textbf{ in} \\
\qquad g(eax, esi, m)
\end{array}
$$

$$
\begin{array}{l}
g(eax, esi, m) \ = \\
\quad \textbf{if } esi \,\&\, esi = 0 \\
\quad \textbf{then } (eax, esi, m) \textbf{ else} \\
\qquad\quad \textbf{let } eax = eax{+}1 \textbf{ in} \\
\qquad\quad \textbf{let } esi = m(esi) \textbf{ in} \\
\qquad\qquad g(eax, esi, m)
\end{array}
$$

Our approach

Decompilation produces the following tail-recursive functions describing the effect of the code, f for x86 and f' for ARM:

$$
\begin{array}{l}
f(eax, esi, m) \;=\\
\quad \mathbf{let}\ eax = eax \otimes eax\ \mathbf{in}\\
\qquad g(eax, esi, m)\\[1ex]
g(eax, esi, m) \;=\\
\quad \mathbf{if}\ esi \,\&\, esi = 0\\
\quad \mathbf{then}\ (eax, esi, m)\ \mathbf{else}\\
\qquad\quad \mathbf{let}\ eax = eax{+}1\ \mathbf{in}\\
\qquad\quad \mathbf{let}\ esi = m(esi)\ \mathbf{in}\\
\qquad\qquad g(eax, esi, m)
\end{array}
\qquad\Bigg|\qquad
\begin{array}{l}
f'(r_0, r_1, m) \;=\\
\quad \mathbf{let}\ r_0 = 0\ \mathbf{in}\\
\qquad g'(r_0, r_1, m)\\[1ex]
g'(r_0, r_1, m) \;=\\
\quad \mathbf{if}\ r_1 = 0\\
\quad \mathbf{then}\ (r_0, r_1, m)\ \mathbf{else}\\
\qquad\quad \mathbf{let}\ r_1 = m(r_1)\ \mathbf{in}\\
\qquad\quad \mathbf{let}\ r_0 = r_0{+}1\ \mathbf{in}\\
\qquad\qquad g'(r_0, r_1, m)
\end{array}
$$

Our approach

Decompilation produces the following tail-recursive functions describing the effect of the code, f for x86 and f' for ARM:

$$
\begin{array}{l}
f(eax, esi, m) \;=\\
\quad \textbf{let}\ eax = eax \otimes eax\ \textbf{in}\\
\quad\quad g(eax, esi, m)\\[1ex]
g(eax, esi, m) \;=\\
\quad \textbf{if}\ esi \,\&\, esi = 0\\
\quad \textbf{then}\ (eax, esi, m)\ \textbf{else}\\
\quad\quad\quad \textbf{let}\ eax = eax{+}1\ \textbf{in}\\
\quad\quad\quad \textbf{let}\ esi = m(esi)\ \textbf{in}\\
\quad\quad\quad\quad g(eax, esi, m)
\end{array}
$$

$$
\begin{array}{l}
f'(r_0, r_1, m) \;=\\
\quad \textbf{let}\ r_0 = 0\ \textbf{in}\\
\quad\quad g'(r_0, r_1, m)\\[1ex]
g'(r_0, r_1, m) \;=\\
\quad \textbf{if}\ r_1 = 0\\
\quad \textbf{then}\ (r_0, r_1, m)\ \textbf{else}\\
\quad\quad\quad \textbf{let}\ r_1 = m(r_1)\ \textbf{in}\\
\quad\quad\quad \textbf{let}\ r_0 = r_0{+}1\ \textbf{in}\\
\quad\quad\quad\quad g'(r_0, r_1, m)
\end{array}
$$

Advantages:
1. no need for knowledge of the next-state function;
2. suitable for proofs in HOL, and
3. proof reuse, $f = f'$ using $w \,\&\, w = w$ and $w \otimes w = 0$.

Produced theorem

How does f relate to the x86 code?

Answer: For each run, the decompiler automatically:

1. generates a function f, and
2. proves a theorem relating the function to the code:

Produced theorem

How does f relate to the x86 code?

Answer: For each run, the decompiler automatically:

1. generates a function f, and
2. proves a theorem relating the function to the code:

$$\{\, (\mathbf{eax}, \mathbf{esi}, \mathbf{m}) \ \mathbf{is}\ (eax, esi, m) * \mathbf{eip}\ p * f_{pre}(eax, esi, m) \,\}$$
$$p : \texttt{31C085F67405408B36EBF7}$$
$$\{\, (\mathbf{eax}, \mathbf{esi}, \mathbf{m}) \ \mathbf{is}\ f(eax, esi, m) * \mathbf{eip}\ (p{+}11) \,\}$$

Here **eax**, **esi**, **m** and **eip** (program counter) assert values of resources and '(x, y, z) **is** (a, b, c)' abbreviates $(x\ a) * (y\ b) * (z\ c)$.

Verification

The decompiler automates machine specific proofs and leaves the user (verifier) to prove properties of the generated function f.

Suppose we have proved

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

for an appropriate definition of *list*.

Verification

The decompiler automates machine specific proofs and leaves the user (verifier) to prove properties of the generated function f.

Suppose we have proved

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

for an appropriate definition of *list*.

Then we have:

$$\{\, (\mathbf{eax}, \mathbf{esi}, \mathbf{m})\ \mathbf{is}\ (eax, esi, m) * \mathbf{eip}\ p * f_{pre}(eax, esi, m)\,\}$$
$$p : \texttt{31C085F67405408B36EBF7}$$
$$\{\, (\mathbf{eax}, \mathbf{esi}, \mathbf{m})\ \mathbf{is}\ f(eax, esi, m) * \mathbf{eip}\ (p{+}11)\,\}$$

Verification

The decompiler automates machine specific proofs and leaves the user (verifier) to prove properties of the generated function f.

Suppose we have proved

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

for an appropriate definition of *list*.

Then we have:

$$list(xs, esi, m) \Rightarrow$$
$$\{\,(\mathbf{eax}, \mathbf{esi}, \mathbf{m})\ \mathbf{is}\ (eax, esi, m) * \mathbf{eip}\ p * f_{pre}(eax, esi, m)\,\}$$
$$p : \texttt{31C085F67405408B36EBF7}$$
$$\{\,(\mathbf{eax}, \mathbf{esi}, \mathbf{m})\ \mathbf{is}\ f(eax, esi, m) * \mathbf{eip}\ (p{+}11)\,\}$$

Verification

The decompiler automates machine specific proofs and leaves the user (verifier) to prove properties of the generated function f.

Suppose we have proved

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

for an appropriate definition of *list*.

Then we have:

$$list(xs, esi, m) \Rightarrow$$
$$\{\, (\mathbf{eax}, \mathbf{esi}, \mathbf{m})\ \mathbf{is}\ (eax, esi, m) * \mathbf{eip}\ p * f_{pre}(eax, esi, m) \,\}$$
$$p : \texttt{31C085F67405408B36EBF7}$$
$$\{\, (\mathbf{eax}, \mathbf{esi}, \mathbf{m})\ \mathbf{is}\ (length(xs), 0, m) * \mathbf{eip}\ (p{+}11) \,\}$$

Verification

The decompiler automates machine specific proofs and leaves the user (verifier) to prove properties of the generated function f.

Suppose we have proved

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

for an appropriate definition of *list*.

Then we have:

$$list(xs, esi, m) \Rightarrow$$
$$\{\, (\mathbf{eax}, \mathbf{esi}, \mathbf{m})\ \mathbf{is}\ (eax, esi, m) * \mathbf{eip}\ p \,\}$$
$$p : \texttt{31C085F67405408B36EBF7}$$
$$\{\, (\mathbf{eax}, \mathbf{esi}, \mathbf{m})\ \mathbf{is}\ (length(xs), 0, m) * \mathbf{eip}\ (p{+}11) \,\}$$

Verification

The decompiler automates machine specific proofs and leaves the user (verifier) to prove properties of the generated function f.

Suppose we have proved

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

for an appropriate definition of *list*.

Then we have:

$$\{\, (\mathbf{eax}, \mathbf{esi}, \mathbf{m})\ \mathbf{is}\ (eax, esi, m) * \mathbf{eip}\ p * \ list(xs, esi, m)\ \,\}$$
$$p : \mathtt{31C085F67405408B36EBF7}$$
$$\{\, (\mathbf{eax}, \mathbf{esi}, \mathbf{m})\ \mathbf{is}\ (length(xs), 0, m) * \mathbf{eip}\ (p{+}11)\,\}$$

Outline of talk:

- motivation for decompilation into logic
- **implementing decompilation**
- compilation

Algorithm

Given some code, the decompiler:

1. derives a specification for each instruction;

Algorithm

Given some code, the decompiler:

1. derives a specification for each instruction;
2. discovers the control flow;

Algorithm

Given some code, the decompiler:

1. derives a specification for each instruction;
2. discovers the control flow;
3. for each code segment:
   a) derives a specification for one pass through the code;
   b) generates a function describing effect of code;
   c) for loops, instantiates special loop rule.

Algorithm

Given some code, the decompiler:

1. derives a specification for each instruction;
2. discovers the control flow;
3. for each code segment:
    a) derives a specification for one pass through the code;
    b) generates a function describing effect of code;
    c) for loops, instantiates special loop rule.
4. composes the top-level specifications and repeats step 3 until all of the code is described by one specification.

Proving loops

Approach: assume existence of termination proof, use induction from termination proof to prove loop.

Proving loops

Approach: assume existence of termination proof, use induction from termination proof to prove loop.

The decompiler models loops as tail-recursive functions k, with appropriate instantiations of F, G and D, where:

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

Proving loops

Approach: assume existence of termination proof, use induction from termination proof to prove loop.

The decompiler models loops as tail-recursive functions k, with appropriate instantiations of F, G and D, where:

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

We define $pre(x)$ to state that there exists and invariant which guarantees that k terminates when applied to x. Here $\prec$ is some well-founded relation.

$$\begin{array}{l} pre(x) = \\ \quad \exists inv.\ inv(x) \wedge \\ \qquad\qquad \exists \prec.\ \forall y.\ inv(y) \wedge G(y) \Rightarrow inv(F(y)) \wedge F(y) \prec y \end{array}$$

Proving loops (continued)

The loop rule, used by the decompiler, for function

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

is the following: for any resource assertions **res** and **res'**,

$$\begin{array}{l}
(\forall x. \quad G(x) \Rightarrow \{\mathbf{res}\ x\}\, c\, \{\mathbf{res}\ \ F(x)\}) \wedge \\
(\forall x. \neg G(x) \Rightarrow \{\mathbf{res}\ x\}\, c\, \{\mathbf{res'}\ D(x)\}) \\
\Rightarrow (\forall x. \{\mathbf{res}\ x * pre(x)\}\, c\, \{\mathbf{res'}\ k(x)\})
\end{array}$$

Proving loops (continued)

The loop rule, used by the decompiler, for function

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

is the following: for any resource assertions **res** and **res'**,

$$\begin{array}{l}(\forall x.\quad G(x) \Rightarrow \{\textbf{res}\ x\}\, c\, \{\textbf{res}\ \ F(x)\}) \wedge \\ (\forall x. \neg G(x) \Rightarrow \{\textbf{res}\ x\}\, c\, \{\textbf{res'}\ D(x)\}) \\ \Rightarrow (\forall x. \{\textbf{res}\ x * pre(x)\}\, c\, \{\textbf{res'}\ k(x)\})\end{array}$$

In our x86 example the loop uses assertions:

$$\begin{array}{rcl}\textbf{res}\ x &=& (\textbf{eax}, \textbf{esi}, \textbf{m})\ \textbf{is}\ x * \textbf{eip}\ p \\ \textbf{res'}\ x &=& (\textbf{eax}, \textbf{esi}, \textbf{m})\ \textbf{is}\ x * \textbf{eip}\ (p{+}9)\end{array}$$

Proving loops (continued)

The loop rule is derived from the following induction, provable from the definition of *pre* and well-founded relation:

$$\begin{aligned}\forall\varphi.\quad & (\forall x.\ G(x) \wedge \varphi(F(x)) \Rightarrow \varphi(x)) \wedge \\ & (\forall x.\ \neg G(x) \Rightarrow \varphi(x)) \\ & \Rightarrow (\forall x.\ \mathit{pre}(x) \Rightarrow \varphi(x))\end{aligned}$$

The proof of the loop rule uses the following composition:

$$\begin{aligned} & \{\textbf{res}\ x\}\, c\, \{\textbf{res}\ F(x)\} \wedge \{\textbf{res}\ F(x)\}\, c\, \{\textbf{res'}\ k(x)\} \\ \Rightarrow\ & \{\textbf{res}\ x\}\, c \cup c\, \{\textbf{res'}\ k(x)\} \\ \Rightarrow\ & \{\textbf{res}\ x\}\, c\, \{\textbf{res'}\ k(x)\}\end{aligned}$$

Decompilation

For most part proof-producing decompilation is just:

1. deriving specifications for individual instructions;
2. composing them; and
3. instantiating a loop rule.

Failure prone heuristic are largely avoided.

This method does not support advanced control-flow, *e.g.* computed jumps and code pointers.

However, it does support non-nested loops, procedure calls and, in an awkward way, procedural recursion.

Outline of talk:

- motivation for decompilation into logic
- implementing decompilation
- **compilation**

Proof-producing compilation

For a simple compiler, given a function f:

- generate code;
- run decompiler to get f';
- automatically prove $f = f'$.

Works well, even for functions f which are hundreds of lines long.

Proof-producing compilation

For a simple compiler, given a function f:

- generate code;
- run decompiler to get f';
- automatically prove $f = f'$.

Works well, even for functions f which are hundreds of lines long.

Each expression in f must implementable in the target language, *e.g.* "let $eax = eax + 1$ in" and "if $eax < 400$ then ... else ..."

However, we can do better...

Proof-producing compilation (continued)

Suppose we have a specification for allocation on a garbage collected heap h, which allows allocation of a new element if the size of the current heap h does not exceed the limit l.

$$\{\textbf{heap}\ (v_1, v_2, v_3, v_4, h, l) * \textbf{eip}\ p * \mathit{size}(h) < l\}$$

$$\ldots\mathit{code}\ldots$$

$$\{\textbf{heap}\ (\mathit{fresh}(h), v_2, v_3, v_4, h[\mathit{fresh}(h) \mapsto (v_1, v_2)], l) * \textbf{eip}\ (p{+}416)\}$$

Proof-producing compilation (continued)

Suppose we have a specification for allocation on a garbage collected heap h, which allows allocation of a new element if the size of the current heap h does not exceed the limit l.

$$\{\textbf{heap}\ (v_1, v_2, v_3, v_4, h, l) * \textbf{eip}\ p * \mathit{size}(h) < l\}$$
$$\ldots \mathit{code} \ldots$$
$$\{\textbf{heap}\ (\mathit{fresh}(h), v_2, v_3, v_4, h[\mathit{fresh}(h) \mapsto (v_1, v_2)], l) * \textbf{eip}\ (p{+}416)\}$$

Such specifications can be fed into the automation so that the compiler can handle:

$$\text{“let } (v_1, h) = (\mathit{fresh}(h), h[\mathit{fresh}(h) \mapsto (v_1, v_2)]) \text{ in”}$$

The side-condition $\mathit{size}(h) < l$ is recorded in the precondition of the theorem from the decompiler.

Conclusions

Decompilation and compilation are based on:

(a) modelling loops as tail-recursion, and

(b) proving (a) correct using termination proofs.

Details described in paper available at: www.cl.cam.ac.uk/mom22

Acknowledgments: I would like to thank Mike Gordon, Konrad Slind, Thomas Tuerk, Anthony Fox, Susmit Sarkar, Peter Sewell, Boris Feigin, Max Bolingbroke, John Regehr, Lu Zhao and Matthew Parkinson for discussions.

Conclusions

Decompilation and compilation are based on:

(a) modelling loops as tail-recursion, and

(b) proving (a) correct using termination proofs.

Details described in paper available at: www.cl.cam.ac.uk/mom22

My questions for you:

- How has the correspondence between loops and tail-recursion been formally proved before?
- Have termination proofs been used for this?

Questions?

Acknowledgments: I would like to thank Mike Gordon, Konrad Slind, Thomas Tuerk, Anthony Fox, Susmit Sarkar, Peter Sewell, Boris Feigin, Max Bolingbroke, John Regehr, Lu Zhao and Matthew Parkinson for discussions.