Extensible proof-producing compilation

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

CC 2009

Motivation

This talk is about compiling functions from the HOL4 theorem prover to machine code.

Motivation

This talk is about compiling functions from the HOL4 theorem prover to machine code.

What is HOL4?

- an interactive and programmable proof assistant
- implements higher-order logic
- used for formalising maths, verification of hardware and software ... (e.g. Anthony Fox has used it for verifying the hardware of an ARM processor)

Motivation

This talk is about compiling functions from the HOL4 theorem prover to machine code.

What is HOL4?

- an interactive and programmable proof assistant
- implements higher-order logic
- used for formalising maths, verification of hardware and software ... (e.g. Anthony Fox has used it for verifying the hardware of an ARM processor)

Aim: user verifies an algorithm, clicks a button and then receives machine code, which is guaranteed (via proof in HOL4) to correctly implement the algorithm.

Example

Given function f as input

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

the compiler generates ARM machine code:

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

Example

Given function f as input

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

the compiler generates ARM machine code:

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

and automatically proves a certificate HOL4 theorem, which states that f is executed by machine code:

$$\begin{array}{ll} \vdash & \{\mathsf{r1}\; r_1 * \mathsf{pc}\; p * \mathsf{s}\} \\ & \;p : \texttt{E351000A 2241100A 2AFFFFFC} \\ & \{\mathsf{r1}\; f(r_1) * \mathsf{pc}\; (p{+}12) * \mathsf{s}\} \end{array}$$

Example, cont.

One can prove properties of f since it lives in HOL4:

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

Here mod is modulus over unsigned machine words.

Example, cont.

One can prove properties of f since it lives in HOL4:

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

Here mod is modulus over unsigned machine words.

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

$$\begin{array}{ll} \vdash & \{\mathsf{r1}\ r_1 * \mathsf{pc}\ p * \mathsf{s}\} \\ & \ p : \texttt{E351000A 2241100A 2AFFFFFC} \\ & \{\mathsf{r1}\ (r_1 \bmod 10) * \mathsf{pc}\ (p{+}12) * \mathsf{s}\} \end{array}$$

Example, cont.

One can prove properties of f since it lives in HOL4:

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

Here mod is modulus over unsigned machine words.

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

$$\vdash \ \{\mathsf{r1}\ r_1 * \mathsf{pc}\ p * \mathsf{s}\}$$
$$p : \texttt{E351000A 2241100A 2AFFFFFC}$$
$$\{\mathsf{r1}\ (r_1 \text{ mod } 10) * \mathsf{pc}\ (p{+}12) * \mathsf{s}\}$$

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

Talk outline

1. how is the proof-producing compiler implemented?
2. how do extensions work? example: LISP interpreter
3. design decisions and related work

Methodology

To compile function f:

1. code generation:
   generate, without proof, machine code from input f;
2. decompilation:
   derive, via proof, a function f' describing the machine code;
3. certification:
   prove $f = f'$.

In TACAS'98, Pnueli et al. call this method translation validation.

Example, code generation

When compiling function f:

$$
\begin{array}{l}
f(r_0, r_1, m) = \\
\quad \text{if } r_0 = 0 \text{ then } (r_0, r_1, m) \text{ else} \\
\quad\quad \text{let } r_1 = m(r_1) \text{ in} \\
\quad\quad \text{let } r_0 = r_0 - 1 \text{ in} \\
\quad\quad\quad f(r_0, r_1, m)
\end{array}
$$

Code generation produces x86 assembly:

Example, code generation

When compiling function f:

$$
\begin{array}{l}
f(r_0, r_1, m) = \\
\quad \text{if } r_0 = 0 \text{ then } (r_0, r_1, m) \text{ else} \\
\quad\quad \text{let } r_1 = m(r_1) \text{ in} \\
\quad\quad \text{let } r_0 = r_0 - 1 \text{ in} \\
\quad\quad\quad f(r_0, r_1, m)
\end{array}
$$

Code generation produces x86 assembly:

```
L1:  test eax, eax
     jz L2
     mov ecx,[ecx]
     dec eax
     jmp L1
L2:
```

Example, code generation

When compiling function f:

$$\begin{array}{l} f(r_0, r_1, m) = \\ \quad \text{if } r_0 = 0 \text{ then } (r_0, r_1, m) \text{ else} \\ \qquad \text{let } r_1 = m(r_1) \text{ in} \\ \qquad \text{let } r_0 = r_0 - 1 \text{ in} \\ \qquad\quad f(r_0, r_1, m) \end{array}$$

Code generation produces x86 assembly, which NASM translates:

```
0:  85C0        L1:  test eax, eax
2:  7405             jz L2
4:  8B09             mov ecx,[ecx]
6:  48               dec eax
7:  EBF7             jmp L1
                L2:
```

Initial input language

The initial input language is designed for ease of code generation:

- all variables must have names of registers r_0, r_1, r_2, stack locations s_1, s_2, or memory functions m, m_1, m_2 etc.
- basic operations over registers are permitted, e.g.
  let $r_1 = r_2 + r_4$ in ...
  let $r_3 = 50$ in ...
- simple comparisons are supported, e.g.
  if $(r_2 = 5) \wedge (r_3 \,\&\, 3 = 0)$ then ... else ...
- tail-recursive function calls allowed.

This language is very restrictive, but can be used as compiler back-end, or extended directly (see later slides).

Example, decompilation

Returning to our example... the second stage of compilation is *decompilation* of the generated code (FMCAD 2008).

Decompilation: derive a function f' describing the code.

Example, decompilation

Returning to our example... the second stage of compilation is *decompilation* of the generated code (FMCAD 2008).

Decompilation: derive a function f' describing the code.

First, theorems describing one pass through the code are derived:

$$
\begin{array}{l}
\mathit{eax} \mathbin{\&} \mathit{eax} = 0 \Rightarrow \\
\{\, (\mathrm{eax}, \mathrm{ecx}, \mathrm{m}) \text{ is } (\mathit{eax}, \mathit{ecx}, \mathit{m}) * \mathrm{eip}\ p * \mathrm{s} \,\} \\
\ p : \texttt{85C074058B0948EBF7} \\
\{\, (\mathrm{eax}, \mathrm{ecx}, \mathrm{m}) \text{ is } (\mathit{eax}, \mathit{ecx}, \mathit{m}) * \mathrm{eip}\ (p{+}9) * \mathrm{s} \,\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{eax} \mathbin{\&} \mathit{eax} \neq 0 \ \wedge\ \mathit{ecx} \in \mathrm{domain}\ \mathit{m} \ \wedge\ (\mathit{ecx} \mathbin{\&} 3 = 0) \Rightarrow \\
\{\, (\mathrm{eax}, \mathrm{ecx}, \mathrm{m}) \text{ is } (\mathit{eax}, \mathit{ecx}, \mathit{m}) * \mathrm{eip}\ p * \mathrm{s} \,\} \\
\ p : \texttt{85C074058B0948EBF7} \\
\{\, (\mathrm{eax}, \mathrm{ecx}, \mathrm{m}) \text{ is } (\mathit{eax}{-}1, \mathit{m}(\mathit{ecx}), \mathit{m}) * \mathrm{eip}\ p * \mathrm{s} \,\}
\end{array}
$$

Example, decompilation, cont.

A special loop rule is used to introduce a tail recursion.

$$\begin{array}{ll} \forall \text{res res' } c. & (\forall x.\ P\ x \wedge G\ x \Rightarrow \{\text{res } x\}\, c\, \{\text{res } (F\ x)\}) \wedge \\ & (\forall x.\ P\ x \wedge \neg(G\ x) \Rightarrow \{\text{res } x\}\, c\, \{\text{res}'\ (D\ x)\}) \Rightarrow \\ & (\forall x.\ \text{pre } x \Rightarrow \{\text{res } x\}\, c\, \{\text{res}'\ (\text{tailrec } x)\}) \end{array}$$

where tailrec and pre are:

$$\begin{array}{rcl} \text{tailrec } x & = & \text{if } (G\ x) \text{ then tailrec } (F\ x) \text{ else } (D\ x) \\ \text{pre } x & = & P\ x \wedge (G\ x \Rightarrow \text{pre } (F\ x)) \end{array}$$

Example, decompilation, cont.

With appropriate instantiations of variables, tailrec satisfies:

$$\begin{array}{l} \text{tailrec}(eax, ecx, m) = \\ \quad \text{if } eax \,\&\, eax = 0 \text{ then } (eax, ecx, m) \text{ else} \\ \quad\quad \text{let } ecx = m(ecx) \text{ in} \\ \quad\quad \text{let } eax = eax - 1 \text{ in} \\ \quad\quad\quad \text{tailrec}(eax, ecx, m) \end{array}$$

and we have a certificate theorem:

$$\begin{array}{l} \text{pre}(eax, ecx, m) \Rightarrow \\ \{\, (\text{eax}, \text{ecx}, \text{m}) \text{ is } (eax, ecx, m) * \text{eip } p * \text{s} \,\} \\ \ p : \texttt{85C074058B0948EBF7} \\ \{\, (\text{eax}, \text{ecx}, \text{m}) \text{ is tailrec}(eax, ecx, m) * \text{eip } (p{+}9) * \text{s} \,\} \end{array}$$

We define decompilation $f' = \text{tailrec}$.

Certification

To compile function f:

1. code generation:
   generate, without proof, machine code from input f;
2. decompilation:
   derive, via proof, a function f' describing the machine code;
3. certification:
   prove $f = f'$.

Example, certification

Since f and f' are instances of tailrec,

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

it is sufficient to prove their components equivalent, in this case:

$$\begin{array}{rcl}
(\lambda(r_0, r_1, m).\ r_0 \neq 0) & = & (\lambda(eax, ecx, m).\ eax\ \&\ eax \neq 0) \\
(\lambda(r_0, r_1, m).\ (r_0{-}1, m(r_1), m)) & = & (\lambda(eax, ecx, m).\ (eax{-}1, m(ecx), m)) \\
(\lambda(r_0, r_1, m).\ (r_0, r_1, m)) & = & (\lambda(eax, ecx, m).\ (eax, ecx, m))
\end{array}$$

Example, certification

Since f and f' are instances of tailrec,

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

it is sufficient to prove their components equivalent, in this case:

$$\begin{array}{rcl}
(\lambda(r_0, r_1, m).\ r_0 \neq 0) & = & (\lambda(eax, ecx, m).\ eax\ \&\ eax \neq 0) \\
(\lambda(r_0, r_1, m).\ (r_0{-}1, m(r_1), m)) & = & (\lambda(eax, ecx, m).\ (eax{-}1, m(ecx), m)) \\
(\lambda(r_0, r_1, m).\ (r_0, r_1, m)) & = & (\lambda(eax, ecx, m).\ (eax, ecx, m))
\end{array}$$

Lightweight optimisations are undone:

- small tweaks, like $eax\ \&\ eax = eax$;
- some instruction reordering;
- conditional execution (for ARM and x86);
- dead-code removal;
- shared-tail elimination (next slides)

Shared-tail elimination

The assignment to r_1 is shared:

$$f(r_1, r_2) \;=\; \text{if } \; r_1 = 0 \; \text{ then let } r_2 = 23 \text{ in let } r_1 = 4 \text{ in } (r_1, r_2)$$
$$\text{else let } r_2 = 56 \text{ in let } r_1 = 4 \text{ in } (r_1, r_2)$$

Another formulation:

$$g(r_1, r_2) \;=\; \text{let } (r_1, r_2) = g_2(r_1, r_2) \text{ in let } r_1 = 4 \text{ in } (r_1, r_2)$$

$$g_2(r_1, r_2) \;=\; \text{if } \; r_1 = 0 \; \text{ then let } r_2 = 23 \text{ in } (r_1, r_2)$$
$$\text{else let } r_2 = 56 \text{ in } (r_1, r_2)$$

Both produce ARM code:

```
 0: E3510000     cmp r1,#0
 4: 03A02017     moveq r2,#23
 8: 13A02038     movne r2,#56
12: E3A01004     mov r1,#4
```

Talk outline

1. how to implement basic proof-producing compiler?
2. how do extensions work? LISP interpreter.
3. design decisions and related work

Extensions

The introduction showed how to prove:

$$\{\mathsf{r1}\ r_1 * \mathsf{pc}\ p * \mathsf{s}\}$$
$$p : \texttt{E351000A 2241100A 2AFFFFFC}$$
$$\{\mathsf{r1}\ (r_1 \bmod 10) * \mathsf{pc}\ (p{+}12) * \mathsf{s}\}$$

Such theorems can be used to extend the compiler's input language, in this case with:

$$\text{let } r_1 = r_1 \bmod 10 \text{ in } \_$$

Extensions, cont.

Example. The extension allows us to compile:

$$
\begin{array}{rcl}
f(r_1, r_2, r_3) & = & \text{let } r_1 = r_1 + r_2 \text{ in} \\
& & \text{let } r_1 = r_1 + r_3 \text{ in} \\
& & \text{let } r_1 = r_1 \text{ mod } 10 \text{ in} \\
& & \quad r_1
\end{array}
$$

Code generation produces "tagged-code":

$$\texttt{E0811002 E0811003}\ \boxed{\texttt{E351000A 2241100A 2AFFFFFC}}$$

The decompiler will know to use the supplied theorem for tagged code blocks. The certification stage is unchanged.

Extensions, cont.

The one-pass theorem is derived using the supplied theorem:

$\{\mathsf{r1}\ r_1 * \mathsf{pc}\ p * \mathsf{s}\}$
p : `E0811002 E0811003 E351000A 2241100A 2AFFFFFC`
$\{\mathsf{r1}\ ((r_1 + r_2 + r_3) \bmod 10) * \mathsf{pc}\ (p{+}20) * \mathsf{s}\}$

Previously proved theorems are used a building blocks.

Example, LISP interpreter

Abstract extensions can also be made.

As a case study, we compiled a small LISP interpreter.

Theorems were proved for primitive LISP operations, e.g.

$$
\begin{array}{l}
(\exists x\ y.\ v_1 = \mathrm{Dot}\ x\ y) \ \Rightarrow \\
\{\ \mathrm{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathrm{pc}\ p\ \} \\
\ p : \texttt{E5933000} \\
\{\ \mathrm{lisp}\ (\mathrm{car}\ v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathrm{pc}\ (p+4)\ \}
\end{array}
$$

$$
\begin{array}{l}
(\mathrm{size}\ v_1 + \mathrm{size}\ v_2 + \mathrm{size}\ v_3 + \mathrm{size}\ v_4 + \mathrm{size}\ v_5 + \mathrm{size}\ v_6) < l \ \Rightarrow \\
\{\ \mathrm{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathrm{s} * \mathrm{pc}\ p\ \} \\
\ p : \texttt{E50A3018 E50A4014 E50A5010} \ \ldots\ \texttt{E51A7008 E51A8004} \\
\{\ \mathrm{lisp}\ (\mathrm{cons}\ v_1\ v_2, v_2, v_3, v_4, v_5, v_6, l) * \mathrm{s} * \mathrm{pc}\ (p+328)\ \}
\end{array}
$$

Here $v_1 \ldots v_6$ are abstract s-expressions and lisp is a heap invariant.

Example, LISP interpreter, cont.

LISP evaluation was defined as a tail-recursive function *lisp_eval* using only variables $v_1 \ldots v_6$, and operations for which the code generator has verified building blocks.

Compilation proceeds as normal and produces:

$$
\begin{array}{l}
\mathit{lisp\_eval\_pre}(v_1, v_2, v_3, v_4, v_5, v_6, l) \;\Rightarrow \\
\{\ \mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{s} * \mathsf{pc}\ p\ \} \\
\ p : \ldots \text{ the generated code } \ldots \\
\{\ \mathsf{lisp}\ (\mathit{lisp\_eval}(v_1, v_2, v_3, v_4, v_5, v_6, l)) * \mathsf{s} * \mathsf{pc}\ (p + 3012)\ \}
\end{array}
$$

This case study has evolved from the one reported in the proceeding. Ask, and I'll tell more about the status of this project.

Talk outline

1. how to implement basic proof-producing compiler?
2. how do extensions work? LISP interpreter.
3. design decisions and related work

Design decisions and related work

Why not verify the compiler? [1]

Why not instrument the code generation to produce proofs? [2,3]

Does the compiler use heuristics to find the proofs? [4]

[1] Leroy, POPL 2006; [2] Pnueli, TACAS 1998; [3] Rinard, CC 1999; [4] Necula, PLDI 2000

Design decisions and related work

Why not verify the compiler? [1]

- Verified compilers are harder to produce. Also requires defining the input language, which restricts the extensibility.

Why not instrument the code generation to produce proofs? [2,3]

Does the compiler use heuristics to find the proofs? [4]

[1] Leroy, POPL 2006; [2] Pnueli, TACAS 1998; [3] Rinard, CC 1999; [4] Necula, PLDI 2000

Design decisions and related work

Why not verify the compiler? [1]

- Verified compilers are harder to produce. Also requires defining the input language, which restricts the extensibility.

Why not instrument the code generation to produce proofs? [2,3]

- It is easier to keep the prover separate and small. That way external tools GAS and NASM can be used.

Does the compiler use heuristics to find the proofs? [4]

---

[1] Leroy, POPL 2006; [2] Pnueli, TACAS 1998; [3] Rinard, CC 1999; [4] Necula, PLDI 2000

Design decisions and related work

Why not verify the compiler? [1]

- Verified compilers are harder to produce. Also requires defining the input language, which restricts the extensibility.

Why not instrument the code generation to produce proofs? [2,3]

- It is easier to keep the prover separate and small. That way external tools GAS and NASM can be used.

Does the compiler use heuristics to find the proofs? [4]

- No, the input language and optimisations are simple so that the prover need not guess what the compiler does.

---

[1] Leroy, POPL 2006; [2] Pnueli, TACAS 1998; [3] Rinard, CC 1999; [4] Necula, PLDI 2000

Design decisions and related work

Why not verify the compiler? [1]

- Verified compilers are harder to produce. Also requires defining the input language, which restricts the extensibility.

Why not instrument the code generation to produce proofs? [2,3]

- It is easier to keep the prover separate and small. That way external tools GAS and NASM can be used.

Does the compiler use heuristics to find the proofs? [4]

- No, the input language and optimisations are simple so that the prover need not guess what the compiler does.

Other questions?

---

[1] Leroy, POPL 2006; [2] Pnueli, TACAS 1998; [3] Rinard, CC 1999; [4] Necula, PLDI 2000