Machine-code verification

Experience of tackling medium-sized case studies using decompilation into logic

ACL2’14, Vienna

Magnus Myreen

Currently at UNIVERSITY OF CAMBRIDGE but soon at UNIVERSITY OF TECHNOLOGY
Why machine code?

Computer systems:

- computer networks
- multi-language implementations
- source code (Java, Lisp, C etc.)
- bytecode or LLVM
- machine code
- hardware
- electric charge

Proofs only target a model of reality.
Why machine code?

Computer systems:

- computer networks
- multi-language implementations
- source code (Java, Lisp, C etc.)
- bytecode or LLVM
- machine code
- hardware
- electric charge

Proofs only target a **model** of reality.

(Tests run on the ‘real thing’, but are not as insightful.)
Why machine code?

Computer systems:
- computer networks
- multi-language implementations
- source code (Java, Lisp, C etc.)
- bytecode or LLVM
- machine code
- hardware
- electric charge

Proofs only target a model of reality.
(Tests run on the ‘real thing’, but are not as insightful.)
Why machine code?

Computer systems:

- computer networks
- multi-language implementations
- source code (Java, Lisp, C etc.)
- bytecode or LLVM
- machine code
- hardware
- electric charge

Ultimately all program verification ought to reach real machine code.

Proofs only target a model of reality.
(Tests run on the ‘real thing’, but are not as insightful.)
Machine code

Machine code,

```
E1510002 B0422001 C0411002 01AFFFFFB
```

is impossible to read, write or maintain manually.
Machine code

Machine code,

\[ \text{E1510002 B0422001 C0411002 01AFFFFFFB} \]

is impossible to read, write or maintain manually.

However, for theorem-prover-based formal verification:

machine code is clean and tractable!
Machine code

Machine code, 

\[ \text{E1510002 B0422001 C0411002 01AFFFFFFB} \]

is impossible to read, write or maintain manually.

However, for theorem-prover-based formal verification:

machine code is clean and tractable!

Reason:

- all types are concrete: word32, word8, bool.
- state consists of a few simple components: a few registers, a memory and some status bits.
- each instruction performs only small well-defined updates.
Challenges of Machine Code
Challenges of Machine Code

machine code

correctness

{P} code {Q}
Challenges of Machine Code

machine code

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

correctness
{P} code {Q}
Challenges of Machine Code

- several large, detailed models
- unstructured code
- very low-level and limited resources

ARM/x86/PowerPC model (1000...10,000 lines each)
This talk

Part 1: my approach (PhD work)

Part 2: verification of existing code

Part 3: construction of correct code
This talk

Part 1: my approach (PhD work)

Part 2: verification of existing code

Part 3: construction of correct code
This talk

Part 1: my approach (PhD work)
  ‣ automation: code to spec
  ‣ automation: spec to code

Part 2: verification of existing code

Part 3: construction of correct code
This talk

Part 1: my approach (PhD work)
  ‣ automation: code to spec
  ‣ automation: spec to code

Part 2: verification of existing code
  ‣ verification of gcc output for microkernel (7,000 lines of C)

Part 3: construction of correct code
This talk

Part 1: my approach (PhD work)
  ‣ automation: code to spec
  ‣ automation: spec to code

Part 2: verification of existing code
  ‣ verification of gcc output for microkernel (7,000 lines of C)

Part 3: construction of correct code
  ‣ verified implementation of Lisp that can run Jared Davis’ Milawa
HOL: fully-expansive LCF-style prover

The aim is to prove deep functional properties of machine code.

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.
Infrastructure

During my PhD, I developed the following infrastructure:

- **func** -> compiler -> (code, thm)
- code -> decompiler -> (func, thm)
- machine-code Hoare triple

...each part will be explained in the next slides.
Models of machine code

Machine models borrowed from work by others:

**ARM model, by Fox [TPHOLs’03]**
- covers practically all ARM instructions, for old and new ARMs
- still actively being developed

**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 → HOL4) of Leroy’s PowerPC model
- instruction decoder added
Hoare triples

Each model can be evaluated, e.g. ARM instruction \texttt{add r0,r0,r0} is described by theorem:

\[
\begin{align*}
|&- \ (\text{ARM_READ_MEM } ((31 > 2) (\text{ARM_READ_REG} \ 15w \ \text{state})) \ \text{state} = \ 0xE0800000w) \land \ \neg \text{state.undefined} \Rightarrow \\
& (\text{NEXT_ARM_MMU} \ cp \ \text{state} = \\
& \quad \text{ARM_WRITE_REG} \ 15w \ (\text{ARM_READ_REG} \ 15w \ \text{state} + 4w) \\
& \quad \text{ARM_WRITE_REG} \ 0w \\
& \quad \ (\text{ARM_READ_REG} \ 0w \ \text{state} + \ \text{ARM_READ_REG} \ 0w \ \text{state}) \ \text{state})
\end{align*}
\]
Hoare triples

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

\[
|- (\text{ARM\_READ\_MEM } ((31 >> 2) (\text{ARM\_READ\_REG} 15w \text{ state})) \text{ state} = 0xE0800000w) \land \neg \text{state.undefined} \implies
\]

\[
(\text{NEXT\_ARM\_MMU} \text{ cp state} =
\text{ARM\_WRITE\_REG} 15w (\text{ARM\_READ\_REG} 15w \text{ state} + 4w)
(\text{ARM\_WRITE\_REG} 0w
(\text{ARM\_READ\_REG} 0w \text{ state} + \text{ARM\_READ\_REG} 0w \text{ state}) \text{ state}))
\]

As a total-correctness machine-code Hoare triple:

\[
|- \text{SPEC ARM\_MODEL}
(aR 0w x * aPC p)
\{(p,0xE0800000w)\}
(aR 0w (x+x) * aPC (p+4w))
\{ R0 (x+x) * PC (p+4) \}
\]

Informal syntax for this talk:
Definition of Hoare triple

\{p\} c \{q\} \iff \forall s \ r. \ (p \ast r \ast \text{code } c) \ s \implies \exists n. \ (q \ast r \ast \text{code } c) \ (\text{run } n \ s)
Definition of Hoare triple

\{p\} \text{c} \{q\} \iff \forall s \, r. \ (p \ast r \ast \text{code c}) \ s \Rightarrow \\
\exists n. \ (q \ast r \ast \text{code c}) \ (\text{run n s})
Definition of Hoare triple

\{p\} c \{q\} \iff \forall s \ r. \ (p \ast r \ast \text{code} \ c) \ s \implies \\
\exists n. \ (q \ast r \ast \text{code} \ c) \ (\text{run} \ n \ s)
Definition of Hoare triple

\(\{p\} \ c \ \{q\} \iff \forall s \ r. \ (p \star r \star \text{code } c) \ s \implies \exists n. \ (q \star r \star \text{code } c) \ (\text{run } n \ s)\)
Definition of Hoare triple

\{p\} c \{q\} \iff \forall s r. (p \ast r \ast \text{code } c) \Rightarrow s \Rightarrow \exists n. (q \ast r \ast \text{code } c) (\text{run } n \ s)

- frame
- code separate
- total correctness
- machine code sem.
Definition of Hoare triple

\[
\{p\} \ c \ \{q\} \iff \forall s \ r. \ (p \ast r \ast \text{code } c) \ s \implies \\
\exists n. \ (q \ast r \ast \text{code } c) \ (\text{run } n \ s)
\]
Definition of Hoare triple

\[ \{ p \} \ c \ \{ q \} \iff \forall s \ r. \ (p \land r \land \text{code } c) \ s \implies \exists n. \ (q \land r \land \text{code } c) \ (\text{run } n \ s) \]

- **Frame**
- **code separate**
- **separating conjunction**
- **total correctness**
- **machine code sem.**

**Program logic** can be used directly for verification.
Definition of Hoare triple

\[ \{ p \} \ c \ \{ q \} \iff \forall s \ r. \ (p * r * \text{code} \ c) \ s \Rightarrow \exists n. \ (q * r * \text{code} \ c) \ (\text{run} \ n \ s) \]

Program logic can be used directly for verification.
But direct reasoning in this embedded logic is tiresome.
Decompiler

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:

```
f(r0, r1, m) = let r0 = 0 in g(r0, r1, m)
g(r0, r1, m) = if r1 = 0 then (r0, r1, m) else let r0 = r0 + 1 in let r1 = m(r1) in g(r0, r1, m)
```
Decompiler

Decompiler automates Hoare triple reasoning.

**Example:** Given some ARM machine code,

\[
\begin{align*}
0 & : \text{E3A00000} & \text{mov r0, #0} \\
4 & : \text{E3510000} & \text{L: cmp r1, #0} \\
8 & : \text{12800001} & \text{addne r0, r0, #1} \\
12 & : \text{15911000} & \text{ldrne r1, [r1]} \\
16 & : \text{1AFFFFF} & \text{bne L}
\end{align*}
\]
Decompiler automates Hoare triple reasoning.

Example: Given some ARM machine code,

\[
\begin{align*}
0: & \text{ E3A00000 } \quad \text{mov } r0, \#0 \\
4: & \text{ E3510000 } \quad \text{L: cmp } r1, \#0 \\
8: & \text{ 12800001 } \quad \text{addne } r0, r0, \#1 \\
12: & \text{ 15911000 } \quad \text{ldrne } r1, [r1] \\
16: & \text{ 1AFFFFFFB } \quad \text{bne L}
\end{align*}
\]

the decompiler automatically extracts a readable function:

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

Decompiler automatically proves a certificate theorem:

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

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 $f$,

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

$$
\forall x \ l \ a \ m. \ \text{list}(l, a, m) \ \Rightarrow \ f_{\text{pre}}(x, a, m)
$$

since properties of $f$ carry over to machine code via the certificate.
Decompression verification example

To verify code: prove properties of function $f$,

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

$$
\forall x \ l \ a \ m. \ \text{list}(l, a, m) \ \Rightarrow \ f_{\text{pre}}(x, a, m)
$$

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

**Proof reuse**: Given similar x86 and PowerPC code:

31C085F67405408B36EBF7
38A000002C140000408200107E80A02E38A500014BFFFFFF0

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

How to decompile:

```
e0810000  add    r0, r1, r0
e1a000a0  lsr    r0, r0, #1
e12fff1e  bx     lr
```
Decompilation

How to decompile:

```
e0810000  add  r0, r1, r0
e1a000a0  lsr  r0, r0, #1
e12fff1e  bx  lr
```
Decompilation

{ R0 i * R1 j * PC p }
p+0 : e0810000
{ R0 (i+j) * R1 j * PC (p+4) }

{ R0 i * PC (p+4) }
p+4 : e1a000a0
{ R0 (i >> 1) * PC (p+8) }

{ LR lr * PC (p+8) }
p+8 : e12fff1e
{ LR lr * PC lr }

How to decompile:

e0810000  add  r0, r1, r0
e1a000a0  lsr  r0, r0, #1
e12fff1e  bx  lr

1. derive Hoare triple theorems using Cambridge ARM model
De-compilation

How to decompile:

1. derive Hoare triple theorems using Cambridge ARM model
2. compose Hoare triples

The code snippet is as follows:

e0810000  add r0, r1, r0
e1a000a0  lsr r0, r0, #1
e12fff1e  bx lr

2. compose Hoare triples
Decompileation

How to decompile:

1. derive Hoare triple theorems using Cambridge ARM model
2. compose Hoare triples
3. extract function
(Loops result in recursive functions.)

avg (i,j) = (i+j)>>1
Decompiler implementation

Implementation:
- ML program which **fully-automatically** performs forward proof,
- **no heuristics** and no dangling proof obligations,
- loops are handled by a special loop rule which introduces tail-recursive functions:

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

with termination and side-conditions \( H \) collected as:

\[
\text{pre}(x) = (\text{if } G(x) \text{ then } \text{pre}(F(x)) \text{ else true}) \land H(x)
\]

Details in Myreen et al. [FMCAD’08].
Comparison of approaches

Decompiler automates Hoare triple reasoning.
Example:

Given some ARM machine code,

\[
\begin{align*}
0: & \text{ E3A00000 } \quad \text{mov} \; r0, \; #0 \\
4: & \text{ E3510000 } \quad \text{L:} \; \text{cmp} \; r1, \; #0 \\
8: & \text{ 12800001 } \quad \text{addne} \; r0, \; r0, \; #1 \\
12: & \text{ 15911000 } \quad \text{ldrne} \; r1, \; [r1] \\
16: & \text{ 1AFFFFFFB } \quad \text{bne} \; L
\end{align*}
\]

the decompiler automatically extracts a readable function:

\[
f(r_0, r_1, m) =
\begin{align*}
& \text{let} \; r_0 = 0 \\
& \text{in} \; g(r_0, r_1, m) \\
\end{align*}
\]

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

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: 1AFFFFFFB bne L
```

the decompiler automatically extracts a readable function:

\[
f(r_0, r_1, m) = \begin{align*}
\text{let } r_0 &= 0 \\
\text{in } g(r_0, r_1, m)
\end{align*}
\]

\[
g(r_0, r_1, m) = \begin{cases} 
(r_0, r_1, m) & \text{if } r_1 = 0 \\
\text{let } r_0 &= r_0 + 1 \\
\text{let } r_1 &= m(r_1) \\
\text{in } g(r_0, r_1, m) & \text{else}
\end{cases}
\]

direct manual proof using definition of instruction set model
Comparison of approaches

direct manual proof using definition of instruction set model
  ‣ tedious and proofs complete tied to model

0: E3A00000  mov r0, #0
4: E3510000  L: cmp r1, #0
8: 12800001  addne r0, r0, #1
12: 15911000  ldrne r1, [r1]
16: 1AFFFFFFFF  bne L
Comparison of approaches

direct manual proof using definition of instruction set model
  ‣ tedious and proofs complete tied to model

symbolic simulation
Comparison of approaches

direct manual proof using definition of instruction set model
  ‣ tedious and proofs complete tied to model

symbolic simulation
  ‣ automatic except at looping points, proofs tied to model
Comparison of approaches

Decompiler automates Hoare triple reasoning.

Example:

Given some ARM machine code,

\[
\begin{align*}
0: & \text{ E3A00000 } \quad \text{mov} \ r0, \ #0 \\
4: & \text{ E3510000 } \quad \text{L: cmp r1, #0} \\
8: & \text{ 12800001 } \quad \text{addne} \ r0, \ r0, \ #1 \\
12: & \text{ 15911000 } \quad \text{ldrne} \ r1, \ [r1] \\
16: & \text{ 1AFFFFFFB } \quad \text{bne L}
\end{align*}
\]

the decompiler automatically extracts a readable function:

\[
f(r0, r1, m) = \begin{cases} 
\text{let } r0 = 0 \text{ in } g(r0, r1, m) & \text{if } r1 = 0 \\
\text{else let } r0 = r0 + 1 \text{ in let } r1 = m(r1) \text{ in } g(r0, r1, m) \end{cases}
\]

direct manual proof using definition of instruction set model

- tedious and proofs complete tied to model

symbolic simulation

- automatic except at looping points, proofs tied to model

proof using program logic
Comparison of approaches

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:

\[
\begin{align*}
f(r0, r1, m) &= \text{let } r0 = 0 \text{ in } \ \ g(r0, r1, m) \\
g(r0, r1, m) &= \text{if } r1 = 0 \text{ then } (r0, r1, m) \text{ else } \text{let } r0 = r0 + 1 \text{ in } \text{let } r1 = m(r1) \text{ in } g(r0, r1, m)
\end{align*}
\]

direct manual proof using definition of instruction set model

- tedious and proofs complete tied to model

symbolic simulation

- automatic except at looping points, proofs tied to model

proof using program logic

- some reusable proofs, but tedious
Comparison of approaches

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: 1AFFFFFFB  bne L

the decompiler automatically extracts a readable function:

\[
\begin{align*}
\text{f}(r0, r1, m) &= \begin{cases} 
0: & \text{mov r0, #0} \\
4: & \text{L: cmp r1, #0} \\
8: & \text{addne r0, r0, #1} \\
12: & \text{ldrne r1, [r1]} \\
16: & \text{bne L}
\end{cases} \\
\text{g}(r0, r1, m) &= \begin{cases} 
\text{if } r1 = 0 & \text{then } (r0, r1, m) \\
\text{else } \begin{cases} 
0: & \text{mov r0, #0} \\
4: & \text{L: cmp r1, #0} \\
8: & \text{addne r0, r0, #1} \\
12: & \text{ldrne r1, [r1]} \\
16: & \text{bne L}
\end{cases}
\end{cases}
\end{align*}
\]

direct manual proof using definition of instruction set model
  ‣ tedious and proofs complete tied to model

symbolic simulation
  ‣ automatic except at looping points, proofs tied to model

proof using program logic
  ‣ some reusable proofs, but tedious

verification condition generation
Comparison of approaches

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: 1AFFFFFFB   bne L
```

the decompiler automatically extracts a readable function:

```markdown
f(r0, r1, m) =
let
r0 = 0
in
g(r0, r1, m)
```

```markdown
g(r0, r1, m) =
if r1 = 0 then
(r0, r1, m)
else
let
r0 = r0 + 1
let
r1 = m(r1)
in
g(r0, r1, m)
```

- **direct manual proof** using definition of instruction set model
  - tedious and proofs complete tied to model

- **symbolic simulation**
  - automatic except at looping points, proofs tied to model

- **proof using program logic**
  - some reusable proofs, but tedious

- **verification condition generation**
  - largely automatic, but requires annotating the machine code(!)
Comparison of approaches

- **Decompilation**
  - Decompiler automates Hoare triple reasoning.
  - Example:
    
    ```
    0: E3A00000   mov r0, #0
    4: E3510000   L: cmp r1, #0
    8: 12800001   addne r0, r0, #1
    12: 15911000  ldrne r1, [r1]
    16: 1AFFFFF8  bne L
    ```

    the decompiler automatically extracts a readable function:

    \[
    f(r0, r1, m) = \begin{align*}
    & \text{let } r0 = 0 \text{ in } \\
    & g(r0, r1, m) = \begin{cases} 
    (r0, r1, m) & \text{if } r1 = 0 \\
    \text{let } r0 = r0 + 1 \text{ in } \text{let } r1 = m(r1) \text{ in } g(r0, r1, m) & \text{else}
    \end{cases}
    \end{align*}
    \]

- **Direct manual proof** using definition of instruction set model
  - tedious and proofs complete tied to model

- **Symbolic simulation**
  - automatic except at looping points, proofs tied to model

- **Proof using program logic**
  - some reusable proofs, but tedious

- **Verification condition generation**
  - largely automatic, but requires annotating the machine code(!)

- **Decompilation into logic**
Comparison of approaches

Decompilation

Decompiler automates Hoare triple reasoning.

Example:

Given some ARM machine code,

\[
0: \text{E3A00000} \quad \text{mov} \ r0, \ #0 \\
4: \text{E3510000} \quad \text{L: cmp} \ r1, \ #0 \\
8: \text{12800001} \quad \text{addne} \ r0, \ r0, \ #1 \\
12: \text{15911000} \quad \text{ldrne} \ r1, [r1] \\
16: \text{1AFFFFFFB} \quad \text{bne} \ L
\]

the decompiler automatically extracts a readable function:

\[
f(r0, r1, m) = \begin{cases} 
\text{let } r0 = 0 \text{ in } g(r0, r1, m) \\
\text{if } r1 = 0 \text{ then } (r0, r1, m) \text{ else let } \\
\text{r0} = \text{r0} + 1 \text{ in let } \\
\text{r1} = m(r1) \text{ in } g(r0, r1, m)
\end{cases}
\]

direct manual proof using definition of instruction set model

\(\quad\) tedious and proofs complete tied to model

symbolic simulation

\(\quad\) automatic except at looping points, proofs tied to model

proof using program logic

\(\quad\) some reusable proofs, but tedious

verification condition generation

\(\quad\) largely automatic, but requires annotating the machine code(!)

decompilation into logic

\(\quad\) model-specific part is automatic, does not req. annotations
Comparison of approaches

Decompiler automates Hoare triple reasoning.

Example:

```
0: E3A00000  mov r0, #0
4: E3510000  L: cmp r1, #0
8: 12800001  addne r0, r0, #1
12: 15911000  ldrne r1, [r1]
16: 1AFFFFFFB  bne L
```

The decompiler automatically extracts a readable function:

```
f(r0, r1, m) =
let r0 =0 in
\( f(r0, r1, m) = \)
if r1 =0 then
(r0, r1, m)
else
let r0 =r0 +1 in
let r1 =m(r1) in
\( f(r0, r1, m) \)
```

direct manual proof using definition of instruction set model
  ‣ tedious and proofs complete tied to model

symbolic simulation
  ‣ automatic except at looping points, proofs tied to model

proof using program logic
  ‣ some reusable proofs, but tedious

verification condition generation
  ‣ largely automatic, but requires annotating the machine code(!)

decompilation into logic
  ‣ model-specific part is automatic, does not req. annotations
  ‣ can implement proof-producing compilation (next slide)
Comparison of approaches

**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: 1AFFFFF8  bne L
```

the decompiler automatically extracts a readable function:

```latex
f(r0, r1, m) =
\begin{align*}
\text{let } r0 &= 0 \\
g(r0, r1, m) &= \text{let } r0 = r0 + 1 \\
&\quad \text{let } r1 = m(r1) \\
&\quad g(r0, r1, m)
\end{align*}
```

**direct manual proof** using definition of instruction set model

- tedious and proofs complete tied to model

**symbolic simulation**

- automatic except at looping points, proofs tied to model

**proof using program logic**

- some reusable proofs, but tedious

**verification**

- symbolic simulation + support for loops (tail-rec.), done over a program logic (not machine model)

**decompilation into logic**

- model-specific part is automatic, does not req. annotations
- can implement *proof-producing compilation* (next slide)
Proof-producing compilation

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
2AFFFFFFFC bcs L
```

and automatically proves a certificate HOL theorem:

$$\vdash \{ \text{R1 } r_1 \ast \text{PC } p \ast s \}$$

$$p : \text{E351000A 2241100A 2AFFFFFFFC}$$

$$\{ \text{R1 } f(r_1) \ast \text{PC } (p+12) \ast s \}$$
Compilation, example cont.

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

\[
\forall x. \ f(x) = x \mod 10
\]

Properties proved of \( f \) translate to properties of the machine code:

\[
\begin{align*}
\vdash & \{ R1 \ r_1 \cdot PC \ p \cdot s \} \\
& p : \text{E351000A 2241100A 2AFFFFFFC} \\
& \{ R1 \ (r_1 \mod 10) \cdot PC \ (p+12) \cdot s \}
\end{align*}
\]

Additional feature: the compiler can use the above theorem to extend its input language with: let \( r_1 = r_1 \mod 10 \) in _
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]
Infrastructure (again)

During my PhD, I developed the following infrastructure:

- decompiler
- compiler
- machine-code Hoare triple
- ARM
- x86
- PowerPC

...each part will be explained in the next slides.
This talk

Part 1:
  ‣ automation: code to spec
  ‣ automation: spec to code

Part 2: verification of existing code
  ‣ verification of gcc output for microkernel (7,000 lines of C)

Part 3:
  ‣ verified that can run Jared Davis’
L4.verified

seL4 = a formally verified general-purpose microkernel

(Work by Gerwin Klein’s team at NICTA, Australia)
L4.verified

seL4 = a formally verified general-purpose microkernel

about 7,000 lines of C code and assembly

(Work by Gerwin Klein’s team at NICTA, Australia)
seL4 = a formally verified general-purpose microkernel

about 7,000 lines of C code and assembly
200,000 lines of Isabelle/HOL proofs

(Work by Gerwin Klein’s team at NICTA, Australia)
Assumptions

L4.verified project assumes correctness of:

- C compiler (gcc)
- inline assembly
- hardware
- hardware management
- boot code
- virtual memory
Assumptions

L4.verified project assumes correctness of:

- C compiler (gcc)
- inline assembly
- hardware
- hardware management
- boot code
- virtual memory

The aim of this work is to remove the first assumption.
Assumptions

L4.verified project assumes correctness of:

- C compiler (gcc)
- inline assembly
- hardware
- hardware management
- boot code
- virtual memory
- Cambridge ARM model

The aim of this work is to remove the first assumption.
Assumptions

L4.verified project assumes correctness of:

- C compiler (gcc)
- inline assembly (?)
- hardware
- hardware management
- boot code (?)
- virtual memory
- Cambridge ARM model

The aim of this work is to remove the first assumption.
Aim: extend downwards

detailed model of C code

low-level design

high-level design

Haskell prototype

real C code

trusted

existing L4.verified work
Aim: extend downwards

- high-level design
- low-level design
- detailed model of C code
- Haskell prototype
- real C code

existing L4.verified work

Aim: remove need to trust C compiler and C semantics
Aim: extend downwards

existing L4.verified work

detailed model of C code

low-level design

high-level design

Haskell prototype

real C code

Aim: remove need to trust C compiler and C semantics
Using Cambridge ARM model

Cambridge ARM model

detailed model of C code

low-level design

Haskell prototype

real C code

high-level design

new extension  existing L4.verified work
Using Cambridge ARM model

- high-level design
- low-level design
- detailed model of C code
- Haskell prototype
- real C code

Cambridge ARM model

new extension
existing L4.verified work

gcc (not trusted)
Using Cambridge ARM model

- high-level design
- low-level design
- detailed model of C code
- Haskell prototype
- real C code
- seL4 machine code
- Cambridge ARM model

existing L4.verified work

new extension

gcc (not trusted)
Using Cambridge ARM model

- high-level design
- low-level design
- detailed model of C code
- machine code as functions
- seL4 machine code
- real C code
- Haskell prototype
- gcc (not trusted)

Cambridge ARM model

existing L4.verified work

new extension
Using Cambridge ARM model

- high-level design
- low-level design
- detailed model of C code
- machine code as functions
- seL4 machine code
- Cambridge ARM model
- Haskell prototype
- real C code
- gcc (not trusted)

new extension
existing L4.verified work
Using Cambridge ARM model

- high-level design
- low-level design
- detailed model of C code
  - refinement proof
  - machine code as functions
    - decompilation
      - seL4 machine code
      - Cambridge ARM model
        - seL4 machine code
        - existing L4.verified work
          - new extension

- Haskell prototype
  - real C code
  - gcc (not trusted)
Approach

- decompilation by me
- refinement proof by Thomas Sewell (NICTA)
Stage 1: decompilation

machine code as functions

seL4 machine code

Cambridge ARM model

decompilation
Sample C code:

```c
uint avg (uint i, uint j) {
    return (i + j) / 2;
}
```
Decompilation

Sample C code:

```c
uint avg (uint i, uint j) {
    return (i + j) / 2;
}
```

machine code:

```
e0810000  add  r0, r1, r0
e1a000a0  lsr  r0, r0, #1
e12fff1e  bx  lr
```
Decompilation

Sample C code:

```c
uint avg (uint i, uint j) {
    return (i + j) / 2;
}
```

Resulting function:

```hs
avg (r0, r1) = let r0 = r1 + r0 in
               let r0 = r0 >> 1 in
               r0
```

Machine code:

```assembly
E0810000  add    r0, r1, r0
E1A000A0  lsr    r0, r0, #1
E12FFF1E  bx     lr
```

decompilation via ARM model
Decompilation

Sample C code:

```c
uint avg (uint i, uint j) {
    return (i + j) / 2;
}
```

machine code:

```
ge0810000  add  r0, r1, r0
e1a000a0  lsr  r0, r0, #1
e12fff1e  bx  lr
```

decompilation via ARM model

Resulting function:

```
avg (r0, r1) = let r0 = r1 + r0 in
             let r0 = r0 >> 1 in
             r0
```

HOL4 certificate theorem:

```
{ R0 i * R1 j * LR lr * PC p }
p : e0810000  e1a000a0  e12fff1e
{ R0 (avg(i,j)) * R1 _ * LR _ * PC lr }
```
Decompilation

Sample C code:
```c
uint avg (uint i, uint j) {
    return (i + j) / 2;
}
```

HOL4 certificate theorem:
```plaintext
\{ R0 i \ast R1 j \ast LR lr \ast PC p \}
p : e0810000 e1a000a0 e12fff1e
\{ R0 (avg(i,j)) \ast R1 _ \ast LR _ \ast PC lr \}
```

Resulting function:
```plaintext
avg (r0, r1) = let r0 = r1 + r0 in
    let r0 = r0 >> 1 in
    r0
```

machine code:
```plaintext
e0810000  add  r0, r1, r0
e1a000a0  lsr  r0, r0, #1
e12fff1e  bx   lr
```

gcc (not trusted)
Decompilation

Sample C code:

```c
uint avg (uint i, uint j) {
    return (i + j) / 2;
}
```

HOL4 certificate theorem:

```
{ R0 i * R1 j * LR lr * PC p }
p : e0810000 e1a000a0 e12fff1e
{ R0 (avg(i,j)) * R1 _ * LR _ * PC lr }
```

Resulting function:

```c
avg (r0, r1) = let r0 = r1 + r0 in
    let r0 = r0 >> 1 in
    r0
```

Machine code:

```
gcc (not trusted)
e0810000 add r0, r1, r0
e1a000a0 lsr r0, r0, #1
e12fff1e bx lr
```

Decompilation:

- bit-string arithmetic
- return instruction
Decompilation

Sample C code:

```c
uint avg (uint i, uint j) {
    return (i + j) / 2;
}
```

machine code:

```
ge0810000  add    r0, r1, r0
e1a000a0  lsr    r0, r0, #1
e12fff1e  bx lr
```

HOL4 certificate theorem:

```hol
{ R0 i * R1 j * LR lr * PC p } p : e0810000 e1a000a0 e12fff1e
{ R0 (avg(i,j)) * R1 _ * LR _ * PC lr }
```

Resulting function:

```
avg (r0, r1) = let r0 = r1 + r0 in
let r0 = r0 >> 1 in r0
```

bit-string arithmetic

bit-string right-shift

return instruction

decompilation via ARM model
Decompilation

Sample C code:

```c
uint avg (uint i, uint j) {
    return (i + j) / 2;
}
```

Machine code:

```
ge0810000  add  r0, r1, r0
e1a000a0  lsr  r0, r0, #1
e12fff1e  bx  lr
```

HOL4 certificate theorem:

```
{ R0 i * R1 j * LR lr * PC p }  
p : e0810000  e1a000a0  e12fff1e  
{ R0 (avg(i,j)) * R1 _ * LR _ * PC lr }  
```

Resulting function:

```
avg (r0, r1) = let r0 = r1 + r0 in 
 let r0 = r0 >> 1 in 
  r0
```

- **bit-string arithmetic**
- **bit-string right-shift**
- **decompilation**
- return instruction
- separation logic:*
Decompiling seL4: Challenges

• seL4 is ~12,000 lines of machine code

• compiled using gcc -O2

• must be compatible with L4.verified proof
Decompiling seL4: Challenges

- seL4 is ~12,000 lines of machine code
  ✓ decompilation is compositional
- compiled using gcc -O2
- must be compatible with L4.verified proof
Decompiling seL4: Challenges

- seL4 is ~12,000 lines of machine code
  - decompilation is compositional
- compiled using gcc -O2
  - gcc implements ARM/C calling convention
- must be compatible with L4.verified proof
Decompiling seL4: Challenges

- seL4 is ~12,000 lines of machine code
  - decompilation is compositional
- compiled using gcc -O2
  - gcc implements ARM/C calling convention
- must be compatible with L4.verified proof
  - stack requires special treatment
Stack visible in m. code

C code:

```c
uint avg8 (uint x0, x1, x2, x3, x4, x5, x6, x7) {
    return (x0+x1+x2+x3+x4+x5+x6+x7) / 8;
}
```
Stack visible in m. code

C code:

```c
uint avg8 (uint x0, x1, x2, x3, x4, x5, x6, x7) {
    return (x0+x1+x2+x3+x4+x5+x6+x7) / 8;
}
```

Some arguments are passed on the stack,
Stack visible in m. code

C code:

```c
uint avg8 (uint x0, x1, x2, x3, x4, x5, x6, x7) {
    return (x0+x1+x2+x3+x4+x5+x6+x7) / 8;
}
```

Some arguments are passed on the stack,

```
add r1, r1, r0
add r1, r1, r2
ldr  r2, [sp]
add r1, r1, r3
add r0, r1, r2
ldmib sp, {r2, r3}
add r0, r0, r2
add r0, r0, r3
ldr  r3, [sp, #12]
add r0, r0, r3
lsr  r0, r0, #3
bx   lr
```

gcc
Stack visible in m. code

C code:

```c
uint avg8 (uint x0, x1, x2, x3, x4, x5, x6, x7) {
    return (x0+x1+x2+x3+x4+x5+x6+x7) / 8;
}
```

Some arguments are passed on the stack, and cause memory ops in machine code... that are not present in C semantics.
Solution

Use separation-logic inspired approach

stack pointer: sp

3 slots of unused but required stack space

rest of stack
Solution

Use separation-logic inspired approach

stack pointer: sp

3 slots of unused but required stack space

rest of stack

m
Solution

Use separation-logic inspired approach

stack sp 3 (s0::s1::s2::s3::s4::ss)

stack pointer: sp

3 slots of unused but required stack space

rest of stack
Solution

Use separation-logic inspired approach

stack pointer: sp

3 slots of unused but required stack space

rest of stack

stack sp 3 (s0::s1::s2::s3::s4::ss) * memory m
Solution

Use separation-logic inspired approach

stack pointer: sp

3 slots of unused but required stack space

rest of stack

stack sp 3 (s0::s1::s2::s3::s4::ss) * memory m

separation logic:*
Solution

Use separation-logic inspired approach

stack pointer: sp

3 slots of unused but required stack space

rest of stack

stack sp 3 (s0::s1::s2::s3::s4::ss) * memory m

disjoint due to *

separation logic:*
Solution (cont.)

Method:

1. static analysis to find stack operations,
2. derive stack-specific Hoare triples,
3. then run decompiler as before.

```
add r1, r1, r0
add r1, r1, r2
ldr  r2, [sp]
add r1, r1, r3
add r0, r1, r2
ldmib sp, {r2, r3}
add r0, r0, r2
add r0, r0, r3
ldr  r3, [sp, #12]
add r0, r0, r3
lsr  r0, r0, #3
bx   lr
```
Solution (cont.)

```assembly
add r1, r1, r0
add r1, r1, r2
ldr  r2, [sp]
add r1, r1, r3
add r0, r1, r2
ldmid sp, {r2, r3}
add r0, r0, r2
add r0, r0, r3
ldr  r3, [sp, #12]
add r0, r0, r3
lsr  r0, r0, #3
bx   lr
```

Method:

1. static analysis to find stack operations,
2. derive stack-specific Hoare triples,
3. then run decompiler as before.
Result

Stack load/stores become straightforward assignments.

avg8(r0, r1, r2, r3, s0, s1, s2, s3) =

let r1 = r1 + r0 in
let r1 = r1 + r2 in
let r2 = s0 in
let r1 = r1 + r3 in
let r0 = r1 + r3 in
let (r2, r3) = (s1, s2) in
let r0 = r0 + r2 in
let r0 = r0 + r3 in
let r3 = s3 in
let r0 = r0 + r3 in
let r0 = r0 >> 3 in
r0

Stack load/stores become straightforward assignments.
Result

Stack load/stores become straightforward assignments.

Additional benefit:
automatically proved certificate theorem
states explicitly stack shape/usage:

\{
  \text{stack sp n (s0::s1::s2::s3::s) } \ast \ldots \ast \text{PC p }
\}

p : \text{code}

\{
  \text{stack sp n (s0::s1::s2::s3::s) } \ast \ldots \ast \text{PC lr }
\}

\begin{verbatim}
ldr r3, [sp, #12]
add r0, r0, r3
lsr r0, r0, #3
bx lr
\end{verbatim}
Result

Stack load/stores become straightforward assignments.

Additional benefit:
automatically proved certificate theorem
states explicitly stack shape/usage:

\[
\{ \text{stack sp n (s0::s1::s2::s3::s) \ast \ldots \ast PC p} \}
\]
\[
p : \text{code}
\]
\[
\{ \text{stack sp n (s0::s1::s2::s3::s) \ast \ldots \ast PC lr} \}
\]

four arguments passed on stack
Result

Stack load/stores become straightforward assignments.

Additional benefit:
- automatically proved certificate theorem
- states explicitly stack shape/usage:

```plaintext
{ stack sp n (s0::s1::s2::s3::s) * ... * PC p }  
p : code
{ stack sp n (s0::s1::s2::s3::s) * ... * PC lr }  
```

Four arguments passed on stack does not require temp space, works for “any n”
Result

Stack load/stores become straightforward assignments.

Additional benefit:

- automatically proved certificate theorem
- states explicitly stack shape/usage:
  \[
  \{ \text{stack sp n (s0::s1::s2::s3::s) \cdots PC p} \}
  \]
  \[
  p : \text{code}
  \]
  \[
  \{ \text{stack sp n (s0::s1::s2::s3::s) \cdots PC lr} \}
  \]
- promises to leave stack unchanged
- does not require temp space, works for “any n”
- four arguments passed on stack
Other C-specifics

- **struct as return value**
  - case of passing `pointer of stack location`
  - stack assertion strong enough
- **switch statements**
  - position dependent
  - must decompile elf-files, not object files
- **infinite loops in C**
  - make `gcc` go weird
  - must be pruned from control-flow graph
Moving on to stage 2

- Detailed model of C code
- Machine code as functions
- seL4 machine code

New extension

Refinement proof

Automatic translation
Moving on to stage 2

- Detailed model of C code
- Machine code as functions
- seL4 machine code

New extension

Refinement proof

Automatic translation
Refinement proof

(Work by Thomas Sewell, NICTA)

detailed model of C code

proof by rewriting

C code as graph

‘guided SMT proof’

machine code as graph

translation (unproved)

machine code as functions
Graph language

- machine code as graph
- machine code as functions
- seL4 machine code

\[
\begin{align*}
\text{translation} & \quad \text{(unproved)} \\
\text{automatic decompilation} & \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \ quarters
Graph language

machine code as graph

↑ automatic decompilation

seL4 machine code
Graph language

Node types:
- state update
- test-and-branch
- call

machine code as graph

\[\text{automatic decompilation}\]

seL4 machine code
Graph language

Node types:
- state update
- test-and-branch
- call

Next pointers:
- node address
- return (from call)
- error
Graph language

Node types:
- state update
- test-and-branch
- call

Next pointers:
- node address
- return (from call)
- error

Theorem: any exec in graph, can be done in machine code
Graph language

Node types:
- state update
- test-and-branch
- call

Next pointers:
- node address
- return (from call)
- error

Theorem: any exec in graph, can be done in machine code

machine code as graph

\[ \uparrow \]
automatic decompilation

seL4 machine code

Potential to suit other applications better, e.g. safety analysis.
Connecting provers

In general, hard. Easy in this case.
Connecting provers

- high-level design
- low-level design
- detailed model of C code
- machine code as functions
- seL4 machine code

In general, hard. Easy in this case.
Connecting provers

- high-level design
- low-level design
- detailed model of C code
- machine code as functions
- machine code as functions
- seL4 machine code

In general, hard. Easy in this case.

Automatic translation of definitions from HOL4 to Isabelle/HOL
Looking back

**Success:** gcc output for -O1 and -O2 on seL4 decompiles.
Looking back

**Success:** gcc output for -O1 and -O2 on seL4 decompiles.

**However:**
- stack analysis brittle and requires expert user to debug,
- latest version avoids stack analysis,
- latest version produces graphs (instead of functions)
Looking back

**Success:** gcc output for -O1 and -O2 on seL4 decompiles.

However:
- stack analysis brittle and requires expert user to debug,
- latest version avoids stack analysis,
- latest version produces graphs (instead of functions)

A *one-fits-all decompilation* target?
- graph — good for automatic analysis/proofs
- functions — readable, good for interactive proofs
Looking back

Success: gcc output for -O1 and -O2 on seL4 decompiles.

However:

stack analysis brittle and requires expert user to debug,
latest version avoids stack analysis,
latest version produces graphs (instead of functions)

A one-fits-all decompilation target?

graph — good for automatic analysis/proofs
functions — readable, good for interactive proofs

Should decompilation be over program logic or machine model?
This talk

Part 1:
  ‣ automation: code to spec
  ‣ automation: spec to code

Part 2:
  ‣ verification of microkernel

Part 3: construction of correct code
  ‣ verified implementation of Lisp that can run Jared Davis’ Milawa
Inspiration: Lisp interpreter

Verified LISP implementations on ARM, x86 and PowerPC

Magnus O. Myreen and Michael J. C. Gordon
Computer Laboratory, University of Cambridge, UK

Abstract. This paper reports on a case study, which we believe is the first to produce a formally verified end-to-end implementation of a functional programming language running on commercial processors. Interpreters for the core of McCarthy’s LISP 1.5 were implemented in ARM, x86 and PowerPC machine code, and proved to correctly parse, evaluate and print LISP s-expressions. The proof of evaluation required working on top of verified implementations of memory allocation and garbage collection. All proofs are mechanised in the HOL4 theorem prover.
A verified Lisp interpreter

Idea: create LISP implementations via compilation.

verified code for LISP primitives car, cdr, cons, etc.

HOL4 functions for LISP parse, eval, print

compiler

decompiler

machine-code Hoare triple

ARM, x86, PowerPC code and certificate theorems

ARM  x86  PowerPC
Lisp formalised

LISP s-expressions defined as data-type SExp:

\[\begin{align*}
\text{Num} & : \mathbb{N} \rightarrow \text{SExp} \\
\text{Sym} & : \text{string} \rightarrow \text{SExp} \\
\text{Dot} & : \text{SExp} \rightarrow \text{SExp} \rightarrow \text{SExp}
\end{align*}\]

LISP primitives were defined, e.g.

\[\begin{align*}
\text{cons} \ x \ y & = \text{Dot} \ x \ y \\
\text{car} \ (\text{Dot} \ x \ y) & = x \\
\text{plus} \ (\text{Num} \ m) \ (\text{Num} \ n) & = \text{Num} \ (m + n)
\end{align*}\]

The semantics of LISP evaluation was taken to be Gordon’s formalisation of ‘LISP 1.5’-like evaluation
Extending the compiler

We define heap assertion ‘lisp (v₁, v₂, v₃, v₄, v₅, v₆, l)’ and prove implementations for primitive operations, e.g.

is_pair v₁ ⇒
{ lisp (v₁, v₂, v₃, v₄, v₅, v₆, l) * pc p }
p : E5934000
{ lisp (v₁, car v₁, v₃, v₄, v₅, v₆, l) * pc (p + 4) }

size v₁ + size v₂ + size v₃ + size v₄ + size v₅ + size v₆ < l ⇒
{ lisp (v₁, v₂, v₃, v₄, v₅, v₆, l) * pc p }
p : E50A3018 E50A4014 E50A5010 E50A600C . . .
{ lisp (cons v₁ v₂, v₂, v₃, v₄, v₅, v₆, l) * pc (p + 332) }

with these the compiler understands:

let v₂ = car v₁ in ...
let v₁ = cons v₁ v₂ in ...
Reminder

How to decompile:

1. derive Hoare triple theorems using Cambridge ARM model
2. compose Hoare triples
3. extract function

(Loops result in recursive functions.)

avg (i,j) = (i+j)>>1
Reminder

{ R0 i * R1 j * PC p }
p+0 : e0810000
{ R0 (i+j) * R1 j * PC (p+4) }

{ R0 i * PC (p+4) }
p+4 : e1a000a0
{ R0 (i >> 1) * PC (p+8) }

{ LR lr * PC (p+8) }
p+8 : e12fff1e
{ LR lr * PC lr }

{ R0 i * R1 j * LR lr * PC p }
p : e0810000 e1a000a0 e12fff1e
{ R0 ((i+j)>>1) * R1 j * LR lr * PC lr }

How to decompile:
1. derive Hoare triple theorems using Cambridge ARM model
2. compose Hoare triples
3. extract function
(Loops result in recursive functions.)

avg (i,j) = (i+j)>>1
Running the Lisp interpreter

To execute verified machine code, we:
1. wrote C wrapper around verified machine code,
2. compiled using `gcc`,
3. checked with `hexdump` that `gcc` didn’t alter the machine code,
4. ran code on real hardware: Nintendo DS lite (ARM) MacBook (x86) old MacMini (PowerPC)

Example: paper gives a definition of `pascal-triangle`, for which:

```
(pascal-triangle '(() 6)
```

returns:

```
((1 6 15 20 15 6 1)
 (1 5 10 10 5 1)
 (1 4 6 4 1)
 (1 3 3 1)
 (1 2 1)
 (1 1)
 (1))
```
Can we do better than a simple Lisp interpreter?
Two projects meet

Jared Davis
A self-verifying theorem prover

Magnus Myreen
Verified Lisp implementations

verified LISP on ARM, x86, PowerPC
Two projects meet

My theorem prover is written in Lisp. Can I try your verified Lisp?

Jared Davis
A self-verifying theorem prover

Magnus Myreen
Verified Lisp implementations

verified LISP on ARM, x86, PowerPC
Two projects meet

My theorem prover is written in Lisp. Can I try your verified Lisp?

Sure, try it.

Jared Davis
A self-verifying theorem prover

Magnus Myreen
Verified Lisp implementations

verified LISP on ARM, x86, PowerPC
Two projects meet

My theorem prover is written in Lisp. Can I try your verified Lisp?  
Sure, try it.

Does your Lisp support ..., ... and ...?

Jared Davis  
A self-verifying theorem prover

Magnus Myreen  
Verified Lisp implementations

verified LISP on ARM, x86, PowerPC
Two projects meet

My theorem prover is written in Lisp. Can I try your verified Lisp?

Sure, try it.

Does your Lisp support ..., ..., and ..., ...

No, but it could ...

Jared Davis
A self-verifying theorem prover

Magnus Myreen
Verified Lisp implementations

verified LISP on ARM, x86, PowerPC
Running Milawa

verified **LISP** on ARM, x86, PowerPC

*(TPHOLs 2009)*
Running Milawa

Milawa’s bootstrap proof:

verified LISP on ARM, x86, PowerPC
(TPHOLs 2009)
Running Milawa

Milawa’s bootstrap proof:

- 4 gigabyte proof file:
  > 500 million unique conses

verified **LISP** on
ARM, x86, PowerPC
(TPHOLs 2009)
Running Milawa

Milawa’s bootstrap proof:

- 4 gigabyte proof file: >500 million unique conses
- Takes 16 hours to run on a state-of-the-art runtime (CCL)

verified LISP on ARM, x86, PowerPC (TPHOLs 2009)
Running Milawa

Milawa’s bootstrap proof:

- 4 gigabyte proof file: >500 million unique conses
- takes 16 hours to run on a state-of-the-art runtime (CCL)

verified **LISP** on ARM, x86, PowerPC (TPHOLs 2009)

hopelessly “toy”
Running Milawa

Milawa’s bootstrap proof:

- 4 gigabyte proof file: greater than 500 million unique conses
- Takes 16 hours to run on a state-of-the-art runtime (CCL)

Contribution:
- A new verified Lisp which is able to host the Milawa thm prover
A short introduction to Milawa

• Milawa is styled after theorem provers such as NQTHM and ACL2,

• has a small trusted logical kernel similar to LCF-style provers,

• ... but does not suffer the performance hit of LCF’s fully expansive approach.
Comparison with LCF approach

FCF-style approach

• all proofs pass through the
  core’s primitive inferences
• extensions steer the core
Comparison with LCF approach

LCF-style approach

- all proofs pass through the core’s primitive inferences
- extensions steer the core
Comparison with LCF approach

**LCF-style approach**
- all proofs pass through the core’s primitive inferences
- extensions steer the core

**the Milawa approach**
- all proofs must pass the core
- the core proof checker can be replaced at runtime
Comparison with LCF approach

LCF-style approach
- all proofs pass through the core’s primitive inferences
- extensions steer the core

the Milawa approach
- all proofs must pass the core
- the core proof checker can be replaced at runtime
Comparison with LCF approach

**LCF-style approach**
- all proofs pass through the core’s primitive inferences
- extensions steer the core

**the Milawa approach**
- all proofs must pass the core
- the core proof checker can be replaced at runtime
Requirements on runtime

Milawa uses a subset of Common Lisp which is for most part first-order pure functions over natural numbers, symbols and conses,

uses primitives:  cons car cdr consp natp symbolp equal + - < symbol-< if

macros:  or and list let let* cond first second third fourth fifth

and a simple form of lambda-applications.

(Lisp subset defined on later slide.)
Requirements on runtime

...but Milawa also

- uses **destructive updates**, hash tables
- prints status messages, **timing data**
- uses Common Lisp’s **checkpoints**
- forces function **compilation**
- makes **dynamic function calls**
- can produce **runtime errors**

(Lisp subset defined on later slide.)
Requirements on runtime

...but Milawa also

- uses destructive updates, hash tables
- prints status messages, timing data
- uses Common Lisp’s checkpoints
- forces function compilation
- makes dynamic function calls
- can produce runtime errors

(Lisp subset defined on later slide.)
Requirements on runtime

...but Milawa also

- uses **destructive updates**, hash tables
- prints status messages, **timing data**
- uses Common Lisp’s **checkpoints**
- forces function **compilation**
- makes **dynamic function calls**
- can produce **runtime errors**

(Lisp subset defined on later slide.)
Runtime must scale

Designed to scale:
Runtime must scale

Designed to scale:

• just-in-time compilation for speed
  ‣ functions compile to native code
Runtime must scale

Designed to scale:

• just-in-time compilation for speed
  ‣ functions compile to native code

• target 64-bit x86 for heap capacity
  ‣ space for $2^{31}$ (2 billion) cons cells (16 GB)
Runtime must scale

Designed to scale:

- just-in-time compilation for speed
  - functions compile to native code
- target 64-bit x86 for heap capacity
  - space for $2^{31}$ (2 billion) cons cells (16 GB)
- efficient scannerless parsing + abbreviations
  - must cope with 4 gigabyte input
Runtime must scale

Designed to scale:

• just-in-time compilation for speed
  ▸ functions compile to native code

• target 64-bit x86 for heap capacity
  ▸ space for $2^{31}$ (2 billion) cons cells (16 GB)

• efficient scannerless parsing + abbreviations
  ▸ must cope with 4 gigabyte input

• graceful exits in all circumstances
  ▸ allowed to run out of space, but must report it
Workflow

1. specified input language: syntax & semantics

2. verified necessary algorithms, e.g.
   - compilation from source to bytecode
   - parsing and printing of s-expressions
   - copying garbage collection

3. proved refinements from algorithms to x86 code

4. plugged together to form read-eval-print loop
Workflow

1. specified input language: syntax & semantics

2. verified necessary algorithms, e.g.
   - compilation from source to bytecode
   - parsing and printing of s-expressions
   - copying garbage collection

3. proved refinements from algorithms to x86 code

4. plugged together to form read-eval-print loop

~30,000 lines of HOL4 scripts
AST of input language

term ::= Const sexp 
      | Var string 
      | App func (term list) 
      | If term term term 
      | LambdaApp (string list) term (term list) 
      | Or (term list) 
      | And (term list) (macro) 
      | List (term list) (macro) 
      | Let ((string \times term) list) term (macro) 
      | LetStar ((string \times term) list) term (macro) 
      | Cond ((term \times term) list) (macro) 
      | First term | Second term | Third term (macro) 
      | Fourth term | Fifth term (macro)

func ::= Define | Print | Error | Funcall 
      | PrimitiveFun primitive | Fun string 

primitive ::= Equal | Symbolp | SymbolLess 
           | Consp | Cons | Car | Cdr | 
           | Natp | Add | Sub | Less
compile: AST → bytecode list

\[
\text{bytecode} ::= \begin{align*}
\text{Pop} & \quad \text{pop one stack element} \\
\text{PopN num} & \quad \text{pop } n \text{ stack elements} \\
\text{PushVal num} & \quad \text{push a constant number} \\
\text{PushSym string} & \quad \text{push a constant symbol} \\
\text{LookupConst num} & \quad \text{push the } n\text{th constant from system state} \\
\text{Load num} & \quad \text{push the } n\text{th stack element} \\
\text{Store num} & \quad \text{overwrite the } n\text{th stack element} \\
\text{DataOp primitive} & \quad \text{add, subtract, car, cons, } \ldots \\
\text{Jump num} & \quad \text{jump to program point } n \\
\text{JumpIfNil num} & \quad \text{conditionally jump to } n \\
\text{DynamicJump} & \quad \text{jump to location given by stack top} \\
\text{Call num} & \quad \text{static function call (faster)} \\
\text{DynamicCall} & \quad \text{dynamic function call (slower)} \\
\text{Return} & \quad \text{return to calling function} \\
\text{Fail} & \quad \text{signal a runtime error} \\
\text{Print} & \quad \text{print an object to stdout} \\
\text{Compile} & \quad \text{compile a function definition}
\end{align*}
\]
How do we get just-in-time compilation?

We have verified compilation algorithm:

\[
\text{compile: AST} \rightarrow \text{bytecode list}
\]

but compiler must produce real x86 code....
How do we get just-in-time compilation?

We have verified compilation algorithm:

\[
\text{compile: } \text{AST} \rightarrow \text{bytecode list}
\]

but compiler must produce real x86 code....

Solution:

• bytecode is represented by numbers in memory that \textit{are} x86 machine code

• we prove that jumping to the memory location of the bytecode executes it
How do we get just-in-time compilation?

Treating code as data:

$$\forall p\ c\ q. \ \{p\} \ c \ \{q\} = \{p \ast \text{code } c\} \emptyset \{q \ast \text{code } c\}$$

Solution:

- bytecode is represented by numbers in memory that are x86 machine code
- we prove that jumping to the memory location of the bytecode executes it

(POPL'10)
How do we get just-in-time compilation?

Treating code as data:

$$\forall p \ c \ q. \ \{p\} \ c \ \{q\} = \{p * \text{code } c\} \emptyset \{q * \text{code } c\}$$

(POPL’10)

Definition of Hoare triple:

$$\{p\} \ c \ \{q\} = \forall s \ r. \ (p * r * \text{code } c) \ s \implies \exists n. \ (q * r * \text{code } c) \ (\text{run } n \ s)$$
I/O and efficient parsing

Jitawa implements a read-eval-print loop:

Use of external C routines adds assumptions to proof:

• reading next string from stdin
• printing null-terminated string to stdout
Read-eval-print loop

- Result of reading **lazily**, writing **eagerly**
- Eval = compile then jump-to-compiled-code
- Specification: read-eval-print until end of input

\[
\text{is_empty} \left( \text{get_input } \text{io} \right) \quad \Rightarrow \\
(k, \text{io}) \xrightarrow{\text{exec}} \text{io}
\]

\[
\neg \text{is_empty} \left( \text{get_input } \text{io} \right) \land \\
\text{next SEXP} \left( \text{get_input } \text{io} \right) = (s, \text{rest}) \land \\
(\text{sexp2term } s, [], k, \text{set_input } \text{rest } \text{io}) \xrightarrow{\text{ev}} (\text{ans}, k', \text{io}') \land \\
(k', \text{append_to_output } (\text{sexp2string } \text{ans}) \text{io}') \xrightarrow{\text{exec}} \text{io''}
\]

\[
(k, \text{io}) \xrightarrow{\text{exec}} \text{io''}
\]
Correctness theorem

Top-level correctness theorem:

\[
\{ \text{init\_state } \mathit{io} \ast \mathit{pc} \ p \ast \langle \text{terminates\_for } \mathit{io} \rangle \}
\]

\[
p : \text{code\_for\_entire\_jitawa\_implementation}
\]

\[
\{ \text{error\_message } \lor \exists \mathit{io}' . \langle ([]), \mathit{io} \rangle \xrightarrow{\text{exec}} \mathit{io}' \} \ast \text{final\_state } \mathit{io}' \}
\]
Correctness theorem:

\[
\{ \text{init\_state } io * \text{pc } p * \langle \text{terminates\_for } io \rangle \} \\
p : \text{code\_for\_entire\_jitawa\_implementation} \\
\{ \text{error\_message } \lor \exists io'. \langle \langle [], io \rangle \xrightarrow{\text{exec}} io' \rangle * \text{final\_state } io' \} 
\]
Correctness theorem

There must be enough memory and I/O assumptions must hold.

Each execution is allowed to fail with an error message.

Top-level correctness theorem:

\[
\{ \text{init	extunderscore state } \ io \ast \ pc \ p \ast \langle \text{terminates	extunderscore for } \ io \rangle \} \\
\ p : \text{code	extunderscore for	extunderscore entire	extunderscore jitawa	extunderscore implementation} \\
\{ \text{error	extunderscore message } \lor \ \exists \ io'. \langle \langle [] , \ io \rangle \xrightarrow{\text{exec}} \ io' \rangle \ast \text{final	extunderscore state } \ io' \} 
\]
Correctness theorem

There must be enough memory and I/O assumptions must hold.

Correctness theorem:

\[
\{ \text{init\_state } io \land \text{pc } p \land \langle \text{terminates\_for } io \rangle \} \\
p : \text{code\_for\_entire\_jitawa\_implementation} \\
\{ \text{error\_message } \lor \exists io'. \langle [\ ], io \rangle \xrightarrow{\text{exec}} io' \} * \text{final\_state } io'
\]

Each execution is allowed to fail with an error message.

If there is no error message, then the result is described by the high-level op. semantics.
Correctness theorem

There must be enough memory and I/O assumptions must hold.

This machine-code Hoare triple holds only for terminating executions.

\[
\{ \text{init\_state } \text{io} \neq \text{pc } p \neq \langle \text{terminates\_for } \text{io} \rangle \}
\]

\[
p : \text{code\_for\_entire\_jitawa\_implementation}
\]

\[
\{ \text{error\_message } \lor \exists \text{io}'. \langle \langle [], \text{io} \rangle \xrightarrow{\text{exec}} \text{io}' \rangle \neq \text{final\_state } \text{io}' \}
\]

Each execution is allowed to fail with an error message.

If there is no error message, then the result is described by the high-level op. semantics.
Correctness theorem

There must be enough memory and I/O assumptions must hold.

This machine-code Hoare triple holds only for terminating executions.

\[
\{ \text{init\_state } io * pc p * \langle \text{terminates\_for } io \rangle \} \\
p : \text{code\_for\_entire\_jitawa\_implementation} \\
\{ \text{error\_message } \lor \exists io'. \langle ([] , io) \xrightarrow{\text{exec}} io' \rangle * \text{final\_state } io' \} \\
\]

Each execution is allowed to fail with an error message.

If there is no error message, then the result is described by the high-level op. semantics.
$ cat verified_code.s

/* Machine code automatically extracted from a HOL4 theorem. */
/* The code consists of 7423 instructions (31840 bytes). */

.byte 0x48, 0xB, 0x5F, 0x18
.byte 0x4C, 0x8B, 0x7F, 0x10
.byte 0x48, 0x8B, 0x47, 0x20
.byte 0x48, 0x8B, 0x4F, 0x28
.byte 0x48, 0x8B, 0x57, 0x08
.byte 0x48, 0x8B, 0x37
.byte 0x4C, 0x8B, 0x47, 0x60
.byte 0x4C, 0x8B, 0x4F, 0x68
.byte 0x4C, 0x8B, 0x57, 0x58
.byte 0x48, 0x01, 0xC1
.byte 0xC7, 0x00, 0x04, 0x4E, 0x49, 0x4C
.byte 0x48, 0xC0, 0x04
.byte 0xC7, 0x00, 0x02, 0x54, 0x06, 0x51
.byte 0xC7, 0x00, 0x01, 0xC1
.byte 0xC7, 0x00, 0x04, 0x4E, 0x49, 0x4C
.byte 0x48, 0xC0, 0x04
.byte 0xC7, 0x00, 0x02, 0x54, 0x06, 0x51
.byte 0xC7, 0x00, 0x01, 0xC1
...
Running Milawa on Jitawa

Running Milawa’s 4-gigabyte bootstrap process:

- CCL 16 hours
- SBCL 22 hours
- Jitawa 128 hours (8x slower than CCL)
Running Milawa on Jitawa

Running Milawa’s 4-gigabyte bootstrap process:

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCL</td>
<td>16 hours</td>
</tr>
<tr>
<td>SBCL</td>
<td>22 hours</td>
</tr>
<tr>
<td>Jitawa</td>
<td>128 hours</td>
</tr>
</tbody>
</table>

Jitawa’s compiler performs almost no optimisations. (8x slower than CCL)
Running Milawa on Jitawa

Running Milawa’s 4-gigabyte bootstrap process:

- CCL: 16 hours
- SBCL: 22 hours
- Jitawa: 128 hours (8x slower than CCL)

Jitawa’s compiler performs almost no optimisations.

Parsing the 4 gigabyte input:

- CCL: 716 seconds (9x slower than Jitawa)
- Jitawa: 79 seconds
Looking back...

The x86 for the compile function was produced as follows:

verified compiler as function in logic

machine-code Hoare triple

verified x86

Very cumbersome....
Looking back…

The x86 for the compile function was produced as follows:

- **verified compiler** as function in logic
- **compiler**
- **decompiler**
- **machine-code Hoare triple**
  - ARM
  - x86
  - PowerPC

Very cumbersome....

...should have compiled the verified compiler using itself!
Bootstrapping the compiler

Instead: we bootstrap the verified compile function, we evaluate the compiler on a deep embedding of itself within the logic:

```
EVAL ``compile COMPILE``
```

derives a theorem:

```
compile COMPILE = compiler-as-machine-code
```

The first (?) bootstrapping of a formally verified compiler.
Bootstrapping the compiler

Instead: we bootstrap the verified compile function, we evaluate the compiler on a deep embedding of itself within the logic:

\[
\text{EVAL `\text{``compile COMPILE``}'}
\]

derives a theorem:

\[
\text{compile COMPILE} = \text{compiler-as-machine-code}
\]

The first(?) bootstrapping of a formally verified compiler.
CakeML: A Verified Implementation of ML

Ramana Kumar * 1  Magnus O. Myreen † 1  Michael Norrish 2  Scott Owens 3

1 Computer Laboratory, University of Cambridge, UK
2 Canberra Research Lab, NICTA, Australia
3 School of Computing, University of Kent, UK

Abstract

We have developed and mechanically verified an ML system called CakeML, which supports a substantial subset of Standard ML. CakeML is implemented as an interactive read-eval-print loop in x86-64 machine code. Our correctness theorem ensures that the CakeML REPL implementation prints only those results permitted by the semantics of CakeML. Our verification effort touches on precision arithmetic, and compiler bootstrapping.

1. Introduction

The last decade has seen a strong interest in verified compilation: and there have been significant, high-profile results, many based on the CompCert compiler for C [1, 14, 16, 29]. This interest is easy to justify: in the context of program verification, an unverified compiler forms a large and complex part of the trusted computing base. However, to our knowledge, none of the existing work on compilers for general-purpose languages has addressed all the dimensions: one, the compilation algorithm as implemented in machine code.
This talk

**Part 1:** my approach (PhD work)
- automation: code to spec
- automation: spec to code

**Part 2:** verification of existing code
- verification of gcc output for microkernel (7,000 lines of C)

**Part 3:** construction of correct code
- verified implementation of Lisp that can run Jared Davis’ Milawa
Summary

Techniques from my PhD

- automation: code to spec
- automation: spec to code

worked for two non-trivial case studies:

- verification of gcc output for microkernel (7,000 lines of C)
- verified implementation of Lisp that can run Jared Davis’ Milawa
Summary

Techniques from my PhD

- automation: code to spec
- automation: spec to code

worked for two non-trivial case studies:

- verification of gcc output for microkernel (7,000 lines of C)
- verified implementation of Lisp that can run Jared Davis’ Milawa

Lessons were learnt:
Summary

Techniques from my PhD

- automation: code to spec
- automation: spec to code

worked for two non-trivial case studies:

- verification of gcc output for microkernel (7,000 lines of C)
- verified implementation of Lisp that can run Jared Davis’ Milawa

Lessons were learnt:

- decompiler shouldn’t try to be smart (stack)
Summary

Techniques from my PhD

› automation: code to spec
› automation: spec to code

worked for two non-trivial case studies:

› verification of gcc output for microkernel (7,000 lines of C)
› verified implementation of Lisp that can run Jared Davis’ Milawa

Lessons were learnt:

› decompiler shouldn’t try to be smart (stack)
› compile the verified compiler with itself!
Summary

Techniques from my PhD

- automation: code to spec
- automation: spec to code

worked for two non-trivial case studies:

- verification of gcc output for microkernel (7,000 lines of C)
- verified implementation of Lisp that can run Jared Davis’ Milawa

Lessons were learnt:

- decompiler shouldn’t try to be smart (stack)
- compile the verified compiler with itself!