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 CHALMERS 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

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 01AFFFFFFB

is impossible to read, write or maintain manually.
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)
› 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.
During my PhD, I developed the following infrastructure:

- **Compiler**
- **Decompiler**
- **Machine-code Hoare triple**
- **ARM**
- **x86**
- **PowerPC**

...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 $\rightarrow$ HOL4) of Leroy’s PowerPC model
- instruction decoder added
Hoare triples

Each model can be evaluated, e.g. ARM instruction\n\texttt{add r0,r0,r0} is described by theorem:\n
\[
\vdash (\text{ARM\_READ\_MEM } ((31 >> 2) (\text{ARM\_READ\_REG 15w state})) \text{ state } = 0xE0800000w) \land \neg \text{state.undefined} \Rightarrow \\
(\text{NEXT\_ARM\_MMU } \text{cp state } = \\
\text{ARM\_WRITE\_REG 15w } (\text{ARM\_READ\_REG 15w state } + 4w) \\
(\text{ARM\_WRITE\_REG 0w} \\
(\text{ARM\_READ\_REG 0w state } + \text{ARM\_READ\_REG 0w state}) \text{ state}))
\]
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)

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)
```
Decompilation, correct?

Decompiler automatically proves a certificate theorem:

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

\[ \{ (R0, R1, M) \text{ is } (r_0, r_1, m) \ast \text{PC } p \ast S \} \]

\[ p : \text{E3A00000 E3510000 12800001 15911000 1AFFFFFFB} \]

\[ \{ (R0, R1, M) \text{ is } f(r_0, r_1, m) \ast \text{PC } (p + 20) \ast S \} \]

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 $f$,

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

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

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

\[
\begin{align*}
\{ & R0 \ i \ * \ R1 \ j \ * \ PC \ p \} \\
\text{p+0} : \\
\{ & R0 \ (i+j) \ * \ R1 \ j \ * \ PC \ (p+4) \} \\
\{ & R0 \ i \ * \ PC \ (p+4) \} \\
\text{p+4} : \\
\{ & R0 \ (i >> 1) \ * \ PC \ (p+8) \} \\
\{ & LR \ lr \ * \ PC \ lr \} \\
\text{p+8} : \\
\{ & LR \ lr \ * \ PC \ lr \} \\
\{ & R0 \ i \ * \ R1 \ j \ * \ LR \ lr \ * \ PC \ lr \} \\
\text{p} : e0810000 \ e1a000a0 \ e12ffff1e \\
\{ & R0 \ ((i+j)>>1) \ * \ R1 \ j \ * \ LR \ lr \ * \ PC \ lr \} \\
\end{align*}
\]

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

- **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:
      - \( f(r0, r1, m) = \) 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 \( g(r0, r1, m) \)

- **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

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

- **Verification**
  - 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:

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>E351000A</td>
<td>L: cmp r1,#10</td>
</tr>
<tr>
<td>2241100A</td>
<td>subcs r1,r1,#10</td>
</tr>
<tr>
<td>2AFFFFFFFC</td>
<td>bcs L</td>
</tr>
</tbody>
</table>

and automatically proves a certificate HOL theorem:

$$\vdash \{ R1 \ r_1 * PC \ p * s \}$$
$$p : E351000A \ 2241100A \ 2AFFFFFFFC$$
$$\{ R1 \ f(r_1) * PC \ (p+12) * s \}$$
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:

\[ \begin{align*}
\vdash & \{ \text{R1 } r_1 \star \text{PC } p \star \text{s} \} \\
\vdash & \{ \text{R1 } (r_1 \mod 10) \star \text{PC } (p+12) \star \text{s} \}
\end{align*} \]

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]
During my PhD, I developed the following infrastructure:

- decompiler
- compiler
- machine-code Hoare triple

...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

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
- Cambridge ARM model

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

existing L4.verified work

high-level design

low-level design

detailed model of C code

Haskell prototype

real C code

Aim: remove need to trust C compiler and C semantics
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

- refinement proof
- decompilation
- gcc (not trusted)

new extension
existing L4.verified work
Approach

• decompilation by me

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

- Cambridge ARM model
- seL4 machine code
- machine code as functions
Decompile 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
```

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
  ✓ decompilation is compositional

• compiled using gcc -O2
  ✓ gcc implements ARM/C calling convention

• must be compatible with L4.verified proof
  ➡ stack requires special treatment
Some arguments are passed on the stack, and cause memory ops in machine code that are not present in C semantics.

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:

```
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

Use separation-logic inspired approach

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

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.

Additional benefit:
- Does not require temp space, works for “any n”
- States explicitly stack shape/usage:
  { stack sp n (s0::s1::s2::s3::s) * ... * PC p } 
  p : code 
  { stack sp n (s0::s1::s2::s3::s) * ... * PC lr } 

Promises to leave stack unchanged
Other C-specifs

• **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
Refinement proof
(Work by Thomas Sewell, NICTA)

detailed model of C code

\[ \downarrow \text{proof by rewriting} \]

C code as graph

\[ \uparrow \text{‘guided SMT proof’} \]

machine code as graph

\[ \downarrow \text{translation (unproved)} \]

machine code as functions
Graph language

machine code as graph

\[ \text{translation (unproved)} \]

machine code as functions

\[ \text{automatic decompilation} \]

seL4 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

\[ \uparrow \]\n
automatic decompilation

seL4 machine code

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

In Isabelle/HOL
in HOL4
new extension
existing L4.verified work

automatic translation of definitions from HOL4 to Isabelle/HOL
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

TPHOLs’09

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:

- Num : \( \mathbb{N} \rightarrow \text{SExp} \)
- Sym : string \( \rightarrow \text{SExp} \)
- Dot : SExp \( \rightarrow \text{SExp} \rightarrow \text{SExp} \)

LISP primitives were defined, e.g.

- \( \text{cons} \ x \ y = \text{Dot} \ x \ y \)
- \( \text{car} \ (\text{Dot} \ x \ y) = x \)
- \( \text{plus} \ (\text{Num} \ m) \ (\text{Num} \ n) = \text{Num} \ (m + n) \)

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 (v_1, v_2, v_3, v_4, v_5, v_6, l)’ and prove implementations for primitive operations, e.g.

\[
\text{is\_pair } v_1 \Rightarrow \\
\{ \text{lisp } (v_1, v_2, v_3, v_4, v_5, v_6, l) * \text{pc } p \} \\
p : \text{E5934000} \\
\{ \text{lisp } (v_1, \text{car } v_1, v_3, v_4, v_5, v_6, l) * \text{pc } (p + 4) \}
\]

\[
\text{size } v_1 + \text{size } v_2 + \text{size } v_3 + \text{size } v_4 + \text{size } v_5 + \text{size } v_6 < l \Rightarrow \\
\{ \text{lisp } (v_1, v_2, v_3, v_4, v_5, v_6, l) * \text{pc } p \} \\
p : \text{E50A3018 E50A4014 E50A5010 E50A600C . . .} \\
\{ \text{lisp } (\text{cons } v_1 v_2, v_2, v_3, v_4, v_5, v_6, l) * \text{pc } (p + 332) \}
\]

with these the compiler understands:

\[
\text{let } v_2 = \text{car } v_1 \text{ in } . . . \\
\text{let } v_1 = \text{cons } v_1 v_2 \text{ in } . . .
\]
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

We change these triples to be about lisp heap. Result: more abstraction.
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:

\[
(pascal-triangle '((1)) '6)
\]

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

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

Sure, try it.

My theorem prover is written in Lisp. 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

Milawa
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)

Contribution:

- a new verified Lisp which is able to host the Milawa thm prover

Verified LISP on ARM, x86, PowerPC

Jitawa: verified LISP with JIT compiler

(TPHOLs 2009)
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

**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

---

work by Jared Davis
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.)
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

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

#### Term

\[
\text{term} ::= \begin{align*}
\text{Const}\ sexp &\mid \text{Var}\ string &\mid \text{App}\ func\ (\text{term}\ list) \\
\text{If}\ term\ term\ term &\mid \text{LambdaApp}\ (\text{string}\ list)\ term\ (\text{term}\ list) \\
\text{Or}\ (\text{term}\ list) &\mid \text{And}\ (\text{term}\ list) &\text{(macro)} \\
\text{List}\ (\text{term}\ list) &\text{(macro)} \\
\text{Let}\ ((\text{string}\times\text{term})\ list)\ term &\text{(macro)} \\
\text{LetStar}\ ((\text{string}\times\text{term})\ list)\ term &\text{(macro)} \\
\text{Cond}\ ((\text{term}\times\text{term})\ list) &\text{(macro)} \\
\text{First}\ term &\mid \text{Second}\ term &\mid \text{Third}\ term &\text{(macro)} \\
\text{Fourth}\ term &\mid \text{Fifth}\ term &\text{(macro)} \\
\end{align*}
\]

#### Sexp

\[
\text{sexp} ::= \begin{align*}
\text{Val}\ num &\mid \text{Sym}\ string &\mid \text{Dot}\ sexp\ sexp \\
\end{align*}
\]

#### Func

\[
\text{func} ::= \begin{align*}
\text{Define} &\mid \text{Print} &\mid \text{Error} &\mid \text{Funcall} \\
\text{PrimitiveFun}\ primitive &\mid \text{Fun}\ string \\
\end{align*}
\]

#### Primitive

\[
\text{primitive} ::= \begin{align*}
\text{Equal} &\mid \text{Symbolp} &\mid \text{SymbolLess} \\
\text{Consp} &\mid \text{Cons} &\mid \text{Car} &\mid \text{Cdr} \\
\text{Natp} &\mid \text{Add} &\mid \text{Sub} &\mid \text{Less} \\
\end{align*}
\]
### compile: AST $\rightarrow$ bytecode list

<table>
<thead>
<tr>
<th>bytecode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pop</td>
<td>pop one stack element</td>
</tr>
<tr>
<td>PopN num</td>
<td>pop $n$ stack elements</td>
</tr>
<tr>
<td>PushVal num</td>
<td>push a constant number</td>
</tr>
<tr>
<td>PushSym string</td>
<td>push a constant symbol</td>
</tr>
<tr>
<td>LookupConst num</td>
<td>push the $n$th constant from system state</td>
</tr>
<tr>
<td>Load num</td>
<td>push the $n$th stack element</td>
</tr>
<tr>
<td>Store num</td>
<td>overwrite the $n$th stack element</td>
</tr>
<tr>
<td>DataOp primitive</td>
<td>add, subtract, car, cons, ...</td>
</tr>
<tr>
<td>Jump num</td>
<td>jump to program point $n$</td>
</tr>
<tr>
<td>JumpIfNil num</td>
<td>conditionally jump to $n$</td>
</tr>
<tr>
<td>DynamicJump</td>
<td>jump to location given by stack top</td>
</tr>
<tr>
<td>Call num</td>
<td>static function call (faster)</td>
</tr>
<tr>
<td>DynamicCall</td>
<td>dynamic function call (slower)</td>
</tr>
<tr>
<td>Return</td>
<td>return to calling function</td>
</tr>
<tr>
<td>Fail</td>
<td>signal a runtime error</td>
</tr>
<tr>
<td>Print</td>
<td>print an object to stdout</td>
</tr>
<tr>
<td>Compile</td>
<td>compile a function definition</td>
</tr>
</tbody>
</table>
How do we get just-in-time compilation?

Treating code as data:

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

(POPL’10)

Definition of Hoare triple:

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

\[
\begin{align*}
is\text{\_empty} (\text{get\_input } \text{io}) & \quad (k, io) \xrightarrow{\text{exec}} io \\
\neg \text{is\_empty (get\_input } \text{io}) \wedge \\
\text{next\_sexp (get\_input } \text{io})) = (s, rest) \wedge \\
(\text{sexp2term } s, [], k, \text{set\_input rest } \text{io}) & \xrightarrow{\text{ev}} (\text{ans}, k', io') \wedge \\
(k', \text{append\_to\_output } (\text{sexp2string } \text{ans}) \text{ io'}) & \xrightarrow{\text{exec}} io'' \\
\end{align*}
\]
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 \ast pc \ p \ast \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' \} \ast \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.

list of numbers
$ cat verified_code.s

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

.byte 0x48, 0x8B, 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 C7, 0x00, 0x04, 0x4E, 0x49, 0x4C
.byte 0x48, 0xC0, 0x04
.byte 0xC7, 0x00, 0x04, 0x49, 0x4C
.byte 0x48, 0xC0, 0x04
.byte 0xC7, 0x00, 0x04, 0x49, 0x4C
.byte 0x48, 0xC0, 0x04
.byte 0xC7, 0x00, 0x04, 0x49, 0x4C
.byte 0x48, 0xC0, 0x04
.byte 0xC7, 0x00, 0x04, 0x49, 0x4C
.byte 0x48, 0xC0, 0x04
.byte 0xC7, 0x00, 0x04, 0x49, 0x4C
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:

The verified compiler was produced as a function in logic, which was then compiled into a machine-code Hoare triple. This triple was then decompiled into ARM, x86, and PowerPC code.

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:

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

derives a theorem:

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

The first (?) bootstrapping of a formally verified compiler.
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 that this REPL implementation prints only those results permitted by the semantics of CakeML. Our verification effort touches on incremental and dynamic compilation, garbage collection, arbitrary-precision arithmetic, and compiler bootstrapping.

Contributions

• Compiler verification; compiler bootstrapping; ML; 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 of these dimensions: one, the compilation algorithm for converting a program from a source string to a list of numbers representing machine code, and two, the execution of that algorithm as implemented in machine code.

1. Introduction

The last decade has seen a strong interest in verified compilation;
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

Lessons were learnt:

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

Questions?