Machine code, formal verification and functional programming

Magnus Myreen
University of Cambridge, UK

LOLA 2013, New Orleans
Machine Code

Machine code is what the CPU executes.

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

Ultimately all program verification ought to reach real machine code.
Machine code

Machine code,

```
E1510002 B0422001 C0411002 01AFFFFFB
```

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.
Some C problems avoided

Machine code verification avoids some challenges in C verification:

- C has annoyingly weak type system, e.g. 
  `union` and cast to/from `void` type

- multiple ambiguities in both syntax and semantics, e.g.
  C syntax preprocessing `cpp`, evaluation orders

- richer set of features compared to plain machine instructions,
  `mdContext->in[mdi++] = *inBuf++`
  in-line assembly in C: `__asm__( ... )`, semantics?

Also, verified C code must be compiled, while verified machine code can be executed ‘as is’.
Verification of Machine Code

Challenges:

Contribution: tools/methods which
• expose as little as possible of the big models to the user
• makes non-automatic proofs independent of the models
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.

Short demo
Talk outline

Part I: Tools and infrastructure

proof-producing decompiler:
  translates machine code into equivalent functions in logic

proof-producing compiler:
  translates functions in logic into correct-by-cons. machine code

Part 2: Case studies

simple verified Lisp interpreter

verified just-in-time compiler for Lisp

verified read-eval-print-loop for a subset of Standard ML
Infrastructure in HOL4

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.
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:

\[
|- (\text{ARM\_READ\_MEM } ((31 >> 2) \ (\text{ARM\_READ\_REG 15w state})) \ state = 0\text{xE0800000w}) \land \neg \text{state.undefined} \Rightarrow
\]
\[
(\text{NEXT\_ARM\_MMU cp state =}
\]
\[
\text{ARM\_WRITE\_REG 15w (ARM\_READ\_REG 15w state + 4w)}
\]
\[
(\text{ARM\_WRITE\_REG 0w}
\]
\[
(\text{ARM\_READ\_REG 0w state + ARM\_READ\_REG 0w state) state})
\]

As a total-correctness machine-code Hoare triple:

\[
|- \text{SPEC ARM\_MODEL}
\]
\[
(aR 0w x \star aPC p)
\]
\[
\{(p,0\text{xE08000000w})\}
\]
\[
(aR 0w (x+x) \star aPC (p+4w))
\]
{ R0 \{ x+x \} * PC (p+4) }

Informal syntax for this talk:

\[
\{ R0 \times \star PC p \}
\]
\[
p : E08000000
\]

Short demo
Definition of Hoare triple

\[ \{ p \} \ c \ \{ q \} \iff \forall s \ r. \ (p \ast r \ast \text{code } c) \ s \implies \exists n. \ (q \ast r \ast \text{code } c) \ (\text{run } n \ s) \]
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: 1AFFFFFFFB bne L

the decompiler automatically extracts a readable function:

\[
  f(r_0, r_1, m) = \text{let } r_0 = 0 \text{ in } g(r_0, r_1, m) \\
  g(r_0, r_1, m) = \text{if } r_1 = 0 \text{ then } (r_0, r_1, m) \text{ else } \text{let } r_0 = r_0 + 1 \text{ in } \text{let } r_1 = m(r_1) \text{ in } g(r_0, r_1, m)
\]
Decompilation, correct?

Decompiler automatically proves a certificate theorem:

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

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.
De.compilation 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.

Proof reuse: Given similar x86 and PowerPC code:

\[
31C085F67405408B36EBF7
\]

\[
38A000002C140000408200107E80A02E38A500014BFFFFF0
\]

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

Decompilation algorithm:

Step 1: evaluate underlying ISA model
(prove Hoare triples for each instruction)

Step 2: construct CFG and find ‘decompilation rounds’
(usually one round per loop)

Step 3: for each round, compose a Hoare triple theorem:

\[
\begin{align*}
\{ \text{pre}[v_0 \ldots v_n] \} \\
\text{code} \\
\{ \text{let } (v'_0 \ldots v'_n) = f(v_0 \ldots v_n) \text{ in } \text{post}[v'_0 \ldots v'_n] \}\end{align*}
\]

if the code contains a loop, apply a loop rule
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].
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:

$$\vdash \{ R1 \; r_1 \times PC \; p \times s \}$$

$$p : E351000A \; 2241100A \; 2AFFFFFFFC$$

$$\{ R1 \; f(r_1) \times PC \; (p+12) \times s \}$$
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:

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

\[ p : E351000A \ 2241100A \ 2AFFFFFFC \]

\[ \{R1 \ (r_1 \mod 10) \ast \ PC \ (p+12) \ast s\} \]

**Additional feature:** the compiler can use the above theorem to extend its input language with: let $r_1 = r_1 \mod 10$ in _
User-defined extensions

Using our theorem about \( \text{mod} \), the compiler accepts:

\[
g(r_1, r_2, r_3) = \begin{align*}
&\text{let } r_1 = r_1 + r_2 \text{ in} \\
&\text{let } r_1 = r_1 + r_3 \text{ in} \\
&\text{let } r_1 = r_1 \text{ mod } 10 \text{ in} \\
&(r_1, r_2, r_3)
\end{align*}
\]

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

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

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]
Talk outline

**Part I**: Tools and infrastructure

proof-producing **decompiler**: translates machine code into equivalent functions in logic

proof-producing **compiler**: translates functions in logic into correct-by-cons. machine code

**Part 2**: Case studies

simple verified **Lisp interpreter**

verified **just-in-time compiler** for Lisp

verified read-eval-print-loop for a subset of **Standard ML**
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, next slide...
Defined using three mutually recursive relations $\rightarrow_{\text{eval}}$, $\rightarrow_{\text{app}}$ and $\rightarrow_{\text{eval\_list}}$.

\[
\begin{align*}
\text{ok\_name } v & : (v, \rho) \rightarrow_{\text{eval}} \rho(v) \\
(p, \rho) \rightarrow_{\text{eval}} \text{nil} \land ([g], \rho) \rightarrow_{\text{eval}} s & : (\langle p \rightarrow e; g \rangle, \rho) \rightarrow_{\text{eval}} s \\
\text{can\_apply } k \text{ args} & : (k, \text{args}, \rho) \rightarrow_{\text{app}} k \text{ args} \\
(e, \rho[\text{args/vars}]) \rightarrow_{\text{eval}} s & : (\lambda[[\text{vars}; e]], \text{args}, \rho) \rightarrow_{\text{app}} s \\
([], \rho) \rightarrow_{\text{eval\_list}} [] & : (\langle [], \rho \rangle) \rightarrow_{\text{eval\_list}} [] \\
\end{align*}
\]

Here $c$, $v$, $k$ and $f$ range over value constants, value variables, function constants and function variables, respectively.
Extending the compiler

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

```plaintext
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:

```plaintext
let v₂ = car v₁ in ...
let v₁ = cons v₁ v₂ in ...
```

Short demo
Verified Lisp interpreters

Evaluation.
1. the compiler was extended with code for car, cons, plus, etc.
2. lisp_eval defined as tail-rec function, for which we proved:

   \[ \forall s \, r. \quad s \xrightarrow{\text{eval}} r \quad \Rightarrow \quad \text{fst} \left( \text{lisp}\_\text{eval} \left( s, \text{nil}, \text{nil}, \text{nil}, \text{nil}, \text{nil}, 1 \right) \right) = r \]

3. the compiler automatically produced correct implementations.

Parsing/printing.
1. high-level definitions parsing/printing functions were defined,

   \[ \forall s. \quad \text{sexp}\_\text{ok} \, s \quad \Rightarrow \quad \text{string2sexp} \left( \text{sexp2string} \, s \right) = s \]

2. low-level definitions were compiled to machine code,
3. manual proof related high- and low-level definitions.
Correctness theorem

The result is an interpreter which parses, evaluates and prints LISP:

\[ \forall s, r, l, p. \]
\[ s \xrightarrow{eval} r \land \text{sexp\_ok } s \land \text{lisp\_eval\_pre}(s, l) \Rightarrow \]
\{ \exists a. \text{R3 } a \ast \text{string } a \ast \text{sexp\_2\_string } s \ast \text{space } s \ast \text{l} \ast \text{pc } p \}

\[ p : \ldots \text{ machine code not shown } \ldots \]
\{ \exists a. \text{R3 } a \ast \text{string } a \ast \text{sexp\_2\_string } r \ast \text{space\' } s \ast \text{l} \ast \text{pc } (p+8968) \}

where:

- \[ s \xrightarrow{eval} r \] is "s evaluates to r in Gordon’s semantics"
- \text{sexp\_ok } s is “s contains no bad symbols”
- \text{lisp\_eval\_pre}(s, l) is “s can be evaluated with heap limit l”
- \text{string } a \text{ str} is “string str is stored in memory at address a”
- \text{space } s \text{ l} is “there is enough memory to setup heap of size l”
Running the Lisp interpreter

Nintendo DS lite (ARM)   MacBook (x86)   old MacMini (PowerPC)

```
(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))
```
Next: 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.

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

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

Jitawa verified LISP on ARM, x86, PowerPC

(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 can be reflectively extended 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
Example of semantics for macros:

\[
\begin{align*}
\text{(App (PrimitiveFun Car) } x,\ env, k, io) & \xrightarrow{ev} (ans, env', k', io') \\
\text{(First } x,\ env, k, io) & \xrightarrow{ev} (ans, env', k', io')
\end{align*}
\]

\text{List (term list) } \quad \text{(macro)}
\text{Let ((string }\times\text{ term) list) term } \quad \text{(macro)}
\text{LetStar ((string }\times\text{ term) list) term } \quad \text{(macro)}
\text{Cond ((term }\times\text{ term) list) } \quad \text{(macro)}
\text{First term | Second term | Third term } \quad \text{(macro)}
\text{Fourth term | Fifth term } \quad \text{(macro)}

\text{func } ::= \text{ Define | Print | Error | Funcall}
\text{ | PrimitiveFun primitive | Fun string}

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

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

An efficient **s-expression parser** (and **printer**) is proved, which deals with abbreviations:

```
(append (cons (cons a b) c)
       (cons (cons a b) c))

(append #1=(cons (cons a b) c)
       #1#)
```
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

\[
\begin{align*}
\text{is\_empty (get\_input } & \text{ io)} \\
\hline
(k, io) & \xrightarrow{\text{exec}} io
\end{align*}
\]

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

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

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

Each execution is allowed to fail with an error message.

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

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 0xC7, 0x00, 0x04, 0x4E, 0x49, 0x4C
.byte 0x48, 0xC0, 0x04
.byte 0xC7, 0x00, 0x02, 0x54, 0x06, 0x51
.byte 0x48, 0x01, 0xC1
.byte 0xC7, 0x00, 0x04, 0x4E, 0x49, 0x4C
.byte 0x48, 0xC0, 0x04
.byte 0xC7, 0x00, 0x02, 0x54, 0x06, 0x51
.byte 0x48, 0xC0, 0x04
...
A short demo:

Jitawa — a verified runtime for Milawa
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
Next: can we do better than Lisp?

(on going work)
The CakeML project

Aim: to do the same for a subset of Standard ML:

- produce verified read-eval-print-loop for ML
- construct a proved-to-be-sound version of HOL light
- synthesise hardware that runs ML programs on ‘bare metal’

Collaborators:

Scott Owens – semantics, type/module systems
Ramana Kumar – compiler verification
Michael Norrish – parsing, general HOL expertise
David Greaves – hardware, FPGAs
Implementation of ML compiler

How to produce compile component?

verified ML-to-x86 compiler as logic function

compiler

decompiler

machine-code Hoare triple

ARM  x86  PowerPC

Very cumbersome....
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
```

We believe this is the first bootstrapping of a formally verified compiler.
Summary

Part 1: Tools and infrastructure

proof-producing decompiler:
  translates machine code into equivalent functions in logic

proof-producing compiler:
  translates functions in logic into correct-by-cons. machine code

Part 2: Case studies

simple verified Lisp interpreter

verified just-in-time compiler for Lisp

verified read-eval-print-loop for a subset of Standard ML