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 01AFFFFB`

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:

machine code

code

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

correctness

{P} code {Q}

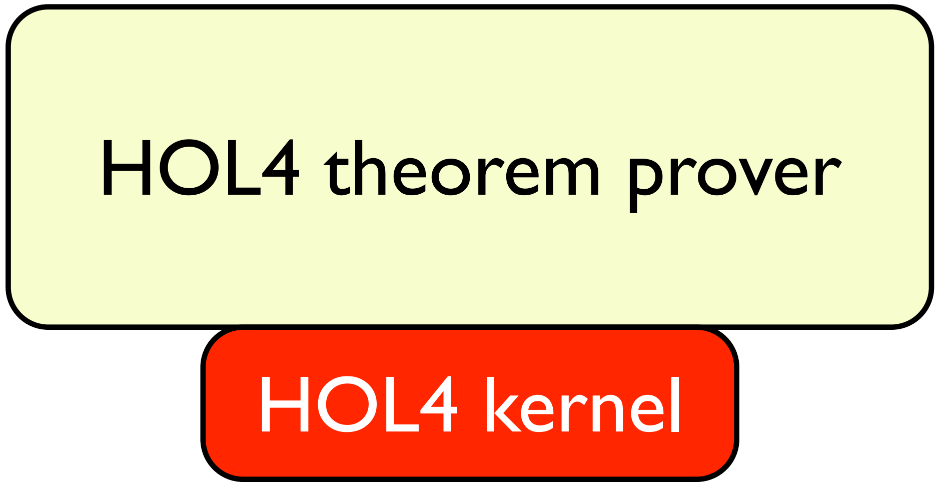
Contribution: tools/methods which

- 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



HOL4 theorem prover

HOL4 kernel

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

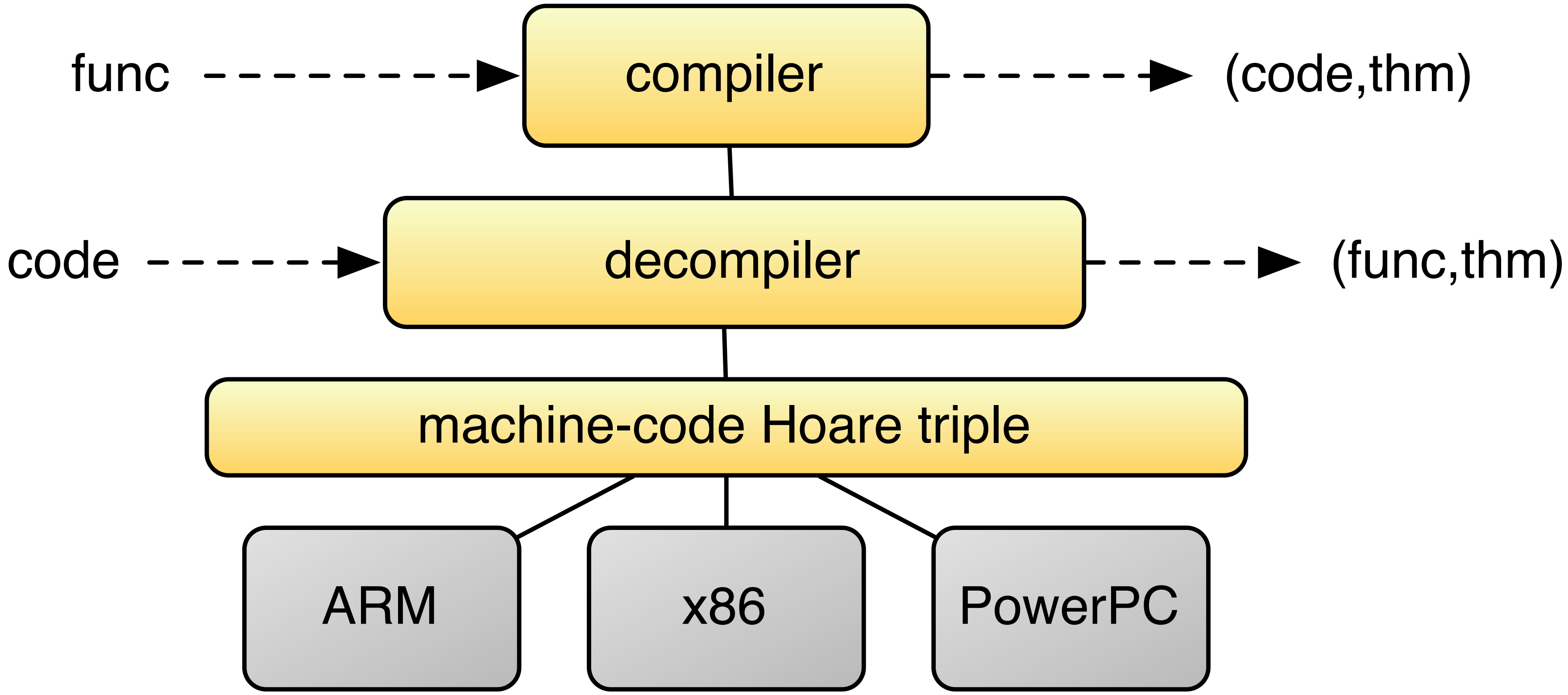
simple verified Lisp interpreter

verified just-in-time compiler for Lisp

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

Infrastructure

During my PhD, I developed the following infrastructure:



. . . 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 `add r0,r0,r0` is described by theorem:

```
|- (ARM_READ_MEM ((31 >< 2) (ARM_READ_REG 15w state)) state =
    0xE0800000w) ∧ ¬state.undefined ⇒
   (NEXT_ARM_MMU cp state =
     ARM_WRITE_REG 15w (ARM_READ_REG 15w state + 4w)
     (ARM_WRITE_REG 0w
      (ARM_READ_REG 0w state + ARM_READ_REG 0w state) state))
```

As a total-correctness machine-code Hoare triple:

```
|- SPEC ARM_MODEL
     (aR 0w x * aPC p)
     {(p,0xE0800000w)}
     (aR 0w (x+x) * aPC (p+4w))
```

Informal syntax for this talk:

$\{\,\mathrm{R0}\;x * \mathrm{PC}\;p\,\}$

p : E0800000

$\{\,\mathrm{R0}\;(x{+}x) * \mathrm{PC}\;(p{+}4)\,\}$

Short demo

Definition of Hoare triple

frame

code separate

$$\begin{aligned}\{p\}\ c\ \{q\} \quad \iff \quad \forall s\ r.\ \ & (p * r * \textsf{code}\ c)\ s \implies \\ & \exists n.\ (q * r * \textsf{code}\ c)\ (\textsf{run}\ n\ s)\end{aligned}$$

total correctness

machine code sem.

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:

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

Decompilation, correct?

Decompiler automatically proves a certificate theorem:

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

$$\{\,(\mathrm{R0}, \mathrm{R1}, \mathrm{M})\ \mathrm{is}\ (r_0, r_1, m) * \mathrm{PC}\ p * \mathrm{S}\,\}$$

p : `E3A00000 E3510000 12800001 15911000 1AFFFFFB`

$$\{\,(\mathrm{R0}, \mathrm{R1}, \mathrm{M})\ \mathrm{is}\ f(r_0, r_1, m) * \mathrm{PC}\ (p + 20) * \mathrm{S}\,\}$$

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,

$$\begin{array}{lcl}\forall x\ l\ a\ m.\ \ list(l,a,m) & \Rightarrow & f(x,a,m) = (length(l),0,m) \\ \forall x\ l\ a\ m.\ \ list(l,a,m) & \Rightarrow & f_{pre}(x,a,m)\end{array}$$

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:

$$\{pre[v_0 \ldots v_n]\}$$
$$code$$
$$\{\text{let } (v_0' \ldots v_n') = f(v_0 \ldots v_n) \text{ in } post[v_0' \ldots v_n']\}$$

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:

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

with termination and side-conditions H collected as:

$$\mathit{pre}(x) = (\text{if } G(x) \text{ then } \mathit{pre}(F(x)) \text{ else true}) \wedge H(x)$$

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
2AFFFFFC         bcs L
```

and automatically proves a certificate HOL theorem:

$$\begin{array}{l} \vdash \;\; \{\,\mathrm{R1}\ {\color{red}r_1} * \mathrm{PC}\ p * \mathrm{s}\,\} \\ \qquad p : \mathtt{E351000A\ 2241100A\ 2AFFFFFC} \\ \qquad \{\,\mathrm{R1}\ {\color{red}f(r_1)} * \mathrm{PC}\ (p{+}12) * \mathrm{s}\,\} \end{array}$$

Compilation, example cont.

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

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

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

$$\begin{array}{ll} \vdash & \{\mathrm{R1}\ r_1 * \mathrm{PC}\ p * \mathrm{s}\} \\ & \ p : \texttt{E351000A 2241100A 2AFFFFFC} \\ & \{\mathrm{R1}\ (r_1\ \mathrm{mod}\ 10) * \mathrm{PC}\ (p{+}12) * \mathrm{s}\} \end{array}$$

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 mod, the compiler accepts:

$$\begin{aligned} g(r_1, r_2, r_3) \;=\; & \text{let } r_1 = r_1 + r_2 \text{ in} \\ & \text{let } r_1 = r_1 + r_3 \text{ in} \\ & \text{let } r_1 = r_1 \text{ mod } 10 \text{ in} \\ & \quad (r_1, r_2, r_3) \end{aligned}$$

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

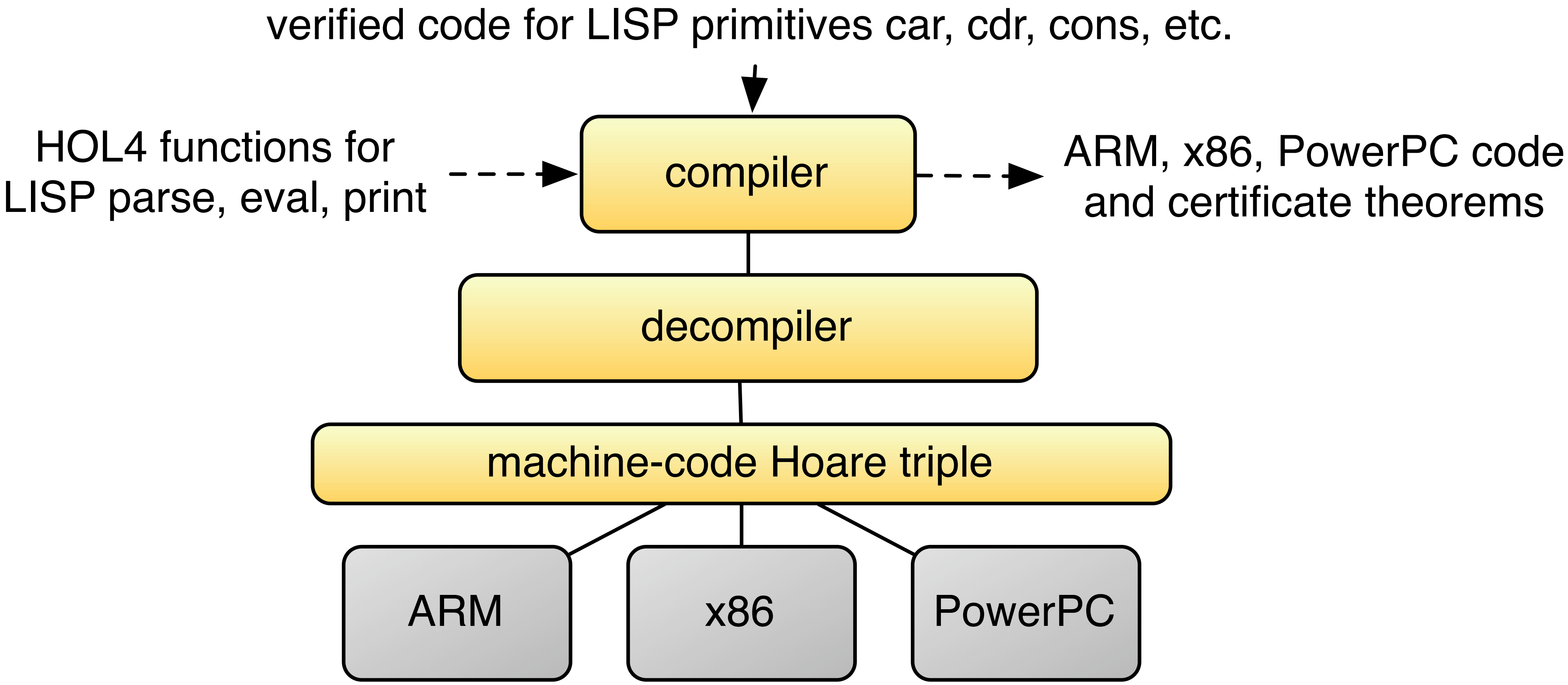
simple verified Lisp interpreter

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

ARM, x86, PowerPC code
and certificate theorems

decompiler

machine-code Hoare triple

ARM

x86

PowerPC

Lisp formalised

LISP s-expressions defined as data-type SExp:

$$\begin{aligned} \mathsf{Num} &: \mathbb{N} \to \mathsf{SExp} \\ \mathsf{Sym} &: \mathsf{string} \to \mathsf{SExp} \\ \mathsf{Dot} &: \mathsf{SExp} \to \mathsf{SExp} \to \mathsf{SExp} \end{aligned}$$

LISP primitives were defined, e.g.

$$\begin{aligned} \mathsf{cons}\ x\ y &= \mathsf{Dot}\ x\ y \\ \mathsf{car}\ (\mathsf{Dot}\ x\ y) &= x \\ \mathsf{plus}\ (\mathsf{Num}\ m)\ (\mathsf{Num}\ n) &= \mathsf{Num}\ (m + n) \end{aligned}$$

The semantics of LISP evaluation was taken to be Gordon's formalisation of 'LISP 1.5'-like evaluation, next slide. . .

Gordon's Lisp semantics from ACL2 workshop 2007

Defined using three mutually recursive relations $\to_{\mathit{eval}}$, $\to_{\mathit{app}}$ and $\to_{\mathit{eval\_list}}$.

$$\frac{\mathit{ok\_name}\ v}{(v, \rho) \to_{\mathit{eval}} \rho(v)} \qquad \frac{}{(c, \rho) \to_{\mathit{eval}} c} \qquad \frac{}{([\,], \rho) \to_{\mathit{eval}} \texttt{nil}}$$

$$\frac{(p, \rho) \to_{\mathit{eval}} \texttt{nil} \wedge ([\mathit{gl}], \rho) \to_{\mathit{eval}} s}{([p \to e; \mathit{gl}], \rho) \to_{\mathit{eval}} s} \qquad \frac{(p, \rho) \to_{\mathit{eval}} x \wedge x \neq \texttt{nil} \wedge (e, \rho) \to_{\mathit{eval}} s}{([p \to e; \mathit{gl}], \rho) \to_{\mathit{eval}} s}$$

$$\frac{\mathit{can\_apply}\ k\ \mathit{args}}{(k, \mathit{args}, \rho) \to_{\mathit{app}} k\ \mathit{args}} \qquad \frac{(\rho(f), \mathit{args}, \rho) \to_{\mathit{app}} s \wedge \mathit{ok\_name}\ f}{(f, \mathit{args}, \rho) \to_{\mathit{app}} s}$$

$$\frac{(e, \rho[\mathit{args}/\mathit{vars}]) \to_{\mathit{eval}} s}{(\lambda[[\mathit{vars}]; e], \mathit{args}, \rho) \to_{\mathit{app}} s} \qquad \frac{(\mathit{fn}, \mathit{args}, \rho[\mathit{fn}/x]) \to_{\mathit{app}} s}{(\mathit{label}[[x]; \mathit{fn}], \mathit{args}, \rho) \to_{\mathit{app}} s}$$

$$\frac{}{([\,], \rho) \to_{\mathit{eval\_list}} [\,]} \qquad \frac{(e, \rho) \to_{\mathit{eval}} s \wedge ([\mathit{el}], \rho) \to_{\mathit{eval\_list}} \mathit{sl}}{([e; \mathit{el}], \rho) \to_{\mathit{eval\_list}} [s; \mathit{sl}]}$$

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

$$
\begin{array}{l}
\text{is\_pair } v_1 \;\Rightarrow \\
\{\ \text{lisp } (v_1, v_2, v_3, v_4, v_5, v_6, l) * \text{pc } p\ \} \\
\ p : \texttt{E5934000} \\
\{\ \text{lisp } (v_1, \text{car } v_1, v_3, v_4, v_5, v_6, l) * \text{pc } (p + 4)\ \}
\end{array}
$$

$$
\begin{array}{l}
\text{size } v_1 + \text{size } v_2 + \text{size } v_3 + \text{size } v_4 + \text{size } v_5 + \text{size } v_6 < l \;\Rightarrow \\
\{\ \text{lisp } (v_1, v_2, v_3, v_4, v_5, v_6, l) * \text{pc } p\ \} \\
\ p : \texttt{E50A3018 E50A4014 E50A5010 E50A600C ...} \\
\{\ \text{lisp } (\text{cons } v_1\ v_2, v_2, v_3, v_4, v_5, v_6, l) * \text{pc } (p + 332)\ \}
\end{array}
$$

with these the compiler understands:

$$
\begin{array}{l}
\text{let } v_2 = \text{car } v_1 \text{ in } \ldots \\
\text{let } v_1 = \text{cons } v_1\ v_2 \text{ in } \ldots
\end{array}
$$

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:

$$\forall s\ r.\quad s \longrightarrow_{eval} r \quad \Rightarrow \quad \mathrm{fst}\ (\mathrm{lisp\_eval}\ (s, \mathrm{nil}, \mathrm{nil}, \mathrm{nil}, \mathrm{nil}, \mathrm{nil}, l)) = r$$

3. the compiler automatically produced correct implementations.

Parsing/printing.

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

$$\forall s.\quad \mathrm{sexp\_ok}\ s \ \Rightarrow\ \mathrm{string2sexp}\ (\mathrm{sexp2string}\ s) = s$$

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:

$$
\begin{array}{l}
\forall s\ r\ l\ p. \\
\quad s \longrightarrow_{eval} r \ \wedge\ \text{sexp\_ok}\ s \ \wedge\ \text{lisp\_eval\_pre}(s, l) \Rightarrow \\
\quad \{\ \exists a.\ \text{R3}\ a * \text{string}\ a\ (\text{sexp2string}\ s) * \text{space}\ s\ l * \text{pc}\ p\ \} \\
\quad\ p : \ldots\ \text{machine code not shown}\ \ldots \\
\quad \{\ \exists a.\ \text{R3}\ a * \text{string}\ a\ (\text{sexp2string}\ r) * \text{space}'\ s\ l * \text{pc}\ (p{+}8968)\ \}
\end{array}
$$

where:

$s \longrightarrow_{eval} r$	is	“s evaluates to r in Gordon’s semantics”
sexp_ok s	is	“s contains no bad symbols”
lisp_eval_pre(s, l)	is	“s can be evaluated with heap limit l”
string a str	is	“string str is stored in memory at address a”
space s 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

Milawa

Magnus Myreen

Verified Lisp implementations

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

Running Milawa

Milawa

Jitawa *verified* **LISP** **LISP**

ARM, x86, PowerPC *with JIT compiler*

(TPHOLs 2009)

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: hopelessly "toy"

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

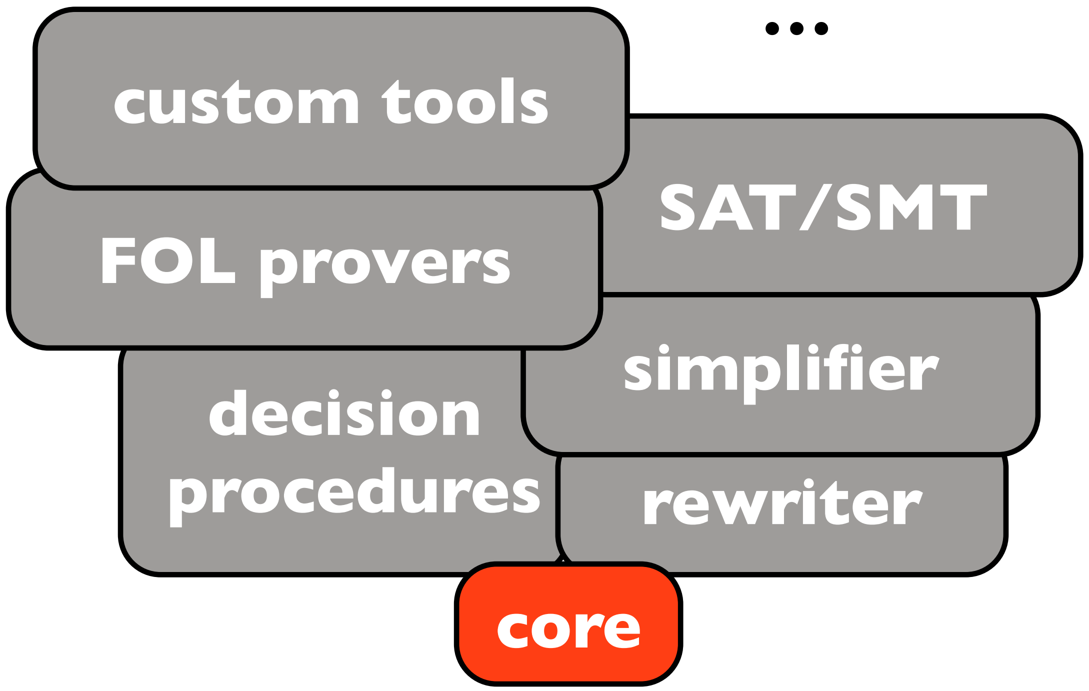
work by Jared Davis

A short introdution 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.

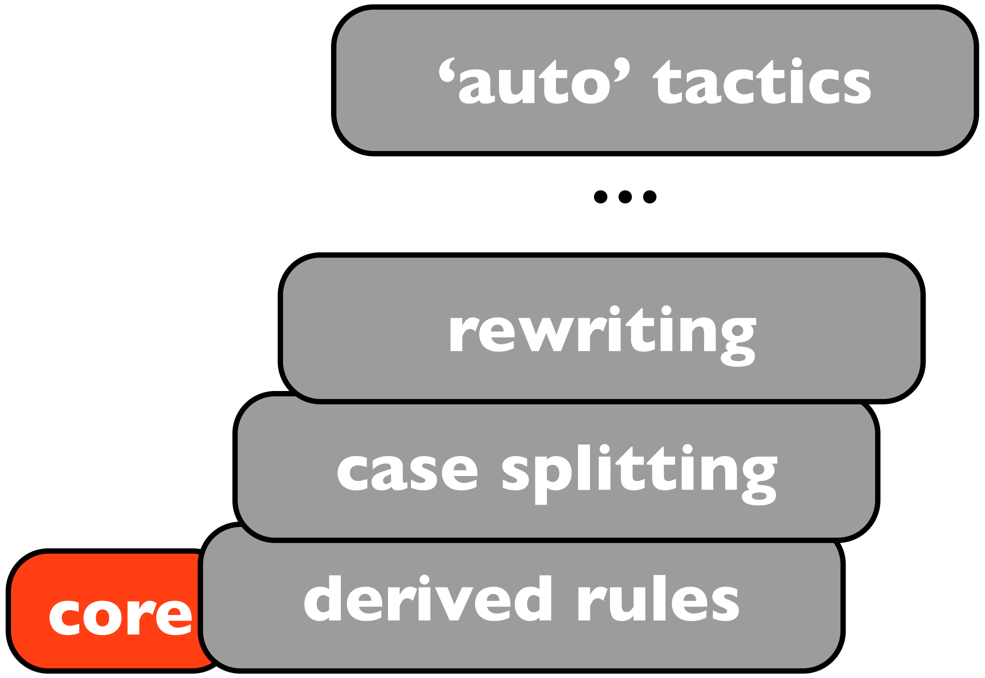
work by Jared Davis

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.)

work by Jared Davis

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

} not necessary

} runtime must support

(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

~30,000 lines of HOL4 scripts

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

AST of input language

Example of semantics for macros:

$$\frac{(\text{App }(\text{PrimitiveFun Car})\ [x], env, k, io) \xrightarrow{\text{ev}} (ans, env', k', io')}{(\text{First } x, env, k, io) \xrightarrow{\text{ev}} (ans, env', k', io')}$$

	$\mid$	List (*term* list)	(macro)
	$\mid$	Let ((*string* × *term*) list) *term*	(macro)
	$\mid$	LetStar ((*string* × *term*) list) *term*	(macro)
	$\mid$	Cond ((*term* × *term*) list)	(macro)
	$\mid$	First *term* $\mid$ Second *term* $\mid$ Third *term*	(macro)
	$\mid$	Fourth *term* $\mid$ Fifth *term*	(macro)
func	::=	Define $\mid$ Print $\mid$ Error $\mid$ Funcall	
	$\mid$	PrimitiveFun *primitive* $\mid$ Fun *string*	
primitive	::=	Equal $\mid$ Symbolp $\mid$ SymbolLess	
	$\mid$	Consp $\mid$ Cons $\mid$ Car $\mid$ Cdr $\mid$	
	$\mid$	Natp $\mid$ Add $\mid$ Sub $\mid$ Less	

compile: AST ⟶ bytecode list

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

How do we get just-in-time compilation?

Treating code as data:

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

(POPL'10)

Definition of Hoare triple:

$$\begin{aligned}\{p\}\ c\ \{q\}\ \ =\ \ \forall s\ r.\ \ & (p * r * \mathsf{code}\ c)\ s\ \Longrightarrow \\ & \exists n.\ (q * r * \mathsf{code}\ c)\ (\mathsf{run}\ n\ s)\end{aligned}$$

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

$$\frac{\text{is\_empty}\ (\text{get\_input}\ io)}{(k, io) \xrightarrow{\text{exec}} io}$$

$$\frac{\begin{array}{l} \neg\text{is\_empty}\ (\text{get\_input}\ io) \wedge \\ \text{next\_sexp}\ (\text{get\_input}\ io)) = (s, rest) \wedge \\ (\text{sexp2term}\ s, [], k, \text{set\_input}\ rest\ \ io) \xrightarrow{\text{ev}} (ans, k', io') \wedge \\ (k', \text{append\_to\_output}\ (\text{sexp2string}\ ans)\ \ io') \xrightarrow{\text{exec}} io'' \end{array}}{(k, io) \xrightarrow{\text{exec}} io''}$$

Correctness theorem

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

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

ne

$$\{\,\mathsf{init\_state}\ io * \mathsf{pc}\ p * \langle \mathsf{terminates\_for}\ io \rangle\,\}$$

$$p : \mathsf{code\_for\_entire\_jitawa\_implementation}$$

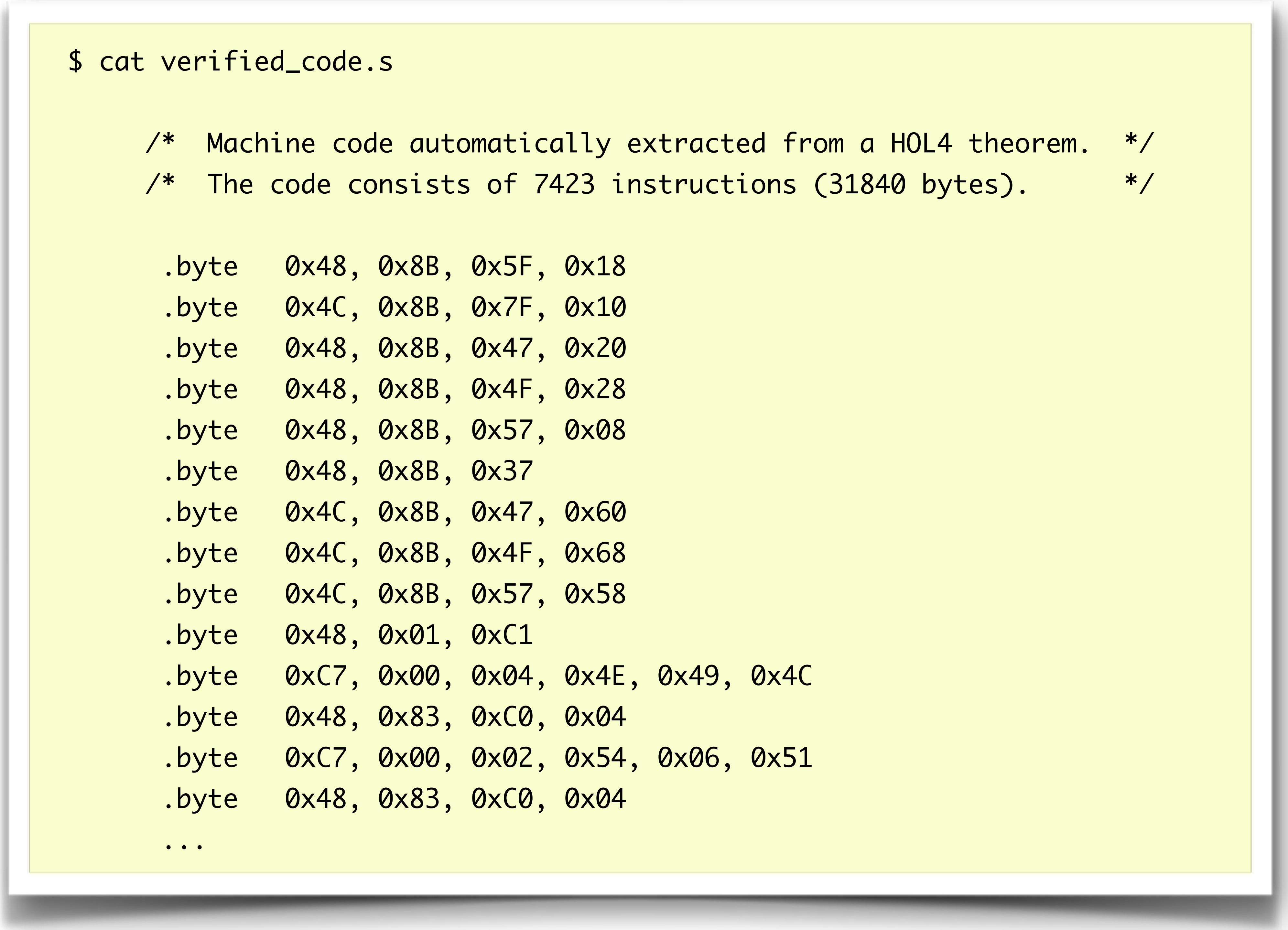
list of numbers

$$\{\,\mathsf{error\_message} \lor \exists io'.\ \langle ([], io) \xrightarrow{\mathsf{exec}} io' \rangle * \mathsf{final\_state}\ io'\,\}$$

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.

Verified code



```
$ 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, 0x83, 0xC0, 0x04
      .byte   0xC7, 0x00, 0x02, 0x54, 0x06, 0x51
      .byte   0x48, 0x83, 0xC0, 0x04
      ...
```

A short demo:

Jitawa — a verified runtime for Milawa

Running Milawa on Jitawa

Running Milawa's 4-gigabyte booststrap 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:

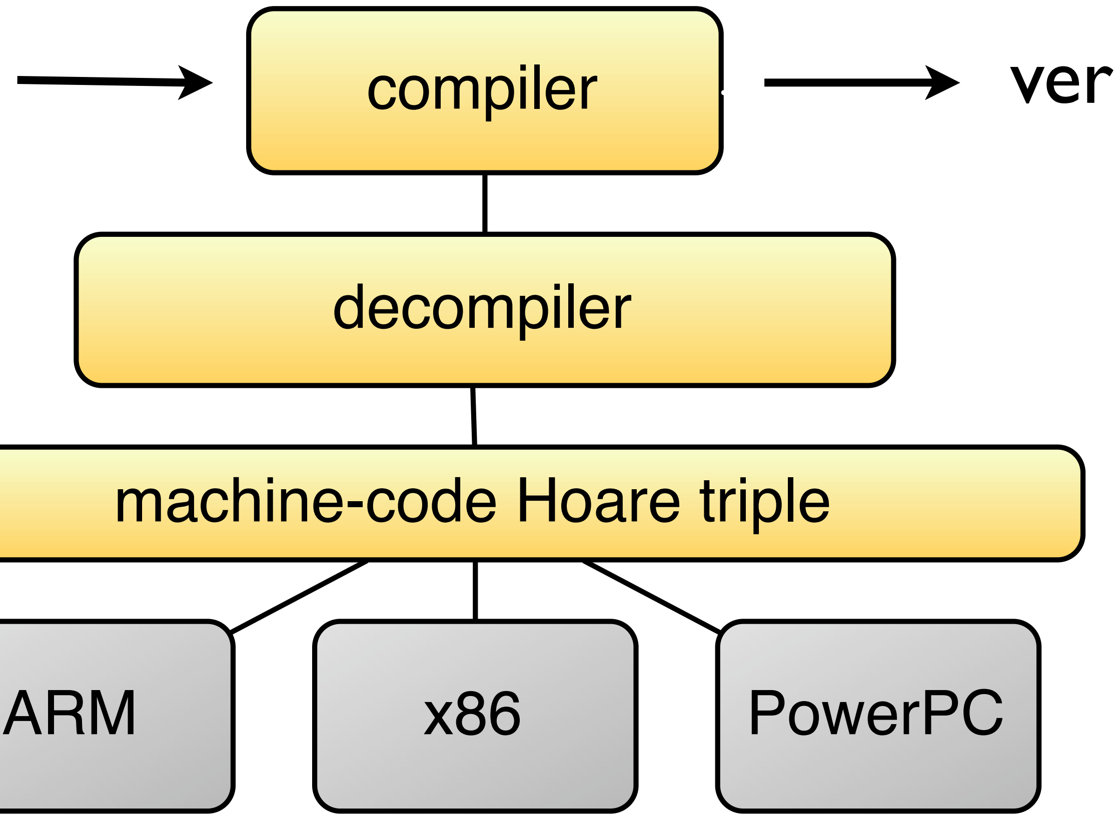
- produce verified read-eval-print-loop for ML
- construct a proved-to-be-sound version of HOL light
- synthesise hardware that runs ML programs on 'bare metal'

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

verified x86

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