A Verified Lisp Implementation for A Verified Theorem Prover

Scheme workshop 2016, Nara, Japan

Magnus O. Myreen — University of Cambridge, but now at Chalmers University of Technology
Jared Davis — Centaur Technology, Inc., but now at Apple

Result:

A Verified Lisp Implementation for A Verified Theorem Prover

Claim:

The most comprehensive proof-based evidence of a theorem prover's soundness to date.

This talk: The Journey

2005:

I'm a PhD student working on verification of machine code (factorial, length of a linked list)

???

???

???

Theme: exploring how to make verification scale.

Result:

A Verified Lisp Implementation for A Verified Theorem Prover

???

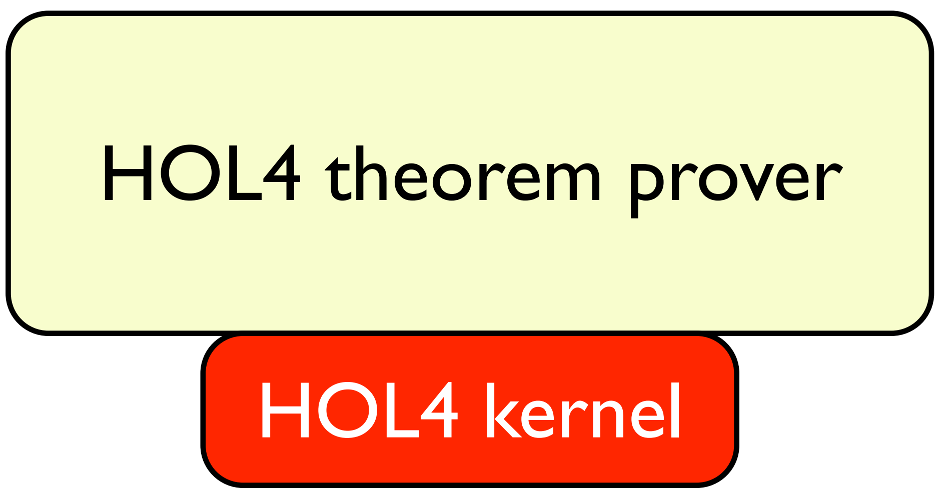
The start:

I'm a PhD student working on verification of machine code (factorial, length of a linked list)

Context: interactive theorem proving

Aim: 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.

Context: interactive theorem proving

photo idea: Larry Paulsson

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.

Challenges of Machine Code

machine code

code

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

correctness

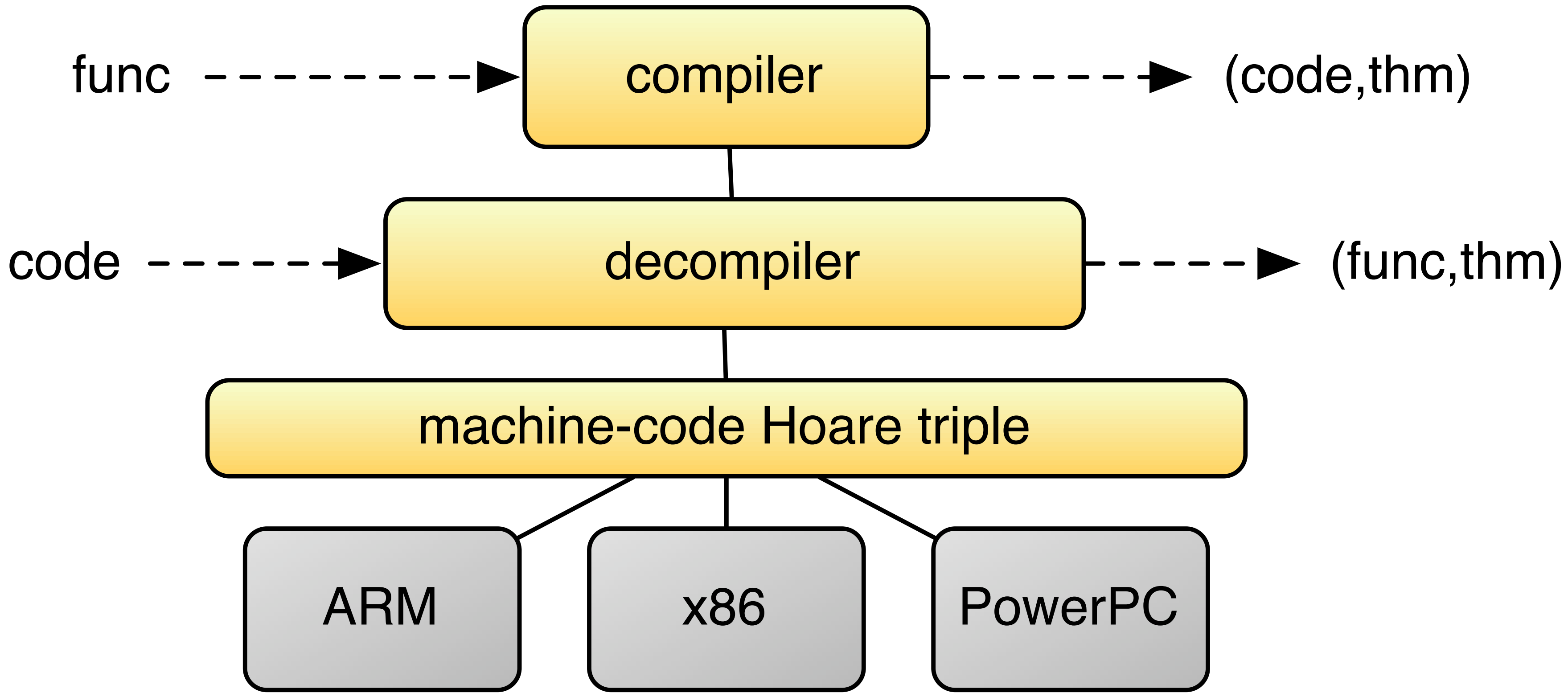
{P} code {Q}

Challenges:

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

Infrastructure

During my PhD, I developed the following infrastructure:



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

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

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

Definition of Hoare triple

separating conjunction

frame

code separate

$$\{p\}\ c\ \{q\} \quad \Longleftrightarrow \quad \forall s\ r.\ \ (p * r * \mathsf{code}\ c)\ s \implies \exists n.\ (q * r * \mathsf{code}\ c)\ (\mathsf{run}\ n\ s)$$

total correctness

machine code sem.

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:

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

Decompilation, correct?

Decompiler automatically proves a certificate theorem:

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

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,

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

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

Proof reuse: Given similar x86 and PowerPC code:

```
31C085F67405408B36EBF7
```

```
38A000002C14000408200107E80A02E38A500014BFFFFF0
```

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

Decompilation how to

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

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

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

2

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

3

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

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
2. compose Hoare triples
3. extract function

(Loops result in recursive functions.)

Decompiler cont.

Implementation:

- ML program which fully automatically performs forward proof
- no heuristics and no dangling proof obligations
- loops result in tail-recursive functions

Case studies:

- verified copying garbage collector
- bignum library routines

Part 2:

I want more automation and abstraction!

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
2AFFFFFC         bcs L
```

and automatically proves a certificate HOL theorem:

$$\vdash \;\{\,\mathsf{R1}\ r_1 * \mathsf{PC}\ p * \mathsf{s}\,\}$$
$$p : \texttt{E351000A 2241100A 2AFFFFFC}$$
$$\{\,\mathsf{R1}\ f(r_1) * \mathsf{PC}\ (p{+}12) * \mathsf{s}\,\}$$

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:

$$\vdash \; \{\mathrm{R1}\ r_1 * \mathrm{PC}\ p * \mathrm{s}\}$$
$$p : \texttt{E351000A 2241100A 2AFFFFFC}$$
$$\{\mathrm{R1}\ (r_1 \bmod 10) * \mathrm{PC}\ (p{+}12) * \mathrm{s}\}$$

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

Implementation

To compile function f:

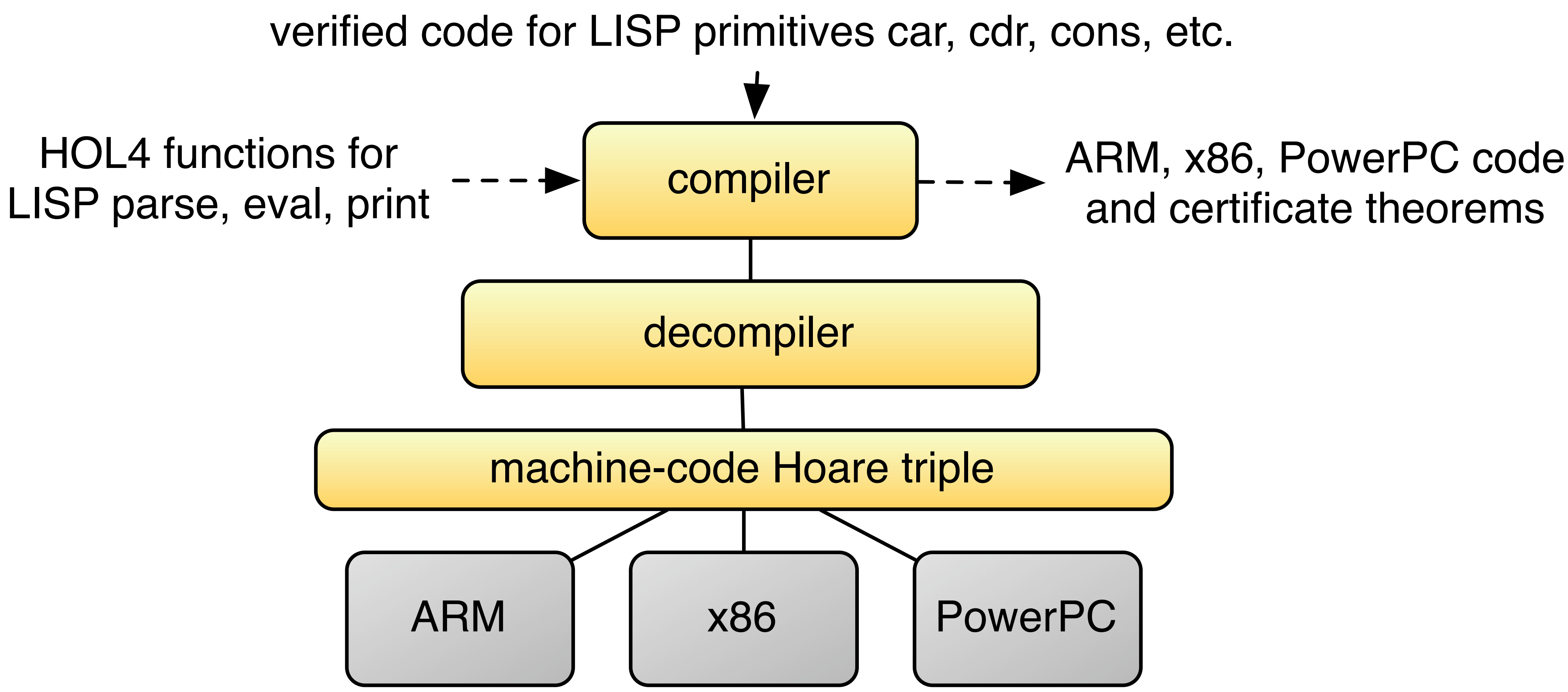
1. generate, without proof, code from input f;
2. decompile, with proof, a function f' from generated code;
3. prove $f = f'$.

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

Infrastructure again

Idea: create LISP implementations via compilation.



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

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.

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

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

with these the compiler understands:

$$\text{let } v_2 = \text{car } v_1 \text{ in } \ldots$$
$$\text{let } v_1 = \text{cons } v_1\ v_2 \text{ in } \ldots$$

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 }

2

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

3

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

How to decompile:

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

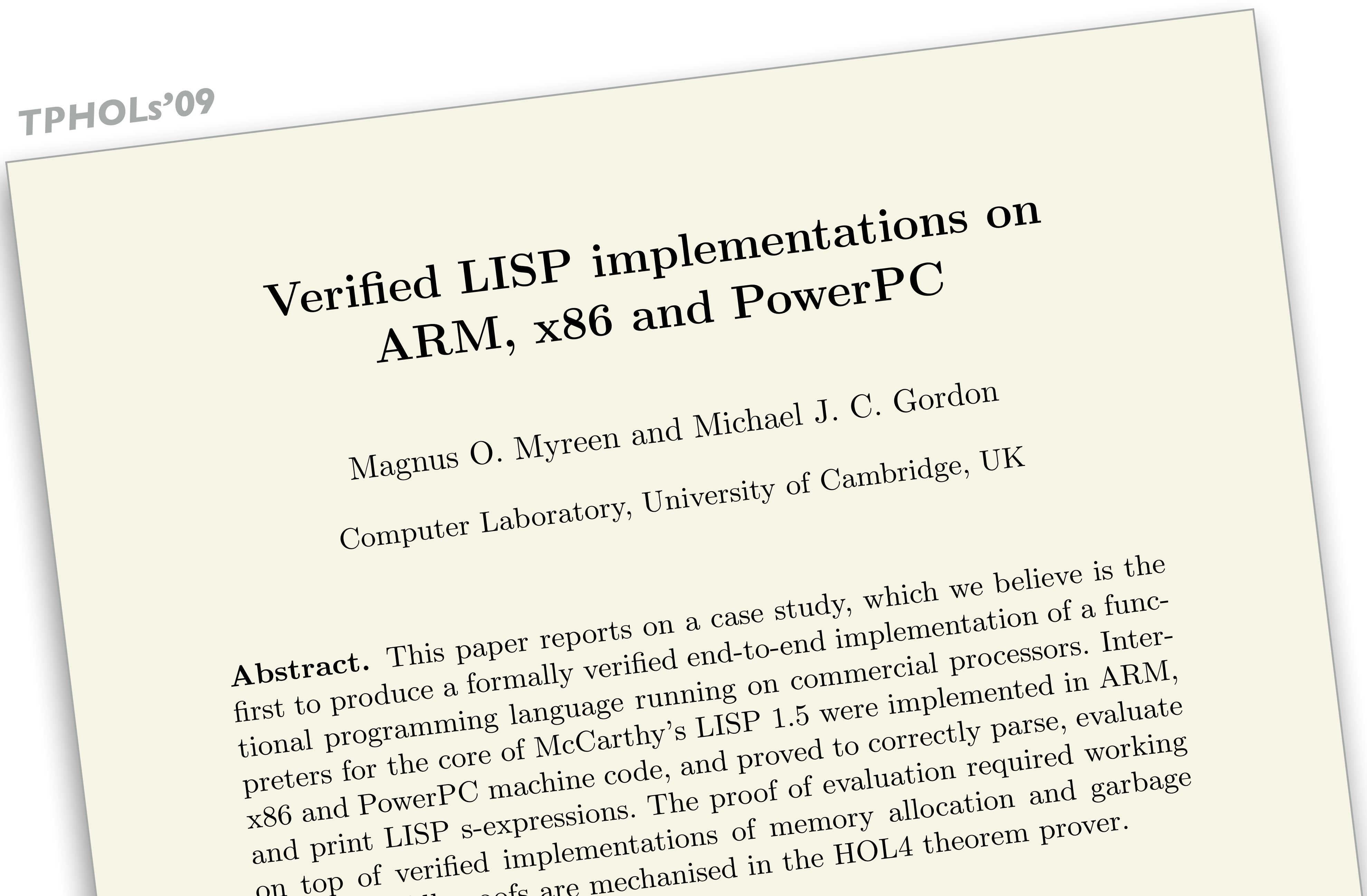
`e12fff1e bx lr`

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

(Loops result in recursive functions.)

The final case study of my PhD

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 ... ofs are mechanised in the HOL4 theorem prover.

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))
```

Part 3:

A sudden need for a serious Lisp implementation.

Two projects meet

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

Umm.. sure!

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

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

hopelessly "toy"

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)

Running Milawa

Milawa

Jitawa: *verified* **LISP**
with JIT compiler

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)

Result:

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

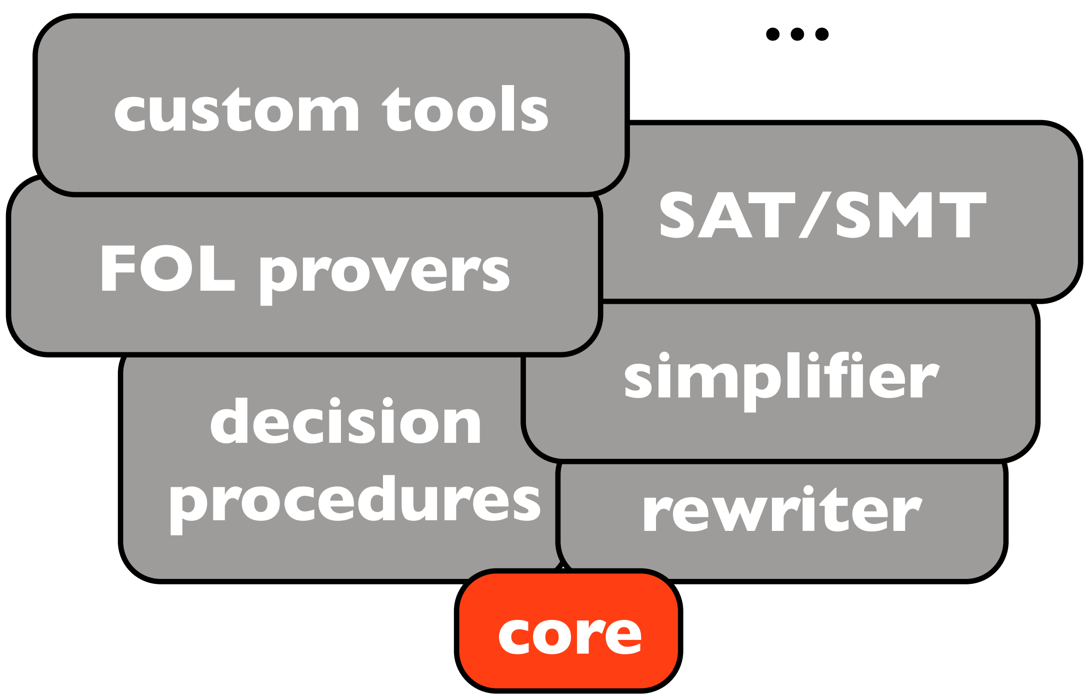
work by Jared Davis

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.

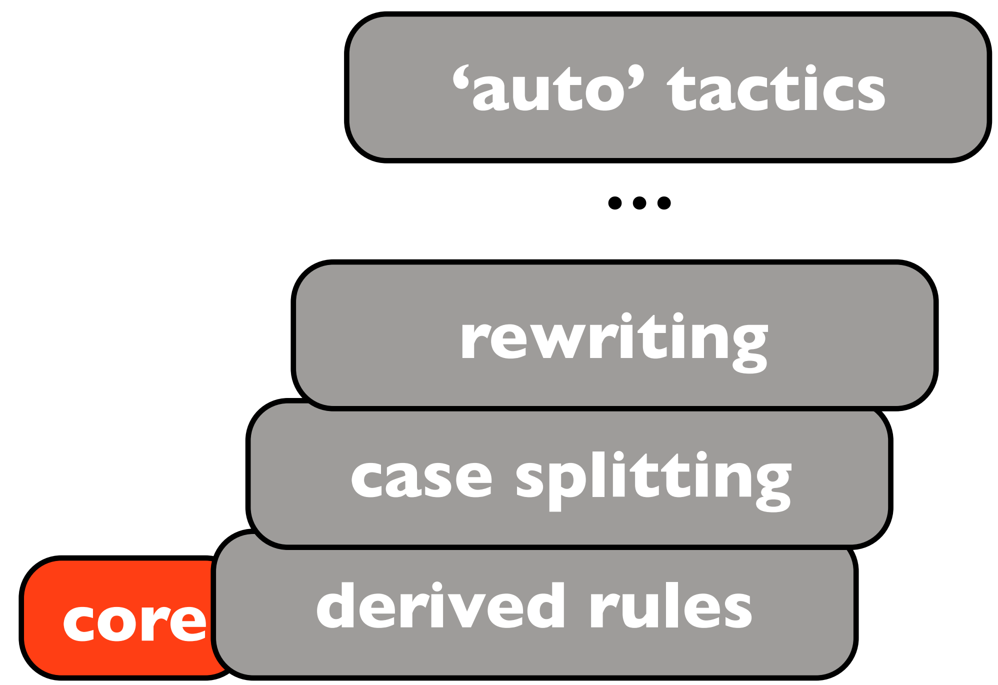
work by Jared Davis

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 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.)

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~~

} not necessary

- forces function compilation
- makes dynamic function calls
- can produce runtime errors

} 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

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* × *term*) list) *term*	(macro)
	|	LetStar ((*string* × *term*) list) *term*	(macro)
	|	Cond ((*term* × *term*) list)	(macro)
	|	First *term* | Second *term* | Third *term*	(macro)
	|	Fourth *term* | Fifth *term*	(macro)

sexp	::=	Val *num*
	|	Sym *string*
	|	Dot *sexp* *sexp*

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

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:

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

Definition of Hoare triple:

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

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

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

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

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

Correctness theorem

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

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

ne

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

list of numbers

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
     ...
```

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.

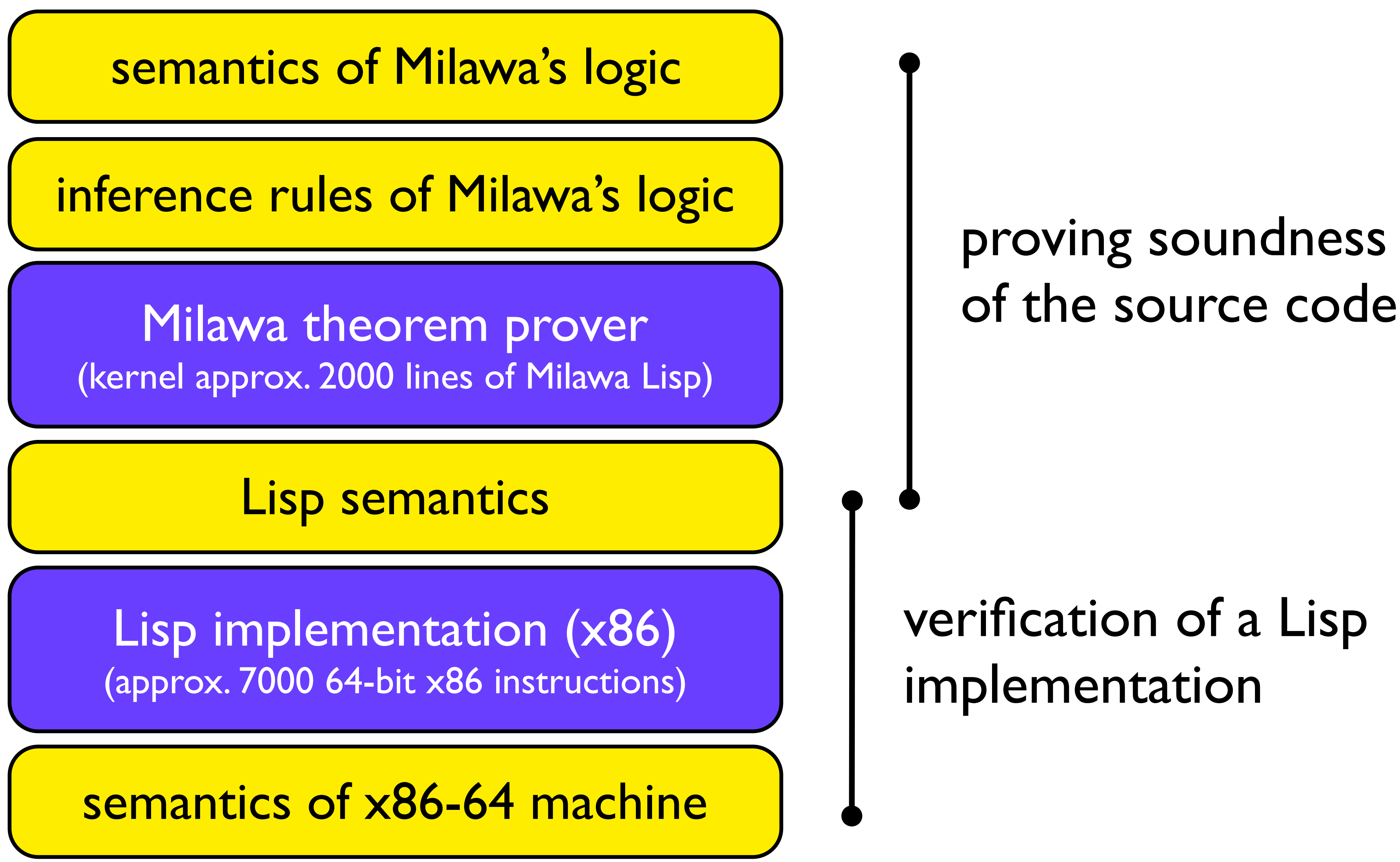
Parsing the 4 gigabyte input:

CCL	716 seconds	(9x slower than Jitawa)
Jitawa	79 seconds	

Part 4:

The end-to-end result

Proving Milawa sound



Milawa

Jitawa *verified* **LISP**

Assumes x86 model, C wrapper, OS, hardware

Milawa

Milawa theorem prover
(kernel approx. 2000 lines of Milawa Lisp)

https://raw.githubusercontent.com/HOL-Theorem-Prover/HOL/master/examples/theorem-prover/milawa-prover/core.lisp

Proving the top-level theorem

The top-level theorem:

relates the logic's semantics
with the execution of the x86 machine code.

Steps:

A. formalise Milawa's logic

- syntax, semantics, inference, soundness

B. prove that Milawa's kernel is faithful to the logic

- run the Lisp parser (in the logic) on Milawa's kernel
- translate (with proof) deep embedding into shallow
- prove that Milawa's (reflective) kernel is faithful to logic

C. connect the verified Lisp implementation

- compose with the correctness thm for Lisp system

Theorem: Milawa is sound down to x86

There must be enough memory and input is Milawa's kernel followed by call to main for some *input*.

$\forall input\ pc.$

$\{$ init_state (milawa_implementation ++ "(milawa-main '$input$)") $*$ pc pc $\}$
pc : code_for_entire_jitawa_implementation
$\{$ error_message $\vee$ (let $result$ = compute_output (parse $input$) in
$\langle$every_line line_ok $result\rangle$ $*$
final_state (output_string $result$ ++ "SUCCESS")) $\}$

Machine code terminates either with error message, or …

… output lines that are all true w.r.t. the semantics of the logic.

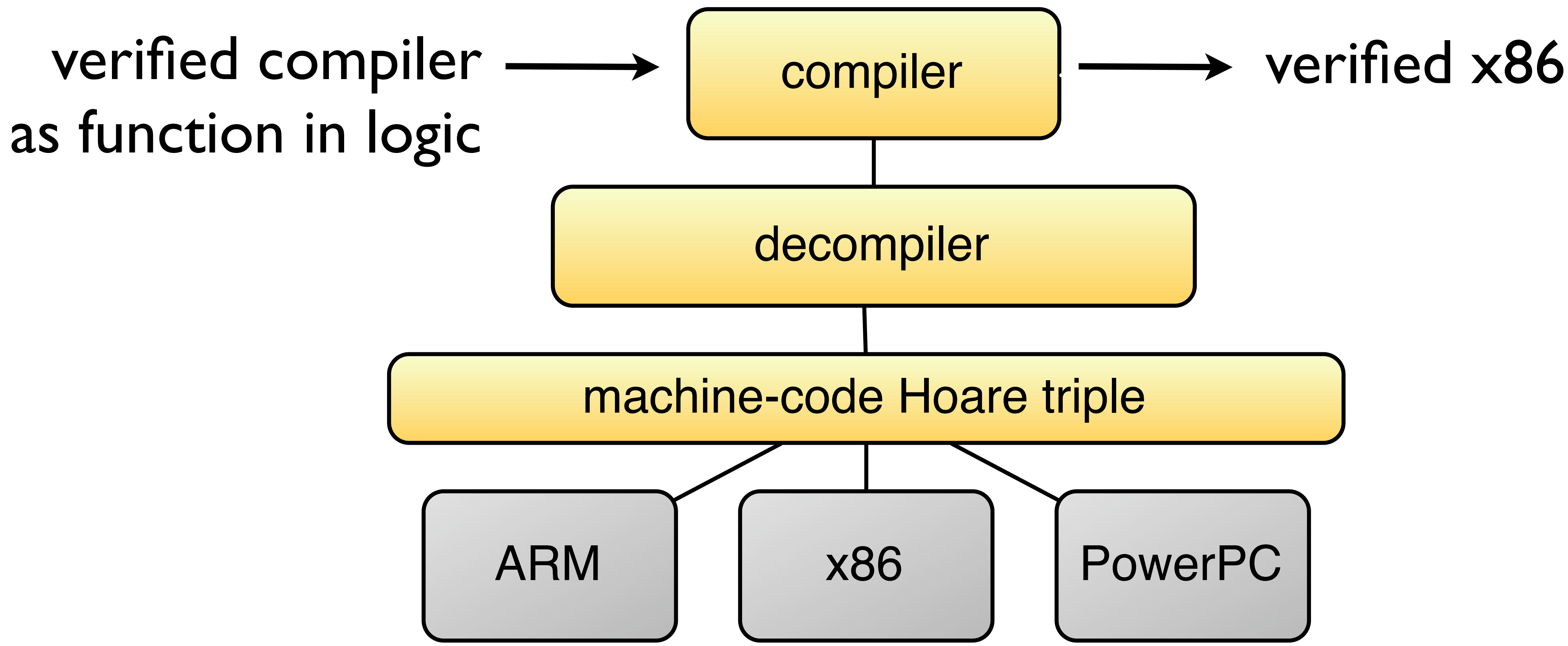
line_ok (π, l) = $(l =$ "NIL"$) \vee$
$(\exists n.\ (l =$ "(PRINT (n …))"$) \wedge$ is_number $n) \vee$
$(\exists \phi.\ (l =$ "(PRINT (THEOREM ϕ))"$) \wedge$ context_ok $\pi \wedge \models_\pi \phi)$

Final Part:

Learning from the mistakes. Doing it better.

A better compiler compiler?

The x86 for the compile function was produced as follows:



A bit cumbersome....

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

Bootstrapping the compiler

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

EVAL ``compile COMPILE``

derives a theorem:

compile COMPILE = compiler-as-machine-code

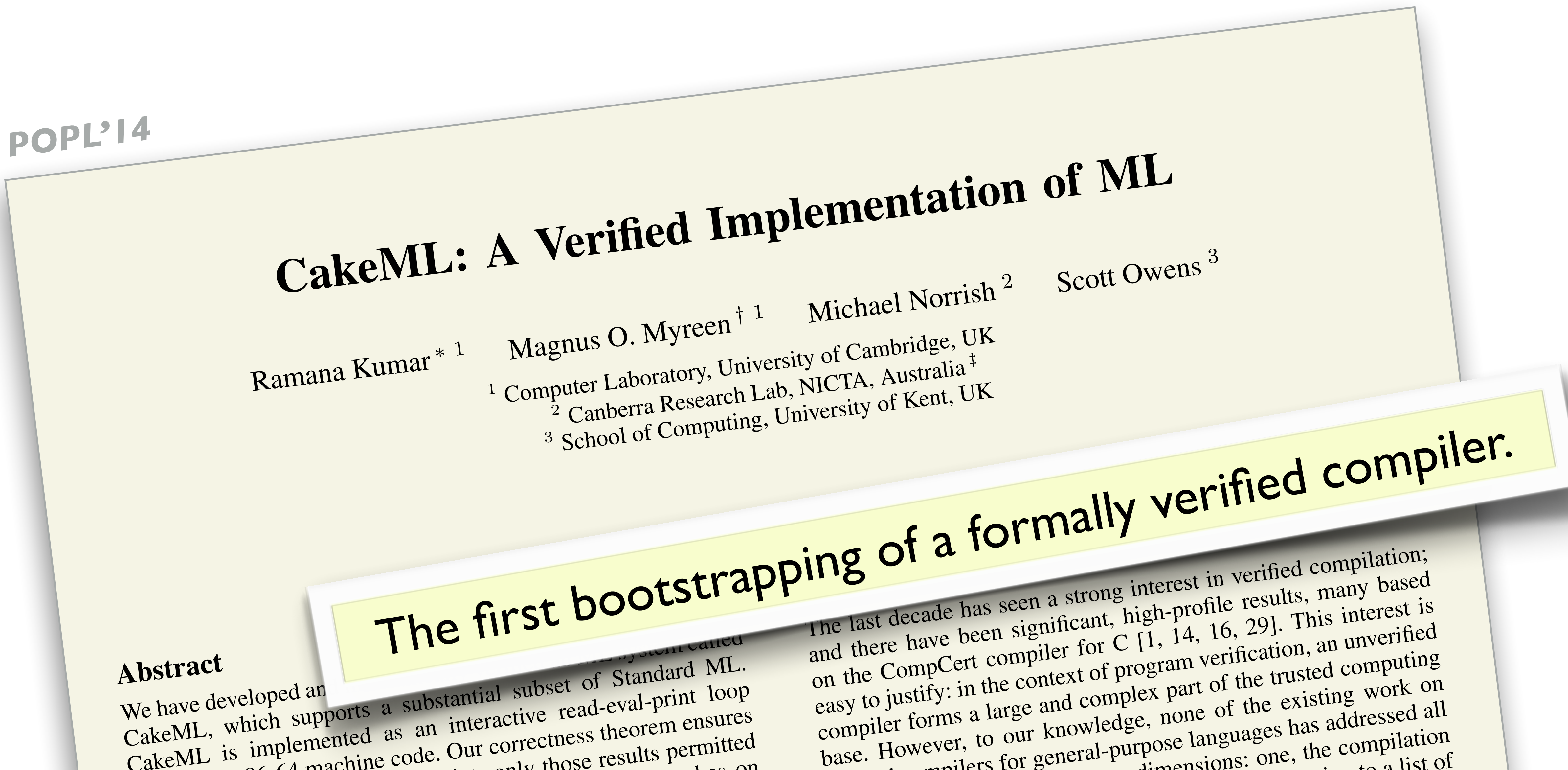
Ramana Kumar
(Uni. Cambridge)

Magnus Myreen
(Uni. Cambridge)

Michael Norrish
(NICTA, ANU)

Scott Owens
(Uni. Kent)

POPL'14



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

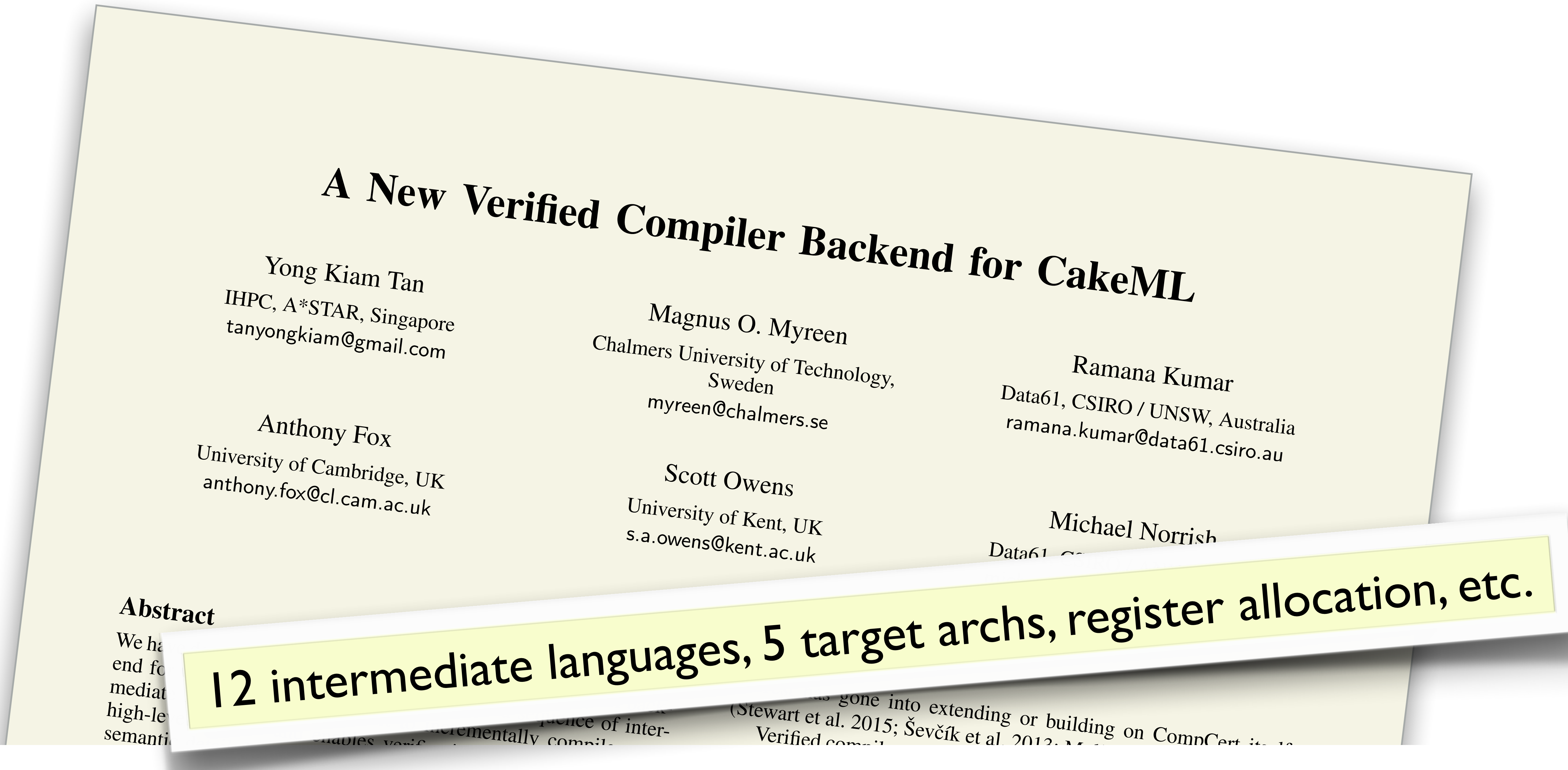
We have developed an ... system called CakeML, which supports a substantial subset of Standard ML. CakeML is implemented as an interactive read-eval-print loop ... machine code. Our correctness theorem ensures ... only those results permitted ...

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 ... dimensions: one, the compilation ...

The first bootstrapping of a formally verified compiler.

Tomorrow at ICFP!

ICFP'16


A New Verified Compiler Backend for CakeML

Yong Kiam Tan
IHPC, A*STAR, Singapore
tanyongkiam@gmail.com

Magnus O. Myreen
Chalmers University of Technology, Sweden
myreen@chalmers.se

Ramana Kumar
Data61, CSIRO / UNSW, Australia
ramana.kumar@data61.csiro.au

Anthony Fox
University of Cambridge, UK
anthony.fox@cl.cam.ac.uk

Scott Owens
University of Kent, UK
s.a.owens@kent.ac.uk

Michael Norrish

Abstract


Magnus Myreen


Yong Kiam Tan


Ramana Kumar


Anthony Fox


Scott Owens


Michael Norrish

Looking back…

2005:

I’m a PhD student working on verification of machine code (factorial, length of a linked list)

???

???

Result:

A Verified Lisp Implementation for A Verified Theorem Prover

???

2005:

I'm a PhD student working on verification of machine code (factorial, length of a linked list)

↓ basic reasoning about real machine code

↓ powerful automation

↓ verification of garbage collectors

↓ synthesis from (abstract) functional specs

↓ verified Lisp interpreters

↓ verified just-in-time compiler for Lisp

↓ *Result:*

A Verified Lisp Implementation for A Verified Theorem Prover

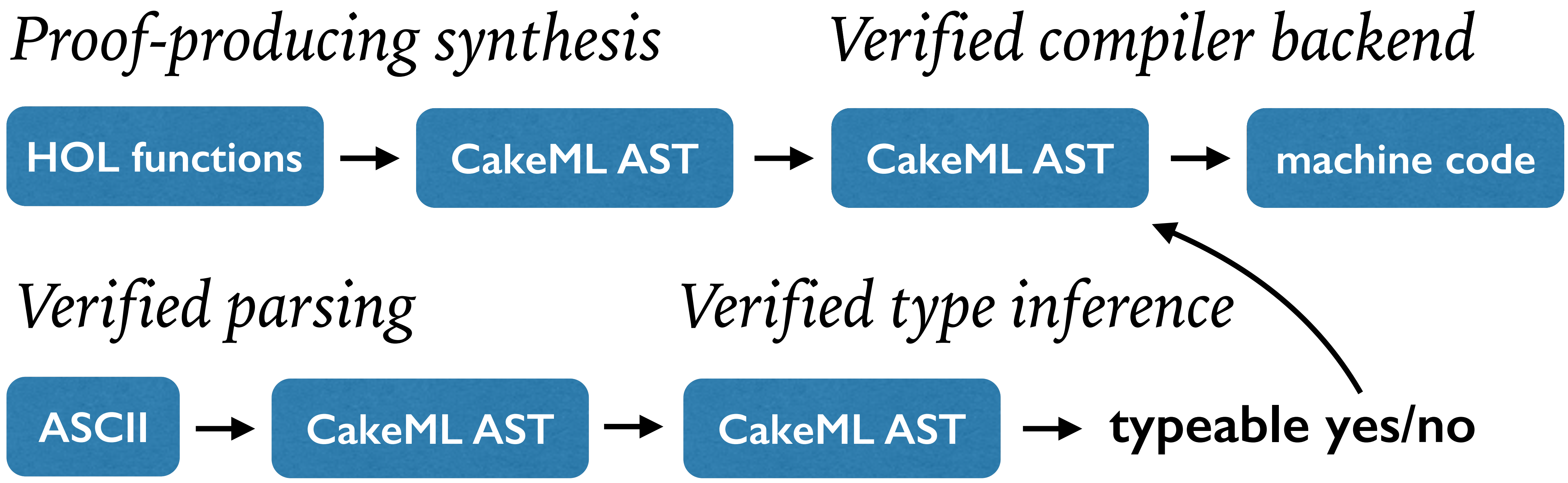
↓ verified compiler bootstrapping (ML)

↓ ...

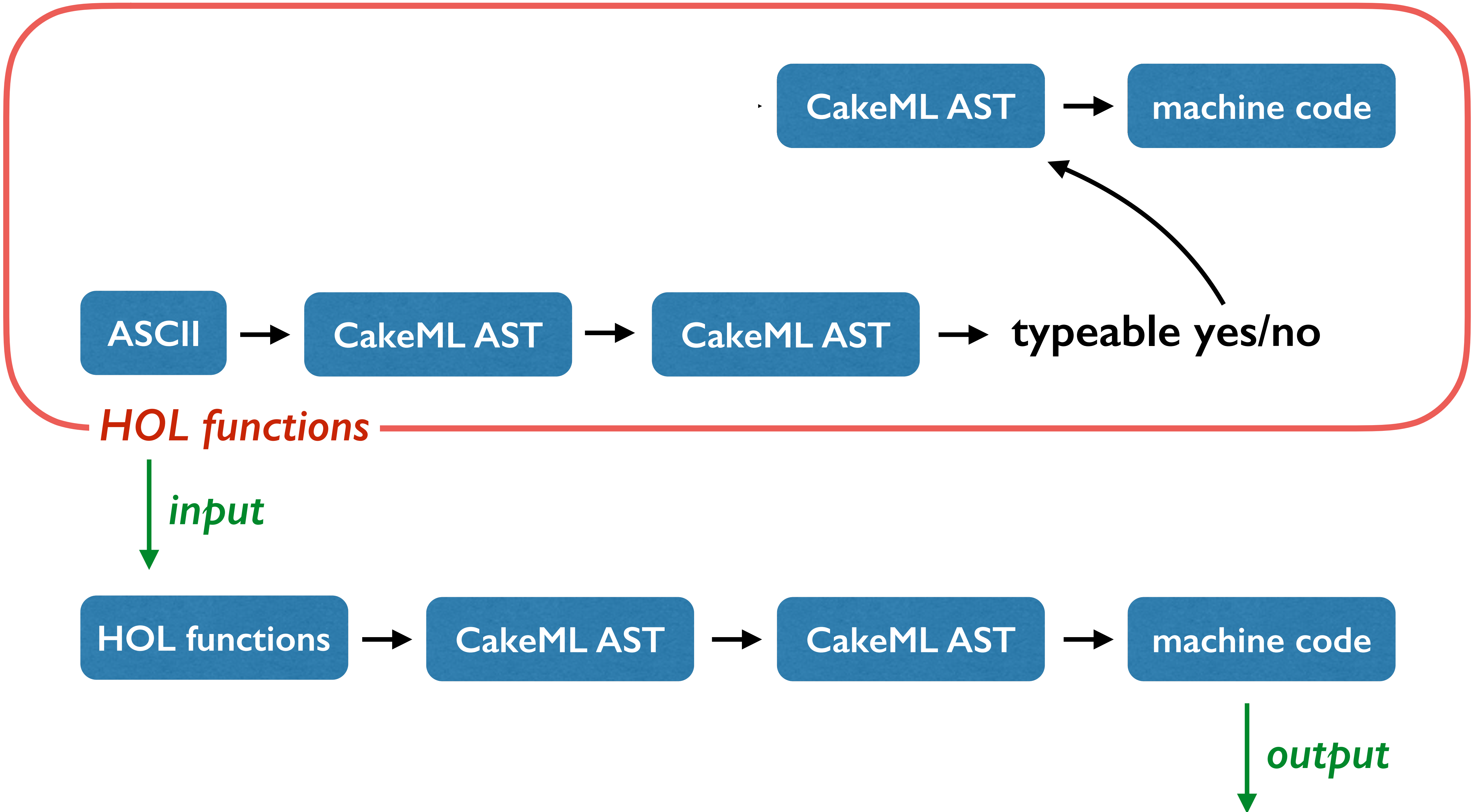
Questions?

Thank you for inviting me!

Intuition for Bootstrapping



Intuition for Bootstrapping



verified x86 implementation of parsing, type inference, and compilation