LFCS seminar, Edinburgh, May 2015

CakeML

A verified implementation of ML

Magnus Myreen

Chalmers University of Technology & University of Cambridge

Joint work with Ramana Kumar, Michael Norrish, Scott Owens and many more

Motivation

CakeML

A verified implementation of ML

Why?

We wanted to know whether it's possible.

functional correctness

Background

From my PhD (2009):

Verified Lisp interpreter
in ARM, x86 and PowerPC machine code

Collaboration with Jared Davis (2011):

Verified Lisp read-eval-print loop
in 64-bit x86 machine code, with dynamic compilation

(plus verification of an ACL2-like theorem prover)

Can we do the same for ML?

A verified implementation of ML

(plus verification of a HOL-like theorem prover?)

Other HOL4 hackers also have relevant interests…

People involved

Ramana Kumar
(Uni. Cambridge)

operational **semantics**

verified **compilation** from CakeML to bytecode

Scott Owens
(Uni. Kent)

verified **type** inferencer

Michael Norrish
(NICTA, ANU)

verified **parsing** (syntax is compatible with SML)

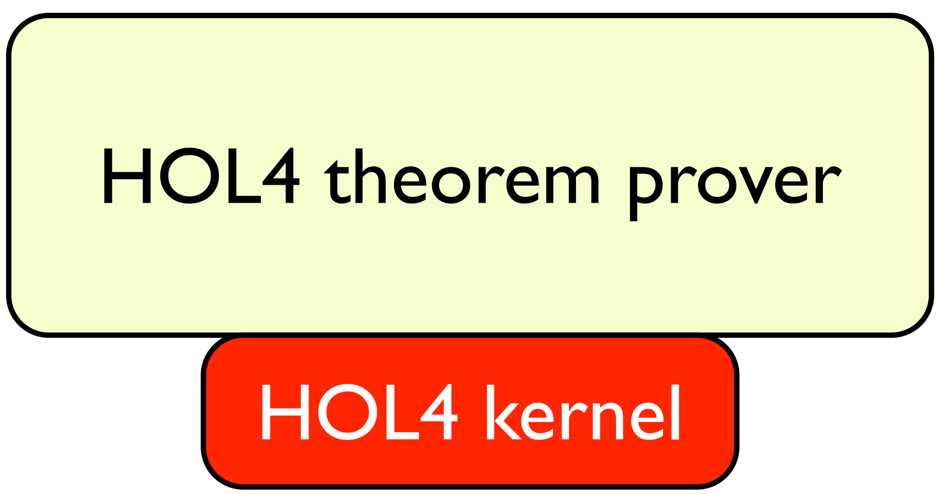
verified **x86** implementations

proof-producing **code generation** from HOL

Magnus Myreen
(Chalmers)

Proofs in HOL4

HOL4 is 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.

Overall aim

to make proof assistants into trustworthy and practical program development platforms

Trustworthy code extraction:

functions in HOL (shallow embedding)

↓ proof-producing translation [ICFP'12, JFP'14]

CakeML program (deep embedding)

↓ verified compilation of CakeML [POPL'14]

x86-64 machine code (deep embedding)

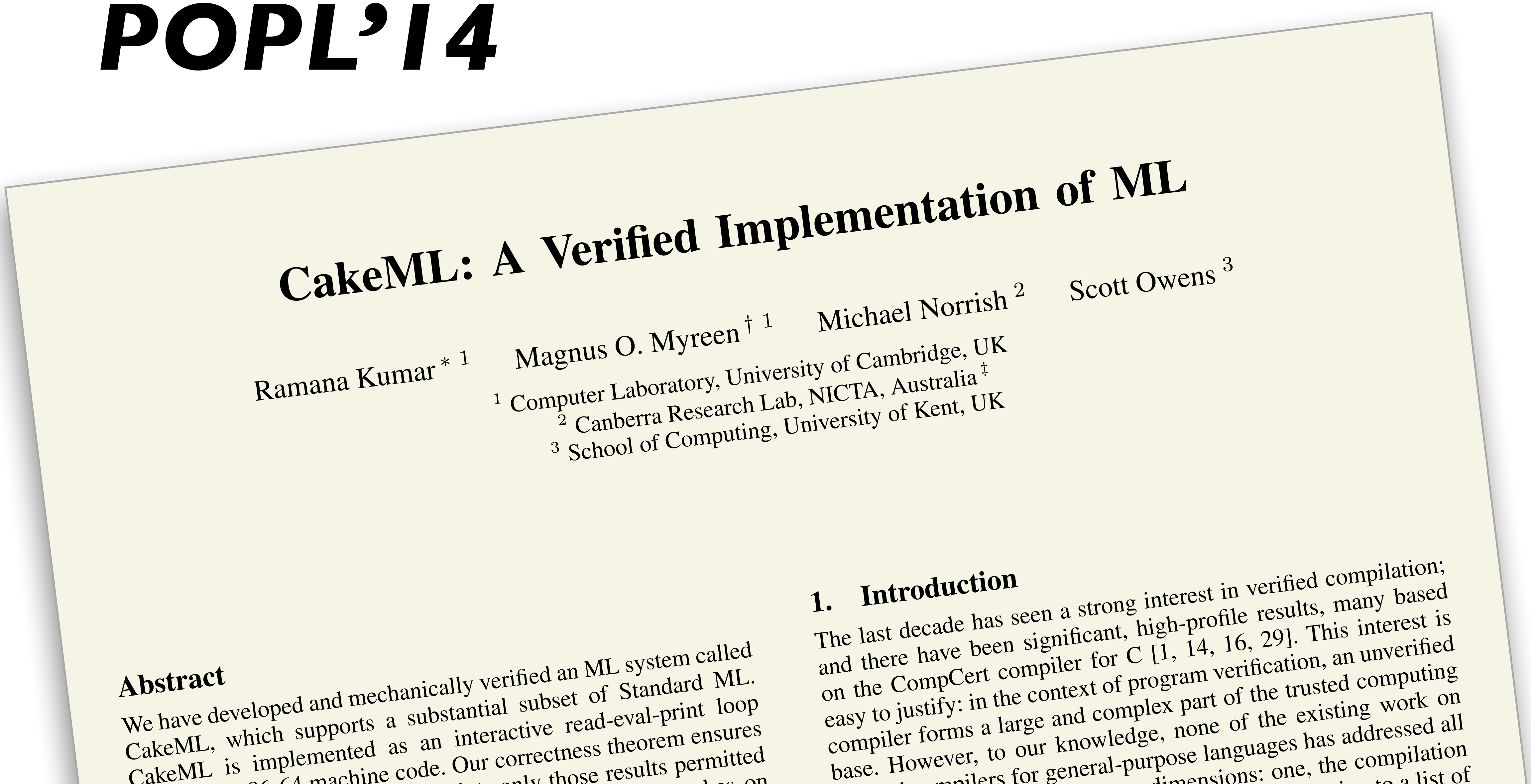
This talk

Part 1: verified implementation of CakeML

Part 2: current status, HOL light, future

Part 1: verified implementation of CakeML

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 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

1. Introduction

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

Dimensions of Compiler Verification

source code

abstract syntax

intermediate language

bytecode

machine code

how far compiler goes

Our verification covers the full spectrum of **both** dimensions.

compiler algorithm

implementation in ML

implementation in machine code

interactive call in read-eval-print loop runtime

the thing that is verified

The CakeML language

Design: *"The CakeML language ~~is~~ was originally designed to be both easy to program in and easy to reason about formally"*

It is still clean, but not always simple.

Reality: CakeML, the language
= Standard ML without I/O or functors

i.e. with almost everything else:

- ✓ higher-order functions
- ✓ mutual recursion and polymorphism
- ✓ datatypes and (nested) pattern matching
- ✓ references and (user-defined) exceptions
- ✓ modules, signatures, abstract types

Contributions of POPL'14 paper

Artefacts

Specifications

Verified Algorithms

Proof techniques

Divergence Preservation

Bootstrapping

light-weight approach to **divergence preservation** with big-step op. sem.

use verified compiler to produce **verified implementation of the compiler**

Proof development where everything fits together.

Approach

Proof by refinement:

Step 1: **specification** of CakeML language

- big-step and small-step operational semantics

Step 2: **functional implementation** in logic

- read-eval-print-loop as verified function in logic

Step 3: production of **verified x86-64 machine code**

- produced mostly by bootstrapping the compiler

Operational semantics

Big-step semantics:

- big-step evaluation relation
- environment semantics (cf. substitution sem.)
- produces TypeError for badly typed evaluations (e.g. `1+nil`)
- stuck = divergence

Equivalent small-step semantics:

- used for type-soundness proof and definition of divergence

Read-eval-print-loop semantics.

Semantics written in Lem, see Mulligan et al. [ICFP'14]

Approach

Proof by refinement:

Step 1: **specification** of CakeML language

- big-step and small-step operational semantics

Step 2: **functional implementation** in logic

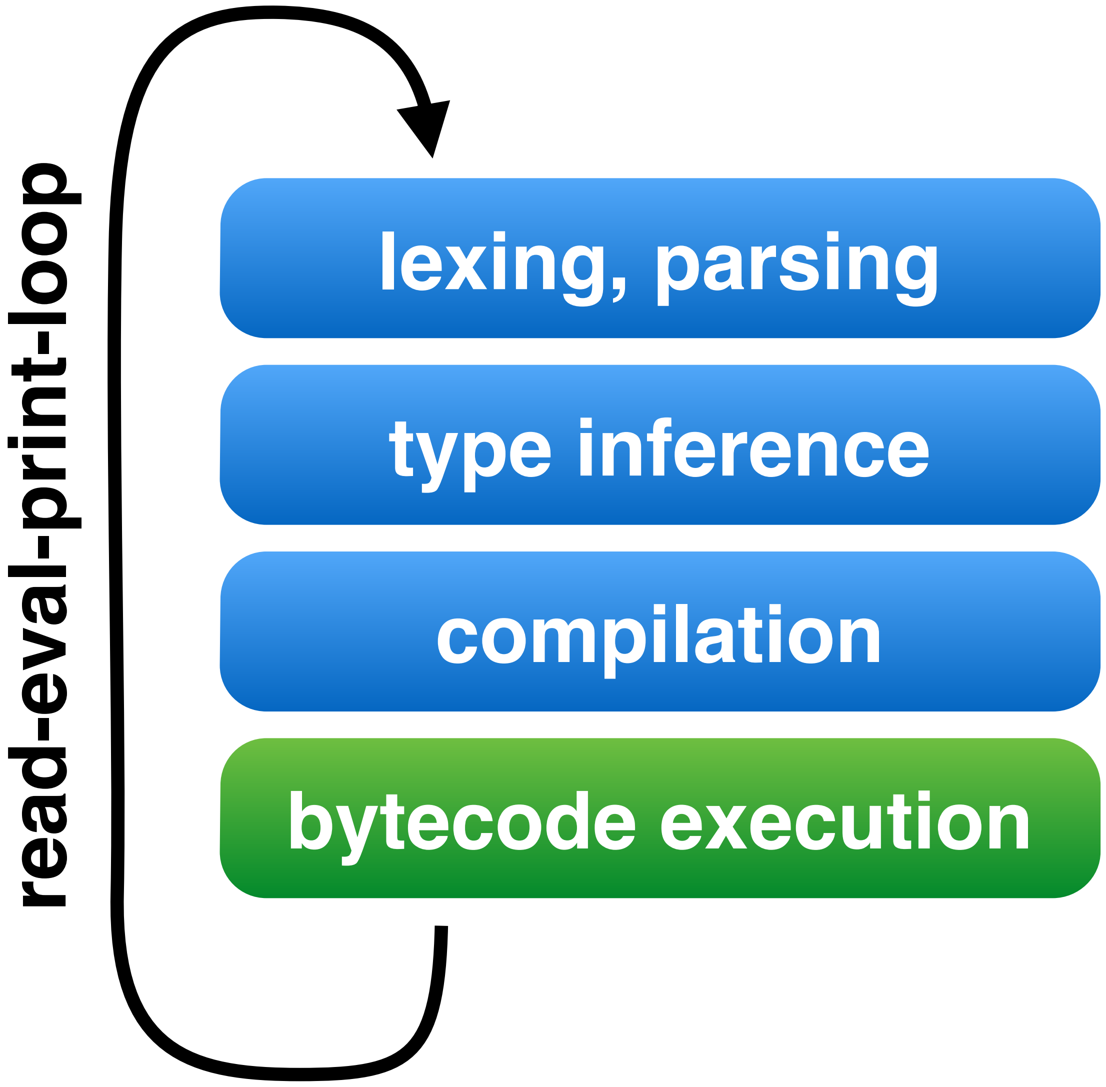
- read-eval-print-loop as verified function in logic

Step 3: production of **verified x86-64 machine code**

- produced mostly by bootstrapping the compiler

Functional implementation

Read-eval-print loop defined as rec. function in the logic:



lexing, parsing

Specification:

Context-free grammar (CFG) for significant subset of SML

Executable lexer.

Implementation:

Parsing-Expression-Grammar (PEG) Parser

- inductive evaluation relation
- executable interpreter for PEGs

Correctness:

Soundness and completeness

- induction on length of token list/parse tree and non-terminal rank

type inference

Specification:

Declarative type system.

Implementation:

Based on Milner's Algorithm W

Purely functional (uses state-exception monad)

Correctness:

new since last month!

Proved sound and *complete* w.r.t. declarative type system

Re-use of previous work on verified unification

compilation

Purpose:

Translates (typechecked) CakeML into CakeML Bytecode.

Implementation:

Translation via one intermediate language (IL).

- de Bruijn indices
- big-step operational semantics

CakeML to IL: makes language more uniform

IL to IL: removes pattern-matching, lightweight opt.

IL to Bytecode: closure conversion, data refinement, tail-call opt.

Semantics of **bytecode execution**

Instructions:

$$
\begin{array}{lll}
bc\_inst & ::= & \textsf{Stack}\ bc\_stack\_op \mid \textsf{PushExc} \mid \textsf{PopExc} \\
& \mid & \textsf{Return} \mid \textsf{CallPtr} \mid \textsf{Call}\ loc \\
& \mid & \textsf{PushPtr}\ loc \mid \textsf{Jump}\ loc \mid \textsf{JumpIf}\ loc \\
& \mid & \textsf{Ref} \mid \textsf{Deref} \mid \textsf{Update} \mid \textsf{Print} \mid \textsf{PrintC char} \\
& \mid & \textsf{Label}\ n \mid \textsf{Tick} \mid \textsf{Stop} \\
bc\_stack\_op & ::= & \textsf{Pop} \mid \textsf{Pops}\ n \mid \textsf{Shift}\ n\ n \mid \textsf{PushInt int} \\
& \mid & \textsf{Cons}\ n\ n \mid \textsf{El}\ n \mid \textsf{TagEq}\ n \mid \textsf{IsBlock}\ n \\
& \mid & \textsf{Load}\ n \mid \textsf{Store}\ n \mid \textsf{LoadRev}\ n \\
& \mid & \textsf{Equal} \mid \textsf{Less} \mid \textsf{Add} \mid \textsf{Sub} \mid \textsf{Mult} \mid \textsf{Div} \mid \textsf{Mod} \\
loc & ::= & \textsf{Lab}\ n \mid \textsf{Addr}\ n
\end{array}
$$

Small-step semantics; values and state:

$$
\begin{array}{lll}
bc\_value & ::= & \textsf{Number int} \mid \textsf{RefPtr}\ n \mid \textsf{Block}\ n\ bc\_value^* \\
& \mid & \textsf{CodePtr}\ n \mid \textsf{StackPtr}\ n \\
bc\_state & ::= \{ & \textsf{stack} : bc\_value^*; \textsf{refs} : n \mapsto bc\_value; \\
& & \textsf{code} : bc\_inst^*;\ \textsf{pc} : n; \textsf{handler} : n; \\
& & \textsf{output} : \textsf{string};\ \textsf{names} : n \mapsto \textsf{string}; \\
& & \textsf{clock} : n^? \quad \}
\end{array}
$$

Semantics of bytecode execution

Sample rules:

$$\frac{\mathsf{fetch}(bs) = \mathsf{Stack}\ (\mathsf{Cons}\ t\ n) \quad bs.\mathsf{stack} = vs \mathbin{@} xs \quad |vs| = n}{bs \to (\mathsf{bump}\ bs)\{\mathsf{stack} = \mathsf{Block}\ t\ (\mathsf{rev}\ vs) :: xs\}}$$

$$\frac{\mathsf{fetch}(bs) = \mathsf{Return} \quad bs.\mathsf{stack} = x :: \mathsf{CodePtr}\ ptr :: xs}{bs \to bs\{\mathsf{stack} = x :: xs;\ \mathsf{pc} = ptr\}}$$

$$\frac{\mathsf{fetch}(bs) = \mathsf{CallPtr} \quad bs.\mathsf{stack} = x :: \mathsf{CodePtr}\ ptr :: xs}{bs \to bs\{\mathsf{stack} = x :: \mathsf{CodePtr}\ (\mathsf{bump}\ bs).\mathsf{pc} :: xs;\ \mathsf{pc} = ptr\}}$$

compilation

Correctness:

Proved in the direction of compilation.

Shape of correctness theorem:

$$(exp, env) \Downarrow_{\mathrm{ev}} val \implies$$
$$\text{``the code for } exp \text{ is installed in } bs \text{ etc.''} \implies$$
$$\exists bs'.\ bs \longrightarrow^{*} bs' \wedge \text{``}bs' \text{ contains } val\text{''}$$

Bytecode semantics step relation

What about divergence?

We want: generated code diverges
if and only if source code diverges

Idea: add logical clock

Big-step semantics:

- has an optional clock component
- clock 'ticks' decrements every time a function is applied
- once clock hits zero, execution stops with a TimeOut

Why do this?

- because now big-step semantics describes both terminating and non-terminating evaluations

either: Result or TimeOut

for every exp env clock

there is some result

$$\forall \mathit{exp}\ \mathit{env}\ \mathit{clock}.\ \exists \mathit{res}.\ (\mathit{exp}, \mathit{env}, \mathsf{Some}\ \mathit{clock}) \Downarrow_{\mathrm{ev}} \mathit{res}$$

produced by the semantics

Divergence

Evaluation diverges if

$$\forall clock.\ (exp, env, \mathsf{Some}\ clock) \Downarrow_{\mathrm{ev}} \mathsf{TimeOut}$$

for all clock values

TimeOut happens

Compiler correctness proved in conventional forward direction:

$$(exp, env) \Downarrow_{\mathrm{ev}} val \implies$$
$$\text{“the code for } exp \text{ is installed in } bs \text{ etc.”} \implies$$
$$\exists bs'.\ bs \rightarrow^* bs' \wedge \text{“}bs' \text{ contains } val\text{”}$$

Bytecode has clock

… that stays in sync with CakeML clock

Theorem: bytecode diverges if and only if CakeML eval diverges

Approach

Proof by refinement:

Step 1: **specification** of CakeML language

- big-step and small-step operational semantics

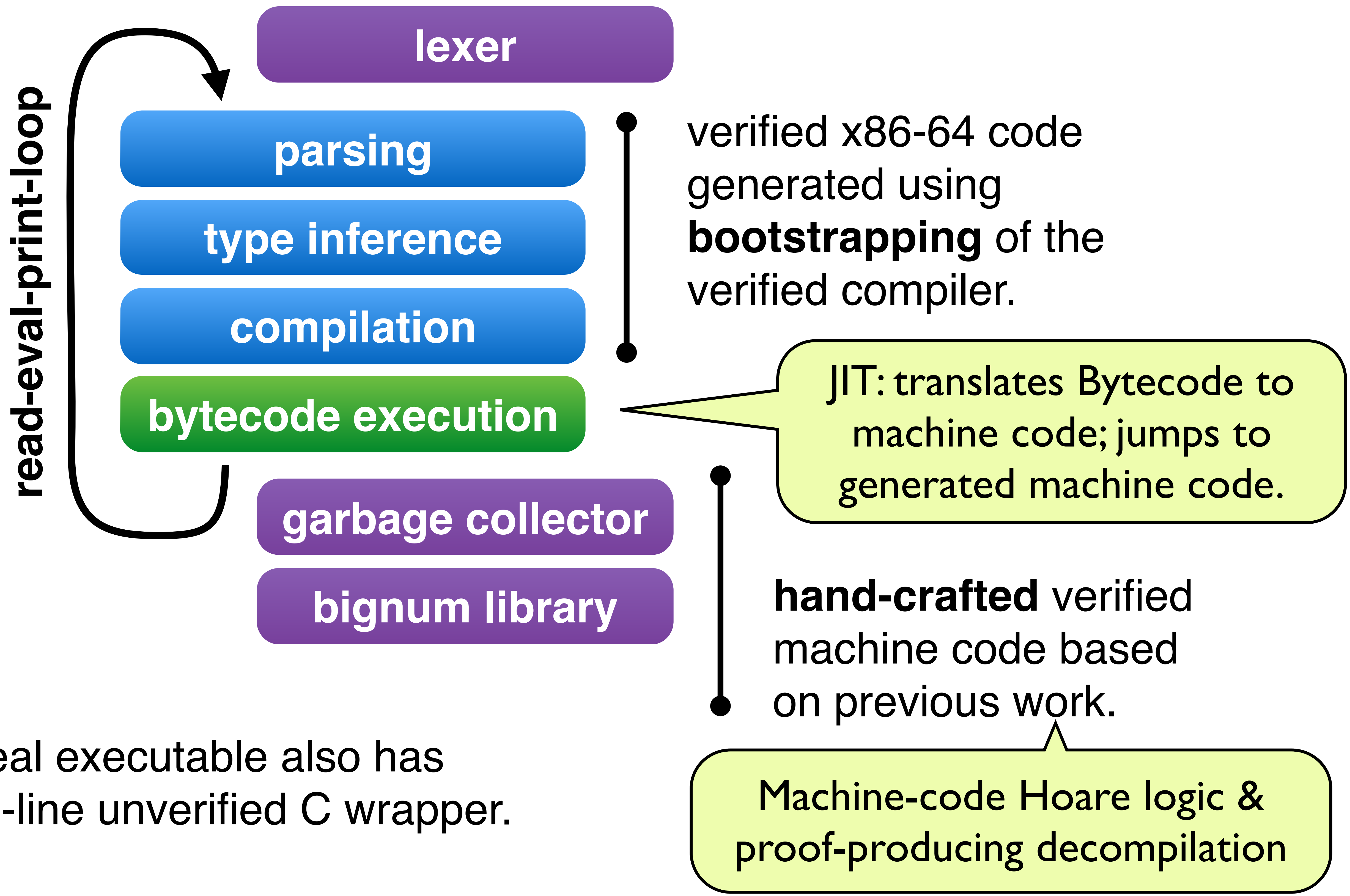
Step 2: **functional implementation** in logic

- read-eval-print-loop as verified function in logic

Step 3: production of **verified x86-64 machine code**

- produced mostly by bootstrapping the compiler

Verified x86-64 machine code



Real executable also has 30-line unverified C wrapper.

Compiling Bytecode into x86-64 mc

Extract of definition:

each bytecode inst. maps to some x86

x64 i Pop = [0x48, 0x58]

x64_code i (x::xs) = x64 i x ++ x64_code (i + len(x64 i x)) xs

Correctness:

Each Bytecode instruction is correctly executed by generated x86-64 code.

$$
\begin{array}{l}
bs \to bs' \implies \\
\text{temporal}\ \{(base, \text{x64\_code}\ 0\ bs.\text{code})\} \\
\quad (\text{now}\ (\text{bc\_heap}\ bs\ (base, aux)) \Rightarrow \\
\qquad \text{later}\ (\text{now}\ (\text{bc\_heap}\ bs'\ (base, aux)) \\
\qquad\qquad \vee\ \text{now}\ (\text{out\_of\_memory\_error}\ aux)))
\end{array}
$$

heap invariant / memory abstraction

Bootstrapping the verified compiler

Idea for in-logic bootstrapping

input: verified compiler function

Trustworthy code extraction:

functions in HOL (shallow embedding)

↓ proof-producing translation [ICFP'12, JFP'14]

CakeML program (deep embedding)

↓ verified compilation of CakeML [POPL'14]

x86-64 machine code (deep embedding)

output: verified implementation of compiler function

Compiling the compiler in logic

parsing

type inference

compilation

function in logic: compile

by proof-producing synthesis [ICFP'12]

CakeML program (COMPILE) such that:

$\vdash$ COMPILE *implements* compile

Proof by evaluation inside the logic:

$\vdash$ compile-to-x64 COMPILE = x64-code

Compiler correctness theorem:

$\vdash$ $\forall$prog. compile-to-x64 prog *implements* prog

Combination of theorems:

$\vdash$ x64-code *implements* compile

Details (build up)

Function in logic:

$$\mathsf{map}\,f\ [\,] = [\,]$$
$$\mathsf{map}\,f\ (h{::}t) = f\ h{::}\mathsf{map}\,f\ t$$

Evaluation in logic:

$$\vdash\ \mathsf{map\ length}\ [[1;\ 1];\ [2];\ [\,]] = [2;\ 1;\ 0]$$

Translation into CakeML produces:

```
fun map v3 v4 =
  case v4
  of [] => []
  |  v2::v1 => v3 v2::(map v3 v1)
```

Translation into CakeML, actual output:

```
map_dec =
 Letrec
  [("map","v3",
    Fun "v4"
     (Mat (Var "v4")
       [(Pcon "nil" [],Con "nil" []);
        (Pcon "::" [Pvar "v2"; Pvar "v1"],
         Con "::"
          [App [Var "v3"; Var "v2"]; App [App [Var "map"; Var "v3"]; Var "v1"]])]))]
```

Details (build up)

Produced proof (called a certificate theorem):

$$\vdash\ \exists env\ c.\ \ \mathsf{EvalDec\ InitEnv\ map\_dec}\ env \wedge \mathsf{Lookup}\ \texttt{"map"}\ env = \mathsf{Some}\ c \wedge ((a \longrightarrow b) \longrightarrow \mathsf{ListTy}\ a \longrightarrow \mathsf{ListTy}\ b)\ \mathsf{map}\ c$$

Translation into CakeML, actual output:

```
map_dec =
 Letrec
  [("map","v3",
   Fun "v4"
    (Mat (Var "v4")
      [(Pcon "nil" [],Con "nil" []);
       (Pcon "::" [Pvar "v2"; Pvar "v1"],
        Con "::"
         [App [Var "v3"; Var "v2"]; App [App [Var "map"; Var "v3"]; Var "v1"]])]))]
```

Evaluation of compilation (in logic)

Example 2 (Compilation by evaluation of map*)*

$\vdash$ CompileDec InitCS map_dec $=$
(MapCS,
[Jump (Lab 12); Label 10; Stack (PushInt 0); Stack (PushInt 1); Ref;
PushPtr (Lab 11); Stack (Load 0); Stack (Load 5); Stack (PushInt 1);
Stack (Cons 0); Stack (... ...); ... ... ; ... ; ...])

Compiler correctness (specialised to terminating case):

$$\vdash \text{Inv}\ env_1\ cs_1\ bs_1 \wedge \text{EvalDec}\ env_1\ dec\ env_2 \wedge \text{CompileDec}\ cs_1\ dec = (cs_2,bc) \Rightarrow$$
$$\exists bs_2.\ (\text{AddCode}\ bs_1\ bc) \rightarrow^* bs_2 \wedge \text{Halted}\ bs_2 \wedge \text{Inv}\ env_2\ cs_2\ bs_2$$

(Some of the) actual details

Translation of compiler into CakeML

$\vdash \exists c.$
EvalDec InitEnv (Struct `"C"` CompileDec_decs) CompEnv $\wedge$
LookupMod `"C"` `"compiledec"` CompEnv $=$ Some c $\wedge$
(CompStateTy $\longrightarrow$ DecTy $\longrightarrow$ PairTy CompStateTy (ListTy BCInstTy)) CompileDec c

Evaluating the compiler on itself

$\vdash$ CompileDec InitCS (Struct `"C"` CompileDec_decs) $=$
(CompCS,CompileDec_bytecode)

Compiler correctness theorem:

$\vdash$ Inv InitEnv InitCS InitBS

$\vdash$ Inv env_1 cs_1 bs_1 $\wedge$ EvalDec env_1 dec env_2 $\wedge$ CompileDec cs_1 $dec = (cs_2, bc) \Rightarrow$
$\exists bs_2.$ (AddCode bs_1 bc) $\rightarrow^*$ bs_2 $\wedge$ Halted bs_2 $\wedge$ Inv env_2 cs_2 bs_2

NB: For a read-eval-print-loop, the details are a bit more involved…

Top-level **correctness theorem**

Top-level correctness theorem

Top-level specification:

initial program, the basis library

$\mathsf{ReplSem}\ \mathsf{Basis}\ input\ \ output$

string

string ending in either Terminate or Diverge

Correctness theorem:

$$
\begin{array}{l}
\vdash \mathsf{TemporalX64}\ \mathsf{ReplX64} \\
\quad (\mathsf{Now}\ (\mathsf{InitialisedX64}\ ms) \Rightarrow \\
\quad\quad \Diamond\ \mathsf{Now}\ (\mathsf{OutOfMemX64}\ ms) \vee \\
\quad\quad \exists\, output. \\
\quad\quad\quad \mathsf{Holds}\ (\mathsf{ReplSem}\ \mathsf{Basis}\ ms.\mathsf{input}\ output) \wedge \\
\quad\quad\quad \texttt{if}\ \mathsf{Diverges}\ output\ \texttt{then}\ \Box\ \Diamond\ \mathsf{Now}\ (\mathsf{RunningX64}\ output\ ms) \\
\quad\quad\quad \texttt{else}\ \Diamond\ \mathsf{Now}\ (\mathsf{TerminatedX64}\ output\ ms))
\end{array}
$$

Numbers

Performance:

Slow: *interpreted* OCaml is 1x faster (… future work!)

Effort:

~100k lines of proof script in HOL4

< 5 man-years, but builds on a lot of previous work

Size:

875,812 instructions of verified x86-64 machine code

implementation generates more instructions at runtime

large due to bootstrapping, naive compiler

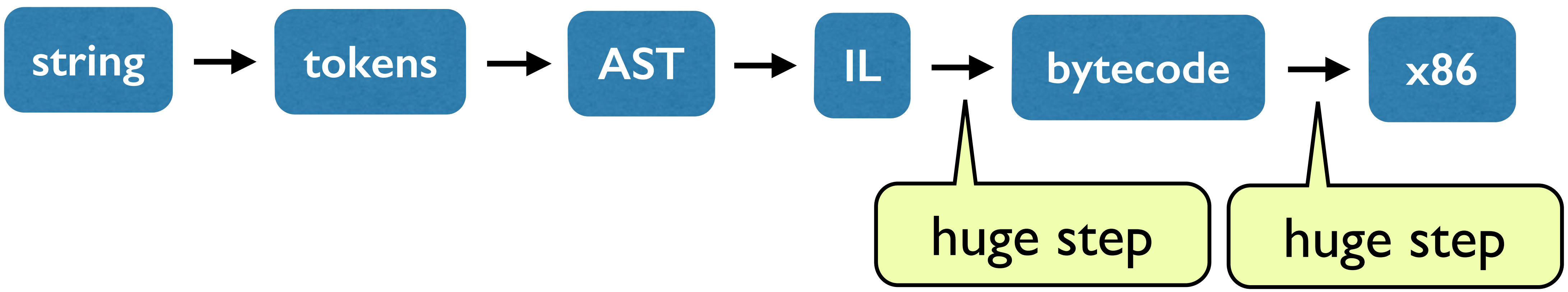
This talk

Part 1: verified implementation of CakeML

Part 2: current status, HOL light, future

Current status

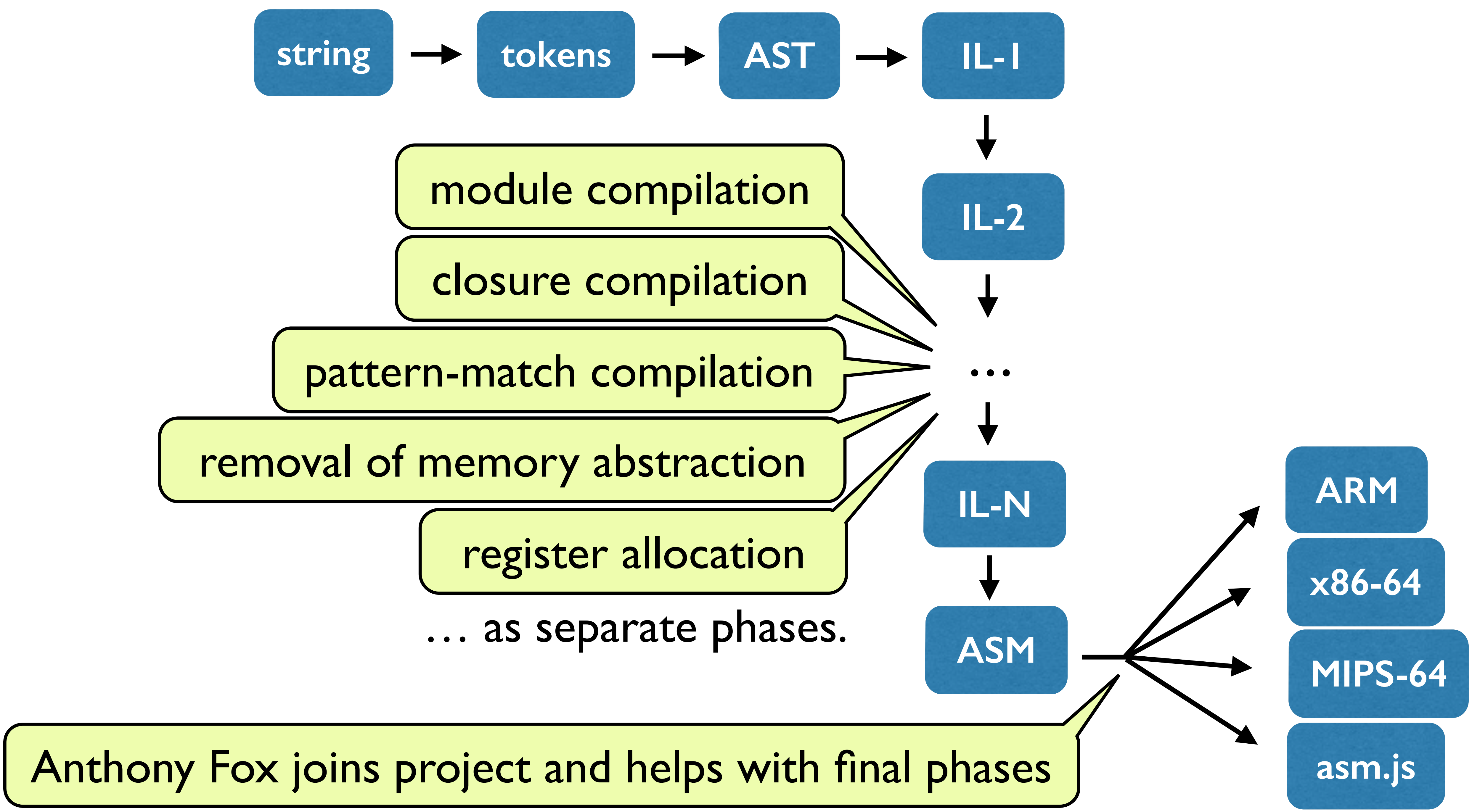
Current compiler:



Bytecode simplified proofs of read-eval-print loop, but made optimisation impossible.

Current work / future plans

Refactored compiler: split into more conventional compiler phases



Verified examples on CakeML

Verification infrastructure:

- have: synthesis tool that maps HOL into CakeML [ICFP'12, JFP'14]
- future: integration with Arthur Charguéraud's characteristic formulae technology [ICFP'10, ICFP'11]

for developing interesting verified examples.

Big example: verified HOL light

ML was originally developed to host theorem provers.

Aim: verified HOL theorem prover.

We have [ITP'13, ITP'14]:

- syntax, semantics and soundness of HOL (stateful, stateless)
- verified implementation of the HOL light kernel in CakeML (produced through synthesis)

Still to do:

- soundness of kernel ⇒ soundness of entire HOL light
- run HOL light standard library on top of CakeML

Freek Wiedijk is translating HOL light sources to CakeML

Summary

Contributions so far:

First **bootstrapping** of a formally **verified compiler**.

New lightweight method for **divergence preservation**.

Current work:

Formally **verified** implementation of **HOL light**.

Verified **I/O** (foreign-function interface)**. seL4**.

Compiler improvements (new ILs, opt, targets).

Long-term aim:

An **ecosystem** of tools and proofs around CakeML lang.

Questions? Suggestions?