Certification of high-level and low-level programs, IHP, Paris, 2014

CakeML

A verified implementation of ML

Ramana Kumar — UNIVERSITY OF CAMBRIDGE

Magnus Myreen — UNIVERSITY OF CAMBRIDGE

Michael Norrish — NICTA

Scott Owens — University of Kent

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)

Michael Norrish
(NICTA, ANU)

operational **semantics**

verified **compilation** from CakeML to bytecode

verified **type** inferencer

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

verified **x86** implementations

proof-producing **code generation** from HOL

Scott Owens
(Uni. Kent)

Magnus Myreen
(Uni. Cambridge)

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)

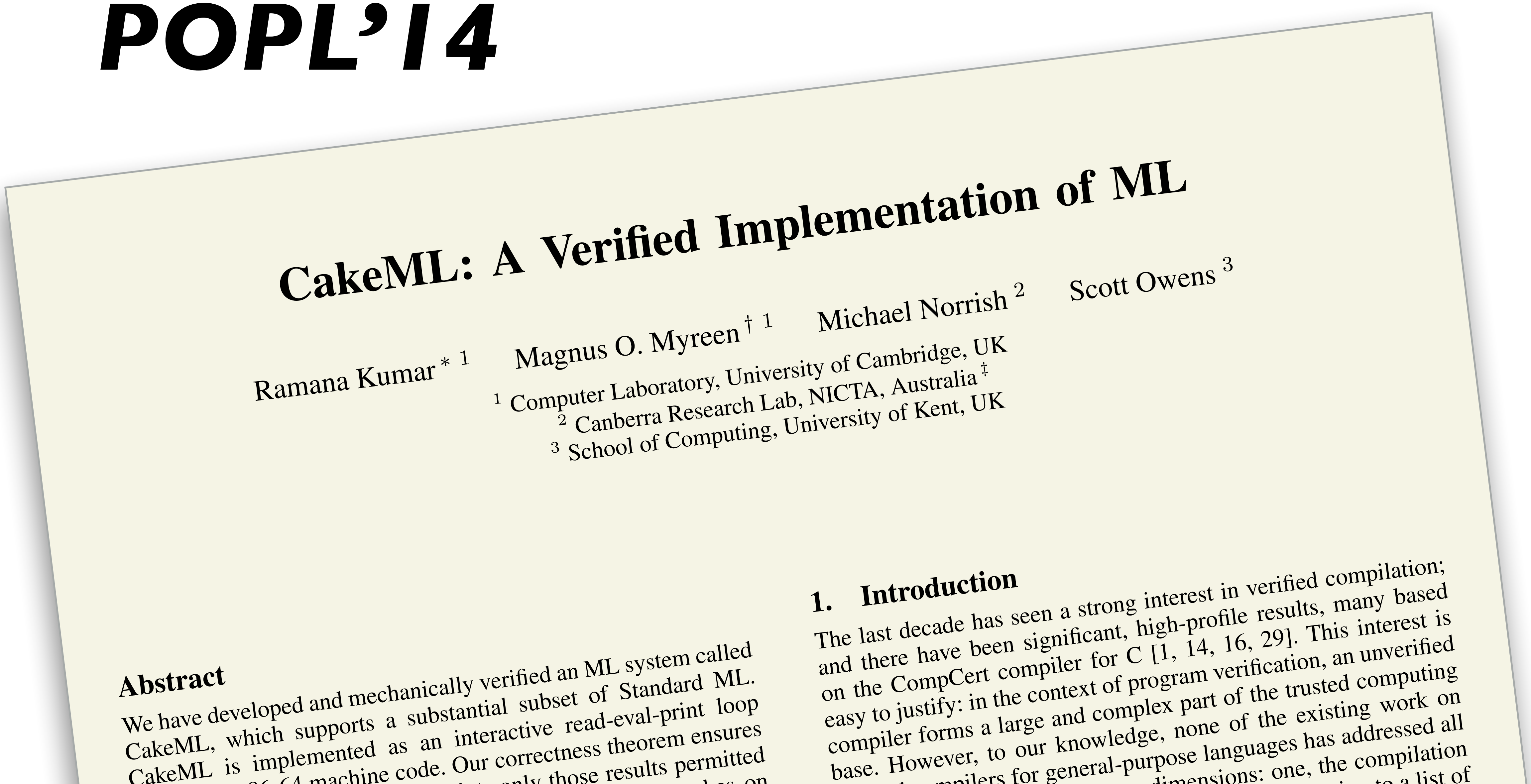
This talk

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

was originally

Design: *"The CakeML language ~~is~~ 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.

main **new technique**: use verified compiler to produce verified implementation

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 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:

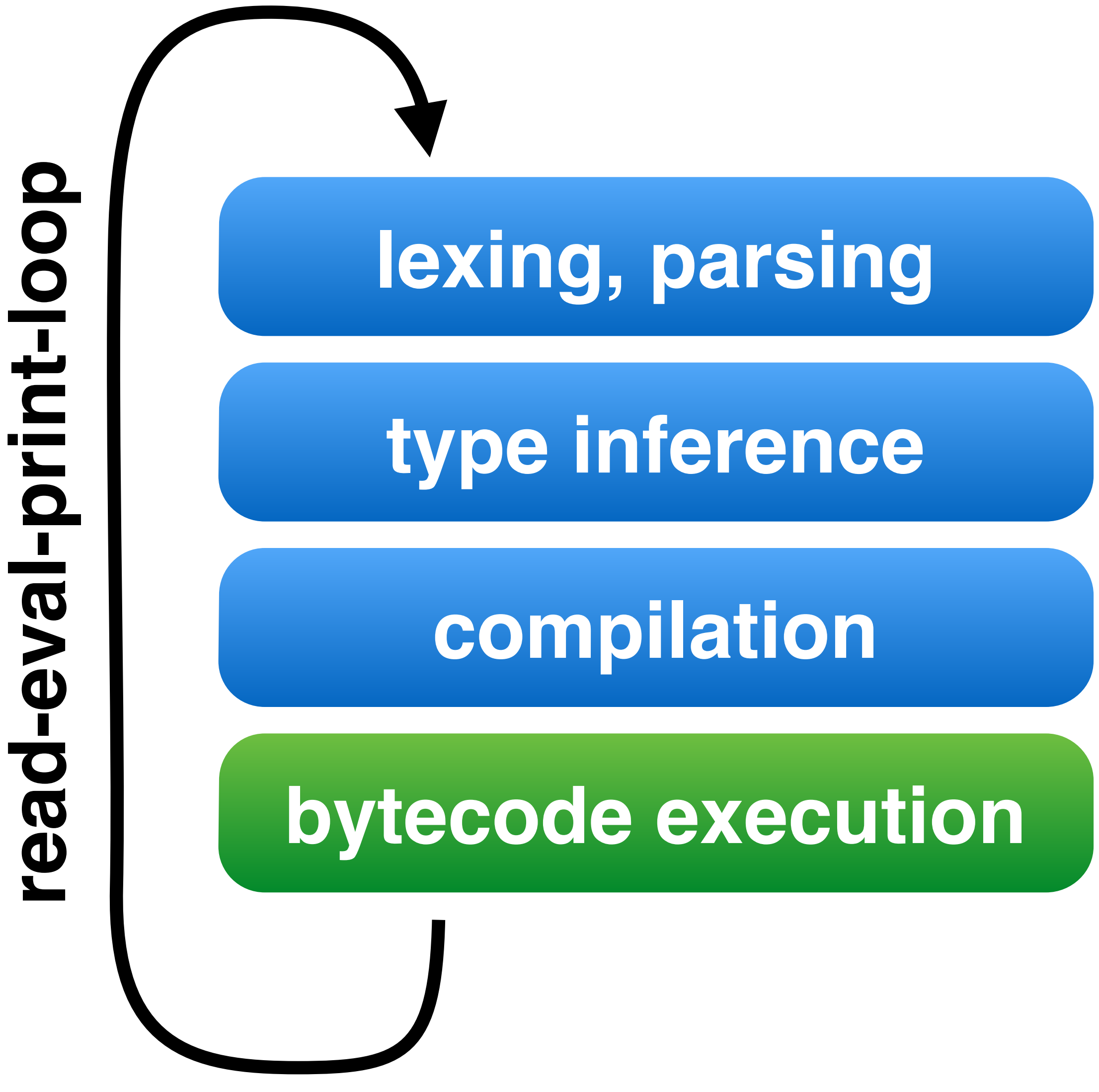
- used for type soundness proof and definition of divergence

Read-eval-print-loop semantics.

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

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:

Proved sound 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 an 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:

bc_inst	::=	Stack bc_stack_op | PushExc | PopExc
	|	Return | CallPtr | Call loc
	|	PushPtr loc | Jump loc | JumpIf loc
	|	Ref | Deref | Update | Print | PrintC char
	|	Label n | Tick | Stop
bc_stack_op	::=	Pop | Pops n | Shift n n | PushInt int
	|	Cons n n | El n | TagEq n | IsBlock n
	|	Load n | Store n | LoadRev n
	|	Equal | Less | Add | Sub | Mult | Div | Mod
loc	::=	Lab n | Addr n

Small-step semantics; values and state:

bc_value	::=	Number int | RefPtr n | Block n bc_value^*
	|	CodePtr n | StackPtr n
bc_state	::= {	stack : bc_value^*; refs : $n \mapsto bc\_value$;
		code : bc_inst^*; pc : n; handler : n;
		output : string; names : $n \mapsto$ string;
		clock : $n^?$ }

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:

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

Bytecode 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

for every exp env clock

there is some result

either: Result or TimeOut

$$\forall exp\ \ env\ \ clock.\ \exists res.\ (exp, env, \textsf{Some}\ clock) \Downarrow_{\text{ev}}\ res$$

produced by the semantics

Divergence

Evaluation diverges if

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

for all clock values

TimeOut happens

Compiler correctness proved in conventional forward direction:

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

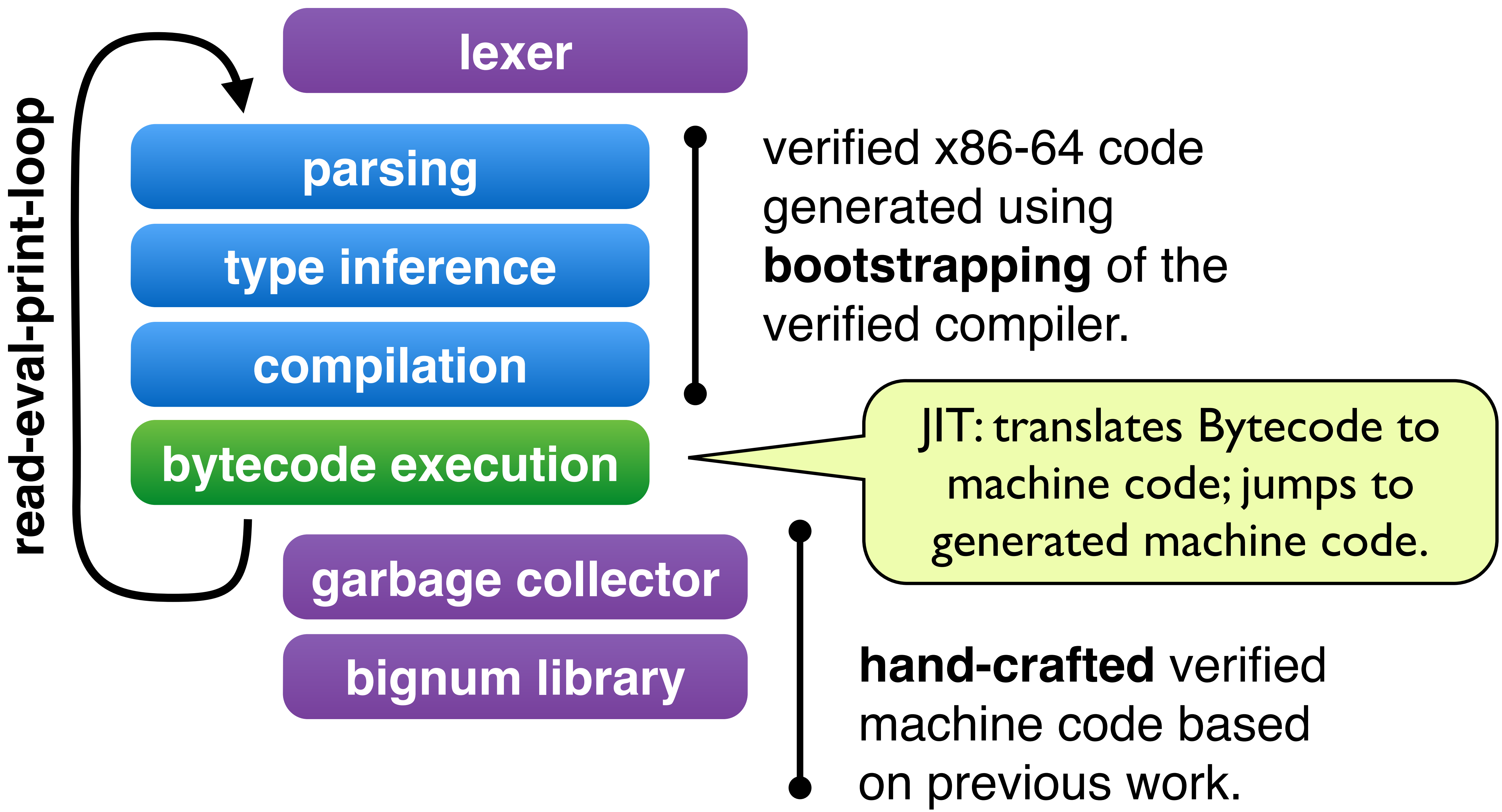
Bytecode has clock

… that stays in sync with CakeML clock

Theorem: bytecode diverges if and only if CakeML eval diverges

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

Verified x86-64 Implementation



Real executable also has 30-line unverified C wrapper.

Translation into x86-64

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 + …) xs

Correctness:

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

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

heap invariant / memory abstraction

Bootstrapping the verified compiler

Production of verified x86-64

parsing

type inference

compilation

function in logic: compile

by proof-producing synthesis [ICFP'12]

CakeML program (COMPILE) such that:

⊢ COMPILE *implements* compile

Proof by evaluation inside the logic:

⊢ compile-to-x64 COMPILE = x64-code

Compiler correctness theorem:

⊢ ∀prog. compile-to-x64 prog *implements* prog

Combination of theorems:

⊢ x64-code *implements* compile

Top-level theorem

Top-level specification:

$\mathsf{REPL}_s\ l\ inp\ res$

result output: list of strings that ends in either Terminate or Diverge

input string

Correctness theorem:

$$
\begin{array}{l}
\mathsf{temporal\ entire\_machine\_code\_implementation} \\
\quad (\mathsf{now}\ (\mathsf{init}\ inp\ aux) \Rightarrow \\
\qquad \mathsf{later}\ ((\exists l\ res.\ \mathsf{repl\_returns}\ (\mathsf{out}\ res)\ aux\ \wedge \\
\qquad\qquad\qquad (\mathsf{REPL}_s\ l\ inp\ res \wedge \mathsf{terminates}\ res)) \\
\qquad\qquad \vee \\
\qquad\qquad (\exists l\ res.\ \mathsf{repl\_diverged}\ (\mathsf{out}\ res)\ aux\ \wedge \\
\qquad\qquad\qquad (\mathsf{REPL}_s\ l\ inp\ res \wedge \neg \mathsf{terminates}\ res)) \\
\qquad\qquad \vee \\
\qquad\qquad \mathsf{now}\ (\mathsf{out\_of\_memory\_error}\ aux)))
\end{array}
$$

Numbers

Performance:

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

Effort:

~70k 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

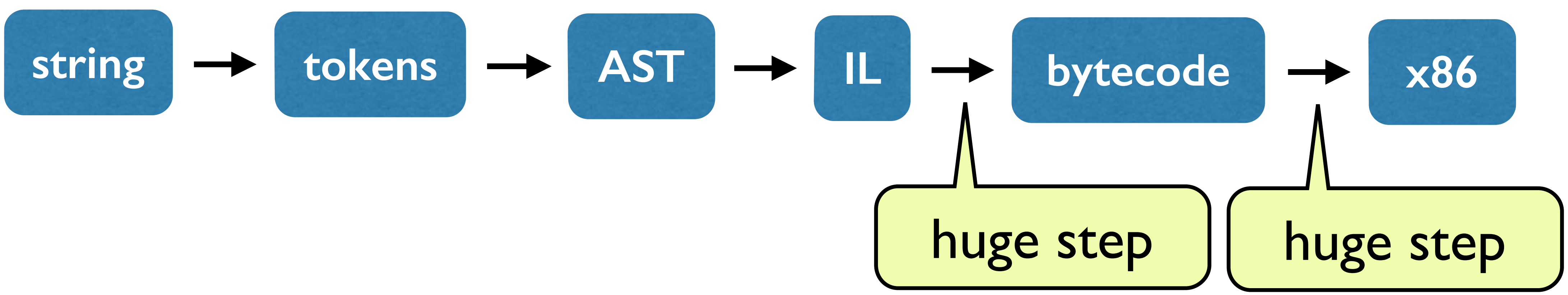
This talk

Part 1: verified implementation of CakeML

Part 2: current status, HOL light, future

Current status

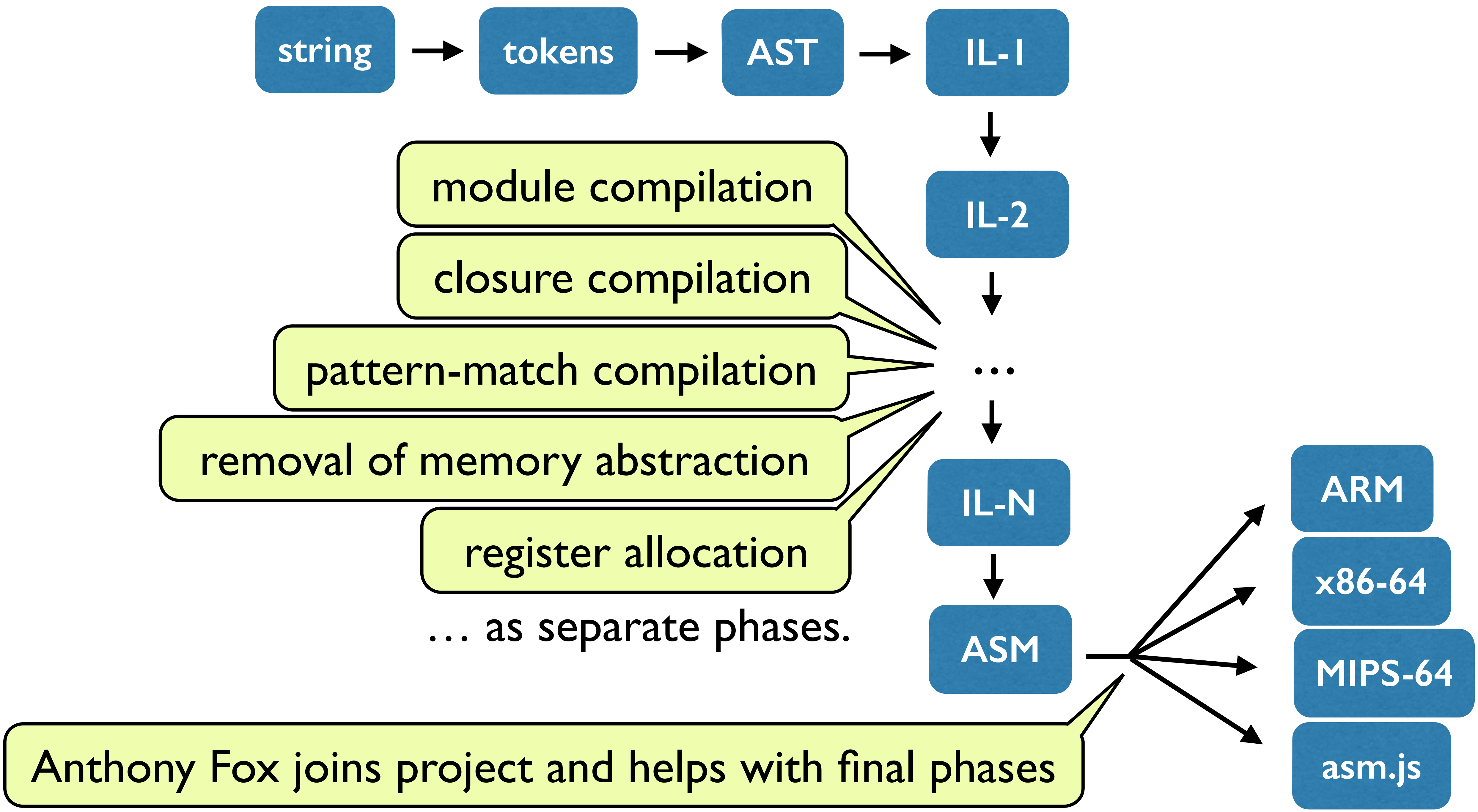
Current compiler:



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

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]
- future: integration with Arthur Charguéraud's characteristic formulae technology [ICFP'10, ICFP'11]

for developing cool verified examples.

Big example: verified HOL light

ML was originally developed to host theorem provers.

Aim: verified HOL theorem prover.

We have:

- 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?