Guest lecture for Compiler Construction, Spring 2015

Verified compilers

Magnus Myréen

Chalmers University of Technology

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

Guest lecture for Compiler Construction, Spring 2015

Verified compilers

What?

- Comes with a machine-checked proof that for any program, which does not generate a compilation error, the source and target programs behave identically. (Precise statement needs more details.)

(Sometimes called *certified* compilers, but that's misleading…)

Trusting the compiler

Bugs

When finding a bug, we go to great lengths to find it in our own code.

- Most programmers trust the compiler to generate correct code
- The most important task of the compiler is to generate correct code

Maybe it is worth the cost?

Cost reduction?

Establishing Compiler Correctness

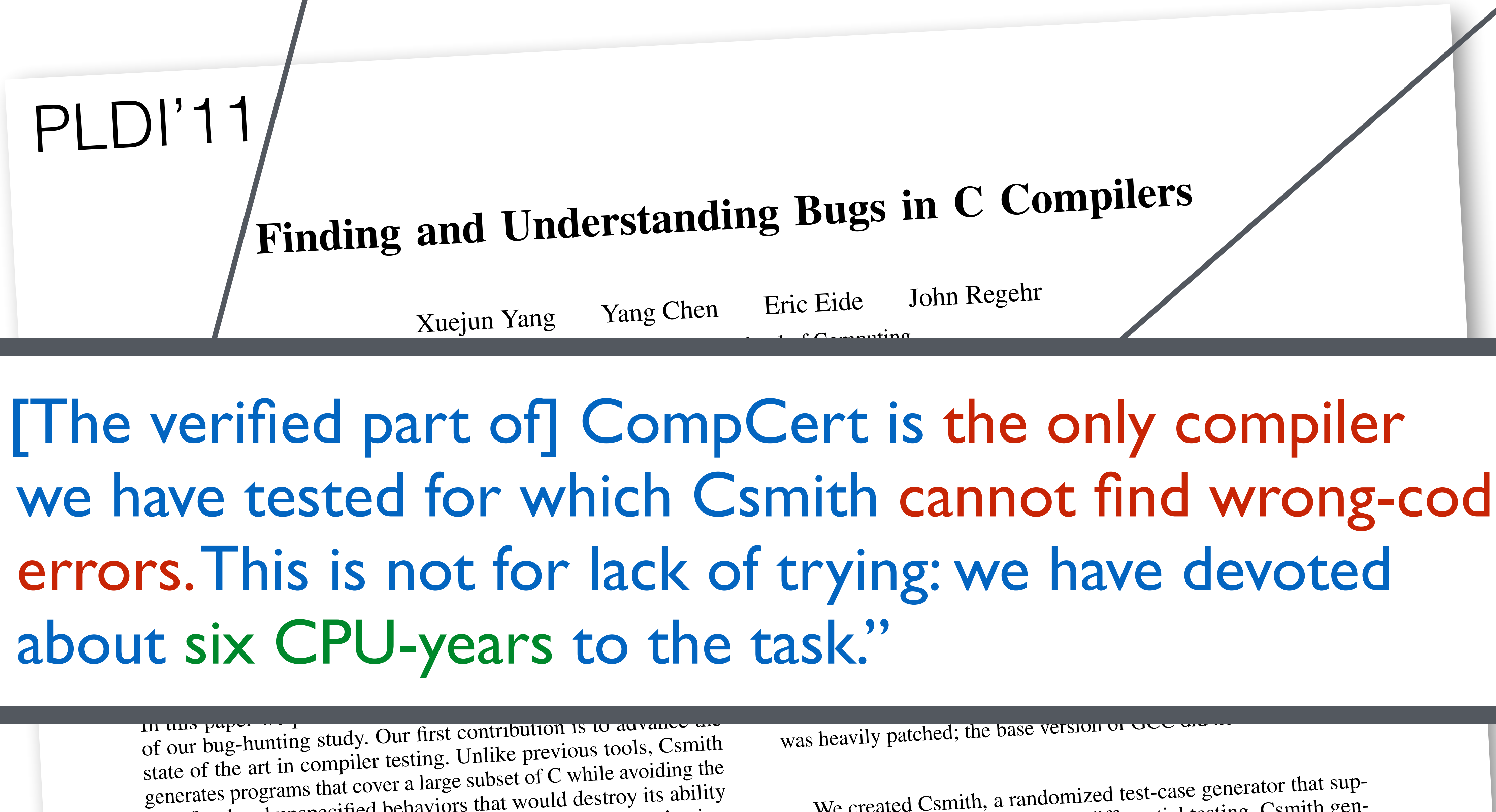
Alternatives

- Proving the correctness of a compiler is prohibitively expensive (however, see the CompCert project)
- Testing is the only viable option

… but with testing you never know you caught all bugs!

All (unverified) compilers have bugs

“ Every compiler we tested was found to crash and also to silently generate wrong code when presented with valid input. ”



PLDI’11

Finding and Understanding Bugs in C Compilers

Xuejun Yang Yang Chen Eric Eide John Regehr

of our bug-hunting study. Our first contribution is to advance the state of the art in compiler testing. Unlike previous tools, Csmith generates programs that cover a large subset of C while avoiding the

was heavily patched; the base version of GCC

We created Csmith, a randomized test-case generator that sup-

> “ [The verified part of] CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task.”

This lecture:
Verified compilers

What? Proof that compiler produces good code.

Why? To avoid bugs, to avoid testing.

How? By mathematical proof…

rest of
this lecture

Proving a compiler correct

like first-order logic, or higher-order logic

Ingredients:

- a formal logic for the proofs
- accurate models of
  - the source language
  - the target language
  - the compiler algorithm

proofs are only about things that live within the logic, i.e. we need to represent the relevant artefacts in the logic

a lot of details… (to get wrong)

Tools:

- a proof assistant (software)

… necessary to use mechanised proof assistant (think, '*Eclipse for logic*') to avoid mistakes, missing details

Accurate model of prog. language

Model of programs:

- syntax — what it looks like
- semantics — how it behaves

e.g. an *interpreter* for the syntax

Major styles of (operational, relational) semantics:

- big-step
- small-step

this style for structured source semantics

this style for unstructured target semantics

… *next slides provide examples.*

Syntax

Source:

```
exp = Num num
    | Var name
    | Plus exp exp
```

Target ‘machine code’:

```
inst = Const name num
     | Move name name
     | Add name name name
```

Target program consists of list of `inst`

Source semantics (big-step)

Big-step semantics as relation ↓ defined by rules, e.g.

$$\frac{}{\texttt{(Num n, env)} \downarrow \texttt{n}} \qquad \frac{\texttt{lookup s in env finds v}}{\texttt{(Var s, env)} \downarrow \texttt{v}}$$

$$\frac{\texttt{(x1, env)} \downarrow \texttt{v1} \qquad \texttt{(x2, env)} \downarrow \texttt{v2}}{\texttt{(Add x1 x2, env)} \downarrow \texttt{v1 + v2}}$$

called “big-step”: each step ↓ describes complete evaluation

Target semantics (small-step)

“small-step”: transitions describe parts of executions

We model the state as a mapping from names to values here.

```
step (Const s n) state = state[s ↦ n]
step (Move s1 s2) state = state[s1 ↦ state s2]
step (Add s1 s2 s3) state = state[s1 ↦ state s2 + state s3]

steps [] state = state
steps (x::xs) state = steps xs (step x state)
```

Compiler function

generated code stores result in register name (n) given to compiler

```
compile (Num k) n = [Const n k]

compile (Var v) n = [Move n v]

compile (Plus x1 x2) n =
   compile x1 n ++ compile x2 (n+1) ++ [Add n n (n+1)]
```

Relies on variable names in source to match variables names in target.

Uses names above n as temporaries.

Correctness statement

Proved using proof assistant — demo!

For every evaluation in the source …

for target state and k, such that …

```
∀x env res.
   (x, env) ↓ res ⇒
   ∀state k.
     (∀i env v. (lookup env i = SOME v) ⇒ (state i = v) ∧ i < k) ⇒
     (let state' = steps (compile x k) state in
        (state' k = res) ∧
        ∀i. i < k ⇒ (state' i = state i))
```

k greater than all var names and state in sync with source env …

… in that case, the result res will be stored at location k in the target state after execution

… and lower part of state left untouched.

Well, that example was simple enough...

But:

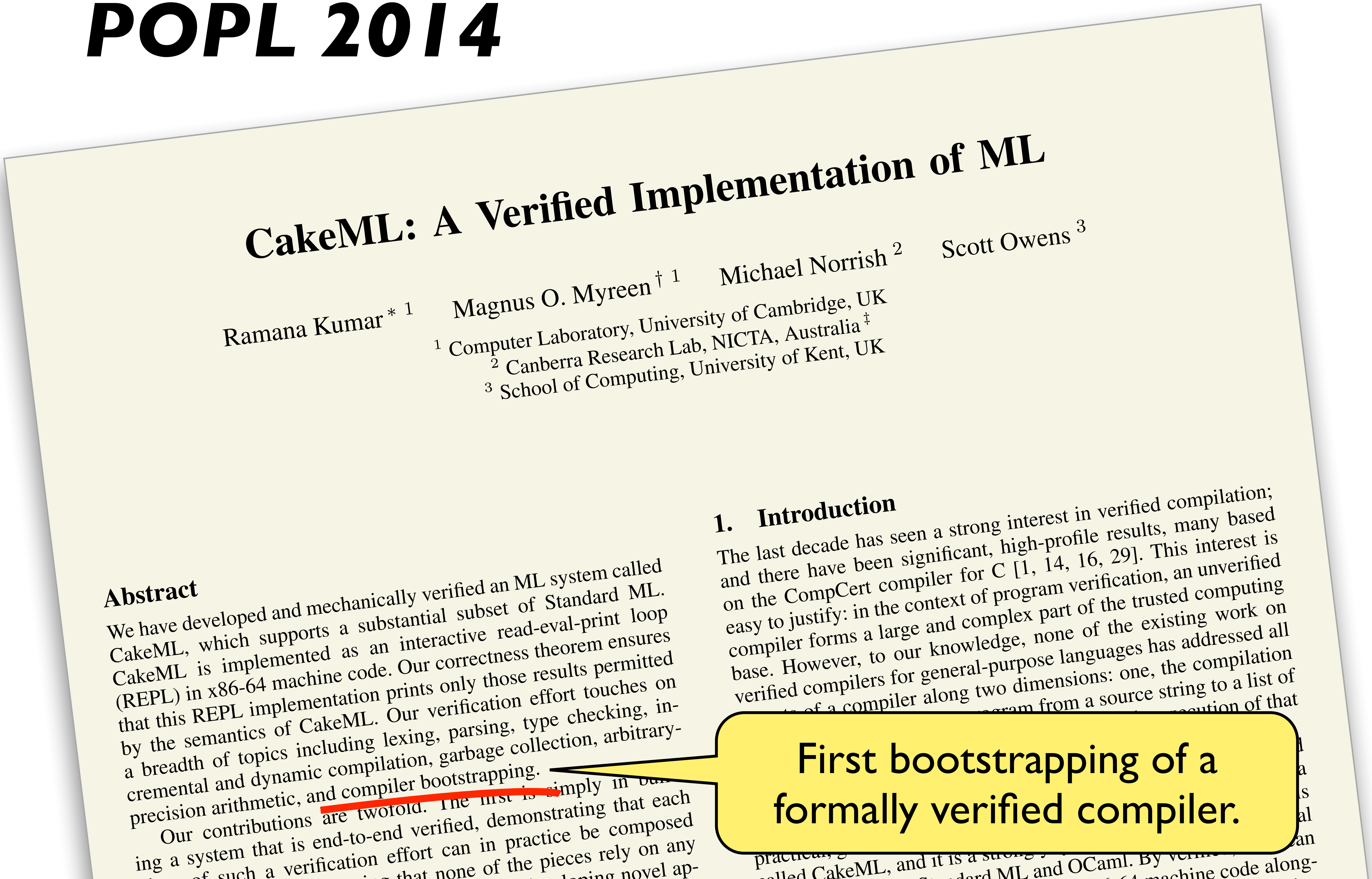
Some people say:

A programming language isn't real until it has a self-hosting compiler

Bootstrapping for verified compilers? Yes!

Scaling up…

POPL 2014



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 (REPL) in x86-64 machine code. Our correctness theorem ensures that this REPL implementation prints only those results permitted by the semantics of CakeML. Our verification effort touches on a breadth of topics including lexing, parsing, type checking, incremental and dynamic compilation, garbage collection, arbitrary-precision arithmetic, and compiler bootstrapping.

Our contributions are twofold. The first is simply in

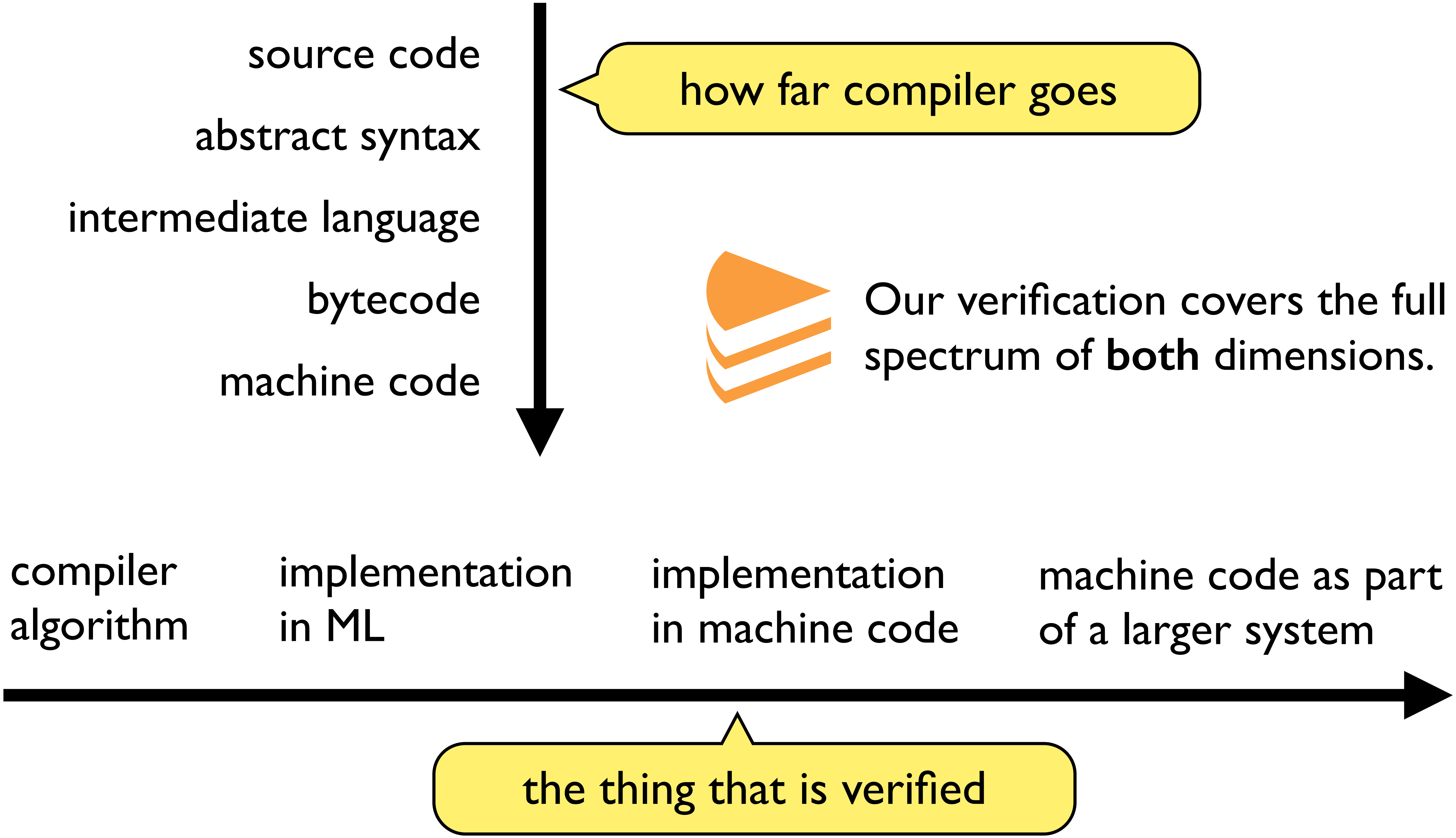
ing a system that is end-to-end verified, demonstrating that each

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 verified compilers for general-purpose languages has addressed all

First bootstrapping of a formally verified compiler.

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

machine code as part of a larger system

the thing that is verified

Idea behind in-logic bootstrapping

input: verified compiler function

Trustworthy code generation:

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

The CakeML at a glance

strict impure functional language

The CakeML 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

The verified machine-code implementation:

parsing, type inference, compilation, garbage collection, bignums etc.

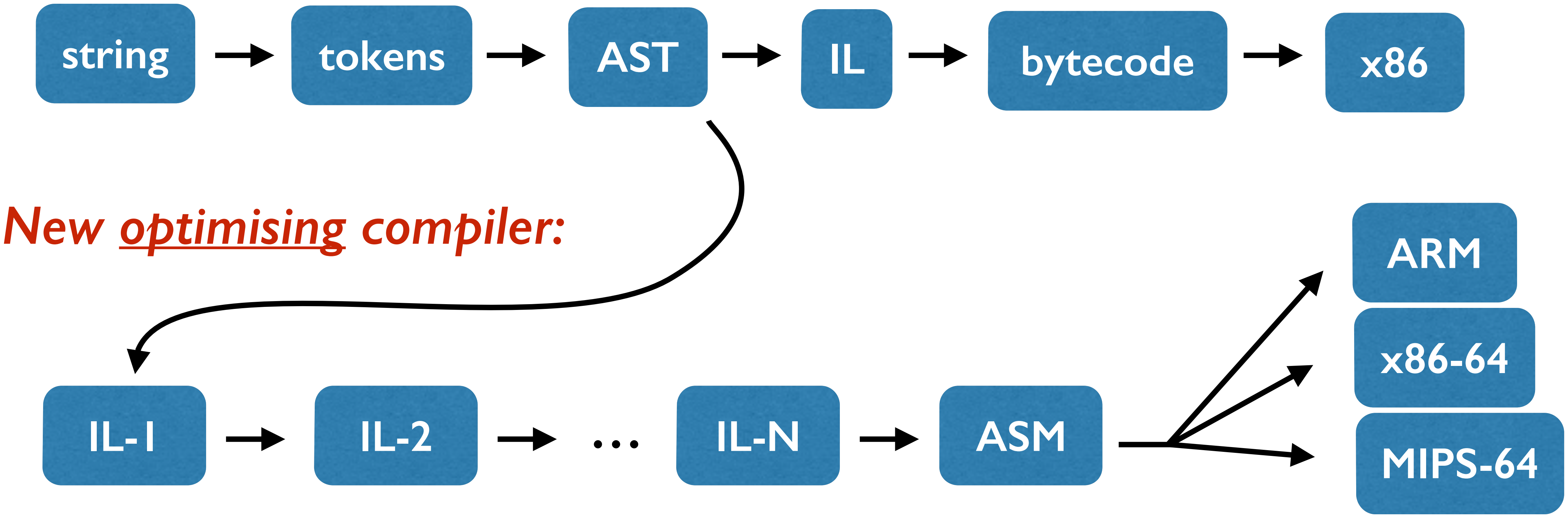
implements a read-eval-print loop (see demo).

The CakeML *compiler verification*

How?

Mostly standard verification techniques as presented in this lecture, but scaled up to large examples. (Four people, two years.)

Compiler:



string → tokens → AST → IL → bytecode → x86

New optimising compiler:

IL-1 → IL-2 → … → IL-N → ASM → ARM, x86-64, MIPS-64

… work in progress (want to join? myreen@chalmers.se)

Compiler verification summary

Ingredients:

- a formal logic for the proofs
- accurate models of
  - the source language
  - the target language
  - the compiler algorithm

Tools:

- a proof assistant (software)

Method:

- (interactively) prove a simulation relation

Questions? Interested?