Compiler Construction, Spring 2020

Verified compilers

Magnus Myreen

Chalmers University of Technology

Mentions joint work with Anthony Fox, Ramana Kumar, Michael Norrish, Scott Owens, Thomas Sewell, Yong Kiam Tan and many more (incl. local MSc students)

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

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

Your program crashes.

Where do you look for the fault?

- Do you look at *your source code*?
- Do *look at the code for the compiler that you used*?

users want to rely on compilers

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

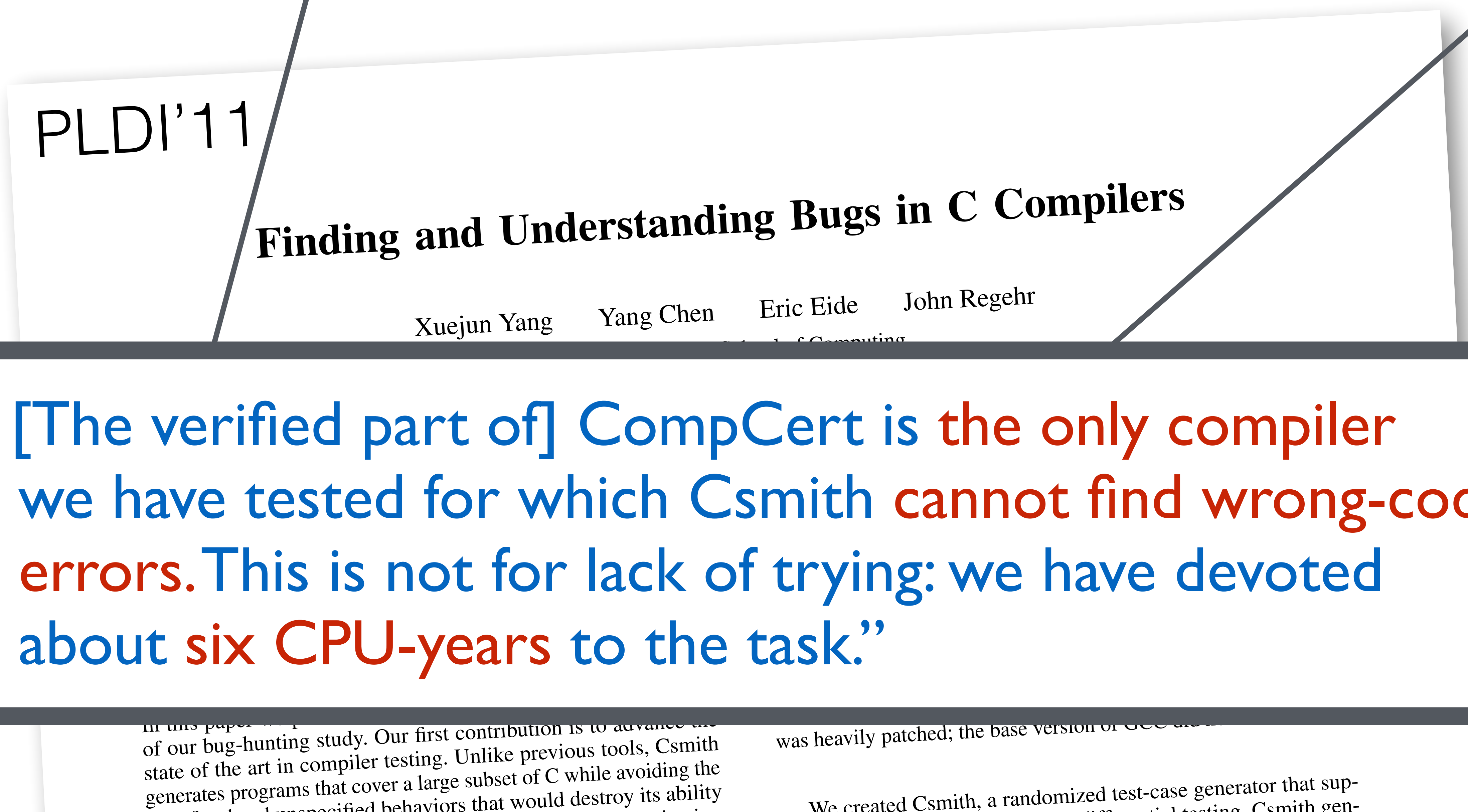
Alternatives

- Proving the correctness of a compiler is prohibitively expensive
- 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;

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."

Motivations

Bugs in compilers are not tolerated by users.

Bugs can be hard to find by testing.

Verified compilers must be used in order for verification of source-level programs to imply guarantees at the level of verified machine code.

Research question: how easy (cheap) can we make compiler verification?

This lecture:

Verified compilers

What? Prove 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 accidentally skipping 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 — this style for structured source semantics
- small-step — 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.

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

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

called "big-step": each step ⇓ describes complete evaluation

Source semantics (…gone wrong)

Real-world semantics are not always clean:

https://www.destroyallsoftware.com/talks/wat

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

```
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)]
```

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

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!

```
∀x env res.
  (x, env) ↓ res ⇒
  ∀state k.
    (∀i 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))
```

For every evaluation in the source …

for target state and k, such that …

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.

Code for the demo:

```
open HolKernel Parse boolLib bossLib lcsymtacs stringTheory combinTheory
     arithmeticTheory finite_mapTheory pairTheory;

val _ = new_theory "demo";

Type name = ``:num``;


(* -- SYNTAX -- *)

(* source *)

Datatype:
  exp = Num num
      | Var name
      | Plus exp exp
End

(* target *)

Datatype:
  inst = Const name num
       | Move name name
       | Add name name name
End

(* -- SEMANTICS -- *)

(* source *)

Inductive eval:
  (T
   ⇒
   eval (Num n, env) n)
  ∧
  ((FLOOKUP env s = SOME v)
   ⇒
   eval (Var s, env) v)
  ∧
  (eval (x1,env) v1 ∧ eval (x2,env) v2
   ⇒
   eval (Plus x1 x2, env) (v1+v2))
End

(* target *)

Definition step_def:
  step (Const s n) state = (s =+ n) state ∧
  step (Move s1 s2) state = (s1 =+ state s2) state ∧
  step (Add s1 s2 s3) state = (s1 =+ state s2 + state s3) state
End

Definition steps_def:
  steps [] state = state ∧
  steps (x::xs) state = steps xs (step x state)
End

(* -- COMPILER -- *)

Definition compile_def:
  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)]
End

(* verification proof *)

Theorem steps_append[simp]:
  ∀xs ys state. steps (xs ++ ys) state = steps ys (steps xs state)
Proof
  Induct \\ fs [steps_def]
QED

Theorem eval_ind = eval_ind |> Q.SPEC ‘λ(x,y) z. P x y z’
                            |> SIMP_RULE (srw_ss()) [FORALL_PROD] |> GEN_ALL;

Theorem compile_correct:
  ∀x env res.
    eval (x, env) res ⇒
    ∀k state.
      (∀i v. (FLOOKUP 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)
Proof
  ho_match_mp_tac eval_ind \\ rpt strip_tac \\ fs [LET_DEF]
  \\ fs [compile_def,steps_def,step_def]
  \\ fs [APPLY_UPDATE_THM] \\ res_tac
  \\ last_x_assum imp_res_tac \\ fs []
  \\ first_x_assum (qspecl_then [`k+1`,`steps (compile x k) state`] mp_tac)
  \\ impl_tac \\ rw [] \\ res_tac \\ fs []
QED

val _ = export_theory();
```

Well, that example was simple enough…

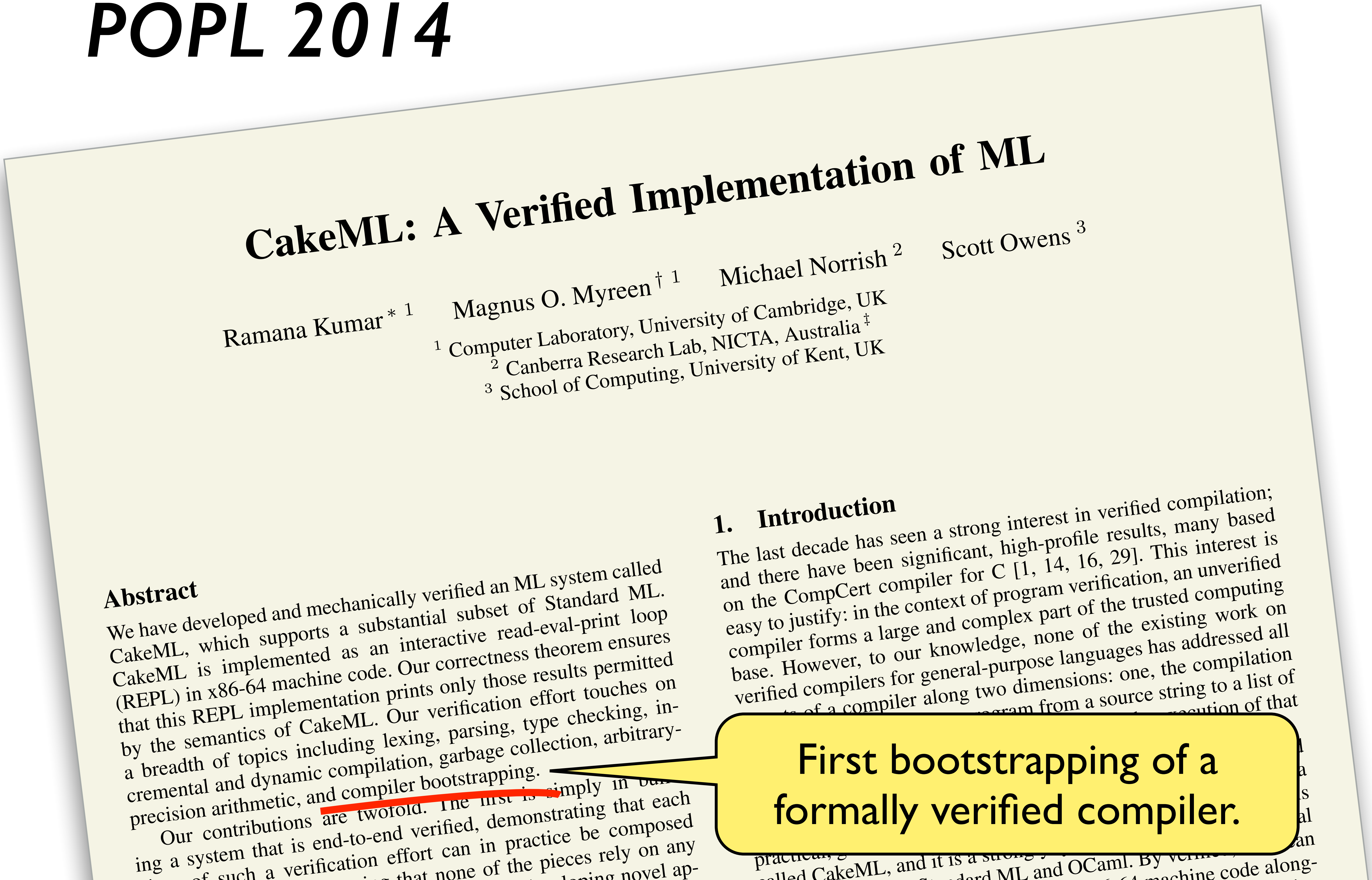
But:

> **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 build- ing a system that is end-to-end verified, demonstrating that each [illegible] of such a verification effort can in practice be composed [illegible] that none of the pieces rely on any [illegible]

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 aspects of a compiler along two dimensions: one, the compilation of a program from a source string to a list of [illegible] execution of that [illegible]

[illegible] called CakeML, and it is a strong [illegible] Standard ML and OCaml. By verif[illegible] x86-64 machine code along-

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,ICFP'16]

x86-64 machine code (deep embedding)

output: verified implementation of compiler function

The CakeML at a glance

The CakeML language
≈ Standard ML without functors

strict impure functional language

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.

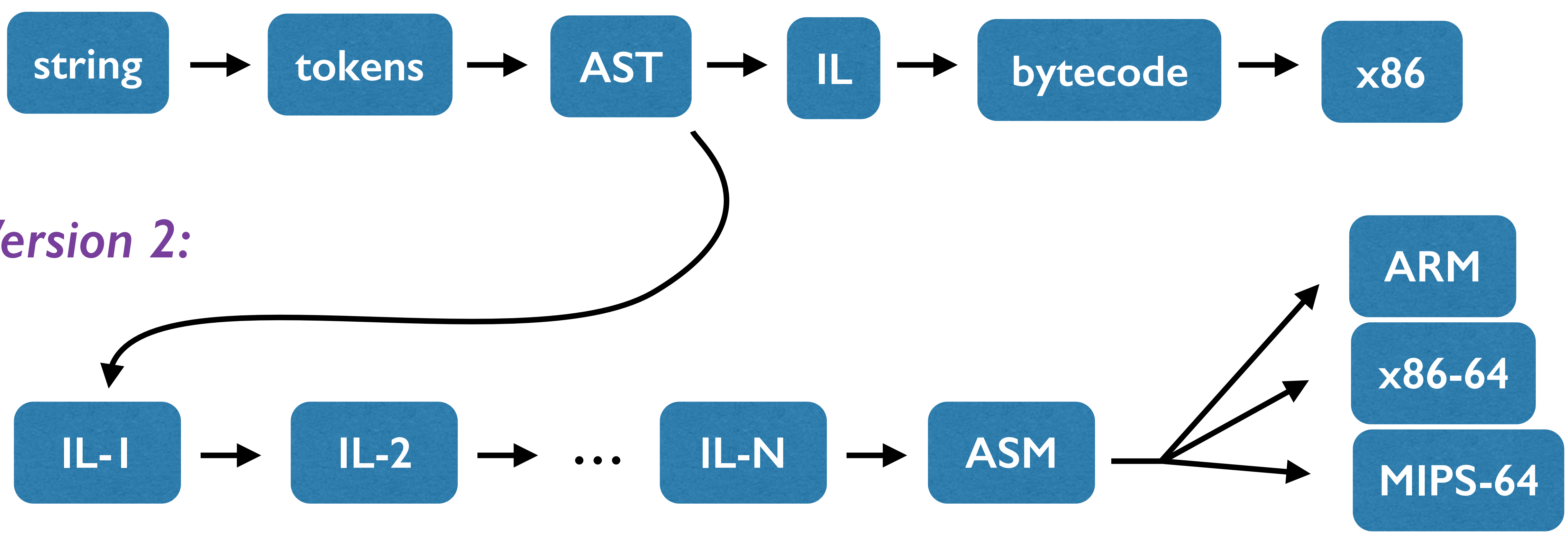
The CakeML compiler verification

How?

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

Version 1:

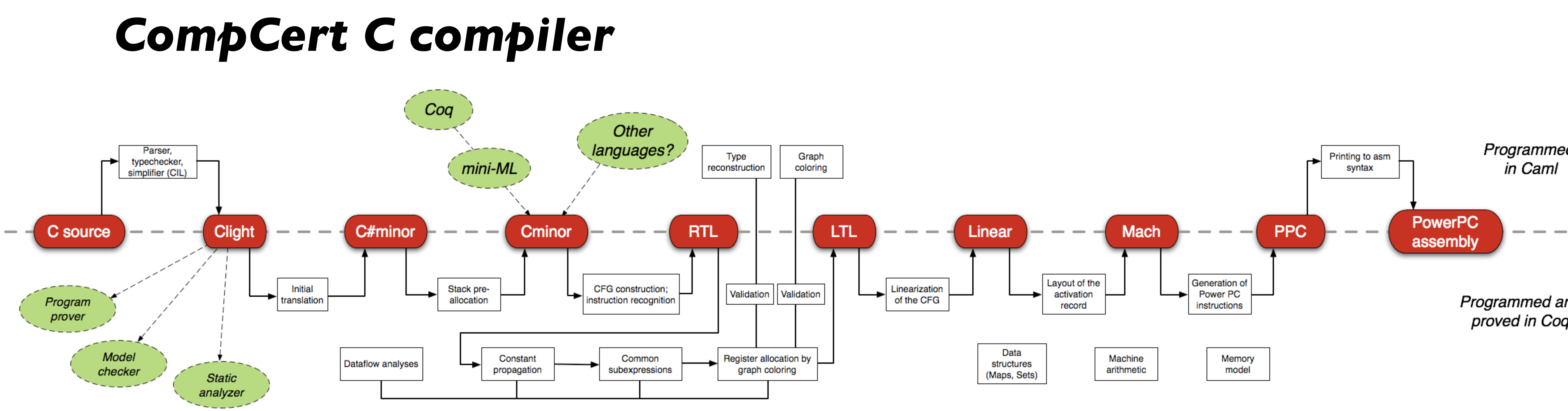
Version 2:



… actively developed (want to join? myreen@chalmers.se)

State of the art

CompCert



Leroy et al. ***Source:*** http://compcert.inria.fr/

Compiles C source code to assembly.

Has **good performance numbers**

Proved correct in Coq.

http://compcert.inria.fr/

CakeML compiler

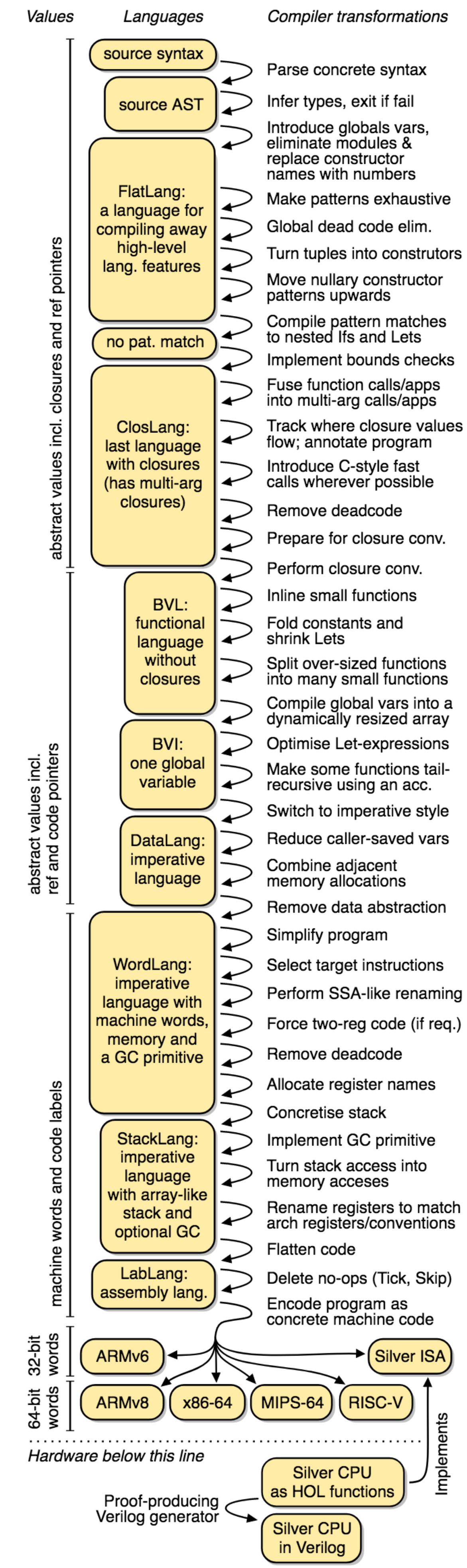
Compiles CakeML concrete syntax to machine code.

Proved correct in HOL4.

Has **mostly good performance numbers** (later lecture)

> Known as the first verified compiler to be bootstrapped.

I'm one of the six developers behind version 2 (diagram to the right).



larger at https://cakeml.org

A spectrum

robust, inflexible

proved to always work correctly

more flexible, but can be fragile

produces a proof for each run

Verified compilers … *Proof-producing compilers*

Pilsner

CompCert C compiler

CakeML compiler

CompCertTSO

Fiat

Cogent

Translation validation for a verified OS kernel

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? — for projects on this, email **myreen@chalmers.se**