Compiler Construction, Spring 2018

Verified compilers

Magnus Myreen

Chalmers University of Technology

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

---

Compiler Construction, Spring 2018

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?

Establishing compiler correctness

Cost reduction?

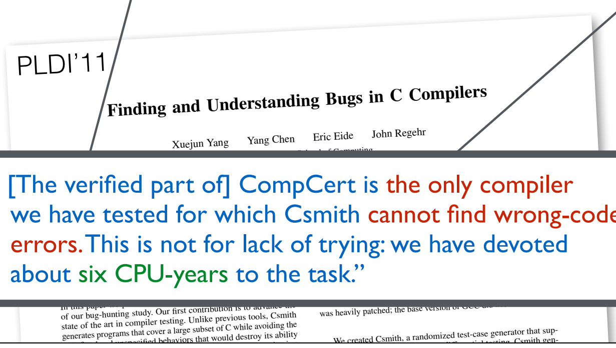
Alternatives

- 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. "



" [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 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

Ingredients:

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

Tools:

- a proof assistant (software)

like first-order logic, or higher-order logic

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)

… 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
- 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.

```
                          lookup s in env finds v
-------------------      ------------------------
(Num n, env) ↓ n          (Var s, env) ↓ v


   (x1, env) ↓ v1          (x2, env) ↓ v2
   -------------------------------------
        (Add x1 x2, env) ↓ 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:

Well, that example was simple enough...

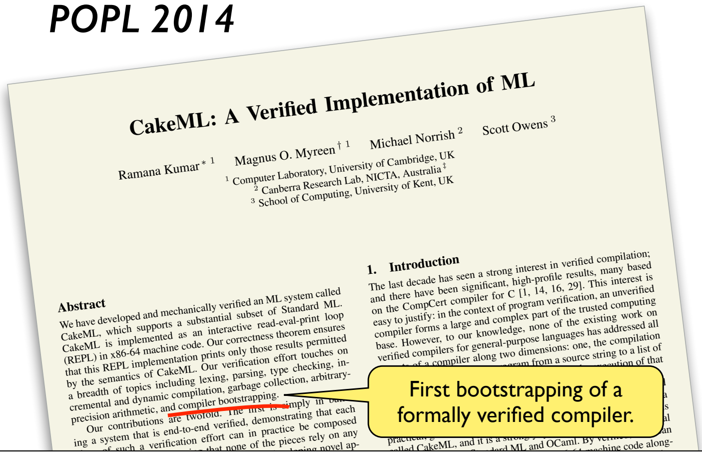
But:

> **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



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 I/O or 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.

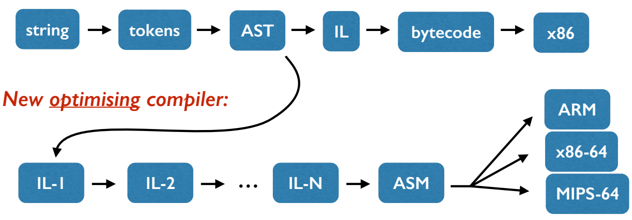
implements a read-eval-print loop (see demo).

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:

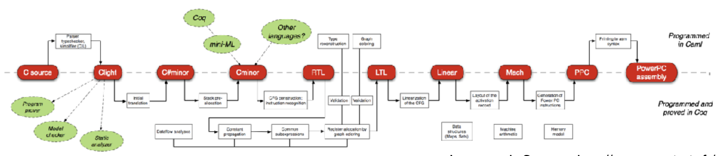


... actively developed (want to join? myreen@chalmers.se)

State of the art

CompCert

CompCert C compiler



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.

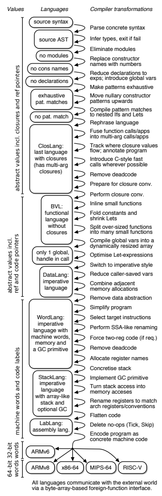
Has **mostly good performance numbers** (later lecture)

Known as the first verified compiler to be bootstrapped.

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

larger on website

https://cakeml.org/



A spectrum

robust, inflexible

proved to always work correctly

Verified compilers ... *Proof-producing compilers*

more flexible, but can be fragile

produces a proof for each run

Pilsner

CompCert C compiler

CakeML compiler

CompCertTSO

Fiat

Cogent

Translation validation for a verified OS kernel

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? — contact me regarding MSc projects on this topic