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?

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."

Mentions joint work with Anthony Fox, Ramana Kumar, Michael Norrish, Scott Owens, Yong Kiam Tan and many more (incl. Chalmers/GU MSc students)
This lecture: **Verified compilers**

What? Prove that compiler produces good code.
Why? To avoid bugs, to avoid testing.
How? By mathematical proof…

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)

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

… next slides provide examples.

Syntax

Source:

\[
\text{exp} = \text{Num num} \\
| \text{Var name} \\
| \text{Plus exp exp}
\]

Target 'machine code':

\[
\text{inst} = \text{Const name num} \\
| \text{Move name name} \\
| \text{Add name name name}
\]

Target program consists of list of inst

Source semantics (big-step)

Big-step semantics as relation \( \downarrow \) defined by rules, e.g.

\[
\text{(Num n, env)} \downarrow n \\
\text{(Var s, env)} \downarrow v
\]

\[
\text{(x1, env)} \downarrow v1 \\
\text{(x2, env)} \downarrow v2
\]

\[
\text{(Add x1 x2, env)} \downarrow v1 + v2
\]

called "big-step": each step \( \downarrow \) 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.

\[
\begin{align*}
\text{step (Const } s \ n) \text{ state } &= \text{state[s} \rightarrow n]\text{,} \\
\text{step (Move } s1 \ s2) \text{ state } &= \text{state[s}1 \rightarrow \text{state } s2]\text{,} \\
\text{step (Add } s1 \ s2 \ s3) \text{ state } &= \text{state[s}1 \rightarrow \text{state } s2 + \text{state } s3]\text{.} \\
\text{steps [] state } &= \text{state} \\
\text{steps } (x::xs) \text{ state } &= \text{steps } xs \text{ (step } x \text{ state)}
\end{align*}
\]

Compiler function

- 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!*

For every evaluation in the source …

\[
\forall x, env, res. \exists v. \forall i. i < k \quad \exists \text{state' } \text{ such that } \text{eval} \text{ (x, env) res } \\
\text{in target state and k, such that …}
\]

\[
\begin{align*}
\text{vi v. (lookup env i = \text{SOME } v} & \Rightarrow \text{ (state } i = v) \wedge i < k) \\
\text{(let state'} = \text{steps } (\text{compile } x k) \text{ state in } \\
\text{(state'} k = \text{res} ) \land \\
\text{vi. } i < k = \text{(state'} i = \text{state } i))
\end{align*}
\]

\[
\begin{align*}
\text{… in that case, the result res will be stored at location } k \text{ in the target state after execution} \\
\text{… and lower part of state left untouched.}
\end{align*}
\]

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!

POPL 2014
Dimensions of Compiler Verification

- source code
- abstract syntax
- intermediate language
- bytecode
- machine code

how far compiler goes

implementation in ML
implementation in machine code
machine code as part of a larger system

the thing that is verified

Our verification covers the full spectrum of both dimensions.

The CakeML at a glance

The CakeML language
≈ Standard ML without 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.)

Version 1:
string → tokens → AST → IL → bytecode → x86

Version 2:
IL-1 → IL-2 → ... → IL-N → ASM → x86-64 → MIPS-64

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

State of the art

CompCert

Compiles C source code to assembly.
Has good performance numbers
Proved correct in Coq.

CompCert C compiler

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

http://compcert.inria.fr/
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).

A spectrum of compilers:

- **Verified compilers**
  - Pilsner
  - CompCert C compiler
  - CakeML compiler
  - CompCertTSO

- **Proof-producing compilers**
  - Fiat
  - Cogent

- Translation validation for a verified OS kernel

**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

Summary