Verified compilers

Magnus Myréen
Chalmers University of Technology

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

Guest lecture for Compiler Construction, Spring 2015
Version control

I highly recommend that you use version control software. Using version control software is an essential practice when developing code. However, do not put your code in a public repository, where others can see your code.

Testing compilers

Trusting the compiler

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

Establishing Compiler Correctness

Alternatives

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

Maybe it is worth the cost?

Cost reduction?

... 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.”
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

Ingredients:
• a formal logic for the proofs
• accurate models of
  • the source language
  • the target language
  • the compiler algorithm

Tools:
• a proof assistant (software)

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 mistakes, missing details.

Like first-order logic, or higher-order logic.
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:**

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

**Target ‘machine code’:**

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

Target program consists of list of \text{inst}
Source semantics (big-step)

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

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

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

\[
(x_1, \text{env}) \downarrow v_1
\]

\[
(x_2, \text{env}) \downarrow v_2
\]

\[
(\text{Add } x_1 x_2, \text{env}) \downarrow v_1 + v_2
\]

called “big-step”: each step \( \downarrow \) 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.

\[
\begin{align*}
\text{step (Const } s \ n) \ \text{state} &= \text{state}[s \mapsto n] \\
\text{step (Move } s_1 \ s_2) \ \text{state} &= \text{state}[s_1 \mapsto \text{state } s_2] \\
\text{step (Add } s_1 \ s_2 \ s_3) \ \text{state} &= \text{state}[s_1 \mapsto \text{state } s_2 + \text{state } s_3]
\end{align*}
\]

\[
\begin{align*}
\text{steps } [] \ \text{state} &= \text{state} \\
\text{steps } (x::xs) \ \text{state} &= \text{steps } xs \ (\text{step } x \ \text{state})
\end{align*}
\]
Compiler function

\[
\text{compile (Num k) } n = [\text{Const } n \ k]
\]

\[
\text{compile (Var v) } n = [\text{Move } n \ v]
\]

\[
\text{compile (Plus x1 x2) } n = \\
\phantom{\text{compile (Plus x1 x2) } n =} \text{compile x1 } n +\text{compile x2 } (n+1) + [\text{Add } n \ n \ (n+1)]
\]

- Uses names above \( n \) as temporaries.
- Relies on variable names in source to match variables names in target.
- Generated code stores result in register name \( (n) \) given to compiler.
Correctness statement

Proved using proof assistant — demo!

For every evaluation in the source ...

∀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))

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.
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!
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 the CakeML system is ultimately x86-64 machine code alongside a mechanically checked theorem in higher-order logic saying that running that machine code causes an input program to yield the expected result. By verified, we mean that the CakeML system is ultimately x86-64 machine code along-side a mechanically checked theorem in higher-order logic saying that running that machine code causes an input program to yield the expected result.

Our contributions are twofold. The first is simply in building a system that is end-to-end verified, demonstrating that each component of such a verification effort can in practice be composed in a way that none of the pieces rely on any assumptions about the others. The other contribution is that we synthesize a CakeML implementation of the compiler from that using our previous technique [22], which puts the compiler on an equal footing with other CakeML programs.

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 of the following properties:

- Bootstrapping: The original motivation for ML was to verify practical programs, and it is a source of frustration that traditional ML compilers are not bootstrapped—i.e., they are not generated from a trusted source, usually a collection of ML programs.
- Incremental compilation: Many real-world ML programs are edited incrementally, so it is important that parsers and other components can be updated without having to recompile the whole compiler.
- Dynamic compilation: Language features such as closures can make it difficult to generate a statically typed compiler for ML.
- Garbage collection: ML programs may contain garbage, so it is important that the compiler does not rely on any assumptions about the input programs.

CakeML addresses all of these issues by verifying a system that is end-to-end verified, demonstrating that each component of such a verification effort can in practice be composed in a way that none of the pieces rely on any assumptions about the others. The second is developing novel approaches to some of the more challenging aspects of the verification. In particular, our formally verified compiler can bootstrapped a system that is end-to-end verified, demonstrating that each component of such a verification effort can in practice be composed in a way that none of the pieces rely on any assumptions about the others. The second is developing novel approaches to some of the more challenging aspects of the verification. In particular, our formally verified compiler can bootstrapped a system that is end-to-end verified, demonstrating that each component of such a verification effort can in practice be composed in a way that none of the pieces rely on any assumptions about the others.
Dimensions of Compiler Verification

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

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

Trust thy code generation:

1. functions in HOL (shallow embedding)
2. proof-producing translation [ICFP'12, JFP'14]
3. CakeML program (deep embedding)
4. verified compilation of CakeML [POPL'14]
5. 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:**

- 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?**