Establishing compiler correctness

What you have to do

Alternatives

• to the test suite; this is known as for testing the compiler

They have a big collection of example programs which are used

Proving the correctness of a compiler is prohibitively expensive

BNFC takes care of lexing and parsing, however, you will have to

BNFC file for

BNFC

Verifying compilers

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

Verifying 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…)

Verifying compilers

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

“[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)

… 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 — this style for structured source semantics
• small-step — this style for unstructured target semantics

… next slides provide examples.

Syntax

Source:

\[
\text{exp} = \text{Num } n \mid \text{Var } n \mid \text{Plus } \exp \exp
\]

Target 'machine code':

\[
\text{inst} = \text{Const } n \mid \text{Move } n \mid \text{Add } n n n
\]

Target program consists of list of inst

Source semantics (big-step)

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

\[
\begin{align*}
\text{(Num } n, \text{ env)} & \downarrow n \\
\text{(Var } s, \text{ env)} & \downarrow v \\
\text{(Add } x_1 x_2, \text{ env)} & \downarrow v_1 + v_2 \\
\end{align*}
\]

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.

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

Compiler function

\[
\text{compile } (\text{Num } k) n = [\text{Const } n k] \\
\text{compile } (\text{Var } v) n = [\text{Move } n v] \\
\text{compile } (\text{Plus } x_1 x_2) n = \\
\text{compile } x_1 n +\!
\text{compile } x_2 (n\!+\!1) +\!
\text{[Add } n n (n\!+\!1)\text{]}
\]

Uses names above n as temporaries.

Correctness statement

\[
\forall x \text{ env res. } \begin{align*}
\text{ (x, env)} & \downarrow \text{res} \\
\forall \text{state } k. \\
& \begin{align*}
\forall i \text{ env v. } (\text{lookup env } i = \text{SOME } v) \\
& \land (\text{state } i = v) \\
& \land i < k
\end{align*}
\end{align*}
\]

For every evaluation in the source ...

v x env res. \\
(x, env) \downarrow res \\
\forall \text{state } k. \\
\forall i < k \\
\land (\text{state'} i = \text{res}) \\
\land (\text{state'} k = \text{state } k)

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

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

… and lower part of state left untouched.

Proved using proof assistant — 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!

Dimensions of Compiler Verification

<table>
<thead>
<tr>
<th>source code</th>
<th>abstract syntax</th>
<th>intermediate language</th>
<th>bytecode</th>
<th>machine code</th>
</tr>
</thead>
<tbody>
<tr>
<td>compiler algorithm</td>
<td>implementation in ML</td>
<td>implementation in machine code</td>
<td>machine code as part of a larger system</td>
<td></td>
</tr>
</tbody>
</table>

how far compiler goes

Our verification covers the full spectrum of both dimensions.

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

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 → x86-64

… actively developed (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?