Formal Verification of Compilers

Zoom on the CompCert project

Arthur Correnson

What could go wrong when we compile a program?

What could go wrong when we compile a program ?

syntax error

internal compiler error

compiler panics

type errors

malformed programs

compiler's stuck and start over

misscompilation

print("hello")

compiler

bin.exe

rm -rf *

What can we do to fix this?

What can we do to fix this?

How to increase our trust in compilers

simple languages ✓ 1

semantics

base language → trivial implementation

How the targets works ? 2

Use only selected instructions

Structured approach "à-la Rust". 3

foundations

marking components of the compiler that are unsafe.

Verify the compiler

↓

Formal proof that the compiler actually compiles the input programs as expected.

Coq

↳ What is the actual theorem we want to prove ?

Standardisation of Programming Languages

Standardisation of Programming Languages

Syntax
Semantics
} Scheme

C standard
↳ Document to explain how the compiler should work

Document to describe the language at a higher level
→ not formal
→ intuitive
→ describe the way a compiler for the language should work

Unifying the ecosystem around your language

Formalisation of Programming Languages

Formalisation of Programming Languages

Semantics

- well defined
- meaning of the language
- Mathematical definition of what it means to execute a program.
- like a standard, but written in Math

Wasm comes with a formal semantic

Type of semantics

$$\boxed{\begin{aligned} expr \;::=\;& \bar{1} \mid \bar{2} \mid \dots \\ &\mid expr \mathbin{\bar{+}} expr \\ &\mid expr \mathbin{\bar{\times}} expr \\ &\vdots \end{aligned}}$$

symbols

denotational semantics

$$[\![\_]\!] : expr \longrightarrow \mathbb{N}$$

$$\bar{n} \longmapsto n$$

$$e_1 \mathbin{\bar{+}} e_2 \longmapsto [\![e_1]\!] + [\![e_2]\!]$$

$stmt ::= \text{while } expr \text{ do } stmt$ $\qquad [\![\text{while } e \text{ do } c]\!] \longmapsto ?$

Operational semantics :

- Relation between program and states.

state

$(s, p) \longrightarrow (s', p')$ $\quad \rightarrow$ relation

if I execute p in s, this gives me new state s' and I still have p' to execute

if $(s, c_1) \rightarrow (s', c_1')$ then $(s, c_1; c_2) \rightarrow (s', c_1'; c_2)$

The challenges of verifying a compiler

The challenges of verifying a compiler

(The example of CompCert, a formally verified C compiler)

compiler : program ⟶ assembly

Theorem compiler_correct : "the function compiler is correct".

For any env of execution,
any input program p :

if p has a behavior in env according to the semantic of our language
compile(p) has the same behavior in env according to assembly semantic,

Non determinism ?

$a + b$? first eval a ?
eval b ?

$(i{+}{+}) + ({+}{+}i)$?

if your semantic is not deterministic,
all order of eval. are "ok"

1 challenge : find the specification we want to prove

2 challenge : how to carry the proof ?

→ Use a proof assistant (Coq)

→ Develop the compiler in Coq

3 challenge : • implement the compiler in a purely functional style

• every function should terminate.

⋮

Time consuming

A compiler written in Rust/Java/C++ cannot be translated easily in Coq. We have to design our compiler in a specific way.

Token

↓

Parse Tree

↓

Typed tree

↓

IR ↺ optimisation

↓

bin

To do the proof in CompCert:

- decompose the pipeline into a <u>lot</u> of IR

- rely on external programs for efficiency and for algos that are too complicated to prove.

  → still work if we can <u>check</u> in Coq that outputs are correct

  → example: register allocation / graph colouring.