x86 verification from scratch

Mario Carneiro

Carnegie Mellon University

September 8, 2019

The challenge

Any software verification tool worth its salt should be able to prove itself correct.

The challenge

Any software verification tool worth its salt should be able to prove itself correct.*

From scratch?

Rule #1 of formalization

Don't try to formalize anything that wasn't designed to be formalized.

- It's possible, but it forces you into a lot of bad design decisions.
- The kinds of changes that formalization needs to impose on a design have to happen early in the process.

From scratch?

Here are some examples of things that were not designed to be formalized:

- C
- C++
- Java
- Scala
- Haskell
- ML
- OCaml
- Lisp
- Lean
- Coq
- Isabelle
- Agda
- HOL Light
- HOL4
- ACL2
- Metamath

Many of these languages present formally defined *interfaces* to users, but the *implementation*, the compiler or theorem prover itself, was not originally intended for formalization.

How much should be proven?

- Software correctness is not measured relative to physical computers, it is measured relative to the abstract model provided by hardware.

How much should be proven?

- Software correctness is not measured relative to physical computers, it is measured relative to the abstract model provided by hardware.
- Therefore: Software correctness is a mathematical statement.

How much should be proven?

- Software correctness is not measured relative to physical computers, it is measured relative to the abstract model provided by hardware.
- Therefore: Software correctness is a mathematical statement.
- Mathematical theorems can be proven.

How much should be proven?

- Software correctness is not measured relative to physical computers, it is measured relative to the abstract model provided by hardware.
- Therefore: Software correctness is a mathematical statement.
- Mathematical theorems can be proven.
- Therefore: Eliminating all software bugs is possible.

How much should be proven?

- Software correctness is not measured relative to physical computers, it is measured relative to the abstract model provided by hardware.
- Therefore: Software correctness is a mathematical statement.
- Mathematical theorems can be proven.
- Therefore: Eliminating all software bugs is possible. What's stopping us?

Metamath Zero

- Designed from the ground up as an efficient backend theorem prover
  - Assembly language for proofs
- Crazy fast (~100MB/s)
  - Can verify entire Metamath library of ~30000 proofs in 500 ms
  - The library of supporting material from PA for this project checks in 5 ms (I/O bound)
- 730 lines of C / 1500 lines assembly (`gcc`)
- Additional tooling written in Haskell for proof authoring (+ VS Code integration)
- Translations from other languages planned
  - MM → MM0 → OpenTheory, Lean currently implemented
- Separate theorem statements from proofs
  - Statements are in a formal abstract like human readable format, say only what is needed to define the problem
  - Proofs are in an efficient binary format, untrusted + checked

Formalizing x86

- For highest confidence, it should be as easy as possible for anyone to audit the proof
  - "Unusual" hardware is not an option
  - Bare metal verification (no OS) is possible but should not be required
- $\Rightarrow$ Intel x86-64 + Linux

Formalizing x86

- For highest confidence, it should be as easy as possible for anyone to audit the proof
  - "Unusual" hardware is not an option
  - Bare metal verification (no OS) is possible but should not be required
- $\Rightarrow$ Intel x86-64 + Linux
- Unfortunately this leaves many questionable parts in the trust base
  - Debuggers, root privilege software, firmware and hardware all have the ability to interrupt the process, modify memory in ways that break the assumptions, and continue the program and there is no way to detect such tampering
  - Large parts of firmware and hardware cannot even be inspected because closed-source
  - Backdoors (firmware update) exist
  - A modern Cartesian demon (can the user see program output?)

Formalizing x86: Sail and K

- The primary source was the Sail formalization, cross referenced against the Intel manual and Felix Cloutier's online opcode tables
- I looked at the K formalization as well, but I had difficulty working with it
  - Work was distributed across thousands of files
  - Not much documentation
  - Much of the spec was autogenerated, and it shows
- The Sail x86 formalization was incomplete and only covered the essential instructions
  - Just what the doctor ordered!
  - I found a few minor bugs
- I did not find any easy way to translate either Sail or K to FOL without writing my own parser for these comparatively complex languages
  - Translations and cross validation should be a high priority now that there are competing specs on the scene
  - Official Intel recognition? (forgive my naivete)

Show me the code!

Demo

Inductive types and recursion

```
def len (l: nat): nat;
theorem len0: $ len 0 = 0 $;
theorem lenS (a b: nat):
  $ len (a : b) = suc (len b) $;
```

- The idea: use abstract `def` + theorems to allow definitions that satisfy arbitrary equations
- Pro: avoids the need to give a concrete definition that may be less clear
- Pro: Very flexible wrt different shapes of induction
  - e.g. definition by recursion on lists from left and from right
- Con: Reader is responsible for checking completeness of the equations
  - It is possible to assert that the equations are exhaustive, but it is not pretty
- Con: If the equations are not exhaustive, the definition can resolve in an arbitrary way

Typing

```
theorem readSIBDisplacementT
  (mod bbase q base l: nat):
$ bbase e. Regs /\
    readSIBDisplacement mod bbase q base l ->
  mod e. Bits 2 /\ mod != 3 /\
    q e. u64 /\ base e. Base /\ l e. List u8 $;
```

- PA is essentially untyped; everything is `nat`
  - `peano.mm0` also includes wff and set := $\mathbb{N} \to$ wff
- Typing is represented explicitly in the logic via elementhood
- Types can be arbitrary predicates, so dependent types are easy
- Automatic typing derivation is handled by the theorem prover, not the logic (a.k.a. weak typing)

High level semantics & IO

- Config := u64 × (Regs → u64) × Flags × Memory
- Memory := u64 → $\mathcal{P}\{\mathtt{R}, \mathtt{W}, \mathtt{X}\}$ × u8
- Step relation $k \rightsquigarrow k'$ is true if k' is the result of executing one instruction from k ($k, k' \in$ Config)
  - Step relation is nondeterministic when the ISA is unspecified, or when I don't care about the result (e.g. most bits of the flags register)
  - $\neg\exists k', k \rightsquigarrow k'$ when k is invalid (would fault) or I don't want to model the result (~43 / ~1500 instructions modelled)
- Step relation terminates at a `syscall` instruction

High level semantics & IO

- KernelState := List u8 × List u8 × Config provides a crude representation of the operating environment (Linux kernel)
- In the augmented state we can define the behavior of syscalls `open`, `read`, `write`, `fstat`, `mmap`
  - We don't model the filesystem so `read` reads a giant block of nondeterminism (which happens to be the proof)
  - `read(stdin)` and `write(stdout)` consume from the input list and push to the output list respectively
- Basic implementation of ELF file format sets up the initial configuration
- Result: end-to-end semantics function
  $\text{eval}_{\text{x86}}(\mathit{elf} : \text{List u8}) : \text{List u8} \to \mathcal{P}(\text{List u8})$

Summary

- Done: Define an easily checkable low level proof language (MM0)
- Done (×2): Write a verifier for the language (`mm0-c`, `mm0-hs`)
- Done: Define Peano Arithmetic, finite set theory, inductive types (`peano.mm0`)
- Done: Define the semantics of the language in itself (`mm0.mm0`)
- Done: **Define the semantics of x86 and ELF** (`x86.mm0`)
- Done: Assert the existence of a correct verifier for the language (`x86-mm0.mm0`)
- Done: Write a theorem prover for the language (MM1)
- Done: Prove `peano.mm0` (`peano.mm1`)
- *TODO: Prove `x86.mm0`*
- *TODO: Build a verifier in the logic and prove it correct (Prove `x86-mm0.mm0`)*