Metamath Zero

or: How to verify a verifier

Mario Carneiro

Carnegie Mellon University

January 9, 2020

What is an interactive theorem prover?

What is an interactive theorem prover?

- Interactive: You and the computer “collaborate” to produce a proof

What is an interactive theorem prover?

- Interactive: You and the computer “collaborate” to produce a proof
- Theorem prover: The result is a mathematical proof

What is an interactive theorem prover?

- Interactive: You and the computer “collaborate” to produce a proof
- Theorem prover: The result is a mathematical proof
- How do we relate things that happen in a computer to mathematical truths?

Synthesis and verification

- Theorem provers have two distinct functions: synthesis and verification
- Synthesis: creating a proof object
  - tactics
  - type checking
  - user interaction
  - elaboration
- Verification: conferring certainty of the result
  - “trusted kernel”
  - soundness of the logic
  - reliability of the software
- We don’t directly interact with the verification side, but it relieves the synthesis side of pressure to be sound

Synthesis and verification

- Theorem provers have two distinct functions: synthesis and verification
- Synthesis: creating a proof object
  - tactics
  - type checking
  - user interaction
  - elaboration
- Verification: conferring certainty of the result
  - “trusted kernel”
  - soundness of the logic
  - reliability of the software
- We don’t directly interact with the verification side, but it relieves the synthesis side of pressure to be sound

An adversarial model of proof input

- Ex: You are running a proof competition with a reward, and want to allow arbitrarily complex proofs written by unknown actors or exploitative machine learning systems
- Need a way to specify a target statement, against which proofs can be checked, and which "can't be fooled"
- Can we prove that the verifier is correct *in implementation*?

An adversarial model of proof input

- Ex: You are running a proof competition with a reward, and want to allow arbitrarily complex proofs written by unknown actors or exploitative machine learning systems
- Need a way to specify a target statement, against which proofs can be checked, and which "can't be fooled"
- Can we prove that the verifier is correct *in implementation*?
- Coq, Lean, Isabelle, HOL4, etc. are all too complex to verify in a reasonable time-frame
- Even if these systems have a "small trusted kernel", this notion is not formalizable in practice because of the complexities of the implementation language
    - CakeML is working on this(?)

The metamathematics of theorem provers

- Let $\mathcal{M} \subseteq \{0,1\}^* \times \{0,1\}^*$ be a machine semantics, where $\mathcal{M}(P, x)$ means that program P on input x terminates and indicates success.
  - For example, $\mathcal{M}(P, x)$ if P encodes a Turing machine that when run on input x stops in finitely many steps with a 1 on the tape
- Let $\mathcal{L} \subseteq \{0,1\}^*$ be a language of assertions
  - For example, $\varphi \in \mathcal{L}$ if φ encodes a statement in FOL
  - We generally want $\varphi \in \mathcal{L}$ to be decidable
- Let $\mathcal{S} \subseteq \mathcal{L}$ be the true, resp. provable assertions
  - For example, $\varphi \in \mathcal{S}$ if $\mathrm{ZFC} \vdash \varphi$

Functional correctness for a theorem prover

Program P is a theorem prover (for $\mathcal{S}$ in $\mathcal{M}$ and $\mathcal{L}$) if for all $\varphi \in \mathcal{L}$, if there exists p such that $\mathcal{M}(P, (\varphi, p))$, then $\varphi \in \mathcal{S}$.

The metamathematics of theorem provers

- p here represents a proof of φ in the system $\mathcal{S}$
- We can simplify this if we let $\mathcal{M}$ be nondeterministic
  - For example, $\mathcal{M}(P, x)$ if P encodes a *nondeterministic* Turing machine that when run on input x can possibly reach a halting state with a 1 on the tape

Functional correctness for a theorem prover

Program P is a theorem prover (for $\mathcal{S}$ in $\mathcal{M}$ and $\mathcal{L}$) if for all $\varphi \in \mathcal{L}$, if there exists p such that $\mathcal{M}(P, (\varphi, p))$, then $\varphi \in \mathcal{S}$.

The metamathematics of theorem provers

- p here represents a proof of φ in the system $\mathcal{S}$
- We can simplify this if we let $\mathcal{M}$ be nondeterministic
  - For example, $\mathcal{M}(P, x)$ if P encodes a *nondeterministic* Turing machine that when run on input x can possibly reach a halting state with a 1 on the tape

Functional correctness for a theorem prover

Program P is a theorem prover (for $\mathcal{S}$ in $\mathcal{M}$ and $\mathcal{L}$) if for all $\varphi \in \mathcal{L}$, if $\mathcal{M}(P, \varphi)$, then $\varphi \in \mathcal{S}$.

Bootstrapping a theorem prover

Functional correctness for a theorem prover

Program P is a theorem prover (for $\mathcal{S}$ in $\mathcal{M}$ and $\mathcal{L}$) if for all $\varphi \in \mathcal{L}$, if $\mathcal{M}(P, \varphi)$, then $\varphi \in \mathcal{S}$.

- This statement is itself a logical formula in FOL, so perhaps we can prove it
- Fix $\mathcal{L} = L_{PA}$ and $\mathcal{S} = \mathrm{Th}(\mathbb{N})$, and let V be a theorem prover for $\mathcal{S}$ (i.e. V is arithmetically sound)
- Then $\mathcal{M}(V, \varphi)$ implies φ is true
- So:
$$\mathcal{M}(V, \ulcorner \forall \varphi \in \mathcal{L}' \ (\mathcal{M}'(P, \varphi) \to \varphi \in \mathcal{S}') \urcorner)$$
$$\implies P \text{ is a theorem prover for } \mathcal{S}'$$
- Finally, let $\mathcal{L} = \mathcal{L}'$, $\mathcal{M} = \mathcal{M}'$, $\mathcal{S} = \mathcal{S}'$, $V = P$ to prove V is a theorem prover

Bootstrapping a theorem prover

A proof of V's correctness inside V

1. Suppose V is a theorem prover for $\mathcal{S} \subseteq \mathrm{Th}(\mathbb{N})$.
2. We can prove $\forall \varphi \in \mathcal{L}\, (\mathcal{M}(V, \varphi) \to \varphi \in \mathcal{S})$ using V, that is, $\mathcal{M}(V, \ulcorner \forall \varphi \in \mathcal{L}\, (\mathcal{M}(V, \varphi) \to \varphi \in \mathcal{S}) \urcorner)$.
3. Therefore $\forall \varphi \in \mathcal{L}\, (\mathcal{M}(V, \varphi) \to \varphi \in \mathcal{S})$, i.e. V is a theorem prover for $\mathcal{S}$.

- This is a circular proof!

Bootstrapping a theorem prover

A proof of V's correctness inside V

1. Suppose V is a theorem prover for $\mathcal{S} \subseteq \mathrm{Th}(\mathbb{N})$.
2. We can prove $\forall \varphi \in \mathcal{L}\, (\mathcal{M}(V, \varphi) \to \varphi \in \mathcal{S})$ using V, that is, $\mathcal{M}(V, \ulcorner \forall \varphi \in \mathcal{L}\, (\mathcal{M}(V, \varphi) \to \varphi \in \mathcal{S}) \urcorner)$.
3. Therefore $\forall \varphi \in \mathcal{L}\, (\mathcal{M}(V, \varphi) \to \varphi \in \mathcal{S})$, i.e. V is a theorem prover for $\mathcal{S}$.

- This is a circular proof!
- This is inevitable: we need *some* root of trust, else everything that runs on the computer is suspect

Bootstrapping a theorem prover

A proof of V's correctness inside V

1. Suppose V is a theorem prover for $\mathcal{S} \subseteq \mathrm{Th}(\mathbb{N})$.
2. We can prove $\forall\varphi \in \mathcal{L}\,(\mathcal{M}(V,\varphi) \to \varphi \in \mathcal{S})$ using V, that is, $\mathcal{M}(V, \ulcorner\forall\varphi \in \mathcal{L}\,(\mathcal{M}(V,\varphi) \to \varphi \in \mathcal{S})\urcorner)$.
3. Therefore $\forall\varphi \in \mathcal{L}\,(\mathcal{M}(V,\varphi) \to \varphi \in \mathcal{S})$, i.e. V is a theorem prover for $\mathcal{S}$.

- This is a circular proof!
- This is inevitable: we need *some* root of trust, else everything that runs on the computer is suspect
- It is still an improvement over the unverified case where (1) is taken on faith

Bootstrapping a theorem prover

Ways to bolster the argument:

- If there are n verifiers $V_1, \ldots, V_n$ such that V_i proves the correctness of V_j for all $i, j \leq n$, then the correctness of any of them implies the correctness of all
  - This is robust for different logics, implementation languages, etc, so you can pick your favorite logic/language to believe
  - Proof porting can reduce the $O(n^2)$ proofs to $O(n)$ work (more on this later)

Bootstrapping a theorem prover

Ways to bolster the argument:

- If there are n verifiers $V_1, \ldots, V_n$ such that V_i proves the correctness of V_j for all $i, j \leq n$, then the correctness of any of them implies the correctness of all
  - This is robust for different logics, implementation languages, etc, so you can pick your favorite logic/language to believe
  - Proof porting can reduce the $O(n^2)$ proofs to $O(n)$ work (more on this later)
- We can prove V correct directly "by inspection":
  - Read V (the verifier source code) and convince oneself that it acts like a verifier (size of V matters!)
  - Use the formal proof to guide this reading (size of the proof matters!)

Prove an existing language correct?

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.

Prove an existing language correct?

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.

Metamath Zero[1]

- Designed from the ground up as an efficient backend theorem prover
  - Assembly language for proofs
- Crazy fast (~300MB/s)
  - Can verify entire Metamath library of ~30000 proofs in 200 ms
  - The library of supporting material from PA for this project checks in 2 ms
- 730 lines of C / 1500 lines assembly (`gcc`)
- Additional verifiers and tooling written in Haskell + Rust
- Translations to/from other languages
  - MM → MM0
  - MM0 → "HOL"
  - MM0 → OpenTheory
  - MM0 → Lean

[1] `https://github.com/digama0/mm0`

Why Metamath?

- The goal is to be able to support a proof of correctness of a nontrivial application (the verifier)
- Every feature costs 10× more when you are formally proving it
  - $\Rightarrow$ no frills
  - Must be efficient enough to handle large proofs
- No other theorem prover I know has this combination of properties:
  - Really simple
  - Really fast
  - General purpose (capable of supporting any axiom system)

Why not Metamath?

- Syntax is really simple and general, but semantics are correspondingly complex
- First order logic is a recognizable subsystem of Metamath, but the axioms must be carefully curated
- Axioms are not O(1)!
    - Definitions are axioms
    - Definition conservativity is not checked by the verifier
- Expressions are strings, not trees
    - Grammar rules become soundness critical
    - Performance penalty for large expressions because of duplication
        - not clear if this is a positive or negative
- "No tactics? How can you even *live* in those conditions?"
    - This is a fallacy. We're talking about verification not synthesis

Metamath Zero = Metamath with a semantics

- FOL with schematic metavariables
  - This is not a theorem of FOL:

1. ax1: $\vdash \varphi \to \varphi \to \varphi$
2. ax1: $\vdash \varphi \to (\varphi \to \varphi) \to \varphi$
3. ax2: $\vdash (\varphi \to (\varphi \to \varphi) \to \varphi) \to (\varphi \to \varphi \to \varphi) \to (\varphi \to \varphi)$
4. MP 2,3: $\vdash (\varphi \to \varphi \to \varphi) \to (\varphi \to \varphi)$
5. MP 1,4: $\vdash \varphi \to \varphi$

Metamath Zero = Metamath with a semantics

- FOL with schematic metavariables
  - This is not a theorem of FOL:

  1. ax1: $\vdash \varphi \to \varphi \to \varphi$
  2. ax1: $\vdash \varphi \to (\varphi \to \varphi) \to \varphi$
  3. ax2: $\vdash (\varphi \to (\varphi \to \varphi) \to \varphi) \to (\varphi \to \varphi \to \varphi) \to (\varphi \to \varphi)$
  4. MP 2,3: $\vdash (\varphi \to \varphi \to \varphi) \to (\varphi \to \varphi)$
  5. MP 1,4: $\vdash \varphi \to \varphi$
- A calculus of open terms
  - $x : \mathrm{var}, \varphi : \mathrm{wff}\ x;\ \varphi \vdash \forall x\, \varphi$ has $y = x \vdash \forall y\, y = x$ as a substitution instance (in context $x, y : \mathrm{var}$)

Metamath Zero = Metamath with a semantics

- FOL with schematic metavariables
  - This is not a theorem of FOL:

1. ax1: $\vdash \varphi \to \varphi \to \varphi$
2. ax1: $\vdash \varphi \to (\varphi \to \varphi) \to \varphi$
3. ax2: $\vdash (\varphi \to (\varphi \to \varphi) \to \varphi) \to (\varphi \to \varphi \to \varphi) \to (\varphi \to \varphi)$
4. MP 2,3: $\vdash (\varphi \to \varphi \to \varphi) \to (\varphi \to \varphi)$
5. MP 1,4: $\vdash \varphi \to \varphi$

- A calculus of open terms
  - $x : \mathrm{var}, \varphi : \mathrm{wff}\ x;\ \varphi \vdash \forall x\, \varphi$ has $y = x \vdash \forall y\, y = x$ as a substitution instance (in context $x, y : \mathrm{var}$)
- Direct (admissible) substitution
  - A substitution is admissible if whenever φ does not depend on x, if $x \mapsto y$ and $\varphi \mapsto t$ then $y \notin \mathrm{Vars}(t)$
- Retains the fast checking loop of Metamath while disallowing the semantic oddities

Expressivity

- The most important property of a foundational system is *expressivity*: the ability to express any mathematical argument or computation with moderate overhead
- Foundations are characterized by what they forbid
  - Pruning the search space
  - Preventing “junk theorems”
  - Preventing certain kinds of mathematical construction
- It is easy to add restrictions on a permissive foundation, but it is not easy to circumvent restrictions in a restrictive foundation
- Type systems and type inference are part of synthesis
  - MM0 has an extremely basic type system (multi-sorted FOL), but you can compile any foundational system to it (PA, ZFC, HOL, MLTT, CIC, HoTT, etc.)

The two parts of verification

- If I want to demonstrate theorem T using V, I have two tasks:
  1. Show that $\mathcal{M}(V, \ulcorner T \urcorner)$ (i.e. exhibit a proof of $\ulcorner T \urcorner$ that V will accept)
  2. Validate that $\ulcorner T \urcorner$ is an encoding of T
- (1) is pure computation, while (2) requires human intervention because T is in the mind of the human
  - For example, T is the Kepler conjecture and $\ulcorner T \urcorner$ is

```
(!V. packing V ==>
  (?c. !r. &1 <= r ==>
    &(CARD(V INTER ball(vec 0,r))) <=
    pi * r pow 3 / sqrt(&18) + c * r pow 2))
```

- MM0 uses separate files for these two tasks
  1. `flyspeck.mmb` is a binary file that contains all the proofs and is designed for fast verification
  2. `flyspeck.mm0` contains a human readable statement of the Kepler conjecture such as the above [2]

[2] flyspeck has not (yet) been translated to MM0

Binary verification

- Proof data is stored as a bytecode stream that drives a stack machine
- Terms are deduplicated, so equality testing is always $O(1)$
- Overall check time is $O(mn)$ where m is the max theorem size
- Trivially parallelizable because the verifier has almost no state

$\sigma ::= e \mid\ \vdash A \mid e \equiv e' \mid e \overset{?}{\equiv} e'$	stack element
$H, S ::= \overline{\sigma}$	heap, stack

Save:	$H; S, \sigma \hookrightarrow H, \sigma; S, \sigma$	
Term t:	$S, \overline{e} \hookrightarrow S, e'$	$\left(\begin{matrix} t : \Gamma' \Rightarrow s\,\overline{x}, \quad \Gamma \vdash \overline{e} :: \Gamma' \\ e' := \mathrm{alloc}(t\,\overline{e}) \end{matrix}\right)$
Ref i:	$H; S \hookrightarrow H; S, H_i$	
Dummy s:	$H; S \hookrightarrow H, e; S, e$	$(e := \mathrm{alloc}(x : s),\ x \text{ fresh})$
Thm T:	$S, \overline{e}^*, A \hookrightarrow S', \vdash A$	$(\mathrm{Unify}(T)\text{: } S; \overline{e}; A \hookrightarrow_{\mathsf{u}} S')$
Hyp:	$\Delta; H; S, A \hookrightarrow \Delta, A; H, \vdash A; S$	
Conv:	$S, A, \vdash B \hookrightarrow S, \vdash A, A \overset{?}{\equiv} B$	
Refl:	$S, e \overset{?}{\equiv} e \hookrightarrow S$	
Symm:	$S, e \overset{?}{\equiv} e' \hookrightarrow S, e' \overset{?}{\equiv} e$	
Cong:	$S, t\,\overline{e} \overset{?}{\equiv} t\,\overline{e'} \hookrightarrow S, \overline{e \overset{?}{\equiv} e'}^*$	
Unfold:	$S, t\,\overline{e}, e' \hookrightarrow S', e' \overset{?}{\equiv} e''$	$(\mathrm{Unify}(t)\text{: } S; \overline{e}; e' \hookrightarrow_{\mathsf{u}} S', t\,\overline{e} \overset{?}{\equiv} e'')$
ConvCut:	$S, e, e' \hookrightarrow S, e \equiv e', e \overset{?}{\equiv} e'$	
ConvRef i:	$S, e \overset{?}{\equiv} e' \hookrightarrow S$	$(H_i = e \equiv e')$
ConvSave:	$H; S, e \equiv e' \hookrightarrow H, e \equiv e'; S$	
USave:	$U; K, \sigma \hookrightarrow_{\mathsf{u}} U, \sigma; K, \sigma$	
UTerm t:	$K, t\,\overline{e} \hookrightarrow_{\mathsf{u}} K, \overline{e}$	
URef i:	$U; K, U_i \hookrightarrow_{\mathsf{u}} U; K$	
UDummy s:	$U; K, x \hookrightarrow_{\mathsf{u}} U, x; K$	$(x : s)$
UHyp:	$S, \vdash A; K \hookrightarrow_{\mathsf{u}} S; K, A$	

The specification file

```
delimiter $ ( ) ~ $;
strict provable sort wff;
term im (a b: wff): wff; infixr im: $->$ prec 25;
term not (a: wff): wff;   prefix not: $~$ prec 40;

-- The Lukasiewicz axioms for propositional logic
axiom ax_1 (a b: wff): $ a -> b -> a $;
axiom ax_2 (a b c: wff):
  $ (a -> b -> c) -> (a -> b) -> a -> c $;
axiom ax_3 (a b: wff):
  $ (~a -> ~b) -> b -> a $;
axiom ax_mp (a b: wff):
  $ a -> b $ >
  $ a $ >
  $ b $;
```

The specification file

```
def iff (a b: wff): wff =
  $ ~((a -> b) -> ~(b -> a)) $;
infixl iff: $<->$ prec 20;
```

or

```
def iff (a b: wff): wff;
infixl iff: $<->$ prec 20;
theorem iff1 (a b: wff):
  $ (a <-> b) -> a -> b $;
theorem iff2 (a b: wff):
  $ (a <-> b) -> b -> a $;
theorem iff3 (a b: wff):
  $ (a -> b) -> (b -> a) -> (a <-> b) $;
```

Synthesis

Synthesis

The MM0 verifier has high standards for the quality of its input. How can we generate such proofs?

Translation

- MM0 lies at the intersection of Metamath and second order logic, and so it has easy translation paths to each
- Metamath has “unusual” databases like Hofstadter’s MIU system, that use strings essentially, but all serious formalization is done in databases in which the term grammar is context-free and unambiguous, so token strings and tree representations are isomorphic
- The MM → MM0 translator can be used to losslessly translate the entire Metamath ZFC library into MM0
  - `set.mm` (34 MB) → `set.mm0` (17 MB) + `set.mmb` (28.5 MB)
  - It can be calibrated to target individual theorems, producing a much smaller `mm0` file
  - `pnt` → `pnt.mm0` (107 KB) + `pnt.mmb` (4.5 MB)

Translation

- MM0's logic is a subset of HOL (FOL with universally quantified second order variables) and there is a translation from MM0 to a HOL-like intermediate language
- This intermediate language also has its own verifier, which can be used to validate the translation
- With minor syntactic transformations, this intermediate language is re-targeted to OpenTheory and Lean
  - Unfortunately the OpenTheory tool runs out of memory on `set.mm`, but it has been validated on smaller targets, perhaps a more efficient verifier can do better

Translation

- The Lean port of `set.mm` checks, and we can prove all the translated MM axioms in lean, so `set.mm` is now proven consistent relative to Lean
- After translation, we can align MM natural numbers to Lean natural numbers, etc. to prove theorems such as:

```
theorem dirith' {n : ℕ} {a : ℤ} (n0 : n ≠ 0)
  (g1 : int.gcd a n = 1) :
  ¬ set.finite {x | nat.prime x ∧ ↑n | ↑x - a}
```

- More complex theorems like the prime number theorem could be aligned but require more theorems in Lean to establish the isomorphisms, e.g. proving uniqueness of the real numbers, the definition of the logarithm, etc.

MM1: the MM0 proof assistant

- MM1 is an extension of MM0 with extra commands to support a Turing-complete programming/tactic language similar to Scheme
- There are two implementations of MM1, in Haskell (`mm0-hs`) and in Rust (`mm0-rs`), as servers using the language server protocol (LSP)
- Performs unification and definition unfolding automatically, and generates `mmb` files if there are no errors
- Provides live error diagnostics, hovers, go-to-definition, etc.
- No big sidebar goal view like Lean/Coq yet, as it's not in LSP. Perhaps we should extend LSP to standardize this

MM1: the MM0 proof assistant

- MM1 is an extension of MM0 with extra commands to support a Turing-complete programming/tactic language similar to Scheme
- There are two implementations of MM1, in Haskell (`mm0-hs`) and in Rust (`mm0-rs`), as servers using the language server protocol (LSP)
- Performs unification and definition unfolding automatically, and generates `mmb` files if there are no errors
- Provides live error diagnostics, hovers, go-to-definition, etc.
- No big sidebar goal view like Lean/Coq yet, as it's not in LSP. Perhaps we should extend LSP to standardize this
- Demo

The correctness proof

The correctness proof

Functional correctness for a theorem prover

Program P is a theorem prover (for $\mathcal{S}$ in $\mathcal{M}$ and $\mathcal{L}$) if for all $\varphi \in \mathcal{L}$, if $\mathcal{M}(P, \varphi)$, then $\varphi \in \mathcal{S}$.

We need to define $\mathcal{M}$, $\mathcal{L}$, and $\mathcal{S}$.

- We could take $\mathcal{M}$ to be the semantics of a functional programming language, but then we would still have to trust the compiler, which will undoubtedly dwarf the size of our verifier.
- We could verify the compiler separately, but this will make the overall job much harder because the compiler doesn't know what it is compiling. A direct approach scales with the task.

The correctness proof

Functional correctness for a theorem prover

Program P is a theorem prover (for $\mathcal{S}$ in $\mathcal{M}$ and $\mathcal{L}$) if for all $\varphi \in \mathcal{L}$, if $\mathcal{M}(P, \varphi)$, then $\varphi \in \mathcal{S}$.

We need to define $\mathcal{M}$, $\mathcal{L}$, and $\mathcal{S}$.

- We could verify down to the hardware, but:
  - Hardware is much less stable at the microarchitectural level; there is a possibility that the hardware will outrun the proof effort
  - Hardware and firmware is usually not open source
  - We can't distribute hardware. MM0 is a github repo, not a hardware manufacturer
- $\Rightarrow$ Verify relative to the ISA
  - The x86 architecture is pathologically backward compatible
  - The ISA is well documented, if complex (but we only need a small part of it)

The correctness proof

Functional correctness for a theorem prover

Program P is a theorem prover (for $\mathcal{S}$ in $\mathcal{M}$ and $\mathcal{L}$) if for all $\varphi \in \mathcal{L}$, if $\mathcal{M}(P, \varphi)$, then $\varphi \in \mathcal{S}$.

We need to define $\mathcal{M}$, $\mathcal{L}$, and $\mathcal{S}$.

- Let $\mathcal{M}(P, x)$ if P is the contents of an ELF file, that when loaded in Linux and executed on an x86-64 ISA, with x provided on `stdin`, terminates after finitely many steps with exit code 0.
  - Some instructions are nondeterministic in the semantics, and reading files other than `stdin` results in blobs of nondeterministic data, so we load the proof file that way
- $\mathcal{L}$ is defined as the grammar of well formed MM0 files.
- $x \in \mathcal{S}$ if the `defs` in the file can be provided definitions such that the theorems are provable from the axioms.

The correctness proof

Functional correctness for a theorem prover

Program P is a theorem prover (for $\mathcal{S}$ in $\mathcal{M}$ and $\mathcal{L}$) if for all $\varphi \in \mathcal{L}$, if $\mathcal{M}(P, \varphi)$, then $\varphi \in \mathcal{S}$.

We need to define $\mathcal{M}$, $\mathcal{L}$, and $\mathcal{S}$.

- Let $\mathcal{M}(P, x)$ if P is the contents of an ELF file, that when loaded in Linux and executed on an x86-64 ISA, with x provided on `stdin`, terminates after finitely many steps with exit code 0.
  - Some instructions are nondeterministic in the semantics, and reading files other than `stdin` results in blobs of nondeterministic data, so we load the proof file that way
- $\mathcal{L}$ is defined as the grammar of well formed MM0 files.
- $x \in \mathcal{S}$ if the `defs` in the file can be provided definitions such that the theorems are provable from the axioms.
- Demo

Next steps

The compiler

- In order to produce proofs at the machine code level, we have to string together theorems about assembly, register allocation, and stack allocation with invariants in order to refine high level correctness properties
- In MM0 computations are performed through the application of carefully chosen sequences of theorems, but since the prover has free choice at each stage it is most like a *nondeterministic* Turing machine
- There is no innate advantage in this setting to proving a program correct once and then executing it deterministically (i.e. Coq-style) over composing correctness theorems in a syntax directed way
    - The nondeterministic choices correspond to efficient unverified side calculations
- Lean has been useful to experiment with different program verification frameworks (e.g. separation logic, matching logic) while building the theory

Proof theory

- The correctness theorem proves that the MM0 verifier checks things according to the MM0 spec, but now we want to relate this to “traditional” proof theory, without the restrictions that MM0 imposes for efficiency
- We will be well situated to reason about proof theory, since we already have a deep embedding of formulas as represented by MM0 inside the MM0 + PA logic
  - For example, we would like to establish that if MM0 $\vdash$ `peano.mm0` + `theorem` : T then PA $\vdash T$
- Gödel’s theorems should be an easy corollary of the work done here

A verified reflective kernel

- Since MM0 has a definition of machine code evaluation, it can reason about an extension of the kernel to include a primitive of the form:
  Given an x86 function P that is proven to be safe, execute it and if it returns a value x, return a proof that the x86 semantics can eventually step to x.

  and prove that this is also correct.
- This primitive would allow the construction of more high level code extraction techniques to prove theorems that would otherwise be too expensive to compute, without giving up any of the strong correctness guarantees.

Conclusion

- MM0 is a principled attempt to address the problem of bootstrapping trust in a theorem prover, while remaining efficient even at relatively large scales.
- It is not a monolith, and I encourage both reimplementations of MM0 and alternative bootstraps in other logics that can help bolster the overall trust base.
- MM0 is not trying to be a replacement for Lean, Coq, Isabelle etc. These are synthesis tools, and MM0 is not.
- While still rough on the edges, MM1 demonstrates that there is no innate barrier to building an interactive theorem prover over a Metamath-like soft typed foundation.

Conclusion

- MM0 is a work in progress, but many aspects are already well developed, and you should try out the language if it interests you.
- I hope that some day soon every major theorem prover will either be verified or have a (practical!) extraction path for their theorems to a verified theorem prover.

`https://arxiv.org/abs/1910.10703`
`https://github.com/digama0/mm0`

Thanks!