Metamath Zero

From Logic, to Proof Assistant, to Verified Compiler

Mario Carneiro

Carnegie Mellon University

June 13, 2022

Introduction to Metamath Zero

Proof as a social process

- Penny (the prover) is a mathematician.
  They have just proved a big theorem T and want to share their work with the world.

Proof as a social process

- Penny (the prover) is a mathematician.
  They have just proved a big theorem T and want to share their work with the world.
- Victor (the verifier) is a professor who is interested in Penny's work.
  They would like to be convinced of the truth of T.

Proof as a social process

- Penny (the prover) is a mathematician.
  They have just proved a big theorem T and want to share their work with the world.
- Victor (the verifier) is a professor who is interested in Penny's work.
  They would like to be convinced of the truth of T.

This is a normal situation in mathematics. The usual process:

- Penny writes a document in plain English going through the logic of the proof, leading readers to have no choice but to accept the truth of T.

Proof as a social process

- Penny (the prover) is a mathematician.
  They have just proved a big theorem T and want to share their work with the world.
- Victor (the verifier) is a professor who is interested in Penny's work.
  They would like to be convinced of the truth of T.

This is a normal situation in mathematics. The usual process:

- Penny writes a document in plain English going through the logic of the proof, leading readers to have no choice but to accept the truth of T.
- Victor reads the proof, and checks that each step contains no logical errors, and thereby is convinced that T is true.

Proof as a social process

- Penny is a mathematician.
  They have just proved a big theorem T and want to share their work with the world.
- Victor is a professor who is interested in Penny's work.
  They would like to be convinced of the truth of T.

(This is an idealization. Other ways this could play out:)

- Penny writes a document in plain English going through the logic of the proof, leading readers to have no choice but to accept the truth of T.
- Victor decides that T is too unlikely to be true and Penny couldn't possibly have proved it, and ignores the proof.

Proof as a social process

- Penny is a mathematician.
  They have just proved a big theorem T and want to share their work with the world.
- Victor is a professor who is interested in Penny's work.
  They would like to be convinced of the truth of T.

(This is an idealization. Other ways this could play out:)

- Penny writes a document in plain English going through the logic of the proof, leading readers to have no choice but to accept the truth of T.
- Victor decides that Penny is trustworthy and does not read the proof very carefully, and is convinced of T.

Proof as a social process

How can computers be used to facilitate this process?

1. Penny can use computers to feasibly *build* larger proofs.
2. Victor can use computers to feasibly *check* larger proofs.

Proof as a social process

How can computers be used to facilitate this process?

1. Penny can use computers to feasibly *build* larger proofs.
2. Victor can use computers to feasibly *check* larger proofs. (?)

(1) is certainly true, but it is not clear how to make (2) true.

Proof as a social process

How can computers be used to facilitate this process?

1. Penny can use computers to feasibly *build* larger proofs.
2. Victor can use computers to feasibly *check* larger proofs. (?)

(1) is certainly true, but it is not clear how to make (2) true.

- If Penny has a *very* large proof, the communication process can break down because Victor will get tired

Proof as a social process

How can computers be used to facilitate this process?

1. Penny can use computers to feasibly *build* larger proofs.
2. Victor can use computers to feasibly *check* larger proofs. (?)

(1) is certainly true, but it is not clear how to make (2) true.

- If Penny has a *very* large proof, the communication process can break down because Victor will get tired
- How can Penny lift the burden on Victor and address the imbalance?

Robo-Victor

Robo-Victor is just like Victor, except it's a robot.

Robo-Victor

Robo-Victor is just like Victor, except it's a robot.

- It can read proofs carefully and ensure that there are no flaws in the proof.

Robo-Victor

Robo-Victor is just like Victor, except it's a robot.

- It can read proofs carefully and ensure that there are no flaws in the proof.
- It never gets tired, and can work much faster than Victor.

Robo-Victor

Robo-Victor is just like Victor, except it's a robot.

- It can read proofs carefully and ensure that there are no flaws in the proof.
- It never gets tired, and can work much faster than Victor.

It looks good. But:

Robo-Victor

Robo-Victor is just like Victor, except it's a robot.

- It can read proofs carefully and ensure that there are no flaws in the proof.
- It never gets tired, and can work much faster than Victor.

It looks good. But:

- Robo-Victor isn't Victor!
- Victor can watch Robo-Victor be convinced without themself being convinced

Proof as a social process: now with robots!

New strategy:

1. Penny writes a document in Robo-Victor's language going through the logic of the proof, leading Robo-Victor to have no choice but to accept the truth of T.
2. Penny convinces Victor that Robo-Victor is a valid proof checker.
3. Victor reads the *statement* written in Robo-Victor's language that determines what Robo-Victor checked, and determines that this is in fact a way to write T.

Proof as a social process: now with robots!

New strategy:

1. Penny writes a document in Robo-Victor's language going through the logic of the proof, leading Robo-Victor to have no choice but to accept the truth of T.
2. Penny convinces Victor that Robo-Victor is a valid proof checker.
3. Victor reads the *statement* written in Robo-Victor's language that determines what Robo-Victor checked, and determines that this is in fact a way to write T.

Proof as a social process: now with robots!

New strategy:

1. Penny writes a document in Robo-Victor's language going through the logic of the proof, leading Robo-Victor to have no choice but to accept the truth of T.
2. Mario (the meta-prover) convinces Victor that Robo-Victor is a valid proof checker.
3. Victor reads the *statement* written in Robo-Victor's language that determines what Robo-Victor checked, and determines that this is in fact a way to write T.

Proof as a social process: now with robots!

New strategy:

1. Penny writes a document in Robo-Victor's language going through the logic of the proof, leading Robo-Victor to have no choice but to accept the truth of T.
2. Mario convinces Victor that Robo-Victor is a valid proof checker.
3. Victor reads the *statement* written in Robo-Victor's language that determines what Robo-Victor checked, and determines that this is in fact a way to write T.

- Victor has two jobs now but it is still a lot less work than the original plan since the hard part is being done by Robo-Victor

Proof as a social process: now with robots!

New strategy:

1. Penny writes a document in Robo-Victor's language going through the logic of the proof, leading Robo-Victor to have no choice but to accept the truth of T.
2. Mario convinces Victor that Robo-Victor is a valid proof checker.
3. Victor reads the *statement* written in Robo-Victor's language that determines what Robo-Victor checked, and determines that this is in fact a way to write T.

- Victor has two jobs now but it is still a lot less work than the original plan since the hard part is being done by Robo-Victor
- Observation: Step 2 is a mathematical statement, that Robo-Victor (a particular computer program) checks proofs according to some rules

Bootstrapping

Bootstrapping refers to the idea of “picking oneself up by their bootstraps” – a self-referential foundation.

Bootstrapping

Bootstrapping refers to the idea of “picking oneself up by their bootstraps” – a self-referential foundation.

Proof of Step 2:

1. Mario writes a document in Robo-Victor's language going through the logic of the proof, leading Robo-Victor to accept that “Robo-Victor is a valid proof checker”.
2. Mario convinces Victor that Robo-Victor is a valid proof checker.
3. Victor reads the statement that Robo-Victor checked, and determines that this is in fact a way to write “Robo-Victor is a valid proof checker”.

Bootstrapping

Bootstrapping refers to the idea of “picking oneself up by their bootstraps” – a self-referential foundation.

Proof of Step 2:

1. Mario writes a document in Robo-Victor’s language going through the logic of the proof, leading Robo-Victor to accept that “Robo-Victor is a valid proof checker”.
2. Mario convinces Victor that Robo-Victor is a valid proof checker.
3. Victor reads the statement that Robo-Victor checked, and determines that this is in fact a way to write “Robo-Victor is a valid proof checker”.

This is a circular argument though, so we need more:

Bootstrapping

Bootstrapping refers to the idea of "picking oneself up by their bootstraps" – a self-referential foundation.

Proof of Step 2: (alternate)

1. Mario writes a document in plain English going through the logic of the proof, leading readers to have no choice but to accept the truth of "Robo-Victor is a valid proof checker".

Bootstrapping

Bootstrapping refers to the idea of “picking oneself up by their bootstraps” – a self-referential foundation.

Proof of Step 2: (alternate)

1. Mario writes a document in plain English going through the logic of the proof, leading readers to have no choice but to accept the truth of “Robo-Victor is a valid proof checker”.
2. Victor reads the proof, and checks that each step contains no logical errors, and thereby is convinced that “Robo-Victor is a valid proof checker” is true.

Bootstrapping

Bootstrapping refers to the idea of “picking oneself up by their bootstraps” – a self-referential foundation.

Proof of Step 2: (alternate)

1. Mario writes a document in plain English going through the logic of the proof, leading readers to have no choice but to accept the truth of “Robo-Victor is a valid proof checker”.
2. Victor reads the proof, and checks that each step contains no logical errors, and thereby is convinced that “Robo-Victor is a valid proof checker” is true.

That is, we can use the circular proof to bolster “good old-fashioned” proof.

Summary

The goal of this project is to build Robo-Victor and prove that it has the desired properties.

Summary

The goal of this project is to build Robo-Victor and prove that it has the desired properties.

- Metamath Zero (MM0) is a specification language which allows you to write theorem statements.

Summary

The goal of this project is to build Robo-Victor and prove that it has the desired properties.

- Metamath Zero (MM0) is a specification language which allows you to write theorem statements.
- The analogue of Robo-Victor is the MM0 verifier.

Introduction to Metamath C

Software without bugs is possible

Software without bugs is possible

- Bugs in software are generally thought to be inevitable

Software without bugs is possible

- Bugs in software are generally thought to be inevitable
- Software correctness is increasingly important as people rely on software in critical infrastructure

Software without bugs is possible

- Bugs in software are generally thought to be inevitable
- Software correctness is increasingly important as people rely on software in critical infrastructure
- Testing is only an incomplete solution, since checking all inputs is infeasible for most programs

Software without bugs is possible

- Bugs in software are generally thought to be inevitable
- Software correctness is increasingly important as people rely on software in critical infrastructure
- Testing is only an incomplete solution, since checking all inputs is infeasible for most programs
- Software correctness is a mathematical question

Software without bugs is possible

- Bugs in software are generally thought to be inevitable
- Software correctness is increasingly important as people rely on software in critical infrastructure
- Testing is only an incomplete solution, since checking all inputs is infeasible for most programs
- Software correctness is a mathematical question
  - Software is a logical construct
  - Software specifications are mathematical statements

Software without bugs is possible

- Bugs in software are generally thought to be inevitable
- Software correctness is increasingly important as people rely on software in critical infrastructure
- Testing is only an incomplete solution, since checking all inputs is infeasible for most programs
- Software correctness is a mathematical question
  - Software is a logical construct
  - Software specifications are mathematical statements

→ Software correctness can be proved by mathematical proof ("deductive verification")

Why doesn't everyone do it?

Why doesn't everyone do it?

It is too hard!

Why doesn't everyone do it?

It is too hard!

- Most programming languages don't even support verification in principle

Why doesn't everyone do it?

It is too hard!

- Most programming languages don't even support verification in principle
- Those that do often only do so at a surface level, leaving users to trust the programming language

Why doesn't everyone do it?

It is too hard!

- Most programming languages don't even support verification in principle
- Those that do often only do so at a surface level, leaving users to trust the programming language
  - Compilers have bugs too

Why doesn't everyone do it?

It is too hard!

- Most programming languages don't even support verification in principle
- Those that do often only do so at a surface level, leaving users to trust the programming language
  - Compilers have bugs too
- Even if the compiler is verified (most aren't), it doesn't help if the compiler faithfully translates your bugs

Why doesn't everyone do it?

It is too hard!

- Most programming languages don't even support verification in principle
- Those that do often only do so at a surface level, leaving users to trust the programming language
  - Compilers have bugs too
- Even if the compiler is verified (most aren't), it doesn't help if the compiler faithfully translates your bugs
- We need a language to help people write *verified programs*

Why doesn't everyone do it?

It is too hard!

- Most programming languages don't even support verification in principle
- Those that do often only do so at a surface level, leaving users to trust the programming language
  - Compilers have bugs too
- Even if the compiler is verified (most aren't), it doesn't help if the compiler faithfully translates your bugs
- We need a language to help people write *verified programs*

Metamath C is a language for writing verified programs.

Metamath Zero Architecture

- MM0: The logic and specification language
  - Simple structure
  - Standalone verifier
  - “small trusted kernel”

Metamath Zero Architecture

- MM0: The logic and specification language
  - Simple structure
  - Standalone verifier
  - “small trusted kernel”
- MM1: The proof assistant – produces MM0 proofs
  - Runs tactics and metaprograms and exports MM0 proofs

Metamath Zero Architecture

- MM0: The logic and specification language
  - Simple structure
  - Standalone verifier
  - "small trusted kernel"
- MM1: The proof assistant – produces MM0 proofs
  - Runs tactics and metaprograms and exports MM0 proofs
- MMC: A proof-producing compiler
  - A programming language for producing (x86) programs with a proof of correctness

Metamath Zero Architecture

- MM0: The logic and specification language
- MM1: The proof assistant
- MMC: A proof-producing compiler

- We use MM1 to write proofs in the MM0 logic

Metamath Zero Architecture

- MM0: The logic and specification language
- MM1: The proof assistant
- MMC: A proof-producing compiler

- We use MM1 to write proofs in the MM0 logic
- We use MM1 as a framework to run the MMC compiler

Metamath Zero Architecture

- MM0: The logic and specification language
- MM1: The proof assistant
- MMC: A proof-producing compiler

- We use MM1 to write proofs in the MM0 logic
- We use MM1 as a framework to run the MMC compiler
- The MMC compiler produces MM0 proofs

Metamath Zero Architecture

- MM0: The logic and specification language
- MM1: The proof assistant
- MMC: A proof-producing compiler

- We use MM1 to write proofs in the MM0 logic
- We use MM1 as a framework to run the MMC compiler
- The MMC compiler produces MM0 proofs
- The MM0 verifier is written in MMC

A simple MM0 file: propositional logic

```
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 $;

-- Assert that ‘P -> P‘ is provable
theorem id (P: wff): $ P -> P $;
```

Peano arithmetic

```
... -- predicate logic

--| The sort of natural numbers, or nonnegative integers.
sort nat;

--| ‘0‘ is a natural number.
term d0: nat; prefix d0: $0$ prec max;
--| The successor operation: ‘suc n‘ is a natural number when ‘n‘ is.
term suc (n: nat): nat;

--| Zero is not a successor. Axiom 1 of Peano Arithmetic.
axiom sucne0 (a: nat): $ suc a != 0 $;
--| The successor function is injective. Axiom 2 of Peano Arithmetic.
axiom sucinj (a b: nat): $ suc a = suc b <-> a = b $;
--| The induction axiom of Peano Arithmetic. If ‘p(0)‘ is true,
--| and ‘p(x)‘ implies ‘p(suc x)‘ for all ‘x‘, then ‘p(x)‘ is true for all ‘x‘.
axiom induction {x: nat} (p: wff x):
  $ [ 0 / x ] p -> A. x (p -> [ suc x / x ] p) -> A. x p $;
```

Peano arithmetic

```
--| Addition of natural numbers, a primitive term constructor in PA.
term add (a b: nat): nat; infixl add: $+$ prec 64;
--| Multiplication of natural numbers, a primitive term constructor in PA.
term mul (a b: nat): nat; infixl mul: $*$ prec 70;

--| Addition respects equality.
axiom addeq (a b c d: nat): $ a = b -> c = d -> a + c = b + d $;
--| Multiplication respects equality.
axiom muleq (a b c d: nat): $ a = b -> c = d -> a * c = b * d $;
--| The base case in the definition of addition.
axiom add0 (a: nat): $ a + 0 = a $;
--| The successor case in the definition of addition.
axiom addS (a b: nat): $ a + suc b = suc (a + b) $;
--| The base case in the definition of multiplication.
axiom mul0 (a: nat): $ a * 0 = 0 $;
--| The successor case in the definition of multiplication.
axiom mulS (a b: nat): $ a * suc b = a * b + a $;
```

Peano arithmetic

- Peano arithmetic is a very simple axiomatic system, but also quite expressive

Peano arithmetic

- Peano arithmetic is a very simple axiomatic system, but also quite expressive

We define:

- Propositional logic
- Predicate logic
- Class theory
- $+, -, *, /$, mod, gcd
- even, odd, disjoint sums
- ordered pairs, cartesian product
- finite functions, class functions
- Integers: $+, -, *, /$, mod
- Bitwise operators
- Recursion, exponentiation
- Lists
- Set operators
- finite sets, finite set theory
- cardinality
- List ops: length, append, repeat, reverse, map, join, filter, zip, ...

Metaprogramming with MM1

- MM1 comes with a metaprogramming language based on Scheme

```
do {
  (display "hello world")        -- hello world
  {2 + 2}                        -- 4
  (def x 5)
  {x + x}                        -- 10
  (def (f y) {y + y})
  (f 3)                          -- 6
  (def (fact x)
    (if {x = 0}
      1
      {x * (fact {x - 1})}))
  (fact 5)                       -- 120
};
```

Metaprogramming with MM1

- MM1 comes with a metaprogramming language based on Scheme
- We can use this to implement tactics to prove simple classes of theorems

Metaprogramming with MM1

- MM1 comes with a metaprogramming language based on Scheme
- We can use this to implement tactics to prove simple classes of theorems
- We can prove basic arithmetic theorems this way:

```
theorem _: $ ,19 * ,120 + ,2 = ,2282 $ = norm_num;
```

Metaprogramming with MM1

- MM1 comes with a metaprogramming language based on Scheme
- We can use this to implement tactics to prove simple classes of theorems
- We can prove basic arithmetic theorems this way:

```
theorem _: $ ,19 * ,120 + ,2 = ,2282 $ = norm_num;
```

  - `theorem _` means it is an example

Metaprogramming with MM1

- MM1 comes with a metaprogramming language based on Scheme
- We can use this to implement tactics to prove simple classes of theorems
- We can prove basic arithmetic theorems this way:

```
theorem _: $ ,19 * ,120 + ,2 = ,2282 $ = norm_num;
```

  - `theorem _` means it is an example
  - `norm_num` is the tactic which proves the theorem

Metaprogramming with MM1

- MM1 comes with a metaprogramming language based on Scheme
- We can use this to implement tactics to prove simple classes of theorems
- We can prove basic arithmetic theorems this way:

```
theorem _: $ ,19 * ,120 + ,2 = ,2282 $ = norm_num;
```

  - `theorem _` means it is an example
  - `norm_num` is the tactic which proves the theorem
  - `,19` calls a preprocessor to render 19 as a term. The actual theorem proved is:

```
theorem _: $ (x1 :x x3) * (x7 :x x8) + x2 = (x8 :x xe :x xa) $;
```

  that is, $\texttt{0x13} \cdot \texttt{0x78} + \texttt{0x2} = \texttt{0x8ea}$ which is the theorem written in hexadecimal

Specifying x86

- x86-64 is the common name for Intel’s instruction set architecture (ISA) that runs on most computers

Specifying x86

- x86-64 is the common name for Intel's instruction set architecture (ISA) that runs on most computers
- For this project I wrote down the specification of a decent chunk of x86-64

Specifying x86

- x86-64 is the common name for Intel's instruction set architecture (ISA) that runs on most computers
- For this project I wrote down the specification of a decent chunk of x86-64

This involves:

- The way the CPU decodes instructions into: prefixes, opcode bytes, Mod/RM and other operand bytes
  - (This is approximately what an assembler does)

Specifying x86

- x86-64 is the common name for Intel's instruction set architecture (ISA) that runs on most computers
- For this project I wrote down the specification of a decent chunk of x86-64

This involves:

- The way the CPU decodes instructions into: prefixes, opcode bytes, Mod/RM and other operand bytes
  - (This is approximately what an assembler does)
- The interpretation of each instruction into execution semantics

Specifying x86

- x86-64 is the common name for Intel's instruction set architecture (ISA) that runs on most computers
- For this project I wrote down the specification of a decent chunk of x86-64

This involves:

- The way the CPU decodes instructions into: prefixes, opcode bytes, Mod/RM and other operand bytes
  - (This is approximately what an assembler does)
- The interpretation of each instruction into execution semantics
  - This requires a model of the registers, instruction pointer, flags, memory, page permissions, exception state

Specifying x86

- x86-64 is the common name for Intel's instruction set architecture (ISA) that runs on most computers
- For this project I wrote down the specification of a decent chunk of x86-64

This involves:

- The way the CPU decodes instructions into: prefixes, opcode bytes, Mod/RM and other operand bytes
  - (This is approximately what an assembler does)
- The interpretation of each instruction into execution semantics
  - This requires a model of the registers, instruction pointer, flags, memory, page permissions, exception state
- To interpret IO, a model of the (Linux) operating system

Specifying x86

- x86-64 is the common name for Intel's instruction set architecture (ISA) that runs on most computers
- For this project I wrote down the specification of a decent chunk of x86-64

This involves:

- The way the CPU decodes instructions into: prefixes, opcode bytes, Mod/RM and other operand bytes
  - (This is approximately what an assembler does)
- The interpretation of each instruction into execution semantics
  - This requires a model of the registers, instruction pointer, flags, memory, page permissions, exception state
- To interpret IO, a model of the (Linux) operating system
  - We focus mainly on the possible inputs and outputs of the program, for simple console applications like the MM0 verifier

Specifying x86

- x86-64 is the common name for Intel's instruction set architecture (ISA) that runs on most computers
- For this project I wrote down the specification of a decent chunk of x86-64

This involves:

- The way the CPU decodes instructions into: prefixes, opcode bytes, Mod/RM and other operand bytes
  - (This is approximately what an assembler does)
- The interpretation of each instruction into execution semantics
  - This requires a model of the registers, instruction pointer, flags, memory, page permissions, exception state
- To interpret IO, a model of the (Linux) operating system
  - We focus mainly on the possible inputs and outputs of the program, for simple console applications like the MM0 verifier
- The ELF file format (the linux equivalent of `.exe`)

Specifying MM0

- We also need a specification for MM0 itself, written in MM0

Specifying MM0

- We also need a specification for MM0 itself, written in MM0
- This involves
  - Parsing the input string into keywords like `theorem`, `def`, etc

Specifying MM0

- We also need a specification for MM0 itself, written in MM0
- This involves
  - Parsing the input string into keywords like `theorem`, `def`, etc
  - Parsing the math text like `$ a + suc b = suc (a + b) $` into a structured representation like `(eq (add a (suc b)) (suc (add a b)))`

Specifying MM0

- We also need a specification for MM0 itself, written in MM0
- This involves
  - Parsing the input string into keywords like `theorem`, `def`, etc
  - Parsing the math text like `$ a + suc b = suc (a + b) $` into a structured representation like `(eq (add a (suc b)) (suc (add a b)))`
  - Describing the underlying proof system, how theorems are proved from axioms

P-THM

$$\frac{(\Gamma'; \overline{A} \vdash B) \in E \qquad \Gamma \vdash \overline{e} :: \Gamma' \qquad \forall i,\ \vdash A_i[\Gamma' \mapsto \overline{e}] \qquad \forall i\, j\, x,\ \Gamma'_i = x \notin \mathrm{V}_{\Gamma'}(\Gamma'_j) \to e_i \notin \mathrm{FV}_{\Gamma}(e_j)}{\vdash B[\Gamma' \mapsto \overline{e}]}$$

Specifying MM0

- We also need a specification for MM0 itself, written in MM0
- This involves
  - Parsing the input string into keywords like `theorem`, `def`, etc
  - Parsing the math text like `$ a + suc b = suc (a + b) $` into a structured representation like `(eq (add a (suc b)) (suc (add a b)))`
  - Describing the underlying proof system, how theorems are proved from axioms
  - Describing how full files are put together from definitions and theorems

P-THM

$$\frac{(\Gamma';\overline{A} \vdash B) \in E \qquad \Gamma \vdash \overline{e} :: \Gamma' \qquad \forall i,\ \vdash A_i[\Gamma' \mapsto \overline{e}] \qquad \forall i\, j\, x,\ \Gamma'_i = x \notin \mathrm{V}_{\Gamma'}(\Gamma'_j) \to e_i \notin \mathrm{FV}_{\Gamma}(e_j)}{\vdash B[\Gamma' \mapsto \overline{e}]}$$

The correctness theorem

- This is everything we need to state the correctness theorem for a verifier:

Functional correctness for a verifier

Program P is a correct theorem prover if for every initial state $s \in \text{init}(P)$, all nondeterministic evaluations do not cause undefined behavior, and after reaching a final state $s \rightsquigarrow^* s'$, if s' is a successful exit state and $\text{input\_consumed}(s') = I$, then I is a valid and provable MM0 file.

The correctness theorem

- This is everything we need to state the correctness theorem for a verifier:

Functional correctness for a verifier

Program P is a correct theorem prover if for every initial state $s \in \mathrm{init}(P)$, all nondeterministic evaluations do not cause undefined behavior, and after reaching a final state $s \rightsquigarrow^* s'$, if s' is a successful exit state and $\mathrm{input\_consumed}(s') = I$, then I is a valid and provable MM0 file.

- Red: definitions from `x86.mm0`
- Blue: definitions from `mm0.mm0`

Metamath C

- Most of this theorem is generic over all programs, not just MM0 verifiers:

Correctness, generalized

Program P is correct to specification T if for every initial state $s \in \text{init}(P)$, all nondeterministic evaluations do not cause undefined behavior, and after reaching a final state $s \rightsquigarrow^* s'$, if s' is a successful exit state and $\text{input\_consumed}(s') = I$ and $\text{output\_produced}(s') = O$, then $T(I, O)$ is true.

Metamath C

- Most of this theorem is generic over all programs, not just MM0 verifiers:

Correctness, generalized

Program P is correct to specification T if for every initial state $s \in \text{init}(P)$, all nondeterministic evaluations do not cause undefined behavior, and after reaching a final state $s \rightsquigarrow^* s'$, if s' is a successful exit state and $\text{input\_consumed}(s') = I$ and $\text{output\_produced}(s') = O$, then $T(I, O)$ is true.

- The Metamath C compiler produces theorems of this form.

Metamath C

- MMC is not a “general-purpose” programming language
  - Someday, it can hope to be about as general purpose as C or Rust, but this is a gargantuan effort for many reasons
- The niche MMC fills is writing executable programs which *provably* satisfy some condition
- Most programs don’t need this property, but correctness is important to some degree in almost every program, and (approximate) type correctness is mainstream

Type system → static analysis → theorem prover

Programming languages have come a long way in terms of provable correctness

- Assembly: bare minimum type system required to make instructions compilable

Type system → static analysis → theorem prover

Programming languages have come a long way in terms of provable correctness

- Assembly: bare minimum type system required to make instructions compilable
- C: Typed pointers, structs

Type system → static analysis → theorem prover

Programming languages have come a long way in terms of provable correctness

- Assembly: bare minimum type system required to make instructions compilable
- C: Typed pointers, structs
- Java: Generic types, type polymorphism

Type system → static analysis → theorem prover

Programming languages have come a long way in terms of provable correctness

- Assembly: bare minimum type system required to make instructions compilable
- C: Typed pointers, structs
- Java: Generic types, type polymorphism
- Haskell: Algebraic data types

Type system → static analysis → theorem prover

Programming languages have come a long way in terms of provable correctness

- Assembly: bare minimum type system required to make instructions compilable
- C: Typed pointers, structs
- Java: Generic types, type polymorphism
- Haskell: Algebraic data types
- Rust: Linear types

Type system → static analysis → theorem prover

Programming languages have come a long way in terms of provable correctness

- Assembly: bare minimum type system required to make instructions compilable
- C: Typed pointers, structs
- Java: Generic types, type polymorphism
- Haskell: Algebraic data types
- Rust: Linear types
- Static analyzers: Value analysis, contract checking

Type system → static analysis → theorem prover

Programming languages have come a long way in terms of provable correctness

- Assembly: bare minimum type system required to make instructions compilable
- C: Typed pointers, structs
- Java: Generic types, type polymorphism
- Haskell: Algebraic data types
- Rust: Linear types
- Static analyzers: Value analysis, contract checking
- Lean: Dependent types, proof objects

Type system → static analysis → theorem prover

Programming languages have come a long way in terms of provable correctness

- Assembly: bare minimum type system required to make instructions compilable
- C: Typed pointers, structs
- Java: Generic types, type polymorphism
- Haskell: Algebraic data types
- Rust: Linear types
- Static analyzers: Value analysis, contract checking
- Lean: Dependent types, proof objects
- **Metamath C**

A type checker is just a simple theorem prover; the study of one naturally leads to the other

Examples: Procedures

- This is a function that takes two 32 bit integers and returns their sum, wrapped to 32 bits

```
proc add2(x: u32, y: u32): u32 {
  return (x + y) as u32;
}
```

Examples: Procedures

- This is a function that takes two 32 bit integers and returns their sum, wrapped to 32 bits

```
proc add2(x: u32, y: u32): u32 {
  return (x + y) as u32;
}
```

- Supports multiple returns and dependent types for writing preconditions and postconditions

```
proc deptypes(x: u32, _: x = 0): y: u32, sn((x + y) as u32) {
  1, sn((x + 1) as u32)
}
```

Examples: Tuples and pattern matching

- This function constructs and destructs some tuples. The `sn(1), sn(2)` return type says that this function returns exactly the values 1 and 2

```
proc tuples(): sn(1), sn(2) {
  let x: (nat, nat) := (1, 2);
  let (one, two) := x;
  sn(one), sn(two)
}
```

Examples: Tuples and pattern matching

- This function constructs and destructs some tuples. The `sn(1), sn(2)` return type says that this function returns exactly the values 1 and 2

```
proc tuples(): sn(1), sn(2) {
  let x: (nat, nat) := (1, 3); // <- changed 2 to 3
  let (one, two) := x;
  sn(one), sn(two) // type error!
}
```

Examples: Control flow

- After an `if` statement, you can capture the property's truth value in a variable:

```
proc if_statement(x: nat) {
  if h: x < 10 {
    // x: nat, h: x < 10
  } else {
    // x: nat, h: ~(x < 10)
  }
}
```

Examples: Control flow

- While loops and assignment:

```
proc while_loop() {
  let b := true;
  let h2 := while h: b {
    // h: b
    b <- false;
  };
  // h2: ~b
}
```

Examples: Numeric types

- There are various fixed size integral types, as well as unbounded integer types

$$\tau ::= \texttt{u8} \mid \texttt{u16} \mid \texttt{u32} \mid \texttt{u64} \mid \texttt{nat} \mid \texttt{i8} \mid \texttt{i16} \mid \texttt{i32} \mid \texttt{i64} \mid \texttt{int} \mid \ldots$$

Examples: Numeric types

- There are various fixed size integral types, as well as unbounded integer types

$$\tau ::= \texttt{u8} \mid \texttt{u16} \mid \texttt{u32} \mid \texttt{u64} \mid \texttt{nat} \mid \texttt{i8} \mid \texttt{i16} \mid \texttt{i32} \mid \texttt{i64} \mid \texttt{int} \mid \ldots$$

- Numeric operations yield their exact untruncated value, so the user must decide how to cast the value back into range:

Examples: Numeric types

- There are various fixed size integral types, as well as unbounded integer types

$$\tau ::= \texttt{u8} \mid \texttt{u16} \mid \texttt{u32} \mid \texttt{u64} \mid \texttt{nat} \mid \texttt{i8} \mid \texttt{i16} \mid \texttt{i32} \mid \texttt{i64} \mid \texttt{int} \mid \ldots$$

- Numeric operations yield their exact untruncated value, so the user must decide how to cast the value back into range:
  - `(x + y): nat`: don't truncate at all (this can only be used in limited ways)

Examples: Numeric types

- There are various fixed size integral types, as well as unbounded integer types

$$\tau ::= \texttt{u8} \mid \texttt{u16} \mid \texttt{u32} \mid \texttt{u64} \mid \texttt{nat} \mid \texttt{i8} \mid \texttt{i16} \mid \texttt{i32} \mid \texttt{i64} \mid \texttt{int} \mid \ldots$$

- Numeric operations yield their exact untruncated value, so the user must decide how to cast the value back into range:
  - `(x + y): nat`: don't truncate at all (this can only be used in limited ways)
  - `(x + y) as u32`: wrap the result

Examples: Numeric types

- There are various fixed size integral types, as well as unbounded integer types

$$\tau ::= \texttt{u8} \mid \texttt{u16} \mid \texttt{u32} \mid \texttt{u64} \mid \texttt{nat} \mid \texttt{i8} \mid \texttt{i16} \mid \texttt{i32} \mid \texttt{i64} \mid \texttt{int} \mid \ldots$$

- Numeric operations yield their exact untruncated value, so the user must decide how to cast the value back into range:
  - `(x + y): nat`: don't truncate at all (this can only be used in limited ways)
  - `(x + y) as u32`: wrap the result
  - `(x + y): u32`: make the type checker prove it is in range (usually only works if the values of x and y are known)

Examples: Numeric types

- There are various fixed size integral types, as well as unbounded integer types

$$\tau ::= \texttt{u8} \mid \texttt{u16} \mid \texttt{u32} \mid \texttt{u64} \mid \texttt{nat} \mid \texttt{i8} \mid \texttt{i16} \mid \texttt{i32} \mid \texttt{i64} \mid \texttt{int} \mid \ldots$$

- Numeric operations yield their exact untruncated value, so the user must decide how to cast the value back into range:
  - `(x + y): nat`: don't truncate at all (this can only be used in limited ways)
  - `(x + y) as u32`: wrap the result
  - `(x + y): u32`: make the type checker prove it is in range (usually only works if the values of x and y are known)
  - `cast(x + y, h): u32`: prove that $x + y < 2^{32}$

Examples: Numeric types

- There are various fixed size integral types, as well as unbounded integer types

$$\tau ::= \texttt{u8} \mid \texttt{u16} \mid \texttt{u32} \mid \texttt{u64} \mid \texttt{nat} \mid \texttt{i8} \mid \texttt{i16} \mid \texttt{i32} \mid \texttt{i64} \mid \texttt{int} \mid \dots$$

- Numeric operations yield their exact untruncated value, so the user must decide how to cast the value back into range:
  - `(x + y): nat`: don't truncate at all (this can only be used in limited ways)
  - `(x + y) as u32`: wrap the result
  - `(x + y): u32`: make the type checker prove it is in range (usually only works if the values of x and y are known)
  - `cast(x + y, h): u32`: prove that $x + y < 2^{32}$
  - `cast(x + y): u32`: assert that $x + y < 2^{32}$ and crash otherwise

Separation logic

MMC's type system includes the basic primitives of separation logic, for expressing complex properties:

Type	Concrete syntax	Typehood predicate $\boxed{a : -}$	Meaning
$\exists x : \tau_1, \tau_2(x)$	`(ex x:` τ_1`,` $\tau_2(x)$`)`	$\exists x : \tau_1, \boxed{a : \tau_2(x)}$	Existential quantification
$\forall x : \tau_1, \tau_2(x)$	`all x:` τ_1`.` $\tau_2(x)$	$\forall x : \tau_1, \boxed{a : \tau_2(x)}$	Universal quantification
$\tau_1 \to \tau_2$	τ_1 `->` τ_2	$\boxed{a : \tau_1} \to \boxed{a : \tau_2}$	Non-separating implication
$\tau_1 \mathrel{-\!\!*} \tau_2$	τ_1 `-*` τ_2	$\boxed{a : \tau_1} \mathrel{-\!\!*} \boxed{a : \tau_1}$	Separating imp. (magic wand)
$\tau_1 \land \tau_2$	τ_1 `&&` τ_2	$\boxed{a : \tau_1} \land \boxed{a : \tau_2}$	Non-separating conjunction
$\tau_1 * \tau_2$	(τ_1, τ_2)	$\boxed{a.0 : \tau_1} * \boxed{a.1 : \tau_2}$	Separating conjunction
$\tau_1 \lor \tau_2$	τ_1 `||` τ_2	$\boxed{a : \tau_1} \lor \boxed{a : \tau_2}$	Disjunction
$\neg\tau$	`~`τ_1	$\neg\boxed{a : \tau}$	Negation
$\ell \mapsto v$	ℓ `|->` v	$\ell \mapsto v$	Points-to assertion
$\boxed{e : \tau}$	`[`e`:` τ`]`	$\boxed{e : \tau}$	Typing assertion
$\lvert\tau\rvert$	`moved(`τ`)`	$\left\lVert\boxed{a : \tau}\right\rVert$	Persistent core of τ

The main function

- The theorem to be proved by the MMC compiler depends on the return type of the `main()` function:

```
proc main(): collatz_conjecture {
  // if this program succeeds, then the collatz conjecture is true
  assert(false) // ...not that I know how to write such a program!
}
```

The Metamath C compiler

- The full language has many features, and it is designed to be proof producing from day 1
- We have to lower all this, while preserving proofs along the way

The Metamath C compiler

- The full language has many features, and it is designed to be proof producing from day 1
- We have to lower all this, while preserving proofs along the way
- Note: the MMC compiler is *not* a "verified compiler" in the sense of CompCert
  - There is a single theorem that asserts that CompCert compiles any C program according to the C spec
  - The MMC compiler instead produces a proof on the fly that *your* program meets *your* spec

The Metamath C compiler

Most of it looks familiar to compiler writers:

The Metamath C compiler

Most of it looks familiar to compiler writers:

- The input is parsed into an abstract syntax tree (AST)

The Metamath C compiler

Most of it looks familiar to compiler writers:

- The input is parsed into an abstract syntax tree (AST)
- The AST is desugared into a simpler AST

The Metamath C compiler

Most of it looks familiar to compiler writers:

- The input is parsed into an abstract syntax tree (AST)
- The AST is desugared into a simpler AST
- The AST is typechecked and translated into HIR (higher-level intermediate representation), type errors are reported

The Metamath C compiler

Most of it looks familiar to compiler writers:

- The input is parsed into an abstract syntax tree (AST)
- The AST is desugared into a simpler AST
- The AST is typechecked and translated into HIR (higher-level intermediate representation), type errors are reported
- The HIR is lowered to MIR, a basic block representation

The Metamath C compiler

Most of it looks familiar to compiler writers:

- The input is parsed into an abstract syntax tree (AST)
- The AST is desugared into a simpler AST
- The AST is typechecked and translated into HIR (higher-level intermediate representation), type errors are reported
- The HIR is lowered to MIR, a basic block representation
- Several optimization passes are run on MIR:
  - Reachability analysis to remove dead blocks

The Metamath C compiler

Most of it looks familiar to compiler writers:

- The input is parsed into an abstract syntax tree (AST)
- The AST is desugared into a simpler AST
- The AST is typechecked and translated into HIR (higher-level intermediate representation), type errors are reported
- The HIR is lowered to MIR, a basic block representation
- Several optimization passes are run on MIR:
  - Reachability analysis to remove dead blocks
  - Ghost analysis determines which variables only need to exist in the proof without any machine representation

The Metamath C compiler

Most of it looks familiar to compiler writers:

- The input is parsed into an abstract syntax tree (AST)
- The AST is desugared into a simpler AST
- The AST is typechecked and translated into HIR (higher-level intermediate representation), type errors are reported
- The HIR is lowered to MIR, a basic block representation
- Several optimization passes are run on MIR:
  - Reachability analysis to remove dead blocks
  - Ghost analysis determines which variables only need to exist in the proof without any machine representation
  - Legalization tries to remove any `nat` intermediates

The Metamath C compiler

- A "monomorphization" pass is run to find out which type instantiations of which functions need to exist in the final binary

The Metamath C compiler

- A “monomorphization” pass is run to find out which type instantiations of which functions need to exist in the final binary
- For each monomorphized function, we determine the stack frame layout, based on the intermediates that exist in the function

The Metamath C compiler

- A “monomorphization” pass is run to find out which type instantiations of which functions need to exist in the final binary
- For each monomorphized function, we determine the stack frame layout, based on the intermediates that exist in the function
- The monomorphized MIR is lowered to VCode (virtual-register code) representation, which is an architecture-dependent IR which looks like x86

The Metamath C compiler

- A “monomorphization” pass is run to find out which type instantiations of which functions need to exist in the final binary
- For each monomorphized function, we determine the stack frame layout, based on the intermediates that exist in the function
- The monomorphized MIR is lowered to VCode (virtual-register code) representation, which is an architecture-dependent IR which looks like x86
- The register allocator produces PCode (physical-register code), which looks like assembly

The Metamath C compiler

- A “monomorphization” pass is run to find out which type instantiations of which functions need to exist in the final binary
- For each monomorphized function, we determine the stack frame layout, based on the intermediates that exist in the function
- The monomorphized MIR is lowered to VCode (virtual-register code) representation, which is an architecture-dependent IR which looks like x86
- The register allocator produces PCode (physical-register code), which looks like assembly
- Branch displacement optimization (BDO) determines which jumps can use the short form

The Metamath C compiler

- A “monomorphization” pass is run to find out which type instantiations of which functions need to exist in the final binary
- For each monomorphized function, we determine the stack frame layout, based on the intermediates that exist in the function
- The monomorphized MIR is lowered to VCode (virtual-register code) representation, which is an architecture-dependent IR which looks like x86
- The register allocator produces PCode (physical-register code), which looks like assembly
- Branch displacement optimization (BDO) determines which jumps can use the short form
- The instructions are assembled into bytes, and the ELF header is added

The Metamath C compiler

A regular compiler would stop there. We have a few more steps to go:

The Metamath C compiler

A regular compiler would stop there. We have a few more steps to go:

- We first re-assemble the emitted byte stream back into the PCode representation, but generating a "proof of assembly" along the way

The Metamath C compiler

A regular compiler would stop there. We have a few more steps to go:

- We first re-assemble the emitted byte stream back into the PCode representation, but generating a “proof of assembly” along the way
- MIR has a proper type system, while PCode maintains a mapping to the bytes and also to the MIR. So we go over each function, and produce a proof of correctness relative to the reconstructed assembly.

The Metamath C compiler: The assembly proof

An example assembly theorem, which parses binary operations involving a register and intermediate argument, like `add rax, 1` (which does `RAX += 1` where `RAX` is one of the general purpose registers):

$$\frac{\text{split}_{1,3}(y) = v, \texttt{0} \quad \text{opSizeW}(\mathit{rex}^?, v) = \mathit{sz} \quad \text{parseModRM}(\mathit{rex}^?, s) \Rightarrow \mathit{opc}, \text{reg } \mathit{dst}, s' \quad \text{parseImm}(\mathit{sz}, s') \Rightarrow \mathit{src} \quad \text{parseBinop}(\mathit{opc}, \mathit{sz}, \mathit{dst}, \text{imm32 } \mathit{src}) \Rightarrow I}{(\texttt{8}\ y)\ s \mathbin{@} p, \mathit{ip}, \mathit{rex}^? \Rightarrow I \text{ inst}}\ \textsc{inst-binop-imm}$$

The Metamath C compiler: The assembly proof

Some example correctness theorems:

CODE-SEQ

$$\frac{\mathcal{B} \vdash \{\mathcal{T}_0\}\ A_1\ \{\mathcal{T}_1\} \qquad \mathcal{B} \vdash \{\mathcal{T}_1\}\ A_2\ \{\mathcal{T}_2\}}{\mathcal{B} \vdash \{\mathcal{T}_0\}\ (A_1;\ A_2)\ \{\mathcal{T}_2\}}$$

CODE-MOV-RR

$$\frac{\mathrm{read}(\mathcal{T}, \mathrm{reg}\ \mathit{src}) \Rightarrow v \qquad \mathrm{write}(\mathcal{T}, \mathrm{reg}\ \mathit{dst}, v) \Rightarrow \mathcal{T}'}{\mathcal{B} \vdash \{\mathcal{T}\}\ (\mathrm{mov.64}\ \mathit{dst}\ \mathit{src})\ \{\mathcal{T}'\}}$$

BLOCK-I

$$\frac{\mathcal{B} \vdash A\ @\ n\ \mathrm{lasm} \qquad \mathcal{B} \vdash \{\mathcal{T}\}\ A\ \{\bot\}}{\mathcal{B} \vdash \mathrm{block}(\mathcal{T})\ @\ n}$$

CODE-JCC

$$\frac{\mathrm{insert}(\mathcal{T}, \tau) \Rightarrow \mathcal{T}_1 \qquad \mathrm{insert}(\mathcal{T}, \neg\tau) \Rightarrow \mathcal{T}_2 \qquad \mathrm{flagCond}(f, \mathit{cond}) \Rightarrow \tau \qquad \mathcal{B} \vdash \mathrm{block}(\mathcal{T}_1)\ @\ \mathit{tgt}}{\mathcal{B} \vdash \{\mathrm{withFlag}(f, \mathcal{T})\}\ (\mathrm{jcc}\ \mathit{cond}\ \mathit{tgt})\ \{\mathcal{T}_2\}}$$

The Metamath C compiler: The correctness proof

These theorems have to juggle a lot of state.

- The global context $\mathcal{G}$ contains the main specification T and the ELF file *elf*

The Metamath C compiler: The correctness proof

These theorems have to juggle a lot of state.

- The global context $\mathcal{G}$ contains the main specification T and the ELF file *elf*
- The procedure context $\mathcal{P}$ adds the return type of the current procedure and whether side effects are legal in this function

The Metamath C compiler: The correctness proof

These theorems have to juggle a lot of state.

- The global context $\mathcal{G}$ contains the main specification T and the ELF file *elf*
- The procedure context $\mathcal{P}$ adds the return type of the current procedure and whether side effects are legal in this function
- The block context $\mathcal{B}$ adds the set of blocks corresponding to back-edges in the control flow (like while loops) which require a `variant` specification because they are in the middle of an inductive proof

The Metamath C compiler: The correctness proof

These theorems have to juggle a lot of state.

- The global context $\mathcal{G}$ contains the main specification T and the ELF file *elf*
- The procedure context $\mathcal{P}$ adds the return type of the current procedure and whether side effects are legal in this function
- The block context $\mathcal{B}$ adds the set of blocks corresponding to back-edges in the control flow (like while loops) which require a `variant` specification because they are in the middle of an inductive proof
- The variable context $\mathcal{V}$ has the values and types of variables that are in scope

The Metamath C compiler: The correctness proof

These theorems have to juggle a lot of state.

- The global context $\mathcal{G}$ contains the main specification T and the ELF file *elf*
- The procedure context $\mathcal{P}$ adds the return type of the current procedure and whether side effects are legal in this function
- The block context $\mathcal{B}$ adds the set of blocks corresponding to back-edges in the control flow (like while loops) which require a `variant` specification because they are in the middle of an inductive proof
- The variable context $\mathcal{V}$ has the values and types of variables that are in scope
- The machine context $\mathcal{M}$ has the assignment of registers and stack values in terms of the variables

The Metamath C compiler: The correctness proof

These theorems have to juggle a lot of state.

- The global context $\mathcal{G}$ contains the main specification T and the ELF file *elf*
- The procedure context $\mathcal{P}$ adds the return type of the current procedure and whether side effects are legal in this function
- The block context $\mathcal{B}$ adds the set of blocks corresponding to back-edges in the control flow (like while loops) which require a `variant` specification because they are in the middle of an inductive proof
- The variable context $\mathcal{V}$ has the values and types of variables that are in scope
- The machine context $\mathcal{M}$ has the assignment of registers and stack values in terms of the variables
- The type context $\mathcal{T} = (\mathcal{V}, \mathcal{M})$ is used as pre- and post-conditions for assembly sequences

The current state of the project

- There are several MM0 verifiers, written in C, Rust, Haskell, and some plans for the MMC verifier which are waiting for the compiler to catch up

The current state of the project

- There are several MM0 verifiers, written in C, Rust, Haskell, and some plans for the MMC verifier which are waiting for the compiler to catch up
- The MM1 proof assistant is fairly stable and has already been used for some pretty big formalization work

The current state of the project

- There are several MM0 verifiers, written in C, Rust, Haskell, and some plans for the MMC verifier which are waiting for the compiler to catch up
- The MM1 proof assistant is fairly stable and has already been used for some pretty big formalization work
- The MMC compiler is mostly working for generating executable programs

The current state of the project

- There are several MM0 verifiers, written in C, Rust, Haskell, and some plans for the MMC verifier which are waiting for the compiler to catch up
- The MM1 proof assistant is fairly stable and has already been used for some pretty big formalization work
- The MMC compiler is mostly working for generating executable programs
- The assembly proof generator is complete

The current state of the project

- There are several MM0 verifiers, written in C, Rust, Haskell, and some plans for the MMC verifier which are waiting for the compiler to catch up
- The MM1 proof assistant is fairly stable and has already been used for some pretty big formalization work
- The MMC compiler is mostly working for generating executable programs
- The assembly proof generator is complete
- The correctness proof generator is not yet complete

The current state of the project

- There are several MM0 verifiers, written in C, Rust, Haskell, and some plans for the MMC verifier which are waiting for the compiler to catch up
- The MM1 proof assistant is fairly stable and has already been used for some pretty big formalization work
- The MMC compiler is mostly working for generating executable programs
- The assembly proof generator is complete
- The correctness proof generator is not yet complete

It is still a research project at this point, but I have every intention to grow this to an industrial strength project eventually.

Conclusion

- The MMC language design is unlike any I have seen, and I think there is a real need for it.
- It still remains to be seen if it is actually usable in practice, but it could be a game-changer, bringing the task of writing formally verified programs down to the level of the average proof assistant user.
- The self-verification of MM0 will set a new standard for what formal verification is really capable of, and my ultimate goal is to get all major theorem provers verified either directly or by translation to MM0 (or another verified language).

Github: `https://github.com/digama0/mm0`
Thesis: `https://digama0.github.io/mm0/thesis.pdf`