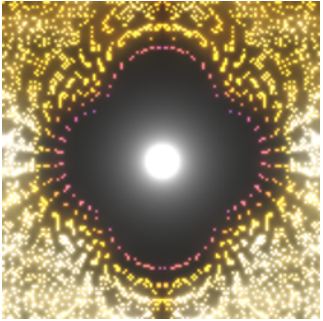
Metamath and Metamath Zero

Mario Carneiro

Carnegie Mellon University

February 7, 2023

Who am I?



Github: `digama0`
Zulip: Mario Carneiro

- Postdoc in Logic at CMU
- Proof engineering since 2013
  - Metamath (maintainer)
  - Lean 3, Lean 4 mathlib (maintainer)
  - Dabbled in Isabelle, HOL Light, Coq, Mizar
  - Metamath Zero (author)
- Proved 37 of Freek's 100 theorems list in Metamath
- Lots of library code in `set.mm` and mathlib
- My PhD thesis was about Metamath Zero
- Say hi at `https://leanprover.zulipchat.com`

Part I: Metamath

Metamath is:

- A computer language for writing mathematical proofs
- A program `metamath.exe` to verify proofs in the Metamath language
- A library of completed proofs in a wide variety of axiomatic systems
  - `set.mm`: Over 40000 proofs deriving consequences of ZFC
    - Covers material in set theory, category theory, real analysis, calculus, number theory, algebra, topology, linear algebra, lattice theory, graph theory
    - 74 from Freek Wiedijk's 100 theorems list, which puts it 4th on the list behind HOL Light, Isabelle, and Coq
  - `iset.mm`: 10000 proofs in intuitionistic ZF
  - `nf.mm`: 5900 proofs in NF set theory
  - `ql.mm`: 1100 proofs in quantum logic
  - Other databases: `hol.mm`, `dtt.mm`, `peano.mm`, `miu.mm`

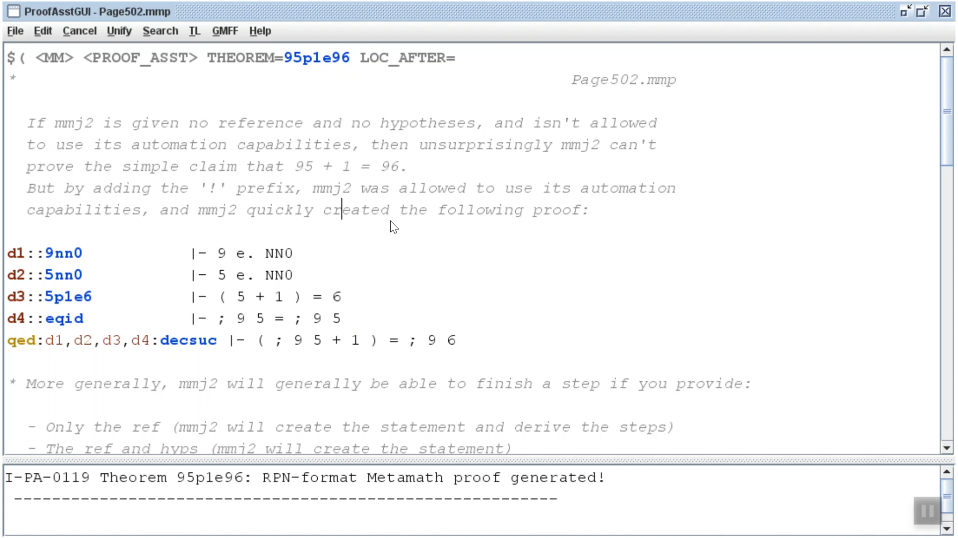
Metamath looks like: (set.mm)

```
$( Function with a domain of two different values.  (Contributed by FL,
   26-Jun-2011.)  (Revised by Mario Carneiro, 26-Apr-2015.) $)
fnprg $p ⊢ ( ( ( A e. V ∧ B e. W ) ∧ ( C e. X ∧ D e. Y ) ∧ A ≠ B )
        → { <. A , C >. , <. B , D >. } Fn { A , B } ) $=
  ( wcel wa wne w3a cop cpr wfun cdm wceq wfn funprg dmpropg 3ad2ant2 df-fn
  sylanbrc ) AEIBFIJZCGIDHIJZABKZLACMBDMNZOUGPABNZQZUGUHRABCDEFGHSUEUDUIUFACB
  DGHTUAUGUHUBUC $.

$( Function with a domain of three different values.  (Contributed by
   Alexander van der Vekens, 5-Dec-2017.) $)
fntpg $p ⊢ ( ( ( X e. U ∧ Y e. V ∧ Z e. W )
                ∧ ( A e. F ∧ B e. G ∧ C e. H )
                ∧ ( X ≠ Y ∧ X ≠ Z ∧ Y ≠ Z ) )
         → { <. X , A >. , <. Y , B >. , <. Z , C >. } Fn { X , Y , Z } ) $=
  ( wcel w3a wne cop cdm wceq csn cun ctp wfun wfn funtpg wa dmsnopg 3ad2ant1
  cpr 3ad2ant2 jca uneq12 syl df-pr syl6eqr dmeqi eqeq1i dmun sylibr 3ad2ant3
  bitri uneq12d df-tp eqtri 3eqtr4g df-fn sylanbrc ) JDMKHMLIMNZAEMZBFMZCGMZN
  ZJKOJLOKLONZNZJAPZKBPZLCPZUAZUBVQQZJKLUAZRVQVSUCABCDEFGHIJKLUDVMVNVOUHZQZVP
  SZQZTZJKUHZLSZTVRVSVMWAWEWCWFVMVNSZQZVOSZQZTZWERZWAWERZVMWKJSZKSZTZWEVMWHWN
  RZWJWORZUEZWKWPRVKVGWSVLVKWQWRVHVIWQVJJAEUFUGVIVHWRVJKBFUFUIUJUIWHWNWJWOUKU
  LJKUMUNWMWGWITZQZWERWLWAXAWEVTWTVNVOUMUOUPXAWKWEWGWIUQUPUTURVKVGWCWFRZVLVJV
  HXBVILCGUFUSUIVAVRVTWBTZQWDVQXCVNVOVPVBUOVTWBUQVCJKLVBVDVQVSVEVF $.

${
  fntp.1 $e ⊢ A e. _V $.
  fntp.2 $e ⊢ B e. _V $.
  fntp.3 $e ⊢ C e. _V $.
  fntp.4 $e ⊢ D e. _V $.
  fntp.5 $e ⊢ E e. _V $.
  fntp.6 $e ⊢ F e. _V $.
  $( A function with a domain of three elements.  (Contributed by NM,
     14-Sep-2011.)  (Revised by Mario Carneiro, 26-Apr-2015.) $)
  fntp $p ⊢ ( ( A ≠ B ∧ A ≠ C ∧ B ≠ C )
          → { <. A , D >. , <. B , E >. , <. C , F >. } Fn { A , B , C } ) $=
    ( wne w3a cop ctp wfun cdm wceq wfn funtp dmtpop a1i df-fn sylanbrc ) ABM
    ACMBCMNZADOBEOCFOPZQUGRABCPZSUGUHTABCDEFGHIJKLUAUIUFADBECFJKLUBUCUGUHUDU
    E $.
$}
```

Metamath looks like: (mmj2)



Metamath looks like: (MPE)

Metamath Proof Explorer

< Previous Next >
Nearby theorems

Mirrors > Home > MPE Home > Th. List > ruc

Structured version Visualization version GIF version

Theorem ruc 14994

Description: The set of positive integers is strictly dominated by the set of real numbers, i.e. the real numbers are uncountable. The proof consists of lemmas ruclem1 14982 through ruclem13 14993 and this final piece. Our proof is based on the proof of Theorem 5.18 of [Truss] p. 114. See ruclem13 14993 for the function existence version of this theorem. For an informal discussion of this proof, see mmcomplex.html#uncountable. For an alternate proof see rucALT 14981. This is Metamath 100 proof #22. (Contributed by NM, 13-Oct-2004.)

Assertion

Ref	Expression
ruc	⊢ ℕ ≺ ℝ

Proof of Theorem ruc

Step	Hyp	Ref	Expression
1		reex 10049	..3 ⊢ ℝ ∈ V
2		nnssre 11046	..3 ⊢ ℕ ⊆ ℝ
3		ssdomg 8023	..3 ⊢ (ℝ ∈ V → (ℕ ⊆ ℝ → ℕ ≼ ℝ))
4	1, 2, 3	mp2 9	.2 ⊢ ℕ ≼ ℝ
5		ruclem13 14993	....5 ⊢ ¬ f:ℕ–onto→ℝ
6		f1ofo 6158	....5 ⊢ (f:ℕ–1-1-onto→ℝ → f:ℕ–onto→ℝ)
7	5, 6	mto 188	...4 ⊢ ¬ f:ℕ–1-1-onto→ℝ
8	7	nex 1731	..3 ⊢ ¬ ∃f f:ℕ–1-1-onto→ℝ
9		bren 7986	..3 ⊢ (ℕ ≈ ℝ ↔ ∃f f:ℕ–1-1-onto→ℝ)
10	8, 9	mtbir 313	.2 ⊢ ¬ ℕ ≈ ℝ
11		brsdom 8000	.2 ⊢ (ℕ ≺ ℝ ↔ (ℕ ≼ ℝ ∧ ¬ ℕ ≈ ℝ))
12	4, 10, 11	mpbir2an 955	1 ⊢ ℕ ≺ ℝ

Colors of variables: wff setvar class

Syntax hints: ¬ wn 3 ∃wex 1704 ∈ wcel 1990 Vcvv 3201 ⊆ wss 3576 *class class class* wbr 4657 –onto→wfo 5900 –1-1-onto→wf1o 5901 ≈ cen 7974 ≼ cdom 7975 ≺ csdm 7976 ℝcr 9957 ℕcn 11042

This theorem was proved from axioms: ax-mp 5 ax-1 6 ax-2 7 ax-3 8 ax-gen 1722 ax-4 1737 ax-5 1839 ax-6 1888 ax-7 1935 ax-8 1992 ax-9 1999 ax-10 2019 ax-11 2034 ax-12 2047 ax-13 2246 ax-ext 2603 ax-rep 4776 ax-sep 4786 ax-nul 4794 ax-pow 4848 ax-pr 4911 ax-un 6968 ax-cnex 10014 ax-resscn 10015 ax-1cn 10016 ax-icn 10017 ax-addcl 10018 ax-addrcl 10019 ax-mulcl 10020 ax-mulrcl 10021 ax-mulcom 10022 ax-addass 10023 ax-mulass 10024 ax-distr 10025 ax-i2m1 10026 ax-1ne0 10027 ax-1rid 10028 ax-rnegex 10029 ax-rrecex 10030 ax-cnre 10031 ax-pre-lttri 10032 ax-pre-lttrn 10033 ax-pre-ltadd 10034 ax-pre-mulgt0 10035 ax-pre-sup 10036

Metamath's good ideas

Metamath is not the most popular theorem prover,
but it has some good ideas that are not shared with its contemporaries.

What makes Metamath unique?

Metamath's good ideas

- Separate proof authoring from proof checking
- Have a simple spec for the logical core

Metamath's good ideas

- Separate proof authoring from proof checking
- Have a simple spec for the logical core

Separate proof authoring from proof checking

- Interactive theorem provers need to support two activities:

Separate proof authoring from proof checking

- Interactive theorem provers need to support two activities:
    - Writing / authoring proofs

Separate proof authoring from proof checking

- Interactive theorem provers need to support two activities:
  - Writing / authoring proofs
  - Checking proofs

Separate proof authoring from proof checking

- Interactive theorem provers need to support two activities:
  - Writing / authoring proofs
  - Checking proofs
- The design criteria for these two are completely different

Separate proof authoring from proof checking

- Interactive theorem provers need to support two activities:
  - Writing / authoring proofs
  - Checking proofs
- The design criteria for these two are completely different
  - Writing happens once, checking happens many times

Separate proof authoring from proof checking

- Interactive theorem provers need to support two activities:
  - Writing / authoring proofs
  - Checking proofs
- The design criteria for these two are completely different
  - Writing happens once, checking happens many times
  - Checking is often performed as part of CI

Separate proof authoring from proof checking

- Interactive theorem provers need to support two activities:
  - Writing / authoring proofs
  - Checking proofs
- The design criteria for these two are completely different
  - Writing happens once, checking happens many times
  - Checking is often performed as part of CI
  - Writing involves human interaction and creativity

Separate proof authoring from proof checking

- Interactive theorem provers need to support two activities:
  - Writing / authoring proofs
  - Checking proofs
- The design criteria for these two are completely different
  - Writing happens once, checking happens many times
  - Checking is often performed as part of CI
  - Writing involves human interaction and creativity
- Writing needs a proof assistant, proof checking needs a kernel

Separate proof authoring from proof checking

- Interactive theorem provers need to support two activities:
  - Writing / authoring proofs
  - Checking proofs
- The design criteria for these two are completely different
  - Writing happens once, checking happens many times
  - Checking is often performed as part of CI
  - Writing involves human interaction and creativity
- Writing needs a proof assistant, proof checking needs a kernel
  - A good proof assistant is big and complex to give a good user experience

Separate proof authoring from proof checking

- Interactive theorem provers need to support two activities:
  - Writing / authoring proofs
  - Checking proofs
- The design criteria for these two are completely different
  - Writing happens once, checking happens many times
  - Checking is often performed as part of CI
  - Writing involves human interaction and creativity
- Writing needs a proof assistant, proof checking needs a kernel
  - A good proof assistant is big and complex to give a good user experience
  - A good kernel is small and trustworthy

Separate proof authoring from proof checking

- Interactive theorem provers need to support two activities:
  - Writing / authoring proofs
  - Checking proofs
- The design criteria for these two are completely different
  - Writing happens once, checking happens many times
  - Checking is often performed as part of CI
  - Writing involves human interaction and creativity
- Writing needs a proof assistant, proof checking needs a kernel
  - A good proof assistant is big and complex to give a good user experience
  - A good kernel is small and trustworthy
    (and ideally fast and not resource-intensive)

Separate proof authoring from proof checking

- Metamath stores *proofs*, not *proof scripts*

```
$( Function with a domain of two different values.  (Contributed by FL,
   26-Jun-2011.)  (Revised by Mario Carneiro, 26-Apr-2015.) $)
fnprg $p ⊢ ( ( ( A e. V ∧ B e. W ) ∧ ( C e. X ∧ D e. Y ) ∧ A ≠ B )
      → { <. A , C >. , <. B , D >. } Fn { A , B } ) $=
  ( wcel wa wne w3a cop cpr wfun cdm wceq wfn funprg dmpropg 3ad2ant2 df-fn
  sylanbrc ) AEIBFIJZCGIDHIJZABKZLACMBDMNZOUGPABNZQZUGUHRABCDEFGHSUEUDUIUFACB
  DGHTUAUGUHUBUC $.

$( Function with a domain of three different values.  (Contributed by
   Alexander van der Vekens, 5-Dec-2017.) $)
fntpg $p ⊢ ( ( ( X e. U ∧ Y e. V ∧ Z e. W )
             ∧ ( A e. F ∧ B e. G ∧ C e. H )
             ∧ ( X ≠ Y ∧ X ≠ Z ∧ Y ≠ Z ) )
         → { <. X , A >. , <. Y , B >. , <. Z , C >. } Fn { X , Y , Z } ) $=
  ( wcel w3a wne cop cdm wceq csn cun ctp wfun wfn funtpg wa dmsnopg 3ad2ant1
  cpr 3ad2ant2 jca uneq12 syl df-pr syl6eqr dmeqi eqeq1i dmun sylibr 3ad2ant3
  bitri uneq12d df-tp eqtri 3eqtr4g df-fn sylanbrc ) JDMKHLIMNZAEMZBFMZCGMZN
  ZJKOJLOKLONZNZJAPZKBPZLCPZUAZUBVQQZJKLUAZRVQVSUCABCDEFGHIJKLUDVMVNVOUHZQZVP
  SZQZTZJKUHZLSZTVRVSVMWAWEWCWFVMVNSZQZVOSZQZTZWERZWAWERZVMWKJSZKSZTZWEVMWHWN
  RZWJWORZUEZWKWPRVKVGWSVLVKWQWRVHVIWQVJJAEUFUGVIVHWRVJKBFUFUIUJUIWHWNWJWOUKU
  LJKUMUNWMWGWITZQZWERWLWAXAWEVTWTVNVOUMUOUPXAWKWEWGWIUQUPUTURVKVGWCWFRZVLVJV
  HXBVILCGUFUSUIVAVRVTWBTZQWDVQXCVNVOVPVBUOVTWBUQVCJKLVBVDVQVSVEVF $.

${
  fntp.1 $e ⊢ A e. _V $.
  fntp.2 $e ⊢ B e. _V $.
  fntp.3 $e ⊢ C e. _V $.
  fntp.4 $e ⊢ D e. _V $.
  fntp.5 $e ⊢ E e. _V $.
  fntp.6 $e ⊢ F e. _V $.
  $( A function with a domain of three elements.  (Contributed by NM,
     14-Sep-2011.)  (Revised by Mario Carneiro, 26-Apr-2015.) $)
  fntp $p ⊢ ( ( A ≠ B ∧ A ≠ C ∧ B ≠ C )
        → { <. A , D >. , <. B , E >. , <. C , F >. } Fn { A , B , C } ) $=
    ( wne w3a cop ctp wfun cdm wceq wfn funtp dmtpop a1i df-fn sylanbrc ) ABM
    ACMBCMNZADOBEOCFOPZQUGRABCPZSUGUHTABCDEFGHIJKLUAUIUFADBECFJKLUBUCUGUHUDU
    E $.
$}
```

Separate proof authoring from proof checking

- Metamath stores *proofs*, not *proof scripts*
- Checking metamath proofs is massively faster than checking Lean, Coq, Isabelle, HOL Light proofs
  - The classic verifier `metamath.exe` checks `set.mm`, a library on the same order of magnitude as Lean mathlib, in 8 seconds

```
$( Function with a domain of two different values.  (Contributed by FL,
   26-Jun-2011.)  (Revised by Mario Carneiro, 26-Apr-2015.) $)
fnprg $p ⊢ ( ( ( A e. V ∧ B e. W ) ∧ ( C e. X ∧ D e. Y ) ∧ A ≠ B )
        → { <. A , C >. , <. B , D >. } Fn { A , B } ) $=
  ( wcel wa wne w3a cop cpr wfun cdm wceq wfn funprg dmpropg 3ad2ant2 df-fn
  sylanbrc ) AEIBFIJZCGIDHIJZABKZLACMBDMNZOUGPABNZQZUGUHRABCDEFGHSUEUDUIUFACB
  DGHTUAUGUHUBUC $.

$( Function with a domain of three different values.  (Contributed by
   Alexander van der Vekens, 5-Dec-2017.) $)
fntpg $p ⊢ ( ( ( X e. U ∧ Y e. V ∧ Z e. W )
                ∧ ( A e. F ∧ B e. G ∧ C e. H )
                ∧ ( X ≠ Y ∧ X ≠ Z ∧ Y ≠ Z ) )
          → { <. X , A >. , <. Y , B >. , <. Z , C >. } Fn { X , Y , Z } ) $=
  ( wcel w3a wne cop cdm wceq csn cun ctp wfun wfn funtpg wa dmsnopg 3ad2ant1
  cpr 3ad2ant2 jca uneq12 syl df-pr syl6eqr dmeqi eqeq1i dmun sylibr 3ad2ant3
  bitri uneq12d df-tp eqtri 3eqtr4g df-fn sylanbrc ) JDMKHMLIMNZAEMZBFMZCGMZN
  ZJKOJLOKLONZNZJAPZKBPZLCPZUAZUBVQQZJKLUAZRVQVSUCABCDEFGHIJKLUDVMVNVOUHZQZVP
  SZQZTZJKUHZLSZTVRVSVMWAWEWCWFVMVNSZQZVOSZQZTZWERZWAWERZVMWKJSZKSZTZWEVMWHWN
  RZWJWORZUEZWKWPRVKVGWSVLVKWQWRVHVIWQVJJAEUFUGVIVHWRVJKBFUFUIUJUIWHWNWJWOUKU
  LJKUMUNWMWGWITZQZWERWLWAXAWEVTWTVNVOUMUOUPXAWKWEWGWIUQUPUTURVKVGWCWFRZVLVJV
  HXBVILCGUFUSUIVAVRVTWBTZQWDVQXCVNVOVPVBUOVTWBUQVCJKLVBVDVQVSVEVF $.

${
  fntp.1 $e ⊢ A e. _V $.
  fntp.2 $e ⊢ B e. _V $.
  fntp.3 $e ⊢ C e. _V $.
  fntp.4 $e ⊢ D e. _V $.
  fntp.5 $e ⊢ E e. _V $.
  fntp.6 $e ⊢ F e. _V $.
  $( A function with a domain of three elements.  (Contributed by NM,
     14-Sep-2011.)  (Revised by Mario Carneiro, 26-Apr-2015.) $)
  fntp $p ⊢ ( ( A ≠ B ∧ A ≠ C ∧ B ≠ C )
          → { <. A , D >. , <. B , E >. , <. C , F >. } Fn { A , B , C } ) $=
    ( wne w3a cop ctp wfun cdm wceq wfn funtp dmtpop a1i df-fn sylanbrc ) ABM
    ACMBCMNZADOBEOCFOPZQUGRABCPZSZUGUHTABCDEFGHIJKLUAUIUFADBECFJKLUBUCUGUHUDU
    E $.
$}
```

Separate proof authoring from proof checking

- Metamath stores *proofs*, not *proof scripts*
- Checking metamath proofs is massively faster than checking Lean, Coq, Isabelle, HOL Light proofs
  - The classic verifier `metamath.exe` checks `set.mm`, a library on the same order of magnitude as Lean mathlib, in 8 seconds
  - An optimized metamath verifier has achieved the same feat in 0.9 seconds

```
$( Function with a domain of two different values.  (Contributed by FL,
   26-Jun-2011.)  (Revised by Mario Carneiro, 26-Apr-2015.) $)
fnprg $p ⊢ ( ( ( A e. V ∧ B e. W ) ∧ ( C e. X ∧ D e. Y ) ∧ A ≠ B )
      → { <. A , C >. , <. B , D >. } Fn { A , B } ) $=
  ( wcel wa wne w3a cop cpr wfun cdm wceq wfn funprg dmpropg 3ad2ant2 df-fn
  sylanbrc ) AEIBFIJZCGIDHIJZABKZLACMBDMNZOUGPABNZQZUGUHRABCDEFGHSUEUDUIUFACB
  DGHTUAUGUHUBUC $.

$( Function with a domain of three different values.  (Contributed by
   Alexander van der Vekens, 5-Dec-2017.) $)
fntpg $p ⊢ ( ( ( X e. U ∧ Y e. V ∧ Z e. W )
             ∧ ( A e. F ∧ B e. G ∧ C e. H )
             ∧ ( X ≠ Y ∧ X ≠ Z ∧ Y ≠ Z ) )
       → { <. X , A >. , <. Y , B >. , <. Z , C >. } Fn { X , Y , Z } ) $=
  ( wcel w3a wne cop cdm wceq csn cun ctp wfun wfn funtpg wa dmsnopg 3ad2ant1
  cpr 3ad2ant2 jca uneq12 syl df-pr syl6eqr dmeqi eqeq1i dmun sylibr 3ad2ant3
  bitri uneq12d df-tp eqtri 3eqtr4g df-fn sylanbrc ) JDMKHLIMNZAEMZBFMZCGMZN
  ZJKOJLOKLONZNZJAPZKBPZLCPZUAZUBVQQZJKLUAZRVQVSUCABCDEFGHIJKLUDVMVNVOUHZQZVP
  SZQZTZJKUHZLSZTVRVSVMWAWEWCWFVMVNSZQZVOSZQZTZWERZWAWERZVMWKJSZKSZTZWEVMWHWN
  RZWJWORZUEZWKWPRVKVGWSVLVKWQWRVHVIWQVJJAEUFUGVIVHWRVJKBFUFUIUJUIWHWNWJWOUKU
  LJKUMUNWMWGWITZQZWERWLWAXAWEVTWTVNVOUMUOUPXAWKWEWGWIUQUPUTURVKVGWCWFRZVLVJV
  HXBVILCGUFUSUIVAVRVTWBTZQWDVQXCVNVOVPVBUOVTWBUQVCJKLVBVDVQVSVEVF $.

${
  fntp.1 $e ⊢ A e. _V $.
  fntp.2 $e ⊢ B e. _V $.
  fntp.3 $e ⊢ C e. _V $.
  fntp.4 $e ⊢ D e. _V $.
  fntp.5 $e ⊢ E e. _V $.
  fntp.6 $e ⊢ F e. _V $.
  $( A function with a domain of three elements.  (Contributed by NM,
     14-Sep-2011.)  (Revised by Mario Carneiro, 26-Apr-2015.) $)
  fntp $p ⊢ ( ( A ≠ B ∧ A ≠ C ∧ B ≠ C )
        → { <. A , D >. , <. B , E >. , <. C , F >. } Fn { A , B , C } ) $=
    ( wne w3a cop ctp wfun cdm wceq wfn funtp dmtpop a1i df-fn sylanbrc ) ABM
    ACMBCMNZADOBEOCFOPZQUGRABCPZSZUGUHTABCDEFGHIJKLUAUIUFADBECFJKLUBUCUGUHUDU
    E $.
$}
```

Metamath's good ideas

- Separate proof authoring from proof checking
- Have a simple spec for the logical core

Have a simple spec for the logical core

- Metamath has a prose specification in the Metamath book

The next section contains the complete specification of the Metamath language. It serves as an authoritative reference and presents the syntax in enough detail to write a parser and proof verifier. The specification is terse and it is probably hard to learn the language directly from it, but we include it here for those impatient people who prefer to see everything up front before looking at verbose expository material. Later sections explain this material and provide examples. We will repeat the definitions in those sections, and you may skip the next section at first reading and proceed to Section 4.2 (p. 116).

4.1 Specification of the Metamath Language

> *Sometimes one has to say difficult things, but one ought to say them as simply as one knows how.*
>
> G. H. Hardy[2]

4.1.1 Preliminaries

A Metamath **database** is built up from a top-level source file together with any source files that are brought in through file inclusion commands (see below). The only characters that are allowed to appear in a Metamath source file are the 94 non-whitespace printable ASCII characters, which are digits, upper and lower case letters, and the following 32 special characters:

```
! " # $ % & ' ( ) * + , - . / :
; < = > ? @ [ \ ] ^ _ ` { | } ~
```

plus the following characters which are the "white space" characters: space (a printable character), tab, carriage return, line feed, and form feed. We will use `typewriter` font to display the printable characters.

A Metamath database consists of a sequence of three kinds of **tokens** separated by **white space** (which is any sequence of one or more white space characters). The set of **keyword** tokens is `${`, `$}`, `$c`, `$v`, `$f`, `$e`, `$d`, `$a`, `$p`, `$.`, `$=`, `$(`, `$)`, `$[`, and `$]`. The last four are called **auxiliary** or preprocessing keywords. A **label** token consists of any combination of letters, digits, and the characters hyphen, underscore, and period. A **math symbol** token may consist of any combination of the 93 printable standard ASCII characters other than space or `$` . All tokens are case-sensitive.

[2] As quoted in [16], p. 273.

Have a simple spec for the logical core

- Metamath has a prose specification in the Metamath book
  - The full spec is 28 pages (with lots of explanation and examples)

The next section contains the complete specification of the Metamath language. It serves as an authoritative reference and presents the syntax in enough detail to write a parser and proof verifier. The specification is terse and it is probably hard to learn the language directly from it, but we include it here for those impatient people who prefer to see everything up front before looking at verbose expository material. Later sections explain this material and provide examples. We will repeat the definitions in those sections, and you may skip the next section at first reading and proceed to Section 4.2 (p. 116).

4.1 Specification of the Metamath Language

Sometimes one has to say difficult things, but one ought to say them as simply as one knows how.

G. H. Hardy[2]

4.1.1 Preliminaries

A Metamath **database** is built up from a top-level source file together with any source files that are brought in through file inclusion commands (see below). The only characters that are allowed to appear in a Metamath source file are the 94 non-whitespace printable ASCII characters, which are digits, upper and lower case letters, and the following 32 special characters:

```
! " # $ % & ' ( ) * + , - . / :
; < = > ? @ [ \ ] ^ _ ` { | } ~
```

plus the following characters which are the "white space" characters: space (a printable character), tab, carriage return, line feed, and form feed. We will use `typewriter` font to display the printable characters.

A Metamath database consists of a sequence of three kinds of **tokens** separated by **white space** (which is any sequence of one or more white space characters). The set of **keyword** tokens is `${`, `$}`, `$c`, `$v`, `$f`, `$e`, `$d`, `$a`, `$p`, `$.`, `$=`, `$(`, `$)`, `$[`, and `$]`. The last four are called **auxiliary** or preprocessing keywords. A **label** token consists of any combination of letters, digits, and the characters hyphen, underscore, and period. A **math symbol** token may consist of any combination of the 93 printable standard ASCII characters other than space or `$` . All tokens are case-sensitive.

[2] As quoted in [16], p. 273.

Have a simple spec for the logical core

- Metamath has a prose specification in the Metamath book
  - The full spec is 28 pages (with lots of explanation and examples)
- There is an emphasis on parsimony

The next section contains the complete specification of the Metamath language. It serves as an authoritative reference and presents the syntax in enough detail to write a parser and proof verifier. The specification is terse and it is probably hard to learn the language directly from it, but we include it here for those impatient people who prefer to see everything up front before looking at verbose expository material. Later sections explain this material and provide examples. We will repeat the definitions in those sections, and you may skip the next section at first reading and proceed to Section 4.2 (p. 116).

4.1 Specification of the Metamath Language

Sometimes one has to say difficult things, but one ought to say them as simply as one knows how.

G. H. Hardy[2]

4.1.1 Preliminaries

A Metamath **database** is built up from a top-level source file together with any source files that are brought in through file inclusion commands (see below). The only characters that are allowed to appear in a Metamath source file are the 94 non-whitespace printable ASCII characters, which are digits, upper and lower case letters, and the following 32 special characters:

```
! " # $ % & ' ( ) * + , - . / :
; < = > ? @ [ \ ] ^ _ ` { | } ~
```

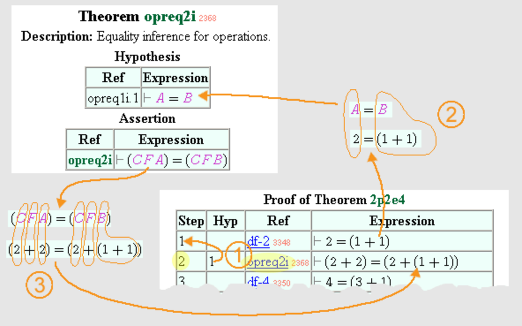
plus the following characters which are the "white space" characters: space (a printable character), tab, carriage return, line feed, and form feed. We will use `typewriter` font to display the printable characters.

A Metamath database consists of a sequence of three kinds of **tokens** separated by **white space** (which is any sequence of one or more white space characters). The set of **keyword** tokens is `${`, `$}`, `$c`, `$v`, `$f`, `$e`, `$d`, `$a`, `$p`, `$.`, `$=`, `$(`, `$)`, `$[`, and `$]`. The last four are called **auxiliary** or preprocessing keywords. A **label** token consists of any combination of letters, digits, and the characters hyphen, underscore, and period. A **math symbol** token may consist of any combination of the 93 printable standard ASCII characters other than space or `$` . All tokens are case-sensitive.

[2] As quoted in [16], p. 273.

Have a simple spec for the logical core

- Metamath has a prose specification in the Metamath book
  - The full spec is 28 pages (with lots of explanation and examples)
- There is an emphasis on parsimony
- The HTML documentation is full of pages of introductory material which assumes no mathematical background



Have a simple spec for the logical core

- Consequence: *Many* verifiers

Have a simple spec for the logical core

- Consequence: *Many* verifiers
  - There are 19 known verifiers

Have a simple spec for the logical core

- Consequence: *Many* verifiers
  - There are 19 known verifiers
  - written in C, C++, C#, Rust, Lua, Haskell, Python, Igor, JavaScript, Mathematica, Julia, Scala, Java, Zig, Lean 4

Have a simple spec for the logical core

- Consequence: *Many* verifiers
  - There are 19 known verifiers
  - written in C, C++, C#, Rust, Lua, Haskell, Python, Igor, JavaScript, Mathematica, Julia, Scala, Java, Zig, Lean 4
  - ...and Turing Machine

Have a simple spec for the logical core

- Consequence: *Many* verifiers
  - There are 19 known verifiers
  - written in C, C++, C#, Rust, Lua, Haskell, Python, Igor, JavaScript, Mathematica, Julia, Scala, Java, Zig, Lean 4
  - ...and Turing Machine
    - A metamath verifier was adapted to prove the best known lower bound on the smallest unprovable-in-ZFC busy beaver number[1]

[1] `https://github.com/sorear/metamath-turing-machines`

Have a simple spec for the logical core

- Consequence: *Many* verifiers
  - There are 19 known verifiers
  - written in C, C++, C#, Rust, Lua, Haskell, Python, Igor, JavaScript, Mathematica, Julia, Scala, Java, Zig, Lean 4
  - ...and Turing Machine
    - A metamath verifier was adapted to prove the best known lower bound on the smallest unprovable-in-ZFC busy beaver number[1]

- Many verifiers are tiny, and some are fast

[1] `https://github.com/sorear/metamath-turing-machines`

Have a simple spec for the logical core

- Consequence: *Many* verifiers
  - There are 19 known verifiers
  - written in C, C++, C#, Rust, Lua, Haskell, Python, Igor, JavaScript, Mathematica, Julia, Scala, Java, Zig, Lean 4
  - ...and Turing Machine
    - A metamath verifier was adapted to prove the best known lower bound on the smallest unprovable-in-ZFC busy beaver number[1]
- Many verifiers are tiny, and some are fast
- There are also multiple *proof assistants*

[1] `https://github.com/sorear/metamath-turing-machines`

Have a simple spec for the logical core

- Consequence: *Many* verifiers
  - There are 19 known verifiers
  - written in C, C++, C#, Rust, Lua, Haskell, Python, Igor, JavaScript, Mathematica, Julia, Scala, Java, Zig, Lean 4
  - ...and Turing Machine
    - A metamath verifier was adapted to prove the best known lower bound on the smallest unprovable-in-ZFC busy beaver number[1]
- Many verifiers are tiny, and some are fast
- There are also multiple *proof assistants*
  - The main ones in use are `MM-PA` and `mmj2`, and another one (`metamath-knife`) is in development

[1] `https://github.com/sorear/metamath-turing-machines`

Have a simple spec for the logical core

- Consequence: *Many* verifiers
  - There are 19 known verifiers
  - written in C, C++, C#, Rust, Lua, Haskell, Python, Igor, JavaScript, Mathematica, Julia, Scala, Java, Zig, Lean 4
  - ...and Turing Machine
    - A metamath verifier was adapted to prove the best known lower bound on the smallest unprovable-in-ZFC busy beaver number[1]
- Many verifiers are tiny, and some are fast
- There are also multiple *proof assistants*
  - The main ones in use are `MM-PA` and `mmj2`, and another one (`metamath-knife`) is in development
- Metamath has also been used for machine learning (Holophrasm, GPT-f)

[1] `https://github.com/sorear/metamath-turing-machines`

Metamath for AI/ML/ATP applications

- Metamath is a very friendly language for bulk processing, because it has such a simple grammar and few core concepts
  - In many cases you can get relevant and accurate information about theorem structure using regexes
  - There is only one kind of proof step (a theorem application), so proofs are just trees of applications and verification is uniform
- It also has a large body of human-curated mathematics, which is good for training and testing automated provers
- Verification and processing is quite fast, so the bottleneck is usually the external processing (the ATP, ML training etc)

Part II: Metamath Zero

Deficiencies of Metamath

- Metamath tries to simultaneously serve the human reader and the computer verifier, but they have divergent needs

```
$( Function with a domain of two different values.  (Contributed by FL,
   26-Jun-2011.)  (Revised by Mario Carneiro, 26-Apr-2015.) $)
fnprg $p ⊢ ( ( ( A e. V ∧ B e. W ) ∧ ( C e. X ∧ D e. Y ) ∧ A ≠ B )
        → { <. A , C >. , <. B , D >. } Fn { A , B } ) $=
  ( wcel wa wne w3a cop cpr wfun cdm wceq wfn funprg dmpropg 3ad2ant2 df-fn
  sylanbrc ) AEIBFIJZCGIDHIJZABKZLACMBDMNZOUGPABNZQZUGUHRABCDEFGHSUEUDUIUFACB
  DGHTUAUGUHUBUC $.

$( Function with a domain of three different values.  (Contributed by
   Alexander van der Vekens, 5-Dec-2017.) $)
fntpg $p ⊢ ( ( ( X e. U ∧ Y e. V ∧ Z e. W )
              ∧ ( A e. F ∧ B e. G ∧ C e. H )
              ∧ ( X ≠ Y ∧ X ≠ Z ∧ Y ≠ Z ) )
         → { <. X , A >. , <. Y , B >. , <. Z , C >. } Fn { X , Y , Z } ) $=
  ( wcel w3a wne cop cdm wceq csn cun ctp wfun wfn funtpg wa dmsnopg 3ad2ant1
  cpr 3ad2ant2 jca uneq12 syl df-pr syl6eqr dmeqi eqeq1i dmun sylibr 3ad2ant3
  bitri uneq12d df-tp eqtri 3eqtr4g df-fn sylanbrc ) JDMKHMLIMNZAEMZBFMZCGMZN
  ZJKOJLOKLONZNZJAPZKBPZLCPZUAZUBVQQZJKLUAZRVQVSUCABCDEFGHIJKLUDVMVNVOUHZQZVP
  SZQZTZJKUHZLSZTVRVSVMWAWEWCWFVMVNSZQZVOSZQZTZWERZWAWERZVMWKJSZKSZTZWEVMWHWN
  RZWJWORZUEZWKWPRVKVGWSVLVKWQWRVHVIWQVJJAEUFUGVIVHWRVJKBFUFUIUJUIWHWNWJWOUKU
  LJKUMUNWMWGWITZQZWERWLWAXAWEVTWTVNVOUMUOUPXAWKWEWGWIUQUPUTURVKVGWCWFRZVLVJV
  HXBVILCGUFUSUIVAVRVTWBTZQWDVQXCVNVOVPVBUOVTWBUQVCJKLVBVDVQVSVEVF $.

${
  fntp.1 $e ⊢ A e. _V $.
  fntp.2 $e ⊢ B e. _V $.
  fntp.3 $e ⊢ C e. _V $.
  fntp.4 $e ⊢ D e. _V $.
  fntp.5 $e ⊢ E e. _V $.
  fntp.6 $e ⊢ F e. _V $.
  $( A function with a domain of three elements.  (Contributed by NM,
     14-Sep-2011.)  (Revised by Mario Carneiro, 26-Apr-2015.) $)
  fntp $p ⊢ ( ( A ≠ B ∧ A ≠ C ∧ B ≠ C )
         → { <. A , D >. , <. B , E >. , <. C , F >. } Fn { A , B , C } ) $=
    ( wne w3a cop ctp wfun cdm wceq wfn funtp dmtpop a1i df-fn sylanbrc ) ABM
    ACMBCMNZADOBEOCFOPZQUGRABCPZSZUGUHTABCDEFGHIJKLUAUIUFADBECFJKLUBUCUGUHUDU
    E $.
$}
```

Deficiencies of Metamath

- Metamath tries to simultaneously serve the human reader and the computer verifier, but they have divergent needs
  - The big block of compressed proof text is very off-putting for newcomers, and not great for source control either

```
$( Function with a domain of two different values.  (Contributed by FL,
   26-Jun-2011.)  (Revised by Mario Carneiro, 26-Apr-2015.) $)
fnprg $p ⊢ ( ( ( A e. V ∧ B e. W ) ∧ ( C e. X ∧ D e. Y ) ∧ A ≠ B )
      → { <. A , C >. , <. B , D >. } Fn { A , B } ) $=
  ( wcel wa wne w3a cop cpr wfun cdm wceq wfn funprg dmpropg 3ad2ant2 df-fn
  sylanbrc ) AEIBFIJZCGIDHIJZABKZLACMBDMNZOUGPABNZQZUGUHRABCDEFGHSUEUDUIUFACB
  DGHTUAUGUHUBUC $.

$( Function with a domain of three different values.  (Contributed by
   Alexander van der Vekens, 5-Dec-2017.) $)
fntpg $p ⊢ ( ( ( X e. U ∧ Y e. V ∧ Z e. W )
             ∧ ( A e. F ∧ B e. G ∧ C e. H )
             ∧ ( X ≠ Y ∧ X ≠ Z ∧ Y ≠ Z ) )
       → { <. X , A >. , <. Y , B >. , <. Z , C >. } Fn { X , Y , Z } ) $=
  ( wcel w3a wne cop cdm wceq csn cun ctp wfun wfn funtpg wa dmsnopg 3ad2ant1
  cpr 3ad2ant2 jca uneq12 syl df-pr syl6eqr dmeqi eqeq1i dmun sylibr 3ad2ant3
  bitri uneq12d df-tp eqtri 3eqtr4g df-fn sylanbrc ) JDMKHLIMNZAEMZBFMZCGMZN
  ZJKOJLOKLONZNZJAPZKBPZLCPZUAZUBVQQZJKLUAZRVQVSUCABCDEFGHIJKLUDVMVNVOUHZQZVP
  SZQZTZJKUHZLSZTVRVSVMWAWEWCWFVMVNSZQZVOSZQZTZWERZWAWERZVMWKJSZKSZTZWEVMWHWN
  RZWJWORZUEZWKWPRVKVGWSVLVKWQWRVHVIWQVJJAEUFUGVIVHWRVJKBFUFUIUJUIWHWNWJWOUKU
  LJKUMUNWMWGWITZQZWERWLWAXAWEVTWTVNVOUMUOUPXAWKWEWGWIUQUPUTURVKVGWCWFRZVLVJV
  HXBVILCGUFUSUIVAVRVTWBTZQWDVQXCVNVOVPVBUOVTWBUQVCJKLVBVDVQVSVEVF $.

${
  fntp.1 $e ⊢ A e. _V $.
  fntp.2 $e ⊢ B e. _V $.
  fntp.3 $e ⊢ C e. _V $.
  fntp.4 $e ⊢ D e. _V $.
  fntp.5 $e ⊢ E e. _V $.
  fntp.6 $e ⊢ F e. _V $.
  $( A function with a domain of three elements.  (Contributed by NM,
     14-Sep-2011.)  (Revised by Mario Carneiro, 26-Apr-2015.) $)
  fntp $p ⊢ ( ( A ≠ B ∧ A ≠ C ∧ B ≠ C )
        → { <. A , D >. , <. B , E >. , <. C , F >. } Fn { A , B , C } ) $=
    ( wne w3a cop ctp wfun cdm wceq wfn funtp dmtpop a1i df-fn sylanbrc ) ABM
    ACMBCMNZADOBEOCFOPZQUGRABCPZSZUGUHTABCDEFGHIJKLUAUIUFADBECFJKLUBUCUGUHUDU
    E $.
$}
```

Deficiencies of Metamath

- Metamath tries to simultaneously serve the human reader and the computer verifier, but they have divergent needs
  - The big block of compressed proof text is very off-putting for newcomers, and not great for source control either
  - In practice you need a tool to read proofs

```
$( Function with a domain of two different values.  (Contributed by FL,
   26-Jun-2011.)  (Revised by Mario Carneiro, 26-Apr-2015.) $)
fnprg $p ⊢ ( ( ( A e. V ∧ B e. W ) ∧ ( C e. X ∧ D e. Y ) ∧ A ≠ B )
      → { <. A , C >. , <. B , D >. } Fn { A , B } ) $=
  ( wcel wa wne w3a cop cpr wfun cdm wceq wfn funprg dmpropg 3ad2ant2 df-fn
  sylanbrc ) AEIBFIJZCGIDHIJZABKZLACMBDMNZOUGPABNZQZUGUHRABCDEFGHSUEUDUIUFACB
  DGHTUAUGUHUBUC $.

$( Function with a domain of three different values.  (Contributed by
   Alexander van der Vekens, 5-Dec-2017.) $)
fntpg $p ⊢ ( ( ( X e. U ∧ Y e. V ∧ Z e. W )
              ∧ ( A e. F ∧ B e. G ∧ C e. H )
              ∧ ( X ≠ Y ∧ X ≠ Z ∧ Y ≠ Z ) )
         → { <. X , A >. , <. Y , B >. , <. Z , C >. } Fn { X , Y , Z } ) $=
  ( wcel w3a wne cop cdm wceq csn cun ctp wfun wfn funtpg wa dmsnopg 3ad2ant1
  cpr 3ad2ant2 jca uneq12 syl df-pr syl6eqr dmeqi eqeq1i dmun sylibr 3ad2ant3
  bitri uneq12d df-tp eqtri 3eqtr4g df-fn sylanbrc ) JDMKHMLIMZAFMHGMZCHMZNKZ
  ZJKOJLOKLONZNJAPZKBPZLCPZUAZUBVQQZJKLUAZRVQVSUCABCDEFGHIJKLUDVMVNVOUHZQZVP
  SZQZTZJKUHZLSZTVRVSVMWAWEWCWFVMVNSZQZVOSZQZTZWERZWAWERZVMWKJSZKSZTZWEVMWHWN
  RZWJWORZUEZWKWPRVKVGWSVLVKWQWRVHVIWQVJJAEUFUGVIVHWRVJKBFUFUIUJUIWHWNWJWOUKU
  LJKUMUNWMWGWITZQZWERWLWAXAWEVTWTVNVOUMUOUPXAWKWEWGWIUQUPUTURVKVGWCWFRZVLVJV
  HXBVILCGUFUSUIVAVRVTWBTZQWDVQXCVNVOVPVBUOVTWBUQVCJKLVBVDVQVSVEVF $.

${
  fntp.1 $e ⊢ A e. _V $.
  fntp.2 $e ⊢ B e. _V $.
  fntp.3 $e ⊢ C e. _V $.
  fntp.4 $e ⊢ D e. _V $.
  fntp.5 $e ⊢ E e. _V $.
  fntp.6 $e ⊢ F e. _V $.
  $( A function with a domain of three elements.  (Contributed by NM,
     14-Sep-2011.)  (Revised by Mario Carneiro, 26-Apr-2015.) $)
  fntp $p ⊢ ( ( A ≠ B ∧ A ≠ C ∧ B ≠ C )
        → { <. A , D >. , <. B , E >. , <. C , F >. } Fn { A , B , C } ) $=
    ( wne w3a cop ctp wfun cdm wceq wfn funtp dmtpop a1i df-fn sylanbrc ) ABM
    ACMBCMNZADOBEOCFOPZQUGRABCPZSZUGUHTABCDEFGHIJKLUAUIUFADBECFJKLUBUCUGUHUDU
    E $.
$}
```

Deficiencies of Metamath

- Metamath tries to simultaneously serve the human reader and the computer verifier, but they have divergent needs
  - The big block of compressed proof text is very off-putting for newcomers, and not great for source control either
  - In practice you need a tool to read proofs
- Metamath automation is decentralized
  - This is nice in principle, but in practice most people won't be writing their own proof assistant

```
$( Function with a domain of two different values.  (Contributed by FL,
   26-Jun-2011.)  (Revised by Mario Carneiro, 26-Apr-2015.) $)
fnprg $p ⊢ ( ( ( A e. V ∧ B e. W ) ∧ ( C e. X ∧ D e. Y ) ∧ A ≠ B )
      → { <. A , C >. , <. B , D >. } Fn { A , B } ) $=
  ( wcel wa wne w3a cop cpr wfun cdm wceq wfn funprg dmpropg 3ad2ant2 df-fn
  sylanbrc ) AEIBFIJZCGIDHIJZABKZLACMBDMNZOUGPABNZQZUGUHRABCDEFGHSUEUDUIUFACB
  DGHTUAUGUHUBUC $.

$( Function with a domain of three different values.  (Contributed by
   Alexander van der Vekens, 5-Dec-2017.) $)
fntpg $p ⊢ ( ( ( X e. U ∧ Y e. V ∧ Z e. W )
               ∧ ( A e. F ∧ B e. G ∧ C e. H )
               ∧ ( X ≠ Y ∧ X ≠ Z ∧ Y ≠ Z ) )
        → { <. X , A >. , <. Y , B >. , <. Z , C >. } Fn { X , Y , Z } ) $=
  ( wcel w3a wne cop cdm wceq csn cun ctp wfun wfn funtpg wa dmsnopg 3ad2ant1
  cpr 3ad2ant2 jca uneq12 syl df-pr syl6eqr dmeqi eqeq1i dmun sylibr 3ad2ant3
  bitri uneq12d df-tp eqtri 3eqtr4g df-fn sylanbrc ) JDMKHMLIMNZAEMZBFMNZCGMZN
  ZJKOJLOKLONZNZJAPZKBPZLCPZUAZUBVQQZJKLUAZRVQVSUCABCDEFGHIJKLUDVMVNVOUHZQZVP
  SZQZTZJKUHZLSZTVRVSVMWAWEWCWFVMVNSZQZVOSZQZTZWERZWAWERZVMWKJSZKSZTZWEVMWHWN
  RZWJWORZUEZWKWPRVKVGWSVLVKWQWRVHVIWQVJJAEUFUGVIVHWRVJKBFUFUIUJUIWHWNWJWOUKU
  LJKUMUNWMWGWITZQZWERWLWAXAWEVTWTVNVOUMUOUPXAWKWEWGWIUQUPUTURVKVGWCWFRZVLVJV
  HXBVILCGUFUSUIVAVRVTWBTZQWDVQXCVNVOVPVBUOVTWBUQVCJKLVBVDVQVSVEVF $.

${
  fntp.1 $e ⊢ A e. _V $.
  fntp.2 $e ⊢ B e. _V $.
  fntp.3 $e ⊢ C e. _V $.
  fntp.4 $e ⊢ D e. _V $.
  fntp.5 $e ⊢ E e. _V $.
  fntp.6 $e ⊢ F e. _V $.
  $( A function with a domain of three elements.  (Contributed by NM,
     14-Sep-2011.)  (Revised by Mario Carneiro, 26-Apr-2015.) $)
  fntp $p ⊢ ( ( A ≠ B ∧ A ≠ C ∧ B ≠ C )
        → { <. A , D >. , <. B , E >. , <. C , F >. } Fn { A , B , C } ) $=
    ( wne w3a cop ctp wfun cdm wceq wfn funtp dmtpop a1i df-fn sylanbrc ) ABM
    ACMBCMNZADOBEOCFOPZQUGRABCPZSZUGUHTABCDEFGHIJKLUAUIUFADBECFJKLUBUCUGUHUDU
    E $.
$}
```

Deficiencies of Metamath

- Metamath tries to simultaneously serve the human reader and the computer verifier, but they have divergent needs
  - The big block of compressed proof text is very off-putting for newcomers, and not great for source control either
  - In practice you need a tool to read proofs
- Metamath automation is decentralized
  - This is nice in principle, but in practice most people won't be writing their own proof assistant
  - Metamath has a reputation for having no automation as a result

```
$( Function with a domain of two different values.  (Contributed by FL,
   26-Jun-2011.)  (Revised by Mario Carneiro, 26-Apr-2015.) $)
fnprg $p ⊢ ( ( ( A e. V ∧ B e. W ) ∧ ( C e. X ∧ D e. Y ) ∧ A ≠ B )
        → { <. A , C >. , <. B , D >. } Fn { A , B } ) $=
  ( wcel wa wne w3a cop cpr wfun cdm wceq wfn funprg dmpropg 3ad2ant2 df-fn
  sylanbrc ) AEIBFIJZCGIDHIJZABKZLACMBDMNZOUGPABNZQZUGUHRABCDEFGHSUEUDUIUFACB
  DGHTUAUGUHUBUC $.

$( Function with a domain of three different values.  (Contributed by
   Alexander van der Vekens, 5-Dec-2017.) $)
fntpg $p ⊢ ( ( ( X e. U ∧ Y e. V ∧ Z e. W )
                ∧ ( A e. F ∧ B e. G ∧ C e. H )
                ∧ ( X ≠ Y ∧ X ≠ Z ∧ Y ≠ Z ) )
          → { <. X , A >. , <. Y , B >. , <. Z , C >. } Fn { X , Y , Z } ) $=
  ( wcel w3a wne cop cdm wceq csn cun ctp wfun wfn funtpg wa dmsnopg 3ad2ant1
  cpr 3ad2ant2 jca uneq12 syl df-pr syl6eqr dmeqi eqeq1i dmun sylibr 3ad2ant3
  bitri uneq12d df-tp eqtri 3eqtr4g df-fn sylanbrc ) JDMKHMLIMNZAEMZBFMZCGMZN
  ZJKOJLOKLONZNZJAPZKBPZLCPZUAZUBVQQZJKLUAZRVQVSUCABCDEFGHIJKLUDVMVNVOUHZQZVP
  SZQZTZJKUHZLSZTVRVSVMWAWEWCWFVMVNSZQZVOSZQZTZWERZWAWERZVMWKJSZKSZTZWEVMWHWN
  RZWJWORZUEZWKWPRVKVGWSVLVKWQWRVHVIWQVJJAEUFUGVIVHWRVJKBFUFUIUJUIWHWNWJWOUKU
  LJKUMUNWMWGWITZQZWERWLWAXAWEVTWTVNVOUMUOUPXAWKWEWGWIUQUPUTURVKVGWCWFRZVLVJV
  HXBVILCGUFUSUIVAVRVTWBTZQWDVQXCVNVOVPVBUOVTWBUQVCJKLVBVDVQVSVEVF $.

${
  fntp.1 $e ⊢ A e. _V $.
  fntp.2 $e ⊢ B e. _V $.
  fntp.3 $e ⊢ C e. _V $.
  fntp.4 $e ⊢ D e. _V $.
  fntp.5 $e ⊢ E e. _V $.
  fntp.6 $e ⊢ F e. _V $.
  $( A function with a domain of three elements.  (Contributed by NM,
     14-Sep-2011.)  (Revised by Mario Carneiro, 26-Apr-2015.) $)
  fntp $p ⊢ ( ( A ≠ B ∧ A ≠ C ∧ B ≠ C )
          → { <. A , D >. , <. B , E >. , <. C , F >. } Fn { A , B , C } ) $=
    ( wne w3a cop ctp wfun cdm wceq wfn funtp dmtpop a1i df-fn sylanbrc ) ABM
    ACMBCMNZADOBEOCFOPZQUGRABCPZSZUGUHTABCDEFGHIJKLUAUIUFADBECFJKLUBUCUGUHUDU
    E $.
$}
```

Deficiencies of Metamath

- Metamath tries to simultaneously serve the human reader and the computer verifier, but they have divergent needs
  - The big block of compressed proof text is very off-putting for newcomers, and not great for source control either
  - In practice you need a tool to read proofs
- Metamath automation is decentralized
  - This is nice in principle, but in practice most people won't be writing their own proof assistant
  - Metamath has a reputation for having no automation as a result
  - Existing MM proof assistants are certainly lacking in small scale automation compared to HOL light, Isabelle, Coq, Lean

```
$( Function with a domain of two different values.  (Contributed by FL,
   26-Jun-2011.)  (Revised by Mario Carneiro, 26-Apr-2015.) $)
fnprg $p ⊢ ( ( ( A e. V ∧ B e. W ) ∧ ( C e. X ∧ D e. Y ) ∧ A ≠ B )
      → { <. A , C >. , <. B , D >. } Fn { A , B } ) $=
  ( wcel wa wne w3a cop cpr wfun cdm wceq wfn funprg dmpropg 3ad2ant2 df-fn
  sylanbrc ) AEIBFIJZCGIDHIJZABKZLACMBDMNZOUGPABNZQZUGUHRABCDEFGHSUEUDUIUFACB
  DGHTUAUGUHUBUC $.

$( Function with a domain of three different values.  (Contributed by
   Alexander van der Vekens, 5-Dec-2017.) $)
fntpg $p ⊢ ( ( ( X e. U ∧ Y e. V ∧ Z e. W )
             ∧ ( A e. F ∧ B e. G ∧ C e. H )
             ∧ ( X ≠ Y ∧ X ≠ Z ∧ Y ≠ Z ) )
         → { <. X , A >. , <. Y , B >. , <. Z , C >. } Fn { X , Y , Z } ) $=
  ( wcel w3a wne cop cdm wceq csn cun ctp wfun wfn funtpg wa dmsnopg 3ad2ant1
  cpr 3ad2ant2 jca uneq12 syl df-pr syl6eqr dmeqi eqeq1i dmun sylibr 3ad2ant3
  bitri uneq12d df-tp eqtri 3eqtr4g df-fn sylanbrc ) JDMKHMLIMZNZAGOBFOCEOZNZ
  ZJKOLOZNZJKOLOZNZJKOLOSNVQNVPSNCDPJKPLVOPSNVLPQZKFPLLZKRLZKRJRPNP
  SZQZTZJKUHZLSZTVRVSVMWAWEWCWFVMVNSZQZVOSZQZTZWERZWAWERZVMWKJSZKSZTZWEVMWHWN
  RZWJWORZUEZWKWPRVKVGWSVLVKWQWRVHVIWQVJJAEUFUGVIVHWRVJKBFUFUIUJUIWHWNWJWOUKU
  LJKUMUNWMWGWITZQZWERWLWAXAWEVTWTVNVOUMUOUPXAWKWEWGWIUQUPUTURVKVGWCWFRZVLVJV
  HXBVILCGUFUSUIVAVRVTWBTZQWDVQXCVNVOVPVBUOVTWBUQVCJKLVBVDVQVSVEVF $.

${
  fntp.1 $e ⊢ A e. _V $.
  fntp.2 $e ⊢ B e. _V $.
  fntp.3 $e ⊢ C e. _V $.
  fntp.4 $e ⊢ D e. _V $.
  fntp.5 $e ⊢ E e. _V $.
  fntp.6 $e ⊢ F e. _V $.
  $( A function with a domain of three elements.  (Contributed by NM,
     14-Sep-2011.)  (Revised by Mario Carneiro, 26-Apr-2015.) $)
  fntp $p ⊢ ( ( A ≠ B ∧ A ≠ C ∧ B ≠ C )
        → { <. A , D >. , <. B , E >. , <. C , F >. } Fn { A , B , C } ) $=
    ( wne w3a cop ctp wfun cdm wceq wfn funtp dmtpop a1i df-fn sylanbrc ) ABM
    ACMBCMNZADOBEOCFOPZQUGRABCPZSUGUHTABCDEFGHIJKLUAUIUFADBECFJKLUBUCUGUHUDU
    E $.
$}
```

Metamath Zero

- Metamath Zero is a project I started in 2019 to solve the problem of bootstrapping a theorem prover

Metamath Zero

- Metamath Zero is a project I started in 2019 to solve the problem of bootstrapping a theorem prover
- Double down on all the things that make Metamath great for metatheory
  - Keep it simple, but expressive

Metamath Zero

- Metamath Zero is a project I started in 2019 to solve the problem of bootstrapping a theorem prover
- Double down on all the things that make Metamath great for metatheory
  - Keep it simple, but expressive
  - The "one rule" of Metamath is a universal computing machine – users can effectively write the language they want to verify using lemmas

Metamath Zero

- Metamath Zero is a project I started in 2019 to solve the problem of bootstrapping a theorem prover
- Double down on all the things that make Metamath great for metatheory
  - Keep it simple, but expressive
  - The "one rule" of Metamath is a universal computing machine – users can effectively write the language they want to verify using lemmas
- Make it more automation friendly

Metamath Zero

- Metamath Zero is a project I started in 2019 to solve the problem of bootstrapping a theorem prover
- Double down on all the things that make Metamath great for metatheory
  - Keep it simple, but expressive
  - The "one rule" of Metamath is a universal computing machine – users can effectively write the language they want to verify using lemmas
- Make it more automation friendly
  - Fix some asymptotic complexity issues in metamath (DAG sharing all the things)

Metamath Zero

- Metamath Zero is a project I started in 2019 to solve the problem of bootstrapping a theorem prover
- Double down on all the things that make Metamath great for metatheory
  - Keep it simple, but expressive
  - The "one rule" of Metamath is a universal computing machine – users can effectively write the language they want to verify using lemmas
- Make it more automation friendly
  - Fix some asymptotic complexity issues in metamath (DAG sharing all the things)
  - Use trees for the internal representation instead of token strings

Metamath Zero

- Metamath Zero is a project I started in 2019 to solve the problem of bootstrapping a theorem prover
- Double down on all the things that make Metamath great for metatheory
  - Keep it simple, but expressive
  - The "one rule" of Metamath is a universal computing machine – users can effectively write the language they want to verify using lemmas
- Make it more automation friendly
  - Fix some asymptotic complexity issues in metamath (DAG sharing all the things)
  - Use trees for the internal representation instead of token strings
  - Have a metaprogramming language for the front end (a tactic language)

Metamath Zero

- Metamath Zero is a project I started in 2019 to solve the problem of bootstrapping a theorem prover
- Double down on all the things that make Metamath great for metatheory
  - Keep it simple, but expressive
  - The "one rule" of Metamath is a universal computing machine – users can effectively write the language they want to verify using lemmas
- Make it more automation friendly
  - Fix some asymptotic complexity issues in metamath (DAG sharing all the things)
  - Use trees for the internal representation instead of token strings
  - Have a metaprogramming language for the front end (a tactic language)
- Give it a more modern-looking syntax
  - shamelessly borrowed from Lean

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 – bootstrap!

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[2] – bootstrap!

[2]Currently the MMC verifier for MM0 is not finished, but there are multiple other MM0 verifiers so it can already be used without the bootstrap.

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:

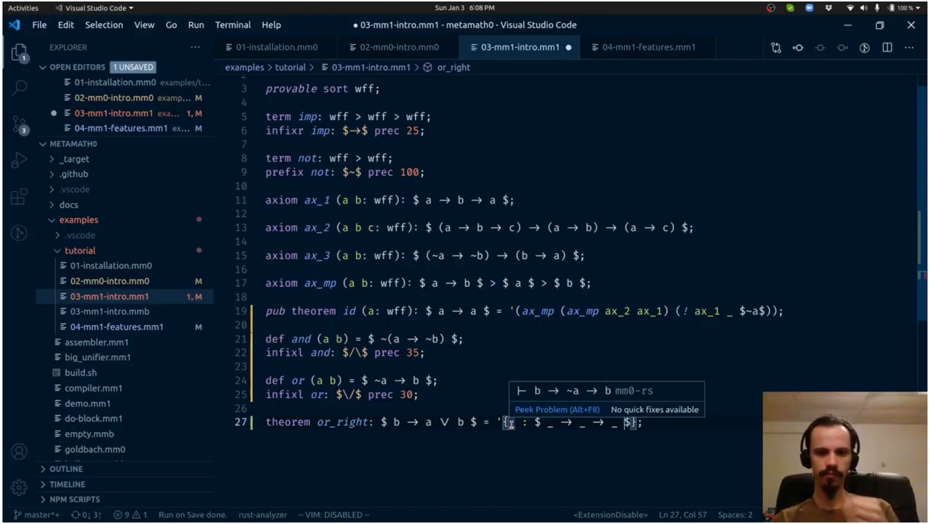
```
theorem _: $ ,19 * ,120 + ,2 = ,2282 $ = norm_num;
```

  - `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, `0x13` · `0x78` + `0x2` = `0x8ea` which is the theorem written in hexadecimal

Working with MM1



From “Metamath Zero (MM0/MM1) tutorial”, `https://youtu.be/A7WfrW7-ifw`

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`)

Metamath C

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

Program correctness

Program P is correct to specification T 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 $\mathrm{input\_consumed}(s') = I$ and $\mathrm{output\_produced}(s') = O$, then $T(I, O)$ is true.

- Red: definitions from `x86.mm0`
- Blue: the user specification

Metamath C

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

Program correctness

Program P is correct to specification T 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 $\mathrm{input\_consumed}(s') = I$ and $\mathrm{output\_produced}(s') = O$, then $T(I, O)$ is true.

- Red: definitions from `x86.mm0`
- Blue: the user specification

- 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

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

- 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 $x + y < 2^{32}$
  - `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
$\|\tau\|$	`moved(`τ`)`	$\left\|\boxed{a : \tau}\right\|$	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 current state of the MM0 project

- MM0 is a new system with not many users, and does not compare to Metamath in terms of formalized material

The current state of the MM0 project

- MM0 is a new system with not many users, and does not compare to Metamath in terms of formalized material
- The project is being developed as open source, and contributions are welcome

The current state of the MM0 project

- MM0 is a new system with not many users, and does not compare to Metamath in terms of formalized material
- The project is being developed as open source, and contributions are welcome
- There are several MM0 verifiers, written in C, Rust, Haskell

The current state of the MM0 project

- MM0 is a new system with not many users, and does not compare to Metamath in terms of formalized material
- The project is being developed as open source, and contributions are welcome
- There are several MM0 verifiers, written in C, Rust, Haskell
- The MMC verifier for MM0 is under construction

The current state of the MM0 project

- MM0 is a new system with not many users, and does not compare to Metamath in terms of formalized material
- The project is being developed as open source, and contributions are welcome
- There are several MM0 verifiers, written in C, Rust, Haskell
- The MMC verifier for MM0 is under construction
- The MM1 proof assistant is fairly stable and has already been used for some pretty big formalization work

The current state of the MM0 project

- MM0 is a new system with not many users, and does not compare to Metamath in terms of formalized material
- The project is being developed as open source, and contributions are welcome
- There are several MM0 verifiers, written in C, Rust, Haskell
- The MMC verifier for MM0 is under construction
- 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, but is still very experimental

The current state of the MM0 project

- MM0 is a new system with not many users, and does not compare to Metamath in terms of formalized material
- The project is being developed as open source, and contributions are welcome
- There are several MM0 verifiers, written in C, Rust, Haskell
- The MMC verifier for MM0 is under construction
- 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, but is still very experimental

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

On program synthesis

- MMC has a strong type system, and things that typecheck must follow their functional specification

On program synthesis

- MMC has a strong type system, and things that typecheck must follow their functional specification
- Program synthesis is necessarily doing deductive verification

On program synthesis

- MMC has a strong type system, and things that typecheck must follow their functional specification
- Program synthesis is necessarily doing deductive verification
- What this language brings to the table is to take those high level proofs and lower them to fully formal proofs about the resulting assembly code

Conclusion

- Metamath is a really simple language which can express complex math

Conclusion

- Metamath is a really simple language which can express complex math
- But we can get the benefits without making things hard for users

Conclusion

- Metamath is a really simple language which can express complex math
- But we can get the benefits without making things hard for users
- The MM1 proof assistant is my vision for how to marry a Metamath-like backend to a Lean-like frontend, and you can play with it today

Conclusion

- Metamath is a really simple language which can express complex math
- But we can get the benefits without making things hard for users
- The MM1 proof assistant is my vision for how to marry a Metamath-like backend to a Lean-like frontend, and you can play with it today
- The MMC language design is similar to a programming language with contracts like Dafny / Why3, but unlike these the proofs aren't just "skin deep", they are synthesized into a full proof at the low level, through the entire compiler

Conclusion

- Metamath is a really simple language which can express complex math
- But we can get the benefits without making things hard for users
- The MM1 proof assistant is my vision for how to marry a Metamath-like backend to a Lean-like frontend, and you can play with it today
- The MMC language design is similar to a programming language with contracts like Dafny / Why3, but unlike these the proofs aren't just "skin deep", they are synthesized into a full proof at the low level, through the entire compiler
- It still remains to be seen if these kind of languages are 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.

Resources

- Metamath: `http://us.metamath.org/`
- Metamath Zero: `https://github.com/digama0/mm0`
- MM0 Youtube tutorial: `https://youtu.be/A7WfrW7-ifw`
- MM0 thesis: `https://digama0.github.io/mm0/thesis.pdf`
- Lean/mathlib: `http://leanprover-community.github.io/`
- Lean Zulip chat: `https://leanprover.zulipchat.com/`
    - Ask me anything on Zulip, I'm there a lot

Thanks!