Types for Programs and Proofs

Lecture 2

Thierry Coquand

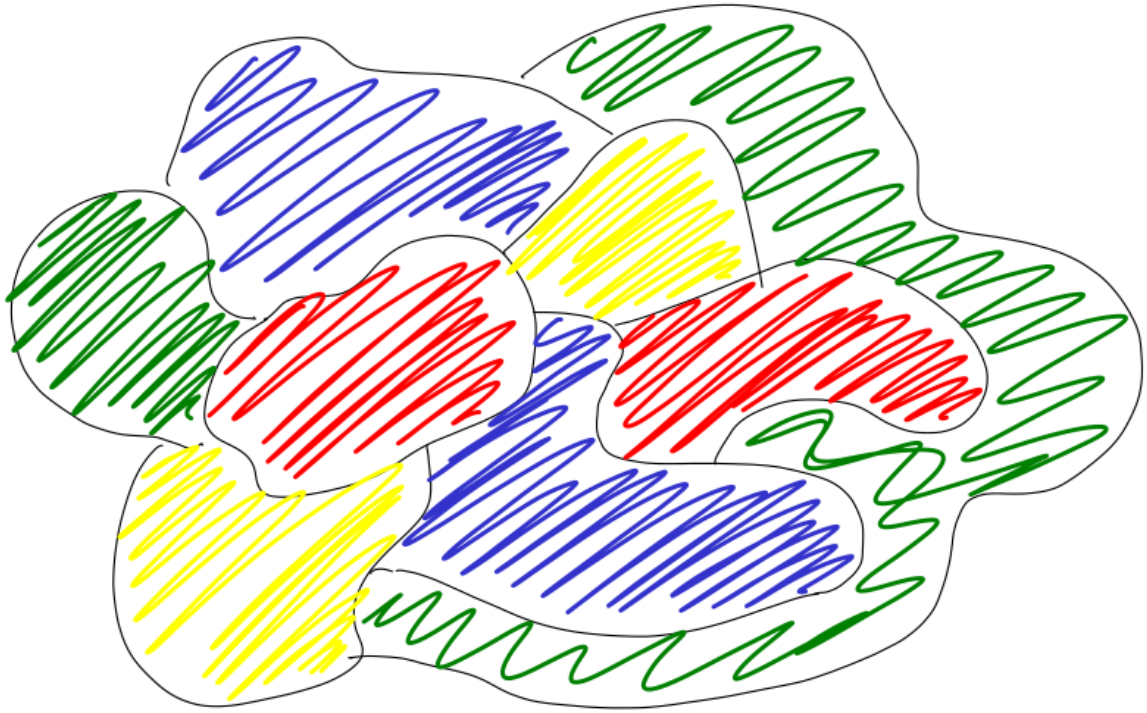
Applications for Proofs

I will use the excellent slides of Mike Shulman, April 2025

Outline

The four-color conjecture

In 1852 Francis Guthrie asked whether it is always possible to color a map so that any two countries that share a border have different colors, using no more than four colors.



The four-color ~~theorem~~ ~~conjecture~~ theorem?

1852 Francis Guthrie makes the conjecture.

The four-color ~~theorem~~ ~~conjecture~~ theorem?

1852 Francis Guthrie makes the conjecture.

1879 Alfred Kempe publishes a proof.

The four-color ~~theorem~~ ~~conjecture~~ theorem?

1852 Francis Guthrie makes the conjecture.

1879 Alfred Kempe publishes a proof.

1890 Percy Heawood finds a subtle flaw in Kempe's proof.

The four-color ~~theorem~~ ~~conjecture~~ theorem?

1852 Francis Guthrie makes the conjecture.

1879 Alfred Kempe publishes a proof.

1890 Percy Heawood finds a subtle flaw in Kempe's proof.

1976 Kenneth Appel and Wolfgang Haken show that Kempe's proof can fail in no more than 1,834 different ways, and use a computer to check that none of those is actually possible.

The four-color ~~theorem~~ ~~conjecture~~ theorem?

1852 Francis Guthrie makes the conjecture.

1879 Alfred Kempe publishes a proof.

1890 Percy Heawood finds a subtle flaw in Kempe's proof.

1976 Kenneth Appel and Wolfgang Haken show that Kempe's proof can fail in no more than 1,834 different ways, and use a computer to check that none of those is actually possible.

1981 Ulrich Schmidt is rumored to have found a flaw in Appel and Haken's proof.

The four-color ~~theorem~~ ~~conjecture~~ theorem?

1852 Francis Guthrie makes the conjecture.

1879 Alfred Kempe publishes a proof.

1890 Percy Heawood finds a subtle flaw in Kempe's proof.

1976 Kenneth Appel and Wolfgang Haken show that Kempe's proof can fail in no more than 1,834 different ways, and use a computer to check that none of those is actually possible.

1981 Ulrich Schmidt is rumored to have found a flaw in Appel and Haken's proof.

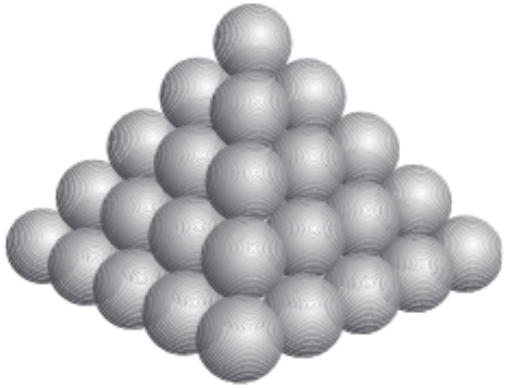
1986 Appel and Haken say that Schmidt's results have been misinterpreted and the proof is correct.

> *"...we plan to publish...an entire emended version of our original proof...one might think that we would miss the pleasures of discussing the latest rumors with our colleagues...but further consideration leads us to believe that facts have never stopped the propagation of a good rumor and so nothing much will change."*

The Kepler conjecture

In 1611, Johannes Kepler conjectured there was no better way of stacking cannonballs on the deck of a ship than the obvious one.

1611



1998

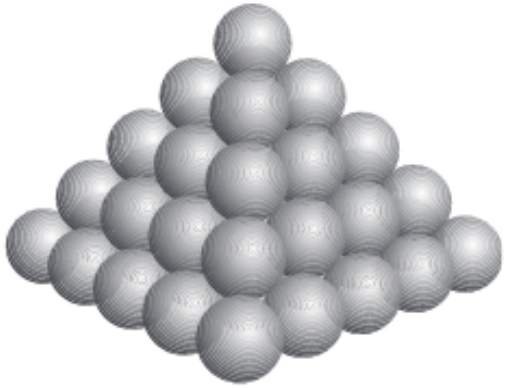
+ 3GB data

After nearly 400 years, Thomas Hales proved this conjecture, again using a computer to check a large number of cases.

The Kepler conjecture

In 1611, Johannes Kepler conjectured there was no better way of stacking cannonballs on the deck of a ship than the obvious one.

1611



1998

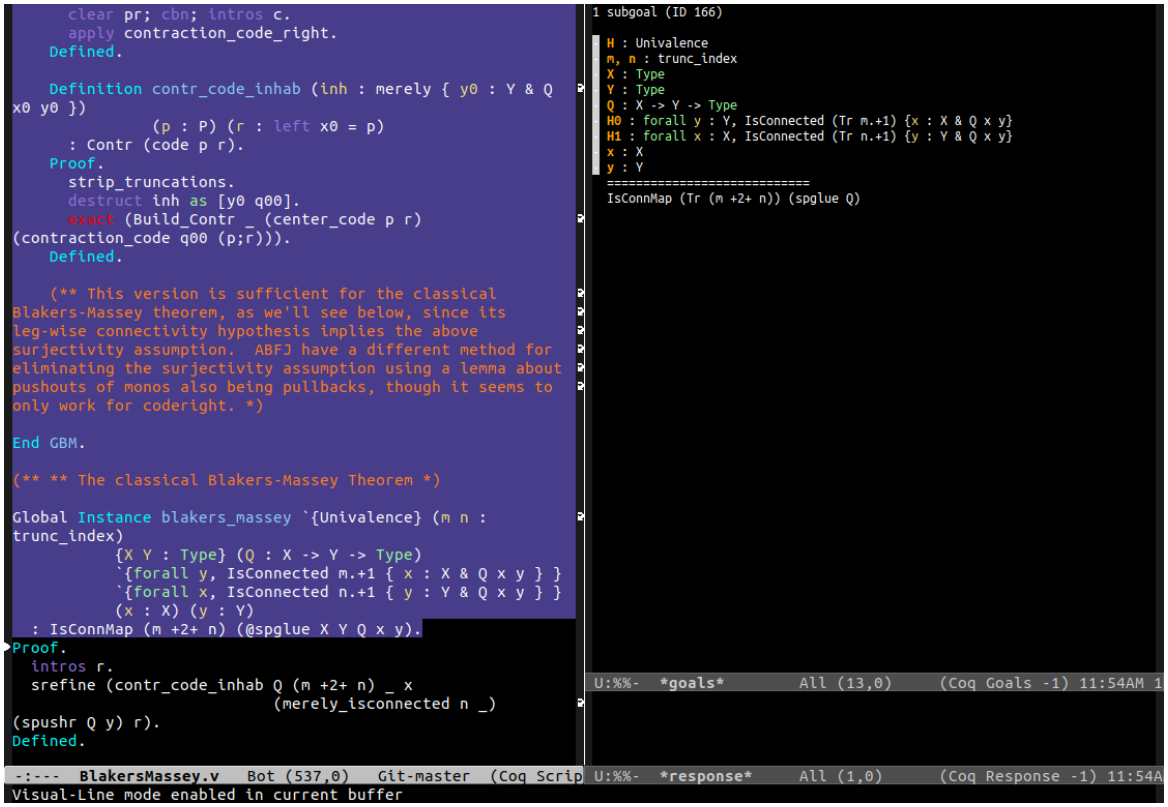
+ 3GB data

After nearly 400 years, Thomas Hales proved this conjecture, again using a computer to check a large number of cases.

> *"The verdict of the referees was that the proof seemed to work, but they just did not have the time or energy to verify everything comprehensively. . . the proof was seemingly beyond the ability of the mathematics community to check thoroughly."* – *Henry Cohn*

Formal verification

By programming a computer to understand proofs, it can check the correctness of a theorem or another program. The programming language that makes this possible is called a proof assistant.



The Four-Color THEOREM

In 2005, a team led by Benjamin Werner and Georges Gonthier formally verified the Appel-Haken proof of the four-color theorem using a computer proof assistant.

Formal Proof—The Four-Color Theorem

Georges Gonthier

The Tale of a Brainteaser

Francis Guthrie certainly did it, when he coined his innocent little coloring puzzle in 1852. He managed to embarrass successively his mathematician brother, his brother's professor, Augustus de Morgan, and all of de Morgan's visitors, who couldn't solve it; the Royal Society, who only realized ten years later that Alfred Kempe's 1879 solution was wrong; and the three following generations of mathematicians who couldn't fix it [19].

Even Appel and Haken's 1976 triumph [2] had a hint of defeat: they'd had a computer do the proof for them! Perhaps the mathematical controversy around the proof died down with their book [3] and with the elegant 1995 revision [13] by Robertson, Saunders, Seymour, and Thomas. However something was still amiss: both proofs combined a textual argument, which could reasonably be checked by inspection, with computer code that could not. Worse, the empirical evidence provided by running code several times with the *same* input is weak, as it is blind to the most common cause of "computer" error: programmer error.

For some thirty years, computer science has been working out a solution to this problem: formal program proofs. The idea is to write code that describes not only *what* the machine should do, but also *why* it should be doing it—a formal proof of correctness. The validity of the proof is an objective mathematical fact that can be checked by a *different* program, whose own validity can be ascertained empirically because it does run on *many* inputs. The main technical difficulty is that formal proofs are very difficult to produce, even with a language rich enough to express all mathematics.

In 2000 we tried to produce such a proof for part of code from [13], just to evaluate how the field had progressed. We succeeded, but now a new question emerged: was the statement of the correctness proof (the *specification*) itself correct? The only solution to that conundrum was to formalize the *entire* proof of the Four-Color Theorem, not just its code. This we finally achieved in 2005.

While we tackled this project mainly to explore the capabilities of a modern formal proof system—at first, to benchmark speed—we were pleasantly surprised to uncover new and rather elegant nuggets of mathematics in the process. In hindsight this might have been expected: to produce a formal proof one must make explicit every single logical step of a proof; this both provides new insight in the structure of the proof, and forces one to use this insight to discover every possible symmetry, simplification, and generalization, if only to cope with the sheer amount of imposed detail. This is actually how all of sections "Combinatorial Hypermaps" (p. 1385) and "The Formal Theorem" (p. 1388) came about. Perhaps this is the most promising aspect of formal proof: it is not merely a method to make absolutely sure we have not made a mistake in a proof, but also a tool that shows us and compels us to understand why a proof works.

In this article, the next two sections contain background material, describing the original proof and the Coq formal system we used. The following two sections describe the sometimes new mathematics involved in the formalization. Then the next two sections go into some detail into the two main parts of the formal proof: reducibility and

Georges Gonthier is a senior researcher at Microsoft Research Cambridge. His email address is gonthier@microsoft.com.

Hales's THEOREM

In 2017, Hales himself led a team that formally verified his proof of the Kepler conjecture.

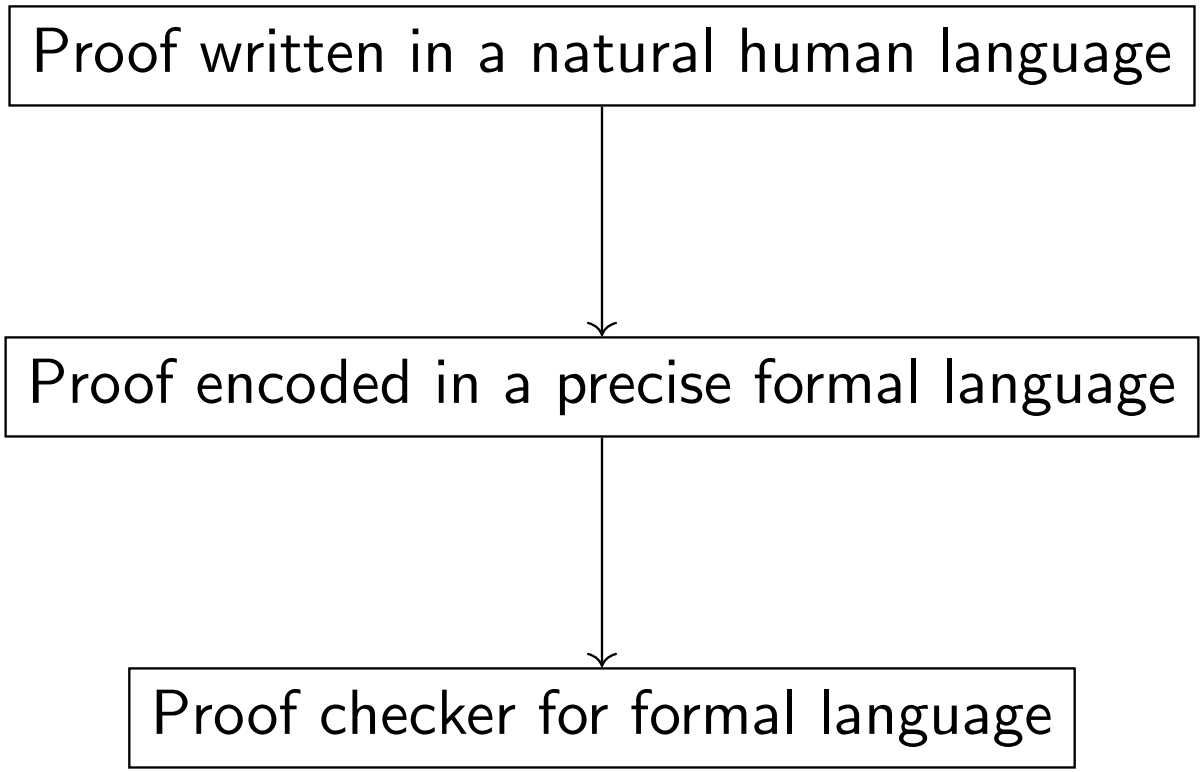
Forum of Mathematics, Pi (2017), Vol. 5, e2, 29 pages
doi:10.1017/fmp.2017.1

A FORMAL PROOF OF THE KEPLER CONJECTURE

THOMAS HALES[1], MARK ADAMS[2,3], GERTRUD BAUER[4],
TAT DAT DANG[5], JOHN HARRISON[6], LE TRUONG HOANG[7],
CEZARY KALISZYK[8], VICTOR MAGRON[9], SEAN MCLAUGHLIN[10],
TAT THANG NGUYEN[7], QUANG TRUONG NGUYEN[1],
TOBIAS NIPKOW[11], STEVEN OBUA[12], JOSEPH PLESO[13], JASON RUTE[14],
ALEXEY SOLOVYEV[15], THI HOAI AN TA[7], NAM TRUNG TRAN[7],
THI DIEP TRIEU[16], JOSEF URBAN[17], KY VU[18] and
ROLAND ZUMKELLER[19]

Outline

How does it work?



So what is a proof, anyway?

Somehow, mathematicians got by for 2000+ years with not much more formal analysis of this question than Aristotle's:

All men are mortal.
Socrates is a man.
Therefore, Socrates is mortal.

General formal notions of proof were finally discovered in the late 19th century by George Boole, Augustus De Morgan, Charles S. Peirce, Gottlob Frege, and others.

Proofs have a definition!

All formal systems for proof share the same basic framework:

- There is a finite and complete list of all the "rules" that can be used in a proof.
- A proof is anything that is constructed by applying these rules.
- Every proof is constructed by using these rules.
- Something that is constructed without using these rules is not a proof.

This is what enables us to write a computer proof checker: it just has to verify that each rule is used correctly.

First-order logic

The most basic formal system for proof is called first-order logic.
The things we prove are called logical formulas, built as follows:

- There are "atomic" formulas, like $x^2 = 3y + 2$ or $a + b < c$.
- If P and Q are formulas, so are
  - $P \wedge Q$, meaning "P and Q"
  - $P \vee Q$, meaning "P or Q"
  - $P \Rightarrow Q$, meaning "if P then Q"
  - $\neg P$, meaning "not P"
- If $P(x)$ is a formula possibly involving a variable x belonging to a set A, then we have formulas:
  - $\exists x \in A.P(x)$, meaning "there exists an $x \in A$ such that $P(x)$"
  - $\forall x \in A.P(x)$, meaning "for all $x \in A$, $P(x)$"

Nested quantifiers

Example

$\forall x \in \mathbb{R}.\exists y \in \mathbb{R}.(x < y)$

"For every real number x, there is a real number y such that $x < y$."

True!

Example

$\exists y \in \mathbb{R}.\forall x \in \mathbb{R}.(x < y)$

"There is a real number y such that for any real number x, $x < y$."

False!

Nested quantifiers

Example

$\forall x \in \mathbb{R}.\exists y \in \mathbb{R}.(x < y)$

"For every real number x, there is a real number y such that $x < y$."

True!

Example

$\exists y \in \mathbb{R}.\forall x \in \mathbb{R}.(x < y)$

"There is a real number y such that for any real number x, $x < y$."

False!

Example

$\forall \varepsilon \in \mathbb{R}_{>0}.\exists \delta \in \mathbb{R}_{>0}.\forall x \in \mathbb{R}.(|x - a| < \delta \Rightarrow |f(x) - L| < \varepsilon)$

"For all positive real numbers ε, there exists a positive real number δ such that for all real numbers x, if $|x - a| < \delta$, then $|f(x) - L| < \varepsilon$."

Natural deduction

Each operator ($\wedge$, $\vee$, $\exists$, $\forall$, ...) is "defined" or "explained" by (1) rule(s) to prove it and (2) rule(s) to use it to prove other things.

Example ($\Rightarrow$, "if-then")

- To prove $P \Rightarrow Q$, we assume P and use this to prove Q.
- To use $P \Rightarrow Q$, if we know or can prove P, we can deduce Q.

Example ($\vee$, "or")

- To prove $P \vee Q$, we can prove P.
- To prove $P \vee Q$, we can prove Q.
- To use $P \vee Q$, we can divide the proof into cases, one of which assumes P, and the other of which assumes Q.

Natural deduction for quantifiers

Example ($\exists$, "there exists")

- To prove $\exists x \in A.\, P(x)$, specify some $a \in A$ and prove $P(a)$.
- To use $\exists x \in A.\, P(x)$, assume $x \in A$ such that $P(x)$.

Example ($\forall$, "for all")

- To prove $\forall x \in A.\, P(x)$, assume $x \in A$ and prove $P(x)$.
- To use $\forall x \in A.\, P(x)$, specify some $a \in A$ and deduce $P(a)$.

Natural deduction for quantifiers

Example ($\exists$, "there exists")

- To prove $\exists x \in A.\, P(x)$, specify some $a \in A$ and prove $P(a)$.
- To use $\exists x \in A.\, P(x)$, assume $x \in A$ such that $P(x)$.

Example ($\forall$, "for all")

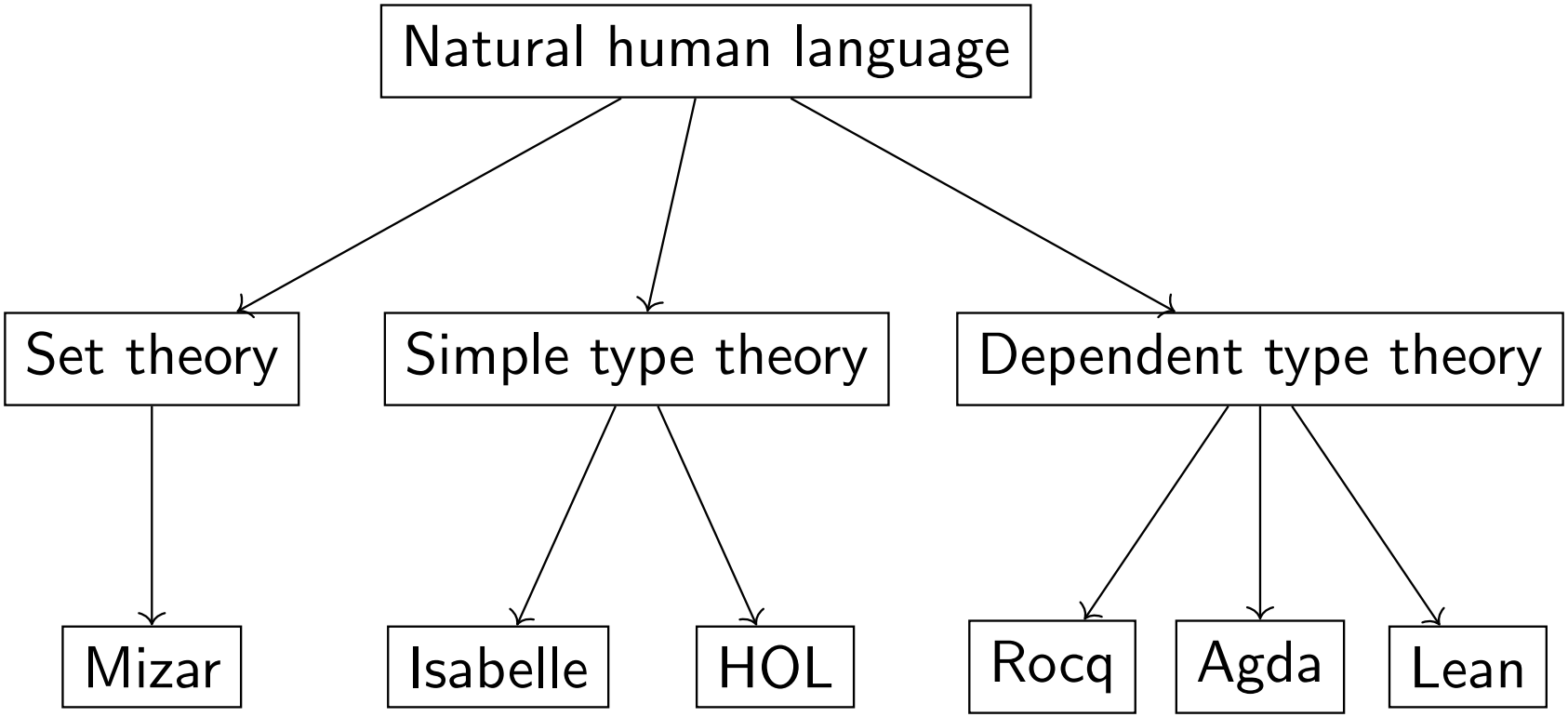
- To prove $\forall x \in A.\, P(x)$, assume $x \in A$ and prove $P(x)$.
- To use $\forall x \in A.\, P(x)$, specify some $a \in A$ and deduce $P(a)$.

All men are mortal.	$\forall x \in \mathrm{Men}.\, \mathrm{Mortal}(x)$
Socrates is a man.	$\mathrm{Socrates} \in \mathrm{Men}$
Therefore, Socrates is mortal.	$\vdash \mathrm{Mortal}(\mathrm{Socrates})$

But first-order logic is much more powerful and flexible than Aristotelian syllogistic.

The proliferation of proof assistants

Olorin isn't flexible enough to formalize all of mathematics, but many proof assistants are:



All formal languages incorporate first-order logic in some way. But they disagree about where to go from there...

Outline

Set theory

First-order logic is great for reasoning about numbers, sets, functions, etc.

But how do we construct numbers, sets, functions?

Answer #1: set theory

- Everything is coded as a set.
- Sets can be heterogeneous: $\{17, \mathrm{blue}, \mathbb{Q}\}$
- Basically one way to construct sets: $\{x \mid P(x)\}$.
- Axioms written in first-order logic say when this is allowed.
  - $\exists x.\forall y.y \notin x$ gives the empty set $\emptyset = \{\}$.
  - $\forall x.\forall y.\exists z.(\forall w.w \in z \iff w = x \vee x = y)$ gives $\{x, y\}$.
  - ...

Type theory

Answer #2: type theory

- Many types of objects: numbers, ordered pairs, functions, ...
  Each is primitive and not coded in terms of other things.
- Sets are homogeneous: all elements have the same type.
  We identify a type with the set of all elements of that type.
- Each type has ways to construct and ways to use its elements.
  - To construct $(a, b) \in A \times B$, need $a \in A$ and $b \in B$.
  - To use $p \in A \times B$, have $\pi_A(p) \in A$ and $\pi_B(p) \in B$.
  - To construct $f \in A \to B$, need $f(x) \in B$ assuming $x \in A$.
  - To use $f \in A \to B$, given $a \in A$ get $f(a) \in B$.

Why type theory?

Most mathematicians are more familiar with set theory...

Why type theory?

Most mathematicians are more familiar with set theory...

...but only one full-scale proof assistant uses set theory (Mizar)

Why type theory?

Most mathematicians are more familiar with set theory...

...but only one full-scale proof assistant uses set theory (Mizar)
...and it's quite old: all modern proof assistants use type theory

Why type theory?

Most mathematicians are more familiar with set theory...

...but only one full-scale proof assistant uses set theory (Mizar)
...and it's quite old: all modern proof assistants use type theory
...and even Mizar has a typed layer on top of the set theory!

Why type theory?

Most mathematicians are more familiar with set theory...

...but only one full-scale proof assistant uses set theory (Mizar)
...and it's quite old: all modern proof assistants use type theory
...and even Mizar has a typed layer on top of the set theory!

Why is type theory so much more popular for proof assistants?

One reason is that it's not enough to just check whether a proof is correct: we also want useful feedback if it isn't.

Explorations in coding

Example

How can we code numbers as sets? Von Neumann's idea was:
$0 = \emptyset$, $1 = \{0\}$, $2 = \{0, 1\}$, $3 = \{0, 1, 2, \dots\}$, $\dots$
Each natural number is the set of all previous ones.

Explorations in coding

Example

How can we code numbers as sets? Von Neumann's idea was:
$0 = \emptyset$, $1 = \{0\}$, $2 = \{0, 1\}$, $3 = \{0, 1, 2, \dots\}$, $\dots$
Each natural number is the set of all previous ones.

Example

How can we code an ordered pair (a, b) as a set?

- Can't use $\{a, b\}$, since $\{a, b\} = \{b, a\}$.
- Can't use $\{a, \{b\}\}$: then $\big(\{x\}, \{y\}\big)$ would equal $\big(\{\{y\}\}, x\big)$.
- But $(a, b) = \{a, \{a, b\}\}$ works: we can prove*

$$\{a, \{a, b\}\} = \{c, \{c, d\}\} \iff a = c \text{ and } b = d$$

Coding collisions

$$0 = \emptyset,\ 1 = \{0\},\ 2 = \{0, 1\},\ 3 = \{0, 1, 2, \dots\},\ \dots$$

$$(a, b) = \{a, \{a, b\}\}$$

With these codings,

$$(0, 0) = \{0, \{0, 0\}\} = \{0, \{0\}\} = \{0, 1\} = 2.$$

Example

If I have a function $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ and I want to compute $f(2, 3)$, but I make a typo and write $f(2)$, a pure set-theory-based proof assistant won't complain, but will happily compute $f(0, 0)$ instead. This will eventually cause a problem, but tracking down the true cause of the problem can be a real pain.

This particular collision $(0, 0) = 2$ can be avoided by using different codings, but any coding will have potential collisions.

Solving the problem in type theory

- The natural numbers are the elements of the type $\mathbb{N}$. Formally, they are expressions of the form

$$0, S0, SS0, SSS0, SSSS0, \ldots$$

  generated by the element 0 and the "successor" operation S. These are primitive, not coded as anything else.
- Ordered pairs are elements of some product type $A \times B$. They are also primitive, not coded as anything else.
- If $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, writing $f(2)$ is a type error, caught immediately by the proof assistant as soon as I write it.

This is one reason why all modern proof assistants are based on some kind of type theory.

Overloading

Another problem is we use the same notation for different things.

Example

When we write $x + y$, we could mean

- addition of integers
- addition of real numbers
- addition of vectors
- addition of matrices
- addition in any abelian group

How is a poor computer supposed to know which we mean?

If everything is a set, it can't! Then x and y are always just sets, and there are infinitely many abelian groups that contain them both, so who knows which one we had in mind?

But in type theory, it can use the types of x and y to guess.

Type theory and programming

Type theory is also closely related to functional programming languages like Lisp, Haskell, and OCaml.

$f : A \times B \to B \times A$
$f(x, y) = (y, x)$

```
f ::  (a, b) -> (b, a)
f (x, y) = (y, x)
```

```
(defun f (xy)
  (cdr xy .  car xy))
```

```
let f :  'a * 'b -> 'b * 'a
   = fun (x, y) -> (y, x)
```

- A constructive definition in type theory runs like a program.
- Proofs using type theory can also reason about programs.

This is another reason type theory is especially well-suited to computer proof assistants, especially when we want to prove correctness of programs as well as mathematical theorems.

Outline

But what kind of type theory?

- In set theory, we start with only first-order logic, and assert axioms in that logic asserting that certain sets exist.

  As we've seen, the proof assistant Mizar uses set theory.

- In simple type theory, we have both
  - type theory, in which we construct sets, pairs, functions, ...
  - first-order logic, in which prove things about them.

  Proof assistants like HOL and Isabelle use simple type theory.

- In dependent type theory, we have only type theory, and we code first-order logic into it.

  Proof assistants like NuPRL, Rocq, Agda, Lean, and Narya use dependent type theory.

How do we code first-order logic into type theory?

Propositions as types

In first-order logic we have natural deduction rules like

- To prove $P \Rightarrow Q$, we assume P and use this to prove Q.
- To use $P \Rightarrow Q$, if we know P, we can deduce Q.

In type theory we have rules like

- To construct $f \in A \to B$, we assume $x \in A$ and construct $f(x) \in B$.
- To use $f \in A \to B$, if we have $a \in A$, we can get $f(a) \in B$.

These are basically the same!!

Idea

We can code logical formulas as types and proofs as elements.

If we represent $P \Rightarrow Q$ as $P \to Q$, the type theory rules for elements of $P \to Q$ will give us precisely the first-order logic rules for proofs involving $P \Rightarrow Q$.

The Curry–Howard correspondence

Logical formula	Meaning	Type
$P \Rightarrow Q$	if P then Q	$P \to Q$
$P \wedge Q$	P and Q	$P \times Q$
$P \vee Q$	P or Q	$P \sqcup Q$
$\forall x \in A, P(x)$	for all $x \in A$, $P(x)$	$\prod_{x \in A} P(x)$
$\exists x \in A, P(x)$	there exists $x \in A$ s.t. $P(x)$	$\coprod_{x \in A} P(x)$

- $P \sqcup Q$ is the disjoint union, which contains copies of P and Q that are "tagged" so they don't overlap.
- Similarly, $\coprod_{x \in A} P(x)$ is the disjoint union of all the $P(x)$.

Dependent types

To consider a type like $\prod_{x \in A} P(x)$, we need a function P whose values $P(x)$ are types.

We call this a dependent type over $x \in A$, or a type family indexed by $x \in A$.

Example

$\mathbb{Z}_{/n}$, the integers mod n, is a type family indexed by $n \in \mathbb{N}$.
One element of $\prod_{n \in \mathbb{N}} \mathbb{Z}_{/n}$ is the family of zeros $\left([0]_n\right)_{n \in \mathbb{N}}$.

We can represent a dependent type as a function $P : A \to \mathcal{U}$, where $\mathcal{U}$ is a universe: a type whose elements are other types.

Dependent types in programming

Proof assistants using dependent type theory are also programming languages, which can give more precise types to functions and eliminate more bugs.

Example

We can define a type `Array(n,A)` of arrays of type `A` of length `n`. Then we can write functions like

```
append : Array(m,A) * Array(n,A) -> Array(m+n,A)
head : Array(n+1,A) -> A
```

In particular, we can guarantee at compile time that there are no out-of-bounds errors: all indices of array accesses are less than the length of the array.

"All errors are type errors!"

Outline

What to mention?

Each step in a proof (or construction) involves

- the application of a particular rule
- to particular known or assumed inputs
- to deduce a certain output.

When writing in natural language, we often mention the input and output statements but not the rule.

> *"Since* $x^2 + 2xy = 3$ *and* $y = 4$*, we have* $x^2 + 8x = 3$*."*

Human readers are good at guessing the rule (here, substitution).

Computers, however, find it much easier to deduce the input and output statements from the rule and the input names.
e.g. if `H1:x^2+2*x*y=3` and `H2:y=4`, the proof "`subst H2 H1`" can be deduced to prove `x^2+8*x=3`.

This is also shorter to write. . . but harder to guess and to read!

Structured proofs

Some proof assistants, like Mizar and Isabelle/Isar and Lurch, allow us to write proofs like in natural language, emphasizing the statements rather than the rules.

```
theorem T2:
  ex x, y st x is irrational & y is irrational &
    x.^.y is rational
proof
  set w = √2;
  H1: w is irrational by INT_2:44,T1;
  w>0 by AXIOMS:22,SQUARE_1:84;
  then (w.^.w).^.w = w.^.(w•w) by POWER:38
    .= w.^.(w^2) by SQUARE_1:58
    .= w.^.2 by SQUARE_1:88
    .= w^2 by POWER:53
    .= 2 by SQUARE_1:88;
  then H2: (w.^.w).^.w is rational by RAT_1:8;
  per cases;
  suppose H3: w.^.w is rational;
    take w, w;
    thus thesis by H1,H3;
  suppose H4: w.^.w is irrational;
    take w.^.w, w;
    thus thesis by H1,H2,H4;
end;
```

Structured proofs

Some proof assistants, like Mizar and Isabelle/Isar and Lurch, allow us to write proofs like in natural language, emphasizing the statements rather than the rules.

```
theorem Drinkers'_Principle: "∃x. Q x ⟶ (∀y. Q y)"
proof cases
  assume "∀y. Q y"
  fix any have "Q any ⟶ (∀y. Q y)" ..
  thus ?thesis ..
next
  assume "¬(∀y. Q y)"
  then obtain y where "¬ Q y" ⟨proof⟩
  hence "Q y ⟶ (∀y. Q y)" ..
  thus ?thesis ..
qed
```

Structured proofs

Some proof assistants, like Mizar and Isabelle/Isar and Lurch, allow us to write proofs like in natural language, emphasizing the statements rather than the rules.

Theorem (composition of continuous is continuous):
Suppose $\langle X, \tau \rangle$ is a topological space, $f: X \to X$, $g: X \to X$, f is continuous, and g is continuous then $g \circ f$ is continuous ✓.
✓

Proof:
Suppose $\langle X, \tau \rangle$ is a topological space, $f: X \to X$, $g: X \to X$, f is continuous, and g is continuous. Then $g \circ f: X \to X$ ✓ by the definition of composition. To prove $g \circ f$ is continuous, we must now show that the inverse image of any open set is open.

So let $U \in \tau$ be an arbitrary open set. Then $g^{\text{inv}}(U) \in \tau$ ✓ by the continuity of g, and $f^{\text{inv}}(g^{\text{inv}}(U)) \in \tau$ ✓ by the continuity of f. Thus, it suffices to show that

$$f^{\text{inv}}(g^{\text{inv}}(U)) = (g \circ f)^{\text{inv}}(U).$$

To prove that, first let $x \in f^{\text{inv}}(g^{\text{inv}}(U))$. Then by the definition of inverse image we have $f(x) \in g^{\text{inv}}(U)$ ✓ and $g(f(x)) \in U$ ✓. But $g(f(x)) = (g \circ f)(x)$ ✓, so $(g \circ f)(x) \in U$ ✓ by substitution. Thus, $x \in (g \circ f)^{\text{inv}}(U)$ ✓ as desired.
✓

Conversely, let $y \in (g \circ f)^{\text{inv}}(U)$. Then $(g \circ f)(y) \in U$ ✓. But once again we have $(g \circ f)(y) = g(f(y))$ ✓ and so $g(f(y)) \in U$ ✓ by substitution. Then $f(y) \in g^{\text{inv}}(U)$ ✓ and $y \in f^{\text{inv}}(g^{\text{inv}}(U))$ ✓.
✓

Thus we have shown that $f^{\text{inv}}(g^{\text{inv}}(U)) = (g \circ f)^{\text{inv}}(U)$ ✓, and so $(g \circ f)^{\text{inv}}(U) \in \tau$ ✓ by substitution.
✓

Therefore $g \circ f$ is continuous ✓.
□
✓

Proof terms

Other proof assistants, like Agda and Rocq and Narya, expect proofs written in a programming language, mentioning only the rules and the inputs.

```
binomial-theorem-Semiring :
  {l : Level} (R : Semiring l) (n : ℕ) (x y : type-Semiring R) →
  mul-Semiring R x y = mul-Semiring R y x →
  power-Semiring R n (add-Semiring R x y) =
  binomial-sum-Semiring R n
    ( λ i →
      mul-Semiring R
      ( power-Semiring R (nat-Fin (succ-ℕ n) i) x)
      ( power-Semiring R (dist-ℕ (nat-Fin (succ-ℕ n) i) n) y))
binomial-theorem-Semiring R zero-ℕ x y H =
  inv
    ( ( sum-one-element-Semiring R
        ( λ i →
          mul-nat-scalar-Semiring R
            ( binomial-coefficient-Fin 0 i)
            ( mul-Semiring R
              ( power-Semiring R (nat-Fin 1 i) x)
              ( power-Semiring R (dist-ℕ (nat-Fin 1 i) 0) y)))) ·
      ( ( left-unit-law-mul-nat-scalar-Semiring R
          ( mul-Semiring R
            ( one-Semiring R)
            ( one-Semiring R))) ·
        ( left-unit-law-mul-Semiring R (one-Semiring R))))
binomial-theorem-Semiring R (succ-ℕ zero-ℕ) x y H =
  ( commutative-add-Semiring R x y) ·
  ( ( ap-binary
      ( add-Semiring R)
      ( ( inv (left-unit-law-mul-Semiring R y)) ·
        ( inv
          ( left-unit-law-mul-nat-scalar-Semiring R
            ( mul-Semiring R (one-Semiring R) y))))
      ( ( inv (right-unit-law-mul-Semiring R x)) ·
```

Proof terms

Other proof assistants, like Agda and Rocq and Narya, expect proofs written in a programming language, mentioning only the rules and the inputs.

```
(** The Eckmann-Hilton argument *)
Definition eckmann_hilton {A : Type} {x:A} (p q : 1 = 1 :> (x = x)) : p @ q = q @ p :=
  (whiskerR_p1 p @@ whiskerL_1p q)^
  @ (concat_p1 _ @@ concat_p1 _)
  @ (concat_1p _ @@ concat_1p _)
  @ (concat_whisker _ _ _ _ p q)
  @ (concat_1p _ @@ concat_1p _)^
  @ (concat_p1 _ @@ concat_p1 _)^
  @ (whiskerL_1p q @@ whiskerR_p1 p).

(** The action of functions on 2-dimensional paths *)

Definition ap02 {A B : Type} (f:A->B) {x y:A} {p q:x=y} (r:p=q) : ap f p = ap f q
  := ap (ap f) r.

Definition ap02_pp {A B} (f:A->B) {x y:A} {p p' p'':x=y} (r:p=p') (r':p'=p'')
  : ap02 f (r @ r') = ap02 f r @ ap02 f r'
  := ap_pp (ap f) r r'.
```

Proof terms

Other proof assistants, like Agda and Rocq and Narya, expect proofs written in a programming language, mentioning only the rules and the inputs.

```
def fΠ (A : Type) (B : A → Type) (fA : isFibrant A)
  (fB : (x : A) → isFibrant (B x))
  : isFibrant ((x : A) → B x)
  := [
| .trr.e ↦ f0 a1 ↦ fB.2 (fA.2 .liftl.1 a1) .trr.1 (f0 (fA.2 .trl.1 a1))
| .trl.e ↦ f1 a0 ↦ fB.2 (fA.2 .liftr.1 a0) .trl.1 (f1 (fA.2 .trr.1 a0))
| .liftr.e ↦ f0 a ⤇
    refl fB.2
        (sym
           (sym (refl fA.2) .id.1 a.2 (fA.2 .liftl.1 a.1) .liftl.1 (refl a.1)))
      .id.1
        (refl f0 (fA.2^(e1) .id.1 a.2 (fA.2 .liftl.1 a.1) .trl.1 (refl a.1)))
        (refl (fB.2 (fA.2 .liftl.1 a.1) .trr.1 (f0 (fA.2 .trl.1 a.1))))
      .trl.1 (fB.2 (fA.2 .liftl.1 a.1) .liftr.1 (f0 (fA.2 .trl.1 a.1)))
| .liftl.e ↦ f1 a ⤇
    refl fB.2
        (sym
           (sym (refl fA.2) .id.1 a.2 (fA.2 .liftr.1 a.0) .liftr.1 (refl a.0)))
      .id.1 (refl (fB.2 (fA.2 .liftr.1 a.0) .trl.1 (f1 (fA.2 .trr.1 a.0))))
        (refl f1 (fA.2^(e1) .id.1 a.2 (fA.2 .liftr.1 a.0) .trr.1 (refl a.0)))
      .trl.1 (fB.2 (fA.2 .liftr.1 a.0) .liftl.1 (f1 (fA.2 .trr.1 a.0)))
| .id.e ↦ f0 f1 ↦
    feqv ((a0 : A.0) (a1 : A.1) (a2 : A.2 a0 a1) → B.2 a2 (f0 a0) (f1 a1))
      (Id Π A.2 B.2 f0 f1) (id_Π_iso A.0 A.1 A.2 B.0 B.1 B.2 f0 f1)
      (fΠ A.0 (a0 ↦ (a1 : A.1) (a2 : A.2 a0 a1) → B.2 a2 (f0 a0) (f1 a1))
         fA.0
         (a0 ↦
          fΠ A.1 (a1 ↦ (a2 : A.2 a0 a1) → B.2 a2 (f0 a0) (f1 a1)) fA.1
            (a1 ↦
             fΠ (A.2 a0 a1) (a2 ↦ B.2 a2 (f0 a0) (f1 a1)) (fA.2 .id.1 a0 a1)
               (a2 ↦ fB.2 a2 .id.1 (f0 a0) (f1 a1)))))]
```

Proof assistance

Many proof assistants also give the user help while constructing or reading a proof.

- In Agda and Narya, the user can leave "holes" in a definition or proof, query the proof assistant about the context and goal for each hole, and then "fill" the hole with a suitable construction or subproof.

`https://agdapad.quasicoherent.io`

Proof assistance

Many proof assistants also give the user help while constructing or reading a proof.

- In Agda and Narya, the user can leave "holes" in a definition or proof, query the proof assistant about the context and goal for each hole, and then "fill" the hole with a suitable construction or subproof.

  `https://agdapad.quasicoherent.io`

- In Rocq and Lean, the user can write a "tactic script" consisting of commands that construct a proof bit by bit, and "step through" the proof one line at a time seeing the context and goal at each step.

  `https://coq.vercel.app/`

Applications for software development

E.g. talks of Sandrine Blazy on CompCert (similar to CakeML)

As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task. The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users

Applications for Proofs

The following talk can be interesting

Chritisan Szegedy 09/3/2025 Autoformalization and Verifiable Superintelligence

In particular, he stresses the dichotomy

Informal/Formal

Validation/Verification

First example: euclidean and reflexive relations are symmetric

The representation of equality may be the first published example of a proof in Dependent Type Theory

de Bruijn, *The mathematical language Automath*, 1969

The example I present is given in

AUTOMATH, A Language for Mathematics, 1973

https://automath.win.tue.nl/archive/pdf/aut030.pdf

Applications for Proofs

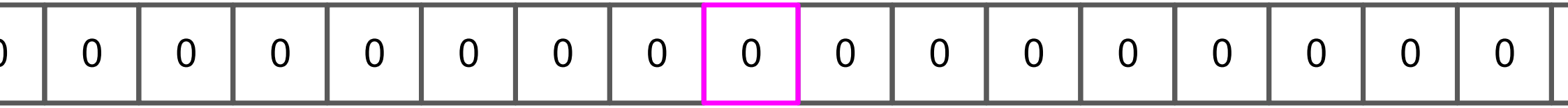
For the next example, I will use the slides of Tristan Stérin at Types 2025

We will represent simple cases of this example in Agda

Does it halt?

	0	1
A	1RB	1LC
B	1RC	1RB
C	1RD	0LE
D	1LA	1LD
E	---	0LA

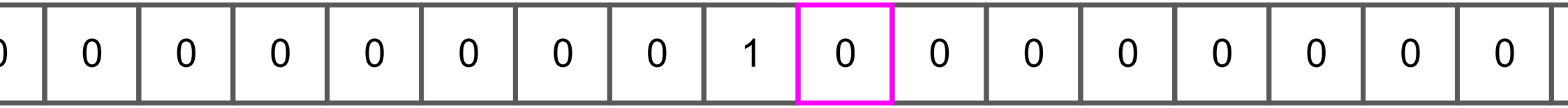
Step #0



Does it halt?

	0	1
A	1RB	1LC
B	1RC	1RB
C	1RD	0LE
D	1LA	1LD
E	---	0LA

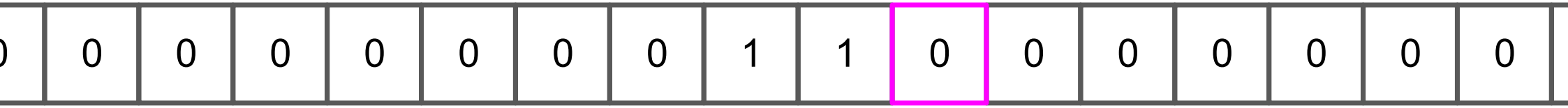
Step #1



Does it halt?

	0	1
A	1RB	1LC
B	1RC	1RB
C	1RD	0LE
D	1LA	1LD
E	---	0LA

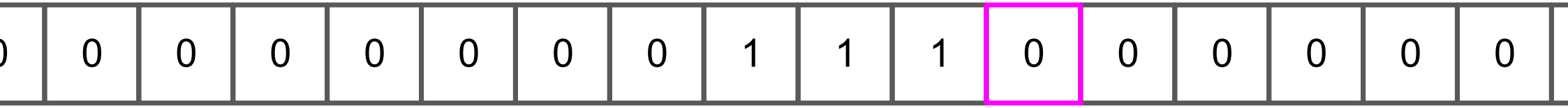
Step #2



Does it halt?

	0	1
A	1RB	1LC
B	1RC	1RB
C	1RD	0LE
D	1LA	1LD
E	---	0LA

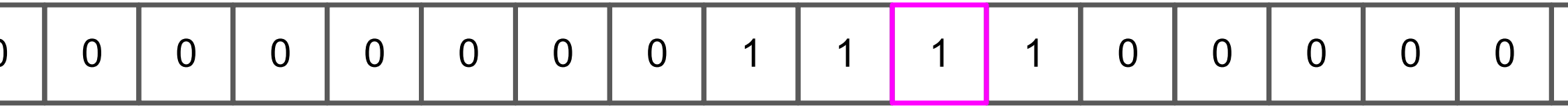
Step #3



Does it halt?

	0	1
A	1RB	1LC
B	1RC	1RB
C	1RD	0LE
D	1LA	1LD
E	---	0LA

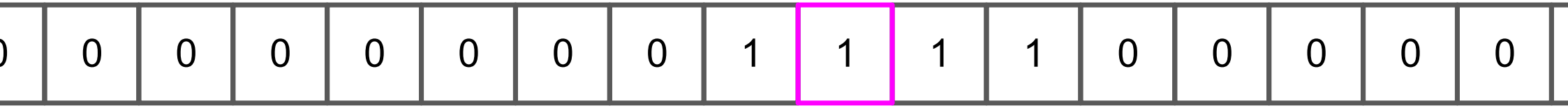
Step #4



Does it halt?

	0	1
A	1RB	1LC
B	1RC	1RB
C	1RD	0LE
D	1LA	1LD
E	---	0LA

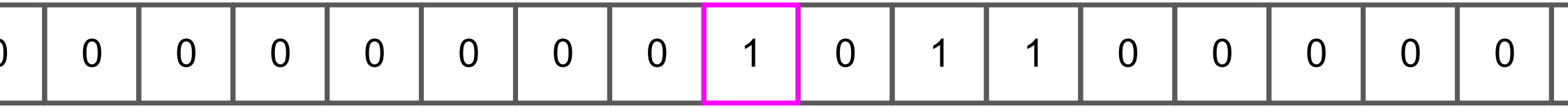
Step #5



Does it halt?

	0	1
A	1RB	1LC
B	1RC	1RB
C	1RD	0LE
D	1LA	1LD
E	---	0LA

Step #6

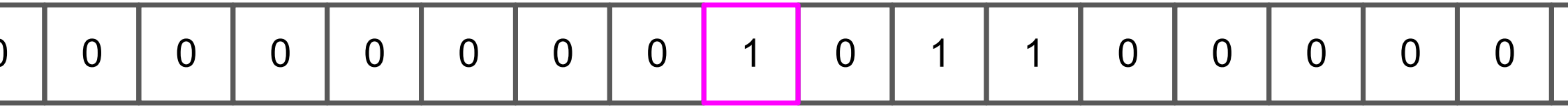


Does it halt?

	0	1
A	1RB	1LC
B	1RC	1RB
C	1RD	0LE
D	1LA	1LD
E	---	0LA

Will we ever read a 0 in state E ?

Step #6



Does it halt?

	0	1
A	1RB	1LC
B	1RC	1RB
C	1RD	0LE
D	1LA	1LD
E	---	0LA

Step #6

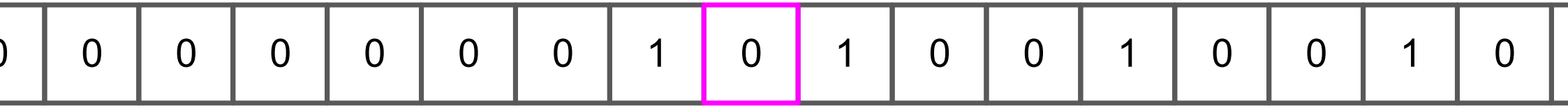
er read a 0 in state E ?

0	0	0	0	0		1	0	0	0	0	0	

Does it halt?
Yes!

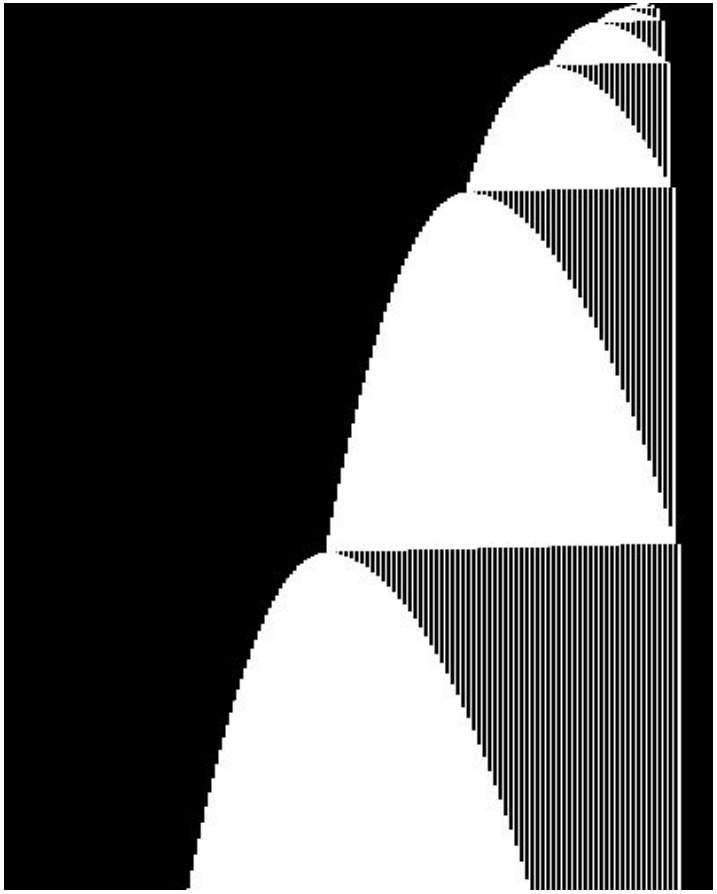
	0	1
A	1RB	1LC
B	1RC	1RB
C	1RD	0LE
D	1LA	1LD
E	---	0LA

Step #47,176,870



The machine has halted!!

Does it halt?
Yes!



	0	1
A	1RB	1LC
B	1RC	1RB
C	1RD	0LE
D	1LA	1LD
E	---	0LA

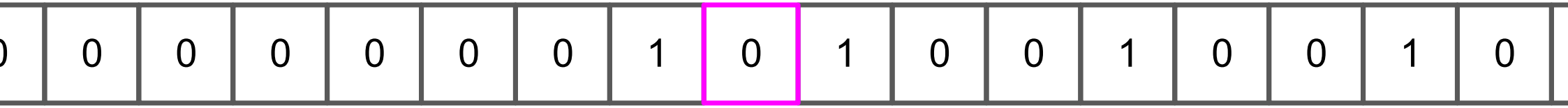
20,000-step *space-time diagram*

- Each row is a successive tape
- White = 1, Black = 0

Does it halt?

	0	1
A	1RB	1LC
B	1RC	1RB
C	1RD	0LE
D	1LA	1LD
E	---	0LA

Step #47,176,870



The machine has halted!!

Can another 5-state machine do better?

The Busy Beaver function

BB(n) = “Maximum number of steps done by a halting 2-symbol Turing machine with n states starting from all-0 memory tape”

T. Radó. On Non-computable Functions. *Bell System Technical Journal*, 41(3):877–884. 1962.

BB(n) = “Maximum algorithmic bang for your buck”

UNCOMPUTABLE

Tibor Radó, 1895 - 1965

The Busy Beaver function

BB(n) = “Maximum number of steps done by a halting 2-symbol Turing machine with n states starting from all-0 memory tape”

T. Radó. On Non-computable Functions. *Bell System Technical Journal*, 41(3):877–884. 1962.

Small busy beaver values:

- BB(1) = 1, BB(2) = 6 [Radó, 1962]
- BB(3) = 21 [Radó and Lin, 1963]
- BB(4) = 107 [Brady, 1983]

Allen Brady, 1934 - 2024

A 4-state Turing machine that runs more than 107 steps never halts (from all-0 tape)

Applications for Proofs

This was solved using computer assistants

For this problem, it is essential to use a computer assistant, like Agda, where we can do proofs about computation

It is also a good example of why computer assistants can be useful: it facilates *collaborative* proofs

"Massively collaborative research projects have a bright future"

One can trust the results proved by somebody else

In this example, students' or anonymous contributions were essential