Type theory in Lean

Riccardo Brasca

Université Paris Cité

October 14th 2023

Plan of the lectures

We will speak about:

Plan of the lectures

We will speak about:

- set theory and its drawbacks as the foundation of mathematics to be used by a proof assistant;

Plan of the lectures

We will speak about:

- set theory and its drawbacks as the foundation of mathematics to be used by a proof assistant;
- an informal introduction to type theory (terms, types and universes);

Plan of the lectures

We will speak about:

- set theory and its drawbacks as the foundation of mathematics to be used by a proof assistant;
- an informal introduction to type theory (terms, types and universes);
- basic constructions in type theory:
  - functions and dependent products;
  - dependent sums;
  - inductive types;

Plan of the lectures

We will speak about:

- set theory and its drawbacks as the foundation of mathematics to be used by a proof assistant;
- an informal introduction to type theory (terms, types and universes);
- basic constructions in type theory:
  - functions and dependent products;
  - dependent sums;
  - inductive types;
- the natural numbers;

Plan of the lectures

We will speak about:

- set theory and its drawbacks as the foundation of mathematics to be used by a proof assistant;
- an informal introduction to type theory (terms, types and universes);
- basic constructions in type theory:
    - functions and dependent products;
    - dependent sums;
    - inductive types;
- the natural numbers;
- various examples of inductive types;

- the notion of equality;

- the notion of equality;
- propositions as a type;

- the notion of equality;
- propositions as a type;
- the logic operators and the Curry-Howard correspondence;

- the notion of equality;
- propositions as a type;
- the logic operators and the Curry-Howard correspondence;
- proof irrelevance;

- the notion of equality;
- propositions as a type;
- the logic operators and the Curry-Howard correspondence;
- proof irrelevance;
- subsingleton elimination;

- the notion of equality;
- propositions as a type;
- the logic operators and the Curry-Howard correspondence;
- proof irrelevance;
- subsingleton elimination;
- more about universes;

- the notion of equality;
- propositions as a type;
- the logic operators and the Curry-Howard correspondence;
- proof irrelevance;
- subsingleton elimination;
- more about universes;
- Mathlib's axioms:
  - propositional extensionality;
  - quotient types;
  - choice;

- the notion of equality;
- propositions as a type;
- the logic operators and the Curry-Howard correspondence;
- proof irrelevance;
- subsingleton elimination;
- more about universes;
- Mathlib's axioms:
  - propositional extensionality;
  - quotient types;
  - choice;
- dependent type theory hell.

If we will have time we will briefly speak about homotopy type theory.

If we will have time we will briefly speak about homotopy type theory.

We will concentrate on the type theory used in Lean, with explicit examples.

If we will have time we will briefly speak about homotopy type theory.

We will concentrate on the type theory used in Lean, with explicit examples.

Disclaimer:

If we will have time we will briefly speak about homotopy type theory.

We will concentrate on the type theory used in Lean, with explicit examples.

Disclaimer: I am not a type theorist

If we will have time we will briefly speak about homotopy type theory.

We will concentrate on the type theory used in Lean, with explicit examples.

Disclaimer: I am not a type theorist, but I know Lean.

References

- Jeremy Avigad, Leonardo de Moura, and Soonho Kong, *Theorem Proving in Lean*, Carnegie Mellon University, 2014.
- Mario Carneiro, *The type theory of lean*, 2019, Master thesis.
- Egbert Rijke, *Introduction to homotopy type theory*, 2022, arXiv:2212.11082.
- The Univalent Foundations Program, *Homotopy type theory: Univalent foundations of mathematics*, `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

To get the repository of the course go to https://github.com/riccardobrasca/TypeTheory/ and follow the instructions there.

To get the repository of the course go to https://github.com/riccardobrasca/TypeTheory/ and follow the instructions there.

The files are TypeTheory/*/name.lean.

To get the repository of the course go to https://github.com/riccardobrasca/TypeTheory/ and follow the instructions there.

The files are TypeTheory/*/name.lean.

If you want to work with a file make a copy and work on it, not on the original version.

Introduction

One of the main goals of a proof assistant is to verify exactness of mathematical proofs

Introduction

One of the main goals of a proof assistant is to verify exactness of mathematical proofs, starting from the axioms.

Introduction

One of the main goals of a proof assistant is to verify exactness of mathematical proofs, starting from the axioms.

One has to choose an axiomatic system.

Introduction

One of the main goals of a proof assistant is to verify exactness of mathematical proofs, starting from the axioms.

One has to choose an axiomatic system.

Usually, mathematicians pick Zermelo-Fraenkel set theory

Introduction

One of the main goals of a proof assistant is to verify exactness of mathematical proofs, starting from the axioms.

One has to choose an axiomatic system.

Usually, mathematicians pick Zermelo-Fraenkel set theory plus the axiom of choice.

ZFC

ZFC is based on propositional logic.

ZFC

ZFC is based on propositional logic. In particular:

- there is a notion of *formulas*, *propositions*;
- we have the basics logic operators, like $\implies$, $\wedge$, $\forall$ etc.

ZFC

ZFC is based on propositional logic. In particular:

- there is a notion of *formulas*, *propositions*;
- we have the basics logic operators, like $\implies$, $\wedge$, $\forall$ etc.

Being a "set" and $\in$ are primitive notions, undefined.

ZFC

ZFC is based on propositional logic. In particular:

- there is a notion of *formulas*, *propositions*;
- we have the basics logic operators, like $\implies$, $\wedge$, $\forall$ etc.

Being a "set" and $\in$ are primitive notions, undefined.

In ZFC *everything is a set*.

There are several axioms that allow to build new set out of old ones

There are several axioms that allow to build new set out of old ones, like the union, the power set etc.

There are several axioms that allow to build new set out of old ones, like the union, the power set etc.

Functions are defined via their graph

There are several axioms that allow to build new set out of old ones, like the union, the power set etc.

Functions are defined via their graph: $f : X \to Y$ is a triple (X, Y, S) where $S \subseteq X \times Y$ is a subset with the property that for all $x \in X$ there is a unique $y \in Y$ such that $(x, y) \in S$.

There are several axioms that allow to build new set out of old ones, like the union, the power set etc.

Functions are defined via their graph: $f\colon X \to Y$ is a triple (X, Y, S) where $S \subseteq X \times Y$ is a subset with the property that for all $x \in X$ there is a unique $y \in Y$ such that $(x, y) \in S$. In particular such an f is itself a set.

There are several axioms that allow to build new set out of old ones, like the union, the power set etc.

Functions are defined via their graph: $f : X \to Y$ is a triple (X, Y, S) where $S \subseteq X \times Y$ is a subset with the property that for all $x \in X$ there is a unique $y \in Y$ such that $(x, y) \in S$. In particular such an f is itself a set.

A "strange" feature of set theory is that

$$\pi \in (\sin : \mathbb{R} \to \mathbb{R})$$

is a valid mathematical statement (hopefully false...).

The set of natural numbers can be defined in several ways.

The set of natural numbers can be defined in several ways. For example

$$
\begin{gathered}
0 := \{\} = \emptyset \\
1 := \{0\} = \{\emptyset\} \\
2 := \{0, 1\} = \{\emptyset, \{\emptyset\}\} \\
\dots \\
n + 1 := \{0, 1, \dots, n\} = n \cup \{n\}
\end{gathered}
$$

The set of natural numbers can be defined in several ways. For example

$$
\begin{aligned}
0 &:= \{\} = \emptyset \\
1 &:= \{0\} = \{\emptyset\} \\
2 &:= \{0, 1\} = \{\emptyset, \{\emptyset\}\} \\
&\dots \\
n + 1 &:= \{0, 1, \dots, n\} = n \cup \{n\}
\end{aligned}
$$

Proposition

3 *is a topology on* 2.

Proof assistants and ZFC

Can we use ZFC as an axiomatic system for a proof assistant?

Proof assistants and ZFC

Can we use ZFC as an axiomatic system for a proof assistant? Yes! (Metamath zero is such an example.)

Proof assistants and ZFC

Can we use ZFC as an axiomatic system for a proof assistant? Yes! (Metamath zero is such an example.)

Most of the modern proof assistants don't use ZFC. Why?

Proof assistants and ZFC

Can we use ZFC as an axiomatic system for a proof assistant? Yes! (Metamath zero is such an example.)

Most of the modern proof assistants don't use ZFC. Why?

Checking of correctness is done by the *kernel* of the proof assistant.

Proof assistants and ZFC

Can we use ZFC as an axiomatic system for a proof assistant? Yes! (Metamath zero is such an example.)

Most of the modern proof assistants don't use ZFC. Why?

Checking of correctness is done by the *kernel* of the proof assistant.

It is a simple software

Proof assistants and ZFC

Can we use ZFC as an axiomatic system for a proof assistant? Yes! (Metamath zero is such an example.)

Most of the modern proof assistants don't use ZFC. Why?

Checking of correctness is done by the *kernel* of the proof assistant.

It is a simple software, easy to check

Proof assistants and ZFC

Can we use ZFC as an axiomatic system for a proof assistant? Yes! (Metamath zero is such an example.)

Most of the modern proof assistants don't use ZFC. Why?

Checking of correctness is done by the *kernel* of the proof assistant.

It is a simple software, easy to check and there are several independent versions.

Proof assistants and ZFC

Can we use ZFC as an axiomatic system for a proof assistant? Yes! (Metamath zero is such an example.)

Most of the modern proof assistants don't use ZFC. Why?

Checking of correctness is done by the *kernel* of the proof assistant.

It is a simple software, easy to check and there are several independent versions.

A bug in the kernel is very unlikely.

To keep the kernel simple, its job must be very easy.

To keep the kernel simple, its job must be very easy.

In practice it only checks *extremely* precise mathematical arguments.

To keep the kernel simple, its job must be very easy.

In practice it only checks *extremely* precise mathematical arguments.

We never write mathematics so precisely

To keep the kernel simple, its job must be very easy.

In practice it only checks *extremely* precise mathematical arguments.

We never write mathematics so precisely, not even when using a proof assistant.

To keep the kernel simple, its job must be very easy.

In practice it only checks *extremely* precise mathematical arguments.

We never write mathematics so precisely, not even when using a proof assistant.

The *elaborator* translates something written by human beings to something totally precise

To keep the kernel simple, its job must be very easy.

In practice it only checks *extremely* precise mathematical arguments.

We never write mathematics so precisely, not even when using a proof assistant.

The *elaborator* translates something written by human beings to something totally precise, that is then checked by the kernel.

To keep the kernel simple, its job must be very easy.

In practice it only checks *extremely* precise mathematical arguments.

We never write mathematics so precisely, not even when using a proof assistant.

The *elaborator* translates something written by human beings to something totally precise, that is then checked by the kernel.

It is a complex software.

To keep the kernel simple, its job must be very easy.

In practice it only checks *extremely* precise mathematical arguments.

We never write mathematics so precisely, not even when using a proof assistant.

The *elaborator* translates something written by human beings to something totally precise, that is then checked by the kernel.

It is a complex software.

Bugs in the elaborator surely exist

To keep the kernel simple, its job must be very easy.

In practice it only checks *extremely* precise mathematical arguments.

We never write mathematics so precisely, not even when using a proof assistant.

The *elaborator* translates something written by human beings to something totally precise, that is then checked by the kernel.

It is a complex software.

Bugs in the elaborator surely exist, but this is not very serious.

What the elaborator has to do

Let's consider the following simple statement

What the elaborator has to do

Let's consider the following simple statement and let's make it more and more precise.

What the elaborator has to do

Let's consider the following simple statement and let's make it more and more precise.

Lemma

Let V be a vector space. For all $x \in V$ we have

$$(1 + 2)x = x + x + x.$$

What the elaborator has to do

Let's consider the following simple statement and let's make it more and more precise.

Lemma

Let K be a field and let V be a K-vector space. For all $x \in V$ we have

$$(1+2)x = x + x + x.$$

What the elaborator has to do

Let's consider the following simple statement and let's make it more and more precise.

Lemma

Let K be a field and let V be a K-vector space. For all $x \in V$ we have

$$(1 +_K 2)x = x + x + x.$$

What the elaborator has to do

Let's consider the following simple statement and let's make it more and more precise.

Lemma

Let K be a field and let V be a K-vector space. For all $x \in V$ we have

$$(1 +_K 2) \cdot x = x + x + x.$$

What the elaborator has to do

Let's consider the following simple statement and let's make it more and more precise.

Lemma

Let K be a field and let V be a K-vector space. For all $x \in V$ we have

$$(1 +_K 2) \cdot_V x = x + x + x.$$

What the elaborator has to do

Let's consider the following simple statement and let's make it more and more precise.

Lemma

Let K be a field and let V be a K-vector space. For all $x \in V$ we have

$$(1 +_K 2) \cdot_{V,K} x = x + x + x.$$

What the elaborator has to do

Let's consider the following simple statement and let's make it more and more precise.

Lemma

Let K be a field and let V be a K-vector space. For all $x \in V$ we have

$$(1 +_K 2) \cdot_{V,K} x = x +_V x +_V x.$$

What the elaborator has to do

Let's consider the following simple statement and let's make it more and more precise.

Lemma

Let K be a field and let V be a K-vector space. For all $x \in V$ we have

$$(1 +_K 2) \cdot_{V,K} x = (x +_V x) +_V x.$$

What the elaborator has to do

Let's consider the following simple statement and let's make it more and more precise.

Lemma

Let K be a field and let V be a K-vector space. For all $x \in V$ we have

$$(1_K +_K 2) \cdot_{V,K} x = (x +_V x) +_V x.$$

What the elaborator has to do

Let's consider the following simple statement and let's make it more and more precise.

Lemma

Let K be a field and let V be a K-vector space. For all $x \in V$ we have

$$(1_K +_K \varphi_{\mathbb{N}/K}(2)) \cdot_{V,K} x = (x +_V x) +_V x.$$

What the elaborator has to do

Let's consider the following simple statement and let's make it more and more precise.

Lemma

Let K be a field and let V be a K-vector space. For all $x \in |V|$ we have

$$(1_K +_K \varphi_{\mathbb{N}/K}(2)) \cdot_{V,K} x = (x +_V x) +_V x.$$

How can the elaborator guess that it needs to use $+_V$?

How can the elaborator guess that it needs to use $+_V$?

"Because $x \in V$".

How can the elaborator guess that it needs to use $+_V$?

"Because $x \in V$". But maybe $V \subseteq W$ for another vector space W, so $x \in W$ too.

How can the elaborator guess that it needs to use $+_V$?

"Because $x \in V$". But maybe $V \subseteq W$ for another vector space W, so $x \in W$ too.

The elaborator has a complex job

How can the elaborator guess that it needs to use $+_V$?

"Because $x \in V$". But maybe $V \subseteq W$ for another vector space W, so $x \in W$ too.

The elaborator has a complex job, and the fact that everything is a set does not help.

Type theory

Type theory is an alternative to set theory as a foundation of mathematics.

Type theory

Type theory is an alternative to set theory as a foundation of mathematics.

It has been introduced by Bertrand Russel around 1910.

Type theory

Type theory is an alternative to set theory as a foundation of mathematics.

It has been introduced by Bertrand Russel around 1910.

Type theory it is its own deductive system (it does not depend on first order logic).

Type theory

Type theory is an alternative to set theory as a foundation of mathematics.

It has been introduced by Bertrand Russel around 1910.

Type theory it is its own deductive system (it does not depend on first order logic).

Sets and $\in$ play no special role, they are mathematical notions with a “normal” definition.

Terms, types and universes

In type theory there are three layer of mathematical objects:

Terms, types and universes

In type theory there are three layer of mathematical objects:

- terms;
- types;
- universes.

Terms, types and universes

In type theory there are three layer of mathematical objects:

- terms;
- types;
- universes.

(Almost) everything is a term.

Terms, types and universes

In type theory there are three layer of mathematical objects:

- terms;
- types;
- universes.

(Almost) everything is a term.

The natural number 0 is a term.

Terms, types and universes

In type theory there are three layer of mathematical objects:

- terms;
- types;
- universes.

(Almost) everything is a term.

The natural number 0 is a term. So is the the function $\sin\colon \mathbb{R} \to \mathbb{R}$ and the “set” $\mathbb{Z}$.

$\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{R}$ are types (but also terms!).

$\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{R}$ are types (but also terms!).

Slogan

Every term x has its own type T, written

$$(x : T).$$

$\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{R}$ are types (but also terms!).

Slogan

Every term x has its own type T, written

$$(x : T).$$

In a sense, the type of x is its own nature.

$\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{R}$ are types (but also terms!).

Slogan

Every term x has its own type T, written

$$(x : T).$$

In a sense, the type of x is its own nature. The fact that the type of x is T is not a (true/false) mathematical statement.

$\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{R}$ are types (but also terms!).

Slogan

Every term x has its own type T, written

$$(x : T).$$

In a sense, the type of x is its own nature. The fact that the type of x is T is not a (true/false) mathematical statement. We cannot prove or disprove it

$\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{R}$ are types (but also terms!).

Slogan

Every term x has its own type T, written

$$(x : T).$$

In a sense, the type of x is its own nature. The fact that the type of x is T is not a (true/false) mathematical statement. We cannot prove or disprove it, we can only *check* the type of a term.

This can looks unfamiliar, but it is very natural in mathematics.

This can looks unfamiliar, but it is very natural in mathematics.

Conjecture (Riemann)

Let $z \in \mathbb{C}$ be a complex number. If $\zeta(z) = 0$ and $\Im(z) > 0$ then $z \in L$, where L is the critical line.

This can looks unfamiliar, but it is very natural in mathematics.

Conjecture (Riemann)

Let $z \in \mathbb{C}$ be a complex number. If $\zeta(z) = 0$ and $\Im(z) > 0$ then $z \in L$, where L is the critical line.

The two symbols $\in$, in the assumption $z \in \mathbb{C}$ and in the conclusion that $z \in L$ have different meaning:

- the first one is about the nature of z, the statement is about a complex number z;
- the second one is a mathematical property.

This can looks unfamiliar, but it is very natural in mathematics.

Conjecture (Riemann)

Let $z \in \mathbb{C}$ be a complex number. If $\zeta(z) = 0$ and $\Im(z) > 0$ then $z \in L$, where L is the critical line.

The two symbols $\in$, in the assumption $z \in \mathbb{C}$ and in the conclusion that $z \in L$ have different meaning:

- the first one is about the nature of z, the statement is about a complex number z;
- the second one is a mathematical property.

In type theory, Riemann hypothesis is a statement about a term z of type $\mathbb{C}$.

$\mathbb{N}$ is a type, but also a term.

$\mathbb{N}$ is a type, but also a term. Its type is Type.

$\mathbb{N}$ is a type, but also a term. Its type is Type.

$$(\mathbb{N} : \mathrm{Type}) \quad (\mathbb{R} : \mathrm{Type}).$$

$\mathbb{N}$ is a type, but also a term. Its type is Type.

$$(\mathbb{N} : \mathrm{Type}) \quad (\mathbb{R} : \mathrm{Type}).$$

Type is also a term!

$\mathbb{N}$ is a type, but also a term. Its type is Type.

$$(\mathbb{N} : \mathrm{Type}) \quad (\mathbb{R} : \mathrm{Type}).$$

Type is also a term! It has type Type 1.

$$(\mathrm{Type} : \mathrm{Type}\ 1).$$

Type and $\mathrm{Type}\ 1$ are *universes*.

Type and $\mathrm{Type}\ 1$ are *universes*.
Lean has a countable non-cumulative hierarchy of universes.

Type and $\mathrm{Type}\ 1$ are *universes*.
Lean has a countable non-cumulative hierarchy of universes.

$$\mathrm{Type}\ 0$$

Type and $\mathrm{Type}\ 1$ are *universes*.
Lean has a countable non-cumulative hierarchy of universes.

$$\mathrm{Type}\ 0 = \mathrm{Type}$$

Type and $\mathrm{Type}\ 1$ are *universes*.
Lean has a countable non-cumulative hierarchy of universes.

$$\mathrm{Type}\ 0 = \mathrm{Type}$$

$$\mathrm{Type}\ 1$$

Type and $\mathrm{Type}\ 1$ are *universes*.
Lean has a countable non-cumulative hierarchy of universes.

$$\mathrm{Type}\ 0 = \mathrm{Type}$$

$$\mathrm{Type}\ 1$$

$$\mathrm{Type}\ 2$$

Type and $\mathrm{Type}\ 1$ are *universes*.
Lean has a countable non-cumulative hierarchy of universes.

$$\mathrm{Type}\ 0 = \mathrm{Type}$$

$$\mathrm{Type}\ 1$$

$$\mathrm{Type}\ 2$$

And so on.

Type and $\mathrm{Type}\ 1$ are *universes*.
Lean has a countable non-cumulative hierarchy of universes.

$$\mathrm{Type}\ 0 = \mathrm{Type}$$

$$\mathrm{Type}\ 1$$

$$\mathrm{Type}\ 2$$

And so on. In general

$$\mathrm{Type}\ n$$

Type and $\mathrm{Type}\ 1$ are *universes*.
Lean has a countable non-cumulative hierarchy of universes.

$$\mathrm{Type}\ 0 = \mathrm{Type}$$

$$\mathrm{Type}\ 1$$

$$\mathrm{Type}\ 2$$

And so on. In general

$$\mathrm{Type}\ n : \mathrm{Type}\ n + 1$$

Type and $\mathrm{Type}\ 1$ are *universes*.
Lean has a countable non-cumulative hierarchy of universes.

$$\mathrm{Type}\ 0 = \mathrm{Type}$$

$$\mathrm{Type}\ 1$$

$$\mathrm{Type}\ 2$$

And so on. In general

$$\mathrm{Type}\ n : \mathrm{Type}\ n + 1$$

At the very bottom there is a special universe: $\mathrm{Prop} : \mathrm{Type}$.

Definitional equality

In Lean there are several notion of equality:

Definitional equality

In Lean there are several notion of equality:

- Syntactic equality: two literally equal expressions ("written using the same keys on the keyboard").

Definitional equality

In Lean there are several notion of equality:

- Syntactic equality: two literally equal expressions ("written using the same keys on the keyboard").
- Mathematical equality (more on this later).

Definitional equality

In Lean there are several notion of equality:

- Syntactic equality: two literally equal expressions ("written using the same keys on the keyboard").
- Mathematical equality (more on this later).
- Definitional equality, denoted today $\equiv$.

Definitional equality

In Lean there are several notion of equality:

- Syntactic equality: two literally equal expressions ("written using the same keys on the keyboard").
- Mathematical equality (more on this later).
- Definitional equality, denoted today $\equiv$.

Two terms are definitionally equal if they are equal "by definition".

Definitional equality

In Lean there are several notion of equality:

- Syntactic equality: two literally equal expressions ("written using the same keys on the keyboard").
- Mathematical equality (more on this later).
- Definitional equality, denoted today $\equiv$.

Two terms are definitionally equal if they are equal "by definition".

$$x \equiv y$$

is *not* a mathematical statement

Definitional equality

In Lean there are several notion of equality:

- Syntactic equality: two literally equal expressions ("written using the same keys on the keyboard").
- Mathematical equality (more on this later).
- Definitional equality, denoted today $\equiv$.

Two terms are definitionally equal if they are equal "by definition".

$$x \equiv y$$

is *not* a mathematical statement, but it can be checked in practice.

There is a precise set of rules to check whether $x \equiv y$.

There is a precise set of rules to check whether $x \equiv y$. For example:

- syntactic equality implies definitional equality;

There is a precise set of rules to check whether $x \equiv y$. For example:

- syntactic equality implies definitional equality;
- the functions $x \mapsto \sin(x)$ and $y \mapsto \sin(y)$ are definitionally equal;

There is a precise set of rules to check whether $x \equiv y$. For example:

- syntactic equality implies definitional equality;
- the functions $x \mapsto \sin(x)$ and $y \mapsto \sin(y)$ are definitionally equal;
- if f is the function $x \mapsto \sin(x)$, then $f(\pi) \equiv \sin(\pi)$;

There is a precise set of rules to check whether $x \equiv y$. For example:

- syntactic equality implies definitional equality;
- the functions $x \mapsto \sin(x)$ and $y \mapsto \sin(y)$ are definitionally equal;
- if f is the function $x \mapsto \sin(x)$, then $f(\pi) \equiv \sin(\pi)$;
- if $(x : T)$ and $(y : S)$ are definitionally equal, then $T \equiv S$;

There is a precise set of rules to check whether $x \equiv y$. For example:

- syntactic equality implies definitional equality;
- the functions $x \mapsto \sin(x)$ and $y \mapsto \sin(y)$ are definitionally equal;
- if f is the function $x \mapsto \sin(x)$, then $f(\pi) \equiv \sin(\pi)$;
- if $(x : T)$ and $(y : S)$ are definitionally equal, then $T \equiv S$;
- ...

There is a precise set of rules to check whether $x \equiv y$. For example:

- syntactic equality implies definitional equality;
- the functions $x \mapsto \sin(x)$ and $y \mapsto \sin(y)$ are definitionally equal;
- if f is the function $x \mapsto \sin(x)$, then $f(\pi) \equiv \sin(\pi)$;
- if $(x : T)$ and $(y : S)$ are definitionally equal, then $T \equiv S$;
- ...

In principle definitional equality is algorithmically decidable.

Slogan

Two terms x and y are definitionally equal if the following proof works.

```
example : x = y := rfl
```

Slogan

Two terms x and y are definitionally equal if the following proof works.

```
example : x = y := rfl
```

This is *not* an equivalence relation

Slogan

Two terms x and y are definitionally equal if the following proof works.

```
example : x = y := rfl
```

This is *not* an equivalence relation, but the "true" definitional equality is.