

Spartan Martin-Löf Type Theory

Andrej Bauer
University of Ljubljana

UniMath School
April 2019, Birmingham, UK

1

Welcome everyone. I am honored to have the opportunity to speak here. I was asked to do an introduction to type theory in one hour for people who know nothing about it.

Benedikt said I can give a lecture which is similar to last one. Well, I can't repeat the same lame joke, so here's a parenthetical one. As last year, Benedikt selected the title for me. But which way did he want to parenthesize?

(Spartan Martin-Löf) Type Theory

Andrej Bauer
University of Ljubljana

UniMath School
April 2019, Birmingham, UK

2

3

Spartan (Martin-Löf Type Theory)

Andrej Bauer
University of Ljubljana

UniMath School
April 2019, Birmingham, UK

4

spartan | 'spɑ:t(ə)n |
adjective
showing or characterized by austerity or a lack of comfort or
luxury: *the accommodation was fairly spartan.*

We are going to consider the basics of type theory. In Birmingham it goes under the name “spartan”.

5

So, what is type theory?

What is type theory?

6

Some say it is a foundation of mathematics.

If this is the case, then type theory has to stand on its own, without assuming logic or set theory.

But in explaining it I *will* rely on our pre-existing mathematical knowledge, because a foundation is not magic that gets you something out of nothing.

A foundation *organizes* mathematics.

It is a *tool* for doing mathematics (and maybe a *language*).

It provides a *method* for doing mathematics.

It *serves* the mathematician first, and the logician second!

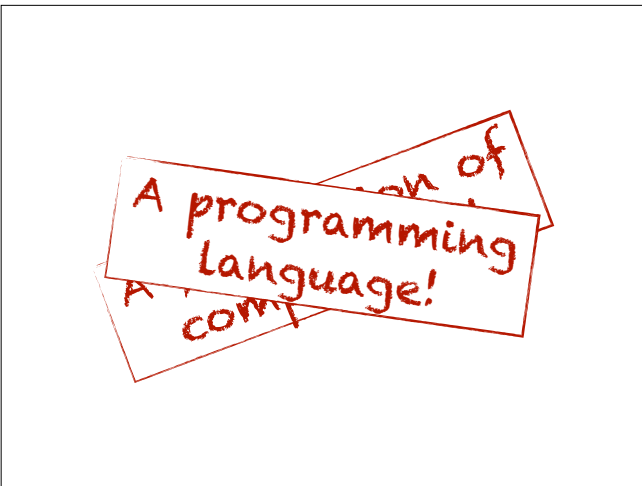
**A foundation
of mathematics**

7



There are several other views of type theory.
We do not have time to go into them, but they are all interesting.

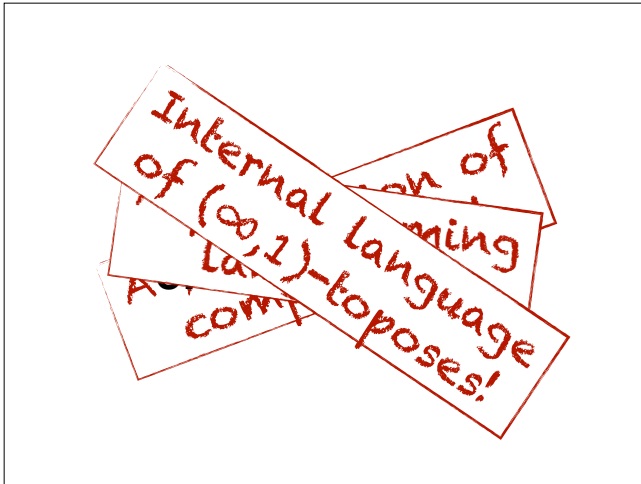
8



Ask your teachers here, they can tell you all about it.

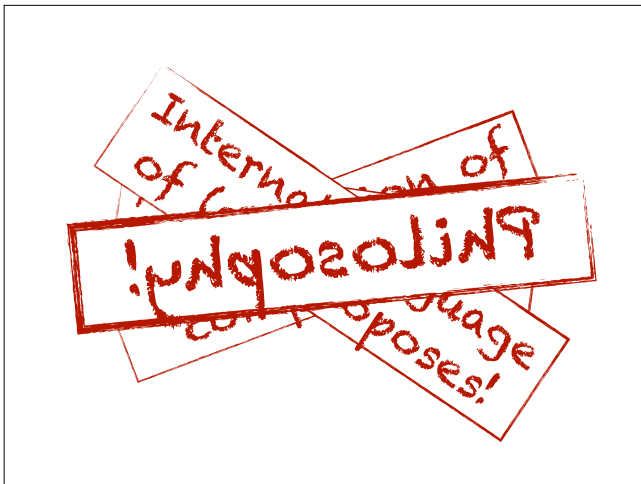
9

Careful though, they may drag you into strange territory.



10

Or they may say incomprehensible things. It's fun!



But let me tell you a secret: the informal style of a typical mathematical text is as close to type theory as it is to Zermelo-Fraenkel set theory, if not closer. This is not to say that anything is wrong with set theory, or that it hasn't had an important role. It's just an observation, supported by evidence: the most successful computer formalizations of mathematics all rely on type theory.

A theory of constructions

11

If I had to give you a single short phrase describing type theory, I would say that it is a **theory of constructions**.

It is not about what is true, but about how to build mathematical objects generally.

A theory of constructions

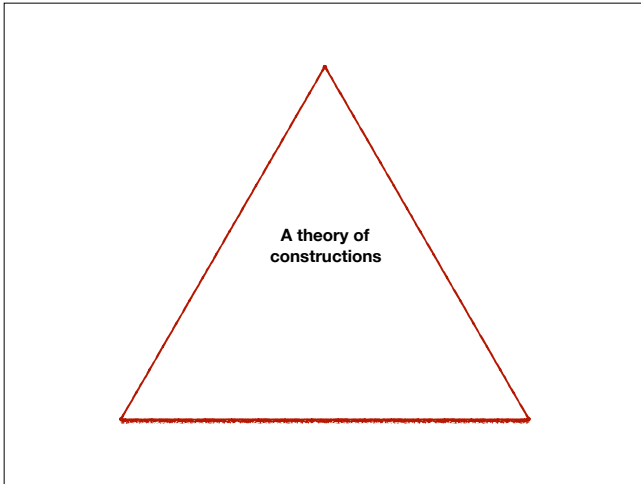
12

Did you know that Euclid's *Elements* contain both geometric *constructions* and *statements* of truth? Proposition 1 says “to erect an equilateral triangle on a line segment”.



13

Mathematics started with a construction!



14

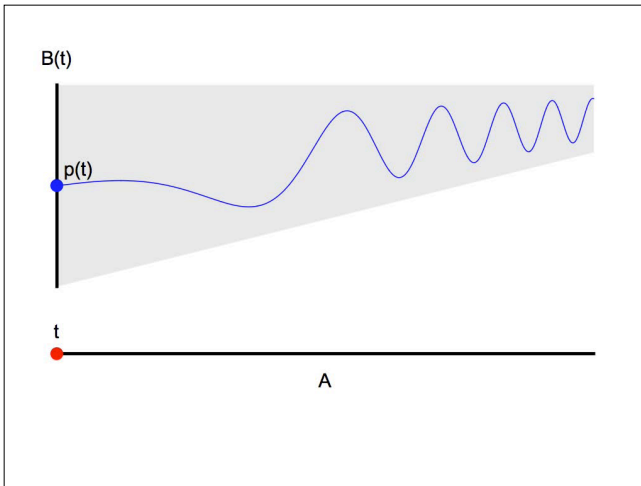
In fact, Euclid's construction is **dependent**: *given* a segment, construct an equilateral triangle which *depends* on it.

To emphasize this aspect of type theory, let us say that it is a theory of *dependent* constructions.

Dependency is a very general idea, so let us have a closer look at it.

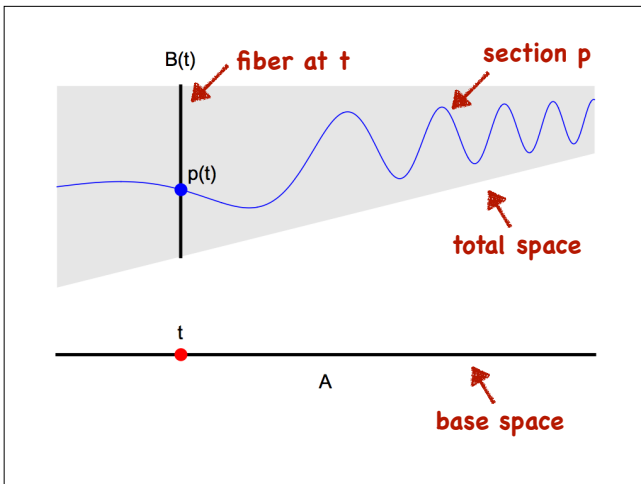
**A theory of
dependent
constructions**

15



We have here a pretty video in which **two** kinds of dependency are presented. First, the point $p(t)$ depends on the point t . As t moves in space A , the point $p(t)$ moves around. Second, the space $B(t)$ depends on t as well. This is very important. As t moves, the space $B(t)$ changes.

16



In geometry and topology each piece of the picture has a name. If you are familiar with it, that is good. If not, do not worry, we will not really use advanced geometry. We have a **base space**. The dependent space is called a **fiber**. We say that a **total space** is **over** the base space. The moving point gives us a map p from the base space to the total space, called a **section**.

Let us for a moment forget this picture and explain how we set up a theory of constructions.

17

Type	A type
Element	$p : A$
Equal types	$A \equiv B$
Equal elements	$p \equiv_A q$

Because I said there was no logic and no set theory, I have to give you something else to work with. Type theory has **judgments**. These are **not** logical statements, they record the fact that something has been constructed, or that two constructions are equal. For instance, we *cannot* make the judgment “A is not a type”.

If type theory is about making houses and cars from Lego blocks, the judgments correspond to the kid running up to their parents and showing them what they’ve built. Nobody ever runs to their parents to show them that they didn’t build anything.

We may and should discuss whether equality is part of “logic”. We will be discussing equality for the rest of the week.

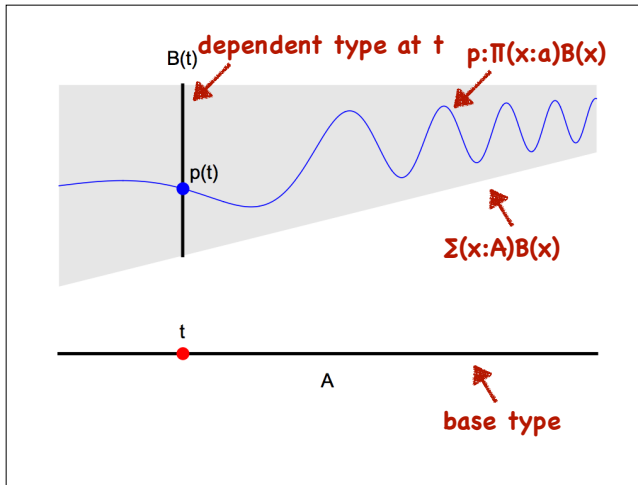
18

Space	A type
Point	$p : A$
Equal spaces	$A \equiv B$
Equal points	$p \equiv q : A$

As I already mentioned, there are many views of type theory, and so there are many ways of imagining what types and elements are. To connect this back to our geometric picture, here is the geometric explanation. Think of types as “spaces”, but take this word in the most general sense you can. It’s like a topological space. It’s like a homotopy type. It’s like a datatype. It’s like a computable space. Some of these explanations may mean nothing to you, but all have been suggested by people.

It is customary then to speak of *points* of a *space*. I emphasize again that a point never exists without being in a space, and in can never be in two spaces at the same time.

19



Going back to our picture, the type-theoretic terminology is as follows. Instead of fibers we have dependent types.

The total space becomes a construction known as “dependent sum”.

Sections are elements of “dependent products”.

20

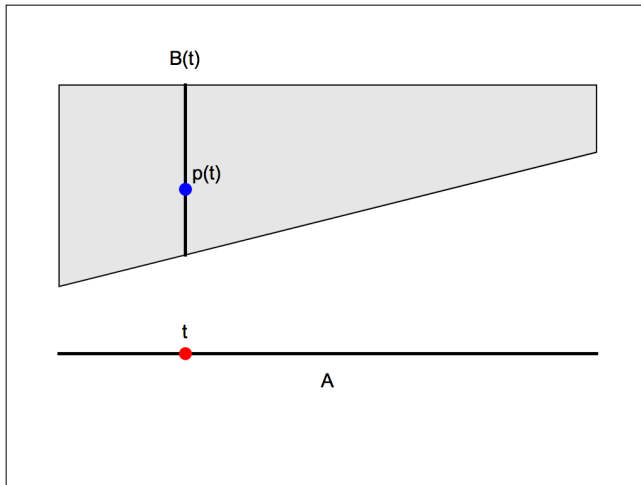
Sum $\Sigma(x:A) B(x)$

- **formation:**
if type $B(x)$ depends on $x : A$, then $\Sigma(x : A) B(x)$ is a type.
- **introduction:**
if $t : A$ and $u : B(t)$ then $(t, u) : \Sigma(x : A) B(x)$.
- **elimination:**
 - if $p : \Sigma(x : A) B(x)$ then $\pi_1(p) : A$.
 - if $p : \Sigma(x : A) B(x)$ then $\pi_2(p) : B(\pi_1(p))$.
- **equations:**
 - $\pi_1(t, u) \equiv t$
 - $\pi_2(t, u) \equiv u$
 - $(\pi_1(p), \pi_2(p)) \equiv p$

Let us look at dependent sums. Every type theoretic construction follows a certain pattern:

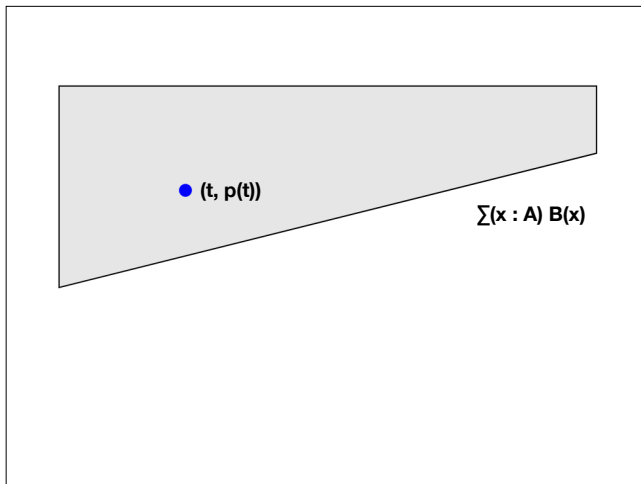
- first we state explain how to form the type, the *formation rule*
- then we give rules for making elements of the type, these are *introduction rules*
- elimination rules explain how we *use* or “*eliminate*” elements of the type
- equations explain what happens when we create an element and take it apart, or vice versa

21



We said that every point is always precisely in one space.
 Consider a point $p(t)$ in the fiber over t . It is a member of $B(t)$. It is *not* a member of the dependent sum!

22



The corresponding point in the dependent sum is the pair $(t, p(t))$.

Note that the dependent sum is a space on its own. It is not a dependent type over A .

Binary product

$$\mathbf{A} \times \mathbf{B} \equiv \sum (_ : \mathbf{A}) \mathbf{B}$$

Anonymous variable
(B does not depend on A)

23

A special case of dependent sum is the usual cartesian product. We get it when $\mathbf{B}(\mathbf{x})$ does not depend on \mathbf{A} , i.e., we have a **constant** dependent type.

Product $\prod(\mathbf{x}:\mathbf{A}) \mathbf{B}(\mathbf{x})$

- given $\mathbf{B}(\mathbf{x})$ over \mathbf{A} , $\prod(\mathbf{x} : \mathbf{A}) \mathbf{B}(\mathbf{x})$ is a type
- if $\mathbf{p}(\mathbf{x}) : \mathbf{B}(\mathbf{x})$ then $\lambda(\mathbf{x}:\mathbf{A}) \mathbf{p}(\mathbf{x}) : \prod(\mathbf{x} : \mathbf{A}) \mathbf{B}(\mathbf{x})$.
- If $\mathbf{f} : \prod(\mathbf{x} : \mathbf{A}) \mathbf{B}(\mathbf{x})$ and $\mathbf{t} : \mathbf{A}$ then $\mathbf{f}(\mathbf{t}) : \mathbf{B}(\mathbf{t})$.
- $(\lambda(\mathbf{x}:\mathbf{A}) \mathbf{p}(\mathbf{x}))(\mathbf{t}) \equiv \mathbf{p}(\mathbf{t})$
- $\lambda(\mathbf{x}:\mathbf{A})(\mathbf{f}(\mathbf{x})) \equiv \mathbf{f}$

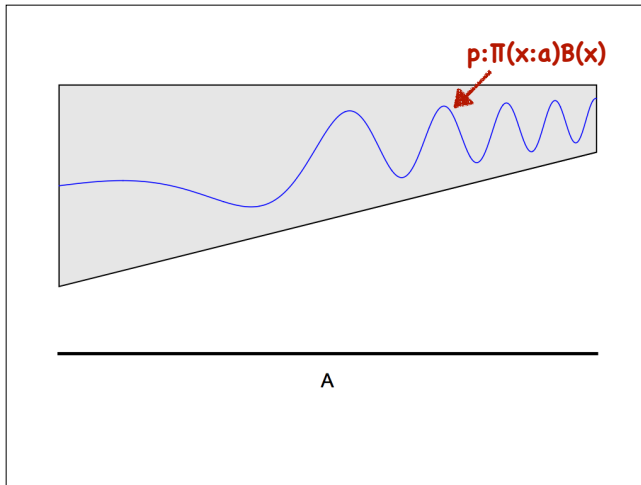
24

Dependent products are spaces of sections. It is difficult to draw a picture of a product, but it is easy to understand what its elements are: they are maps.

How is a map constructed? If for every fiber $\mathbf{B}(\mathbf{x})$ we have a point $\mathbf{p}(\mathbf{x})$ in it, then we can form a map. The λ thing should be read as: “the map which takes \mathbf{x} to $\mathbf{p}(\mathbf{x})$ ”.

To use a map, we apply it to an argument.

25



It is difficult to draw a picture of the product, but it is easy to see one of its elements: a section over A.

26

Function space

$$A \rightarrow B \equiv \prod (_ : A) B$$

If we form a dependent product over the constant dependent type we get the (non-dependent) function space. The elements are ordinary maps from A to B.

So far we have not given any ways of constructing a type from scratch. We need some *primitive* types from which all other constructions start.

Universe

- There is a type **Type**.
- The elements of **Type** are types.
- A dependent type is a map $\mathbf{B} : \mathbf{A} \rightarrow \mathbf{Type}$.
- Beware of paradoxes:
there can be no type of all types!
- $\mathbf{Type}_0 : \mathbf{Type}_1 : \mathbf{Type}_2 : \dots$

27

If you think about it, types themselves get constructed too. But anything that is constructed should have a type, so there should be a type of types. Such a type is called a *universe*.

Its elements are types. A dependent type is a map from the base space to the universe.

This is all very neat, but we have to beware of paradoxes. An early version of type theory stated that the universe was its own element, and that turned out to lead to the same sort of trouble as the set of all sets. If you need more than one universe you can make a hierarchy of them.

Basic types

- **Unit** – element is **tt**
- **Empty** – no elements
- **Bool** – elements **true** and **false**
- **N** – natural numbers

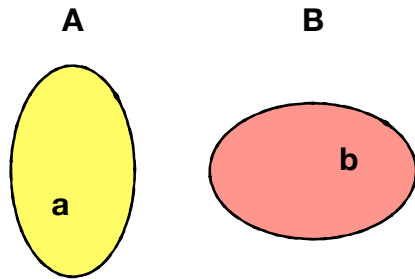
28

We do not have the time to carefully give the rules for other primitive types, so let us just list them.

- the unit type has just one element, called the “unit” (to confuse people), in Coq it is **tt**
- the empty type has no elements. It is instructive to give the elimination principle for **Empty**.
- **Bool** is the type with two elements, called “true” and “false”. Beware: this is not logic! It is just a type. We might have called its elements “left” and “right”.
- The natural numbers are also a basic type. We will look at it shortly.

29

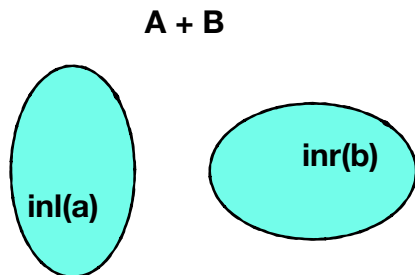
Simple sum $A + B$



Let us mention one other construction. Given types A and B , we can form their *simple sum*, also called coproduct.

30

Simple sum $A + B$



Once again, an element a of A cannot be an element of $A + B$ at the same time. We *tag* it to indicate that it is an element of $A + B$, where “inl” stands for “left injection” and “inr” for “right injection”. It is the tagging that distinguishes the simple sum from a union: it’s possible to make the sum of two equal summands $A + A$ without overlapping them.

In set theory this construction is known as disjoint union, where we usually tag the elements by making pairs of the form $(0, a)$ and $(1, b)$. Using primitive tags inl and inr is more principled.

Natural numbers \mathbf{N}

- $0 : \mathbf{N}$
- If $n : \mathbf{N}$ then $\mathbf{S}(n) : \mathbf{N}$.
- If $P : \mathbf{N} \rightarrow \mathbf{Type}$ and $e : P(0)$ and $f : \prod(x:\mathbf{N}) P(x) \rightarrow P(\mathbf{S}(x))$ then $\mathbf{ind_nat} P e f : \prod(x:\mathbf{N}) P(x)$.
- $\mathbf{ind_nat} P e f 0 \equiv e$
- $\mathbf{ind_nat} P e f (\mathbf{S} n) \equiv f n (\mathbf{ind_nat} P e f n)$

31

The natural numbers are shown in this slide. I am not going to drag you through the rules. Instead I am going to give an exercise: relate the first three rules to Peano axioms, and explain what the equations say in terms of programming.

Path space

- If $t : \mathbf{A}$ and $u : \mathbf{A}$ then $\mathbf{Paths}_{\mathbf{A}}(t, u)$ is a type.
- If $t : \mathbf{A}$ then $\mathbf{idpath}(t) : \mathbf{Paths}_{\mathbf{A}}(t, t)$.
- We write $t = u$ for $\mathbf{Paths}_{\mathbf{A}}(t, u)$.

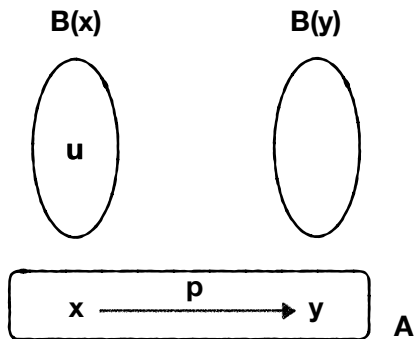
32

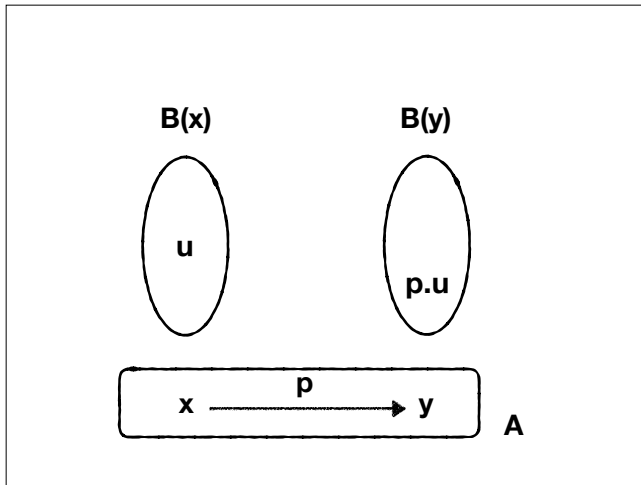
We now come to the most important type.

We shall adapt our view of what a type is: types do not tell us just

Transport

- Given a type A and $B : A \rightarrow \text{Type}$
- If $\alpha : \text{Paths}_A(x, y)$ and $s : B(x)$ then $\text{transport } B \ \alpha \ s : B(y)$.
- $\text{transport } B \ (\text{idpath}(x)) \ s \equiv s$.





Caveats

- Old people call path spaces “*identity types*” and use the notation $\text{Id}_A(x, y)$.
- Older people call path spaces “*propositional equality*” and they call equality “*judgmental equality*”.
- Half of the definition of path spaces is missing. The other half will be given later this week.
- Path spaces *are* the equality you are used to. Really!

We have only scratched the surface. You will find out the rest about path spaces throughout the week. But let me also say this (thanks to Peter Lumsdaine for warning me about it): the path spaces correspond not only to, well, spaces of paths, but also to equality as usually used in mathematics.

The equalities between types, $A \equiv B$ and $p \equiv_A q$, are *not* the ones you usually want or need, and in fact they are *invisible* in Coq. The difficulty is that in most mathematics you only see a very trivial kind of path spaces. Martín Escardó will speak more about this point.

Proofs as constructions

“Every natural number is even or odd.”

37

We banished logic, but we do not want to give it up forever.

There is an ingenious way of doing logic by construction. We can prove logical statements by constructing elements of types.

For instance, consider the statement “every number is even or odd”. How can we “prove” it by constructing something?

Proofs as constructions

“Every natural number is even or odd.”

$\prod(n:\text{nat}) \sum(m:\text{nat}) (n = 2m) + (n = 2m+1)$

38

Ask what an element of this type does. It is a map, which takes a number n and gives a pair. The pair tells you how to decompose n into a lowest bit and the rest of it. In essence, it *computes* the fact that every number is even or odd.

How do we *deny* P?

By constructing an element in
 $P \rightarrow \text{Empty}$

39

How do we show that something *isn't* the case?
For that we can use the empty type.

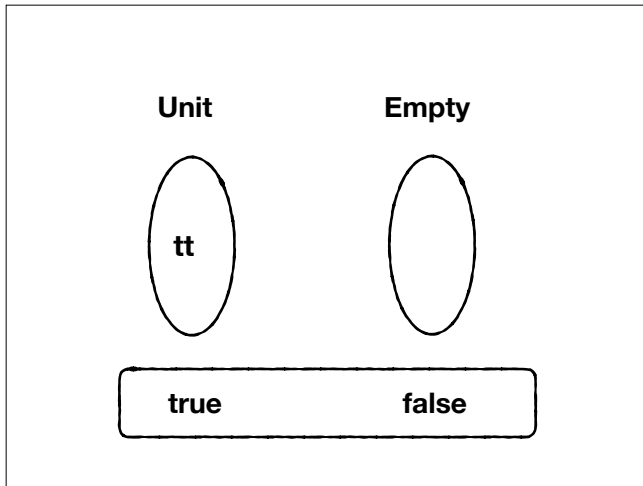
(false = true) \rightarrow Empty

```
Definition sanity : true = false  $\rightarrow$  Empty :=  
  fun (p : true = false)  $\Rightarrow$   
    transport  
      (ind_bool (fun _  $\Rightarrow$  Type) Unit Empty)  
      p  
      tt.
```

40

Let us walk through an example by showing that true isn't false. We need to construct map which takes a path p from true to false and turns it into an element of Empty.
Hopefully by mid-week you will be able to understand how such a map is defined in Coq, shown here. Let us draw the picture that this construction encodes.

41

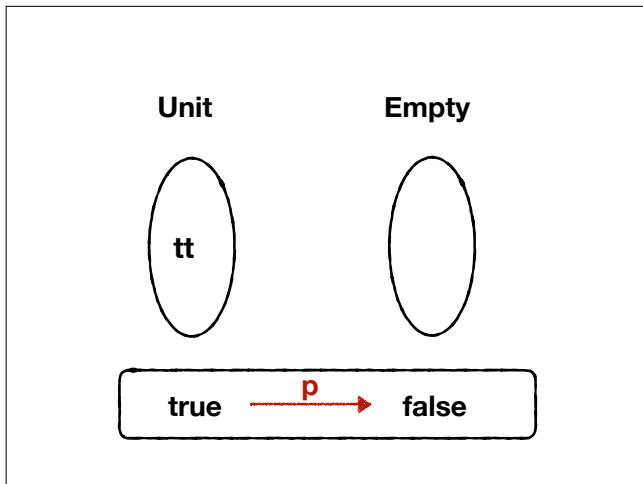


We start with a dependent type over Bool.

The fiber at true is the Unit type. The fiber at false is the Empty type.

(We did not actually say how such a dependent type is obtained, but let us say we can get it.)

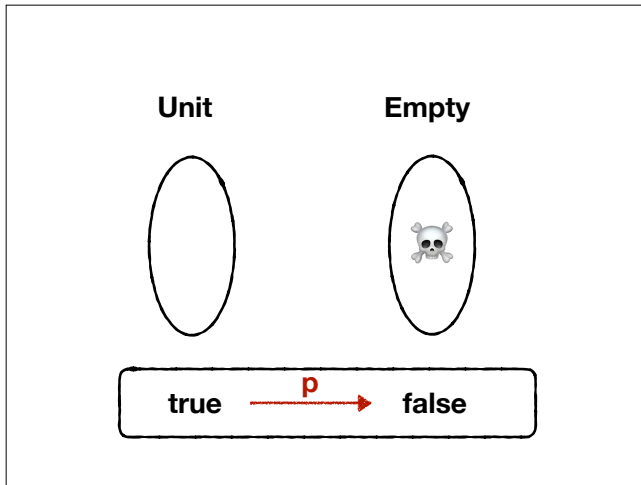
42



What happens if there is a path from true to false? Can we make an element of Empty?

43

Yes, we just transport tt across p .



44

We are not done!

- We left out precise rules!
- What about spheres, reals, groups, etc?
- Can paths be composed, or inverted?
- What is a path between paths?
- What is a path between types in **Type**?

I have to stop here. Bear in mind that this was only the introductory talk, which attempted to explain one possible intuition about type theory. You will learn other ways of looking at type theory throughout the week.

And please ask everyone many questions. Here is a list to get you started.

Further material

- Euclid: *Elements*
- Daniel Grayson: *An introduction to univalent foundations for mathematicians* ([arXiv:1711.01477](https://arxiv.org/abs/1711.01477))
- [UniMath](#) library
- Talk to people here!

45

If you would like some further reading, I recommend that you look at Euclid's elements and see what proportion is construction, and what proportion are statements.

A very good introduction to univalent foundations was written by Daniel Grayson.

Of course, for the brave there is always the Coq code.

Thank you.
