Computer Formalisation of Mathematics

Philipp G. Haselwarter[1]

Anja Petković[1]

[1]University of Ljubljana, Slovenia

Meeting of young mathematicians,
Women of mathematics on the Mediterranean shores,
Bled, 27.09.2019

[1]This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-17-1-0326.

1. As you may have noticed, I am not Philipp. Since we are working on very similar things, we decided to give this talk together.
2. We would like to talk about formalisation of mathematics - WHY and HOW we do this. Let us first shortly comment on the "WHY" part.

1. So, in principle we agree, that theorems should be true. But are they always true?
2. Let us remind ourselves how we make sure published papers are mistakes-free. We have reviewers, who go over every step of the proof carefully and try to find a mistake in the result. For big results, we may have 10 or more reviewers, but it boils down to people checking other people's work. This sounds bad.
3. Mistakes happen not just to students, but to big shot mathematicians too. One of the big shots this happened to was Vladimir Voevodsky.

- Theorems should be true.

1. So, in principle we agree, that theorems should be true. But are they always true?
2. Let us remind ourselves how we make sure published papers are mistakes-free. We have reviewers, who go over every step of the proof carefully and try to find a mistake in the result. For big results, we may have 10 or more reviewers, but it boils down to people checking other people's work. This sounds bad.
3. Mistakes happen not just to students, but to big shot mathematicians too. One of the big shots this happened to was Vladimir Voevodsky.

- Theorems should be true.
- But are they always?

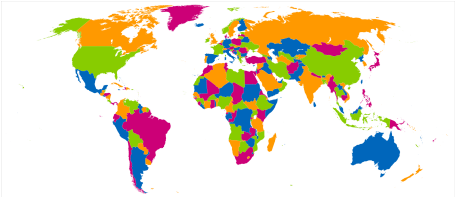
1. So, in principle we agree, that theorems should be true. But are they always true?
2. Let us remind ourselves how we make sure published papers are mistakes-free. We have reviewers, who go over every step of the proof carefully and try to find a mistake in the result. For big results, we may have 10 or more reviewers, but it boils down to people checking other people's work. This sounds bad.
3. Mistakes happen not just to students, but to big shot mathematicians too. One of the big shots this happened to was Vladimir Voevodsky.

Figure: Vladimir Voevodsky, 1966 - 2017

1. Vladimir Voevodsky was a russian mathematician, who got a Fields medal in 2002 for his proof of Milnor's Conjecture.
2. In 1991 he defined ∞-groupoids and proved they constitute models for homotopy types
3. In 2003, twelve years after the proof was published in English, a preprint appeared on the web in which his author, Carlos Simpson, very politely claimed that he has constructed a counter-example to the theorem. Since he was sure there was no mistake, Vladimir ignored that. In the Fall of 2013, he suddenly understood that Carlos Simpson was correct.
4. Vladimir was the founder of Univalent Mathematics, one of the basics for formalizing maths in computers.

Other reasons

- combinatorial proofs



Finish at: 4 min

1. There are several combinatorial proofs, which are solved by "manually" checking lots of special cases. The most famous one is the proof of the 4 color theorem. There were many, who doubted its correctness, especially because the computer checked a lot of configurations.
2. Another very wide range of motivating examples would be the mathematical view of computing. Once we go into that, we can mention self-driving cars, which we want to make sure is bug-free, drones, rockets if you want.

Other reasons

- combinatorial proofs

- mathematical view of computing

Finish at: 4 min

1. There are several combinatorial proofs, which are solved by "manually" checking lots of special cases. The most famous one is the proof of the 4 color theorem. There were many, who doubted its correctness, especially because the computer checked a lot of configurations.
2. Another very wide range of motivating examples would be the mathematical view of computing. Once we go into that, we can mention self-driving cars, which we want to make sure is bug-free, drones, rockets if you want.

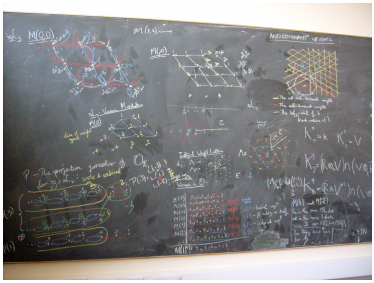
Computer representation of mathematical constructions

Want: (1) precise (2) capture mathematical practice (3) "compute"

1. 6 Thank you Anja. A language for formalised mathematics has to be precise. It should also capture mathematical practice, meaning that everyday mathematical concepts should be easy to represent rather than encoded. And of course we want the computer to do some of the work for us. After all, automation is where computers shine. I will call a "computer system for formalised maths" a "proof assistant". If I use words you never heard before please interrupt me. pause
2. Here we have an example of "everyday mathematical practice" at work. And in fact, this is too informal even for non-expert humans.
3. By the way, your jobs are safe for now. The creative part of maths still needs to be done by humans. pause
4. This is a theorem in Rudin's book on analysis. It's readable by non-experts, but it won't do for computers, because the Latex code is not semantic: it relies on the reader's experience to convey the meaning of "lim sup" and "implies", etc. It is also incomplete, just look at these dots here in line one, and this hand-wavy "of course" business in line two of the proof. pause
5. Maybe now you're thinking, "ah but Matlab can do computer maths". Here are some simple facts, checked by Matlab. But the result here is just a "logical 1". There is no proof explaining *why* these facts are true. So while Matlab and similar systems can compute, they're not proof assistants. pause
6. Finally here we have a statement of two theorems that a computer can understand. Lets read the first one. The Feit Thompson theorem states that a for a group G of finite type, if the order of G is odd, then G is solvable. And there's a proof here too, in fact we just apply a lemma, and the computer checked that all steps of deduction in the proof and the proof of the lemma and so on were correct. What you see here is a proof assistant based on a formal language called type theory.
7. Type theory is not the only choice possible, but it's one of the most successful ones, and it's the one we study, so let's have a look at it.

Computer representation of mathematical constructions

Want: (1) precise (2) capture mathematical practice (3) "compute"

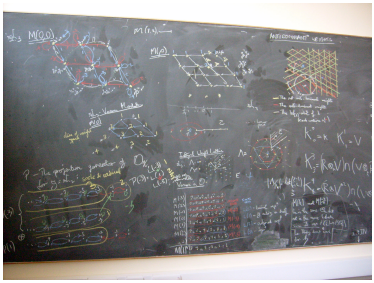


(a) too informal for (most) humans

1. 6 Thank you Anja. A language for formalised mathematics has to be precise. It should also capture mathematical practice, meaning that everyday mathematical concepts should be easy to represent rather than encoded. And of course we want the computer to do some of the work for us. After all, automation is where computers shine. I will call a "computer system for formalised maths" a "proof assistant". If I use words you never heard before please interrupt me. pause
2. Here we have an example of "everyday mathematical practice" at work. And in fact, this is too informal even for non-expert humans.
3. By the way, your jobs are safe for now. The creative part of maths still needs to be done by humans. pause
4. This is a theorem in Rudin's book on analysis. It's readable by non-experts, but it won't do for computers, because the Latex code is not semantic: it relies on the reader's experience to convey the meaning of "lim sup" and "implies", etc. It is also incomplete, just look at these dots here in line one, and this hand-wavy "of course" business in line two of the proof. pause
5. Maybe now you're thinking, "ah but Matlab can do computer maths". Here are some simple facts, checked by Matlab. But the result here is just a "logical 1". There is no proof explaining *why* these facts are true. So while Matlab and similar systems can compute, they're not proof assistants. pause
6. Finally here we have a statement of two theorems that a computer can understand. Lets read the first one. The Feit Thompson theorem states that a for a group G of finite type, if the order of G is odd, then G is solvable. And there's a proof here too, in fact we just apply a lemma, and the computer checked that all steps of deduction in the proof and the proof of the lemma and so on were correct. What you see here is a proof assistant based on a formal language called type theory.
7. Type theory is not the only choice possible, but it's one of the most successful ones, and it's the one we study, so let's have a look at it.

Computer representation of mathematical constructions

Want: (1) precise (2) capture mathematical practice (3) "compute"



(a) too informal for (most) humans

1.14 Theorem *If $f_n : X \to [-\infty, \infty]$ is measurable, for $n = 1, 2, 3, \ldots$, and*

$$g = \sup_{n \geq 1} f_n, \qquad h = \limsup_{n \to \infty} f_n,$$

then g and h are measurable.

PROOF $g^{-1}((\alpha, \infty]) = \bigcap_{n=1}^{\infty} f_n^{-1}((\alpha, \infty])$. Hence Theorem 1.12(c) implies that g is measurable. The same result holds of course with inf in place of sup, and since

$$h = \inf_{k \geq 1} \left\{ \sup_{i \geq k} f_i \right\},$$

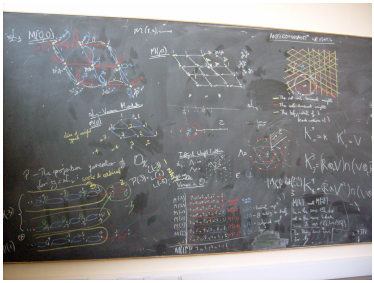
it follows that h is measurable. ////

(b) too informal for computers

1. 6 Thank you Anja. A language for formalised mathematics has to be precise. It should also capture mathematical practice, meaning that everyday mathematical concepts should be easy to represent rather than encoded. And of course we want the computer to do some of the work for us. After all, automation is where computers shine. I will call a "computer system for formalised maths" a "proof assistant". If I use words you never heard before please interrupt me. pause
2. Here we have an example of "everyday mathematical practice" at work. And in fact, this is too informal even for non-expert humans.
3. By the way, your jobs are safe for now. The creative part of maths still needs to be done by humans. pause
4. This is a theorem in Rudin's book on analysis. It's readable by non-experts, but it won't do for computers, because the Latex code is not semantic: it relies on the reader's experience to convey the meaning of "lim sup" and "implies", etc. It is also incomplete, just look at these dots here in line one, and this hand-wavy "of course" business in line two of the proof. pause
5. Maybe now you're thinking, "ah but Matlab can do computer maths". Here are some simple facts, checked by Matlab. But the result here is just a "logical 1". There is no proof explaining *why* these facts are true. So while Matlab and similar systems can compute, they're not proof assistants. pause
6. Finally here we have a statement of two theorems that a computer can understand. Lets read the first one. The Feit Thompson theorem states that a for a group G of finite type, if the order of G is odd, then G is solvable. And there's a proof here too, in fact we just apply a lemma, and the computer checked that all steps of deduction in the proof and the proof of the lemma and so on were correct. What you see here is a proof assistant based on a formal language called type theory.
7. Type theory is not the only choice possible, but it's one of the most successful ones, and it's the one we study, so let's have a look at it.

Computer representation of mathematical constructions

Want: (1) precise (2) capture mathematical practice (3) "compute"



(a) too informal for (most) humans

1.14 Theorem *If $f_n : X \to [-\infty, \infty]$ is measurable, for $n = 1, 2, 3, \ldots$, and*

$$g = \sup_{n \geq 1} f_n, \qquad h = \limsup_{n \to \infty} f_n,$$

then g and h are measurable.

PROOF $g^{-1}((\alpha, \infty]) = \bigcup_{n=1}^{\infty} f_n^{-1}((\alpha, \infty])$. Hence Theorem 1.12(c) implies that g is measurable. The same result holds of course with inf in place of sup, and since

$$h = \inf_{k \geq 1} \left\{ \sup_{i \geq k} f_i \right\},$$

it follows that h is measurable. ////

(b) too informal for computers

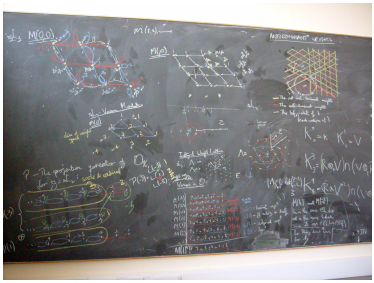
```
syms x
logical(1 < 2 & x == x)

ans =
  logical 1

isAlways(sin(x)/cos(x) == tan(x))

ans =
  logical 1
```

(c) Matlab – computes, but no proofs

1. 6 Thank you Anja. A language for formalised mathematics has to be precise. It should also capture mathematical practice, meaning that everyday mathematical concepts should be easy to represent rather than encoded. And of course we want the computer to do some of the work for us. After all, automation is where computers shine. I will call a "computer system for formalised maths" a "proof assistant". If I use words you never heard before please interrupt me. pause
2. Here we have an example of "everyday mathematical practice" at work. And in fact, this is too informal even for non-expert humans.
3. By the way, your jobs are safe for now. The creative part of maths still needs to be done by humans. pause
4. This is a theorem in Rudin's book on analysis. It's readable by non-experts, but it won't do for computers, because the Latex code is not semantic: it relies on the reader's experience to convey the meaning of "lim sup" and "implies", etc. It is also incomplete, just look at these dots here in line one, and this hand-wavy "of course" business in line two of the proof. pause
5. Maybe now you're thinking, "ah but Matlab can do computer maths". Here are some simple facts, checked by Matlab. But the result here is just a "logical 1". There is no proof explaining *why* these facts are true. So while Matlab and similar systems can compute, they're not proof assistants. pause
6. Finally here we have a statement of two theorems that a computer can understand. Lets read the first one. The Feit Thompson theorem states that a for a group G of finite type, if the order of G is odd, then G is solvable. And there's a proof here too, in fact we just apply a lemma, and the computer checked that all steps of deduction in the proof and the proof of the lemma and so on were correct. What you see here is a proof assistant based on a formal language called type theory.
7. Type theory is not the only choice possible, but it's one of the most successful ones, and it's the one we study, so let's have a look at it.

Computer representation of mathematical constructions

Want: (1) precise (2) capture mathematical practice (3) "compute"



(a) too informal for (most) humans

1.14 Theorem *If $f_n : X \to [-\infty, \infty]$ is measurable, for $n = 1, 2, 3, \ldots$, and*

$$g = \sup_{n \geq 1} f_n, \qquad h = \limsup_{n \to \infty} f_n,$$

then g and h are measurable.

PROOF $g^{-1}((\alpha, \infty]) = \bigcup_{n=1}^{\infty} f_n^{-1}((\alpha, \infty])$. Hence Theorem 1.12(c) implies that g is measurable. The same result holds of course with inf in place of sup, and since

$$h = \inf_{k \geq 1} \left\{ \sup_{i \geq k} f_i \right\},$$

it follows that h is measurable. ////

(b) too informal for computers

```
syms x
logical(1 < 2 & x == x)

ans =
  logical 1

isAlways(sin(x)/cos(x) == tan(x))

ans =
  logical 1
```

(c) Matlab – computes, but no proofs

```
Theorem Feit_Thompson (gT : finGroupType) (G : {group gT}) :
  odd #|G| → solvable G.
Proof. exact: (minSimpleOdd_ind no_minSimple_odd_group).
Qed.

Theorem simple_odd_group_prime (gT : finGroupType)
        (G : {group gT}) :
  odd #|G| → simple G → prime #|G|.
Proof. exact: (minSimpleOdd_prime no_minSimple_odd_group).
Qed.
```

(d) Type Theory ✓

1. 6 Thank you Anja. A language for formalised mathematics has to be precise. It should also capture mathematical practice, meaning that everyday mathematical concepts should be easy to represent rather than encoded. And of course we want the computer to do some of the work for us. After all, automation is where computers shine. I will call a "computer system for formalised maths" a "proof assistant". If I use words you never heard before please interrupt me. pause
2. Here we have an example of "everyday mathematical practice" at work. And in fact, this is too informal even for non-expert humans.
3. By the way, your jobs are safe for now. The creative part of maths still needs to be done by humans. pause
4. This is a theorem in Rudin's book on analysis. It's readable by non-experts, but it won't do for computers, because the Latex code is not semantic: it relies on the reader's experience to convey the meaning of "lim sup" and "implies", etc. It is also incomplete, just look at these dots here in line one, and this hand-wavy "of course" business in line two of the proof. pause
5. Maybe now you're thinking, "ah but Matlab can do computer maths". Here are some simple facts, checked by Matlab. But the result here is just a "logical 1". There is no proof explaining *why* these facts are true. So while Matlab and similar systems can compute, they're not proof assistants. pause
6. Finally here we have a statement of two theorems that a computer can understand. Lets read the first one. The Feit Thompson theorem states that a for a group G of finite type, if the order of G is odd, then G is solvable. And there's a proof here too, in fact we just apply a lemma, and the computer checked that all steps of deduction in the proof and the proof of the lemma and so on were correct. What you see here is a proof assistant based on a formal language called type theory.
7. Type theory is not the only choice possible, but it's one of the most successful ones, and it's the one we study, so let's have a look at it.

The language of type theory

- base types: $\mathbb{N}, \bot, \top, \mathbb{R}, \mathrm{Type}, \ldots$. Write $\vdash A$ type.

1. 9 First we need to represent the objects of mathematical study. These objects are called types. There's the type of natural numbers, and an empty type, a singleton type, the real numbers. There's even a type of all types, up to some cardinal if you're a set theorist. These are what we call the "base types".
2. We also like to combine existing types and construct new ones out of them: given two types we can take their cartesian product products A times B, we can take disjoint unions, written A plus B. We can take the product of all B_is over some indexing type, here the natural numbers, and we can consider the type of functions from A to B.
3. Now that we have types, we can think about the bits that make up a specific type. We call these bits *terms*. They're a bit like elements, but a term is always a term of a fixed type. For example, "42 is a natural number" becomes "42 is a term of type N", and we can talk about vectors in $\mathbb{R}^n$ as terms of type $\mathbb{R}^n$.
4. The example of vectors is interesting because it showcases a common theme in mathematical practice: The type we construct, $\mathbb{R}^n$, is *dependent* on n, which is itself a *term* of type natural number.
5. To give you another example of where dependency occurs, consider the type of continuous functions from the interval $[a, b]$ to the reals. This type mentions the pair a and b, which is a term of type R times R.
6. Our second requirement for a formal language was that it should model mathematical practice. Type theory fits this requirement, it has a notion of dependency built in.
7. In fact, a dependent type is a type valued function. The type of continuous functions from some interval to the reals can be seen as a function that's waiting to fix the endpoints of the interval in questions.

The language of type theory

- base types: $\mathbb{N}, \bot, \top, \mathbb{R}, \mathrm{Type}, \ldots$. Write $\vdash A$ type.
- type formers: Given $\vdash A$ type and $\vdash B$ type, can form $\vdash A \times B$ type , $\vdash A + B$ type , $\vdash \Pi_{(i:\mathbb{N})} B_i$ type , $\vdash \mathbb{N} \to \mathbb{R}$ type , ...

1. 9 First we need to represent the objects of mathematical study. These objects are called types. There's the type of natural numbers, and an empty type, a singleton type, the real numbers. There's even a type of all types, up to some cardinal if you're a set theorist. These are what we call the "base types".
2. We also like to combine existing types and construct new ones out of them: given two types we can take their cartesian product products A times B, we can take disjoint unions, written A plus B. We can take the product of all B_is over some indexing type, here the natural numbers, and we can consider the type of functions from A to B.
3. Now that we have types, we can think about the bits that make up a specific type. We call these bits *terms*. They're a bit like elements, but a term is always a term of a fixed type. For example, "42 is a natural number" becomes "42 is a term of type N", and we can talk about vectors in $\mathbb{R}^n$ as terms of type $\mathbb{R}^n$.
4. The example of vectors is interesting because it showcases a common theme in mathematical practice: The type we construct, $\mathbb{R}^n$, is *dependent* on n, which is itself a *term* of type natural number.
5. To give you another example of where dependency occurs, consider the type of continuous functions from the interval $[a, b]$ to the reals. This type mentions the pair a and b, which is a term of type R times R.
6. Our second requirement for a formal language was that it should model mathematical practice. Type theory fits this requirement, it has a notion of dependency built in.
7. In fact, a dependent type is a type valued function. The type of continuous functions from some interval to the reals can be seen as a function that's waiting to fix the endpoints of the interval in questions.

The language of type theory

- base types: $\mathbb{N}, \bot, \top, \mathbb{R}, \mathrm{Type}, \ldots$. Write $\vdash A$ type.
- type formers: Given $\vdash A$ type and $\vdash B$ type, can form $\vdash A \times B$ type , $\vdash A + B$ type , $\vdash \Pi_{(i:\mathbb{N})} B_i$ type , $\vdash \mathbb{N} \to \mathbb{R}$ type , ...
- the *terms* of a type are the "elements"
  Eg. "42 is a term of type $\mathbb{N}$" is written as $\vdash 42 : \mathbb{N}$.
  Write "given a $n : \mathbb{N}$, v is a vector in $\mathbb{R}^n$" as $n : \mathbb{N} \vdash x : \mathbb{R}^n$

1. 9 First we need to represent the objects of mathematical study. These objects are called types. There's the type of natural numbers, and an empty type, a singleton type, the real numbers. There's even a type of all types, up to some cardinal if you're a set theorist. These are what we call the "base types".
2. We also like to combine existing types and construct new ones out of them: given two types we can take their cartesian product products A times B, we can take disjoint unions, written A plus B. We can take the product of all B_is over some indexing type, here the natural numbers, and we can consider the type of functions from A to B.
3. Now that we have types, we can think about the bits that make up a specific type. We call these bits *terms*. They're a bit like elements, but a term is always a term of a fixed type. For example, "42 is a natural number" becomes "42 is a term of type N", and we can talk about vectors in $\mathbb{R}^n$ as terms of type $\mathbb{R}^n$.
4. The example of vectors is interesting because it showcases a common theme in mathematical practice: The type we construct, $\mathbb{R}^n$, is *dependent* on n, which is itself a *term* of type natural number.
5. To give you another example of where dependency occurs, consider the type of continuous functions from the interval $[a, b]$ to the reals. This type mentions the pair a and b, which is a term of type R times R.
6. Our second requirement for a formal language was that it should model mathematical practice. Type theory fits this requirement, it has a notion of dependency built in.
7. In fact, a dependent type is a type valued function. The type of continuous functions from some interval to the reals can be seen as a function that's waiting to fix the endpoints of the interval in questions.

The language of type theory

- base types: $\mathbb{N}, \bot, \top, \mathbb{R}, \mathrm{Type}, \ldots$. Write $\vdash A$ type.
- type formers: Given $\vdash A$ type and $\vdash B$ type, can form $\vdash A \times B$ type , $\vdash A + B$ type , $\vdash \Pi_{(i:\mathbb{N})} B_i$ type , $\vdash \mathbb{N} \to \mathbb{R}$ type , ...
- the *terms* of a type are the "elements"
  Eg. "42 is a term of type $\mathbb{N}$" is written as $\vdash 42 : \mathbb{N}$.
  Write "given a $n : \mathbb{N}$, v is a vector in $\mathbb{R}^n$" as $n : \mathbb{N} \vdash x : \mathbb{R}^n$
- vectors form a *dependent* type: $\mathbb{R}^n$ mentions n, a term of type $\mathbb{N}$.

1. 9 First we need to represent the objects of mathematical study. These objects are called types. There's the type of natural numbers, and an empty type, a singleton type, the real numbers. There's even a type of all types, up to some cardinal if you're a set theorist. These are what we call the "base types".
2. We also like to combine existing types and construct new ones out of them: given two types we can take their cartesian product products A times B, we can take disjoint unions, written A plus B. We can take the product of all B_is over some indexing type, here the natural numbers, and we can consider the type of functions from A to B.
3. Now that we have types, we can think about the bits that make up a specific type. We call these bits *terms*. They're a bit like elements, but a term is always a term of a fixed type. For example, "42 is a natural number" becomes "42 is a term of type N", and we can talk about vectors in $\mathbb{R}^n$ as terms of type $\mathbb{R}^n$.
4. The example of vectors is interesting because it showcases a common theme in mathematical practice: The type we construct, $\mathbb{R}^n$, is *dependent* on n, which is itself a *term* of type natural number.
5. To give you another example of where dependency occurs, consider the type of continuous functions from the interval $[a, b]$ to the reals. This type mentions the pair a and b, which is a term of type R times R.
6. Our second requirement for a formal language was that it should model mathematical practice. Type theory fits this requirement, it has a notion of dependency built in.
7. In fact, a dependent type is a type valued function. The type of continuous functions from some interval to the reals can be seen as a function that's waiting to fix the endpoints of the interval in questions.

The language of type theory

- base types: $\mathbb{N}, \bot, \top, \mathbb{R}, \mathrm{Type}, \ldots$. Write $\vdash A$ type.
- type formers: Given $\vdash A$ type and $\vdash B$ type, can form $\vdash A \times B$ type , $\vdash A + B$ type , $\vdash \Pi_{(i:\mathbb{N})} B_i$ type , $\vdash \mathbb{N} \to \mathbb{R}$ type , ...
- the *terms* of a type are the "elements" Eg. "42 is a term of type $\mathbb{N}$" is written as $\vdash 42 : \mathbb{N}$. Write "given a $n : \mathbb{N}$, v is a vector in $\mathbb{R}^n$" as $n : \mathbb{N} \vdash x : \mathbb{R}^n$
- vectors form a *dependent* type: $\mathbb{R}^n$ mentions n, a term of type $\mathbb{N}$.
- dependency is common : consider $\mathcal{C}([a, b], \mathbb{R})$. Here, a pair of terms $\langle a, b \rangle : \mathbb{R} \times \mathbb{R}$ occurs in the type!

1. 9 First we need to represent the objects of mathematical study. These objects are called types. There's the type of natural numbers, and an empty type, a singleton type, the real numbers. There's even a type of all types, up to some cardinal if you're a set theorist. These are what we call the "base types".
2. We also like to combine existing types and construct new ones out of them: given two types we can take their cartesian product products A times B, we can take disjoint unions, written A plus B. We can take the product of all B_is over some indexing type, here the natural numbers, and we can consider the type of functions from A to B.
3. Now that we have types, we can think about the bits that make up a specific type. We call these bits *terms*. They're a bit like elements, but a term is always a term of a fixed type. For example, "42 is a natural number" becomes "42 is a term of type N", and we can talk about vectors in $\mathbb{R}^n$ as terms of type $\mathbb{R}^n$.
4. The example of vectors is interesting because it showcases a common theme in mathematical practice: The type we construct, $\mathbb{R}^n$, is *dependent* on n, which is itself a *term* of type natural number.
5. To give you another example of where dependency occurs, consider the type of continuous functions from the interval $[a, b]$ to the reals. This type mentions the pair a and b, which is a term of type R times R.
6. Our second requirement for a formal language was that it should model mathematical practice. Type theory fits this requirement, it has a notion of dependency built in.
7. In fact, a dependent type is a type valued function. The type of continuous functions from some interval to the reals can be seen as a function that's waiting to fix the endpoints of the interval in questions.

The language of type theory

- base types: $\mathbb{N}, \bot, \top, \mathbb{R}, \mathrm{Type}, \ldots$. Write $\vdash A$ type.
- type formers: Given $\vdash A$ type and $\vdash B$ type, can form $\vdash A \times B$ type , $\vdash A + B$ type , $\vdash \Pi_{(i:\mathbb{N})} B_i$ type , $\vdash \mathbb{N} \to \mathbb{R}$ type , ...
- the *terms* of a type are the "elements" Eg. "42 is a term of type $\mathbb{N}$" is written as $\vdash 42 : \mathbb{N}$. Write "given a $n : \mathbb{N}$, v is a vector in $\mathbb{R}^n$" as $n : \mathbb{N} \vdash x : \mathbb{R}^n$
- vectors form a *dependent* type: $\mathbb{R}^n$ mentions n, a term of type $\mathbb{N}$.
- dependency is common : consider $\mathcal{C}([a, b], \mathbb{R})$. Here, a pair of terms $\langle a, b \rangle : \mathbb{R} \times \mathbb{R}$ occurs in the type!
- a dependent type is a Type-valued function: $\vdash \mathcal{C}([-, -], \mathbb{R}) : \mathbb{R} \times \mathbb{R} \to \mathrm{Type}$

1. 9 First we need to represent the objects of mathematical study. These objects are called types. There's the type of natural numbers, and an empty type, a singleton type, the real numbers. There's even a type of all types, up to some cardinal if you're a set theorist. These are what we call the "base types".
2. We also like to combine existing types and construct new ones out of them: given two types we can take their cartesian product products A times B, we can take disjoint unions, written A plus B. We can take the product of all B_is over some indexing type, here the natural numbers, and we can consider the type of functions from A to B.
3. Now that we have types, we can think about the bits that make up a specific type. We call these bits *terms*. They're a bit like elements, but a term is always a term of a fixed type. For example, "42 is a natural number" becomes "42 is a term of type N", and we can talk about vectors in $\mathbb{R}^n$ as terms of type $\mathbb{R}^n$.
4. The example of vectors is interesting because it showcases a common theme in mathematical practice: The type we construct, $\mathbb{R}^n$, is *dependent* on n, which is itself a *term* of type natural number.
5. To give you another example of where dependency occurs, consider the type of continuous functions from the interval $[a, b]$ to the reals. This type mentions the pair a and b, which is a term of type R times R.
6. Our second requirement for a formal language was that it should model mathematical practice. Type theory fits this requirement, it has a notion of dependency built in.
7. In fact, a dependent type is a type valued function. The type of continuous functions from some interval to the reals can be seen as a function that's waiting to fix the endpoints of the interval in questions.

Types and their terms

How do we construct terms of a type? We specify the generators! We call them "constructors" ; defined type by type, for example

- $\bot$ is the empty type : no constructors

1. 11 Let's quickly talk about how we can construct terms. Each type has its own ideas about what terms should this type should look like. Think of it like this: every type comes with its own generators. Because we like to build things, we call these generators "constructors".
2. For example the empty type has no constructors. It's empty.
3. The singleton type has a single constructor, written as star here.
4. The natural numbers have two constructors: zero, and, if we already have a natural number m, we can form its successor, m plus one.
5. To form a term in the product type, we need a term in each component, an a and a b.
6. To form a function, we can assume that we have some variable in the domain, and use that variable to construct a term in the codomain.

Types and their terms

How do we construct terms of a type? We specify the generators! We call them "constructors" ; defined type by type, for example

- $\bot$ is the empty type : no constructors
- $\top$ is the singleton : a single constructor $* : \top$

1. 11 Let's quickly talk about how we can construct terms. Each type has its own ideas about what terms should this type should look like. Think of it like this: every type comes with its own generators. Because we like to build things, we call these generators "constructors".
2. For example the empty type has no constructors. It's empty.
3. The singleton type has a single constructor, written as star here.
4. The natural numbers have two constructors: zero, and, if we already have a natural number m, we can form its successor, m plus one.
5. To form a term in the product type, we need a term in each component, an a and a b.
6. To form a function, we can assume that we have some variable in the domain, and use that variable to construct a term in the codomain.

Types and their terms

How do we construct terms of a type? We specify the generators! We call them "constructors" ; defined type by type, for example

- $\bot$ is the empty type : no constructors
- $\top$ is the singleton : a single constructor $* : \top$
- $0 : \mathbb{N}$. Given some $m : \mathbb{N}$, we can form $m + 1 : \mathbb{N}$.

1. 11 Let's quickly talk about how we can construct terms. Each type has its own ideas about what terms should this type should look like. Think of it like this: every type comes with its own generators. Because we like to build things, we call these generators "constructors".
2. For example the empty type has no constructors. It's empty.
3. The singleton type has a single constructor, written as star here.
4. The natural numbers have two constructors: zero, and, if we already have a natural number m, we can form its successor, m plus one.
5. To form a term in the product type, we need a term in each component, an a and a b.
6. To form a function, we can assume that we have some variable in the domain, and use that variable to construct a term in the codomain.

Types and their terms

How do we construct terms of a type? We specify the generators! We call them "constructors" ; defined type by type, for example

- $\bot$ is the empty type : no constructors
- $\top$ is the singleton : a single constructor $* : \top$
- $0 : \mathbb{N}$. Given some $m : \mathbb{N}$, we can form $m + 1 : \mathbb{N}$.
- If $a : A$ and $b : B$, then $\langle a, b \rangle : A \times B$.

1. 11 Let's quickly talk about how we can construct terms. Each type has its own ideas about what terms should this type should look like. Think of it like this: every type comes with its own generators. Because we like to build things, we call these generators "constructors".
2. For example the empty type has no constructors. It's empty.
3. The singleton type has a single constructor, written as star here.
4. The natural numbers have two constructors: zero, and, if we already have a natural number m, we can form its successor, m plus one.
5. To form a term in the product type, we need a term in each component, an a and a b.
6. To form a function, we can assume that we have some variable in the domain, and use that variable to construct a term in the codomain.

Types and their terms

How do we construct terms of a type? We specify the generators! We call them "constructors" ; defined type by type, for example

- $\bot$ is the empty type : no constructors
- $\top$ is the singleton : a single constructor $* : \top$
- $0 : \mathbb{N}$. Given some $m : \mathbb{N}$, we can form $m + 1 : \mathbb{N}$.
- If $a : A$ and $b : B$, then $\langle a, b \rangle : A \times B$.
- Functions? Assume a variable x of type A and form a term $e : B$. That's a function $\lambda x{:}A\,.\,e : A \to B$

1. 11 Let's quickly talk about how we can construct terms. Each type has its own ideas about what terms should this type should look like. Think of it like this: every type comes with its own generators. Because we like to build things, we call these generators "constructors".
2. For example the empty type has no constructors. It's empty.
3. The singleton type has a single constructor, written as star here.
4. The natural numbers have two constructors: zero, and, if we already have a natural number m, we can form its successor, m plus one.
5. To form a term in the product type, we need a term in each component, an a and a b.
6. To form a function, we can assume that we have some variable in the domain, and use that variable to construct a term in the codomain.

Constructions are good, but where are the theorems?

1. 14 Now we have seen how mathematical constructions are represented as types. But the original question that Anja asked was about using a computer to check the proofs of theorems. We haven't seen any logic, any propositions yet in type theory. So how do we represent theorems and proofs in type theory? pause
2. The key insight that Curry and Howard had was that we already have everything we need : propositions *are* types. What I mean by that is that we can take the usual notion of logic you are acquainted with and interpret it in type theory.
3. Let's see how this works. The propositions are types, and a proof of a proposition A is simply a term t of inhabiting that type. pause
4. Here's a little dictionary that explains how to translate a logical statement into type theory. The idea is the following: To prove the conjunction of A and B, we have to provide a proof for A, and a proof for B. In other words, we construct a pair of proofs. But that's just a term of type A times B. If, for example, we want to give a proof that A implies B, we have to construct a proof of B, assuming that we have a proof of A. That's exactly a function from A to B. Truth is interpreted as the singleton type, falsehood as the empty type.
5. We also find dependent types again, namely in the form of predicates.

Constructions are good, but where are the theorems?

Insight (Curry, Howard, ca 1960): propositions *are* types!
And a proof of a proposition A is... a term $t : A$.

1. 14 Now we have seen how mathematical constructions are represented as types. But the original question that Anja asked was about using a computer to check the proofs of theorems. We haven't seen any logic, any propositions yet in type theory. So how do we represent theorems and proofs in type theory? pause
2. The key insight that Curry and Howard had was that we already have everything we need : propositions *are* types. What I mean by that is that we can take the usual notion of logic you are acquainted with and interpret it in type theory.
3. Let's see how this works. The propositions are types, and a proof of a proposition A is simply a term t of inhabiting that type. pause
4. Here's a little dictionary that explains how to translate a logical statement into type theory. The idea is the following: To prove the conjunction of A and B, we have to provide a proof for A, and a proof for B. In other words, we construct a pair of proofs. But that's just a term of type A times B. If, for example, we want to give a proof that A implies B, we have to construct a proof of B, assuming that we have a proof of A. That's exactly a function from A to B. Truth is interpreted as the singleton type, falsehood as the empty type.
5. We also find dependent types again, namely in the form of predicates.

Constructions are good, but where are the theorems?

Insight (Curry, Howard, ca 1960): propositions *are* types!
And a proof of a proposition A is... a term $t : A$.

A dictionary:

English	Type Theory
True	$\top$
False	$\bot$
A and B	$A \times B$
A or B	$A + B$
If A then B	$A \to B$
A if and only if B	$(A \to B) \times (B \to A)$
Not A	$A \to \bot$
Predicate on A	$A \to \mathrm{Type}$
For all $x{:}A$, B	$\Pi_{x:A} B$
There exists $x{:}A$ s.t. B	$\Sigma_{x:A} B$

1. 14 Now we have seen how mathematical constructions are represented as types. But the original question that Anja asked was about using a computer to check the proofs of theorems. We haven't seen any logic, any propositions yet in type theory. So how do we represent theorems and proofs in type theory? pause
2. The key insight that Curry and Howard had was that we already have everything we need : propositions *are* types. What I mean by that is that we can take the usual notion of logic you are acquainted with and interpret it in type theory.
3. Let's see how this works. The propositions are types, and a proof of a proposition A is simply a term t of inhabiting that type. pause
4. Here's a little dictionary that explains how to translate a logical statement into type theory. The idea is the following: To prove the conjunction of A and B, we have to provide a proof for A, and a proof for B. In other words, we construct a pair of proofs. But that's just a term of type A times B. If, for example, we want to give a proof that A implies B, we have to construct a proof of B, assuming that we have a proof of A. That's exactly a function from A to B. Truth is interpreted as the singleton type, falsehood as the empty type.
5. We also find dependent types again, namely in the form of predicates.

How do we now prove stuff?

$$\neg(P \vee Q) \Rightarrow (\neg P \wedge \neg Q)$$

Finish demo by: 19 min

1. this dictionary shows that we introduced proof relevance: it matters what construction we used, and our implications are now functions!
2. we need to change the way we view propositions, not just what is valid, but what we can prove
3. there are truncation details with disjunction, but let us not get into that
4. how do we actually prove stuff? By constructing proof terms
5. exercise: $((P + Q) \to \emptyset) \to ((P \to \emptyset) \times (Q \to \emptyset))$ First explicitly and then using tactics
6. demonstrate a mistake in coq
7. fail (checks something actually fails)
8. a real example?

How do we now prove stuff?

$$\neg(P \vee Q) \Rightarrow (\neg P \wedge \neg Q)$$

Finish demo by: 19 min

1. this dictionary shows that we introduced proof relevance: it matters what construction we used, and our implications are now functions!
2. we need to change the way we view propositions, not just what is valid, but what we can prove
3. there are truncation details with disjunction, but let us not get into that
4. how do we actually prove stuff? By constructing proof terms
5. exercise: $((P + Q) \to \emptyset) \to ((P \to \emptyset) \times (Q \to \emptyset))$ First explicitly and then using tactics
6. demonstrate a mistake in coq
7. fail (checks something actually fails)
8. a real example?

Cute, but what can you do?

Some cool formalisation projects using the Coq proof assistant:

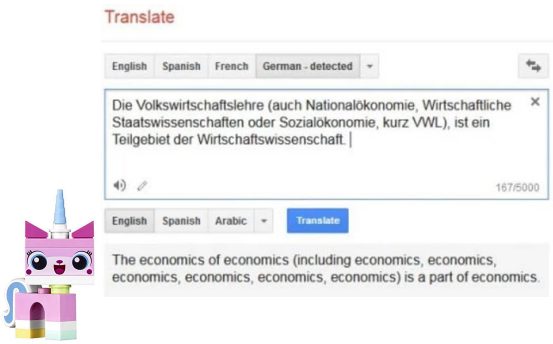
- formalise some serious maths : Odd order theorem (part of classification of finite simple groups), four colour theorem, lots of homotopy theory, ...
- verify computer programs : a C compiler, research papers about programming languages, crypto protocols, proof assistants, ...

Other cool things using type theory:

- lots of models – lots of type theories!
- eg. homotopy type theory – talk to Egbert here
- Andromeda : a proof assistant that lets you define your own type theory, developed at FMF in Ljubljana

1. 21 Okay so now that we convinced you that proof assistants based on type theory can be used for formalisation *in principle*, let me give you some examples of what has been done *in practice*.
2. First I'd like to mention some cool formalisation projects. The Coq proof assistants has been used to verify some serious theorems. The odd order theorem states that every finite group of odd order is solvable. When it was first published, it was infamous for its long proof. The four colour theorem was mentioned by Anja at the beginning of the talk, and the thousands of different cases have been checked using the Coq prover. In recent years people have formalised lots of homotopy theory.
3. People have also verified the correctness of substantial computer programs, like a C compiler or cryptographic protocols that are used in the code of the Firefox browser today. And of course people felt the urge to verify proof assistants themselves.
4. Finally, type theory is also a lot of fun if you're sure you never make mistakes and don't need to formalise anything.
5. Actually, there is not just one type theory, but there are lots of them, just like there are lots of different sub-languages for different fields of maths. I don't have time to go into this, but Egbert here can tell you more about it. He's also an expert about the connection between type theory and homotopy theory.
6. At FMF we developed a proof assistant that allows you to define your own type theory and prove things with it. And that's what Anja and I work on.

Why not use AI?



Soon, but not yet.