Introduction to formalizing number theory

Anne Baanen

Vrije Universiteit Amsterdam

Intercity number theory seminar
2022-06-03

Formalizing mathematics

No foundations needed!

This talk is not about foundations of maths.

If the computer scientists and logicians did their job well, formalization doesn't need to involve any foundations.

What is formalization?

Computer formalization is the translation of mathematics from a natural language like English to a formal computer language.

What is formalization?

Computer formalization is the translation of mathematics from a natural language like English to a formal computer language.

Formal implies precise, but high-level formal languages exist.

What is formalization?

Computer formalization is the translation of mathematics from a natural language like English to a formal computer language.

Formal implies precise, but high-level formal languages exist.

Examples of computer languages:

- Agda
- Coq
- Isabelle/HOL
- Lean
- MetaMath
- Mizar
- ...

What is formalization?

Formalization is manual:

- Read maths textbook
- Make sure you understand completely
- Write computer code

What is formalization?

Formalization (typesetting) is manual:

- Read maths textbook (blackboard)
- Make sure you understand completely
- Write computer code (in LaTeX)

What is formalization?

Formalization (typesetting) is manual:

- Read maths textbook (blackboard)
- Make sure you understand completely
- Write computer code (in LaTeX)

Automated theorem provers try to generate proofs automatically. Automation can handle the easy proofs, leaving the hard parts to manual formalization.

Interactive theorem proving

An interactive theorem prover or proof assistant interprets the code you write.

Interactive theorem proving

An interactive theorem prover or proof assistant interprets the code you write.

Python works with numbers and lists, Lean works with functions and proofs.

Interactive theorem proving

Demo: there are infinitely many primes

As I wrote code, I could check the available hypotheses and goal,
the documentation for theorems and functions
and see errors immediately.

Interactive theorem proving

As I wrote code, I could check the available hypotheses and goal,
the documentation for theorems and functions
and see errors immediately.

This was an elementary example but it scales up.
Formalizing the finiteness of the class number works the same (given appropriate definitions and lemmas).

Getting definitions right

Formalization is not just theorem proving: we need to get definitions right too.

Getting definitions right

Formalization is not just theorem proving: we need to get definitions right too.

A definition in Lean is followed by little lemmas:

```
def prime (p : ℕ) := irreducible p
lemma not_prime_zero : ¬ prime 0
lemma not_prime_one : ¬ prime 1
lemma prime.ne_zero {p : ℕ} : prime p → p ≠ 0
lemma prime.pos {p : ℕ} : prime p → 0 < p
```

Proofs should be easy once you get your definitions right.

How to read computer languages

Functions and the : relation

Lean writes `f x` for "$f(x)$",
`n : ℕ` for "$n \in \mathbb{N}$",
and `p_prime : nat.prime p` for "the claim that p is prime".

Functions and the : relation

Lean writes `f x` for "$f(x)$",
`n :` $\mathbb{N}$ for "$n \in \mathbb{N}$",
and `p_prime : nat.prime p` for "the claim that p is prime".

This is because Lean is based on types instead of sets.
$\mathbb{N}$ is the type of all natural numbers
and `nat.prime p` is the type of all proofs that `p` is prime.

Types and sets

Think of types as sets, except each element can only have *one* type.

`(0 : ℕ) = (0 : ℝ)` raises a type error,
just like `1 + "green"` in Python.

Types and sets

Think of types as sets, except each element can only have *one* type.

`(0 : ℕ) = (0 : ℝ)` raises a type error,
just like `1 + "green"` in Python.

Still, Lean understands `(0 : ℕ)` and `(0 : ℝ)`: how does `0` have two types?

Kernel and Elaborator

Lean has a kernel that understands only pure type theory,
and an elaborator that turns high-level language into type theory.

We only need to trust the correctness of a simple kernel, while the elaborator protects us humans from the foundations.

Kernel and Elaborator

Lean has a kernel that understands only pure type theory,
and an elaborator that turns high-level language into type theory.

We only need to trust the correctness of a simple kernel, while the elaborator protects us humans from the foundations.

My computer has a CPU that understands only machine code,
and a compiler that turns high-level language into machine code.

Kernel and Elaborator

The elaborator supplies "obvious" values that the user can leave out, turning `(0 : ℕ)` into `@has_zero.zero ℕ nat.has_zero` and `nat.prime.pos p_prime` into `@nat.prime.pos p p_prime`.

Kernel and Elaborator

The elaborator supplies "obvious" values that the user can leave out, turning `(0 : ℕ)` into `@has_zero.zero ℕ nat.has_zero` and `nat.prime.pos p_prime` into `@nat.prime.pos p p_prime`.

A tactic is a little program inside the elaborator to supply (less obvious) proofs that the user can leave out, such as the proof that `k ≠ 1` that `linarith` supplied.

Equality

Lean also understands "obvious" equalities:
we proved `nat.prime p` by showing `nat.min_fac k` is prime. This works because `p` is defined to be `nat.min_fac k`.

Not all obvious equalities hold by definition:
to show `p | 1` using `p | n! + 1 - n!` we need a theorem.

Equality

Lean also understands "obvious" equalities:
we proved `nat.prime p` by showing `nat.min_fac k` is prime. This works because `p` is defined to be `nat.min_fac k`.

Not all obvious equalities hold by definition:
to show `p | 1` using `p | n! + 1 - n!` we need a theorem.

A good definition will make all equalities that are obvious also hold by definition.

Mathematical libraries

The Lean mathematical library

The proof of infinitely many primes used definitions from the Lean mathematical library mathlib.

The goal of mathlib is to provide a coherent collection of Lean code for as much mathematics as is feasible.

Examples of mathematical libraries

Like languages, many libraries exist:

- Agda standard library
- Coq Mathematical Components
- Isabelle/HOL Archive of Formal Proofs
- ...

Examples of mathematical libraries

Like languages, many libraries exist:

- Agda standard library
- Coq Mathematical Components
- Isabelle/HOL Archive of Formal Proofs
- ...

Libraries range from intuitionistic to classical, tightly integrated to modular, focussed to universalist, powerful to selfexplanatory.

Tight integration in mathlib

mathlib aims for self-consistency and power over modularity and explanatory value.

This means you can't just pick a random textbook definition or theorem and start translating it: you pick the best definition and most powerful theorem.

Tight integration in mathlib

mathlib aims for self-consistency and power over modularity and explanatory value.

This means you can't just pick a random textbook definition or theorem and start translating it: you pick the best definition and most powerful theorem.

We want to ensure finite-dimensionality applies to finite field extensions over $\mathbb{Q}$ and to the Euclidean plane over $\mathbb{R}$.

We cannot make excuses to the computer: we have to get everything right everywhere.

Tight integration in mathlib

Drawback: mathlib is unstable.

An inconvenient definition of vector space will be replaced (breaking everything that uses the original definition).

Tight integration in mathlib

Drawback: mathlib is unstable.

An inconvenient definition of vector space will be replaced (breaking everything that uses the original definition).

At least the computer can help us determine which usages broke (and if you're clever, fix the breakage!)

Tight integration in mathlib

Advantage: mathlib is unstable.

Any project that makes use of mathlib is liable to break,
so just contribute your work directly to mathlib,
and the mathlib contributors will keep it up to date.

This means mathlib will grow as a tightly-integrated library.

Why formalize?

Practical application 1: verification

The computer can verify correctness of all details of a proof.
We can be sure a theorem is proved if the proof is computer-checked.

Especially useful for new, long, complicated, computer-generated proofs:

1. Condensed mathematics
2. Four colour theorem
3. Kepler conjecture
4. Odd order theorem

Practical application 1: verification

Reviewing a paper becomes easier if the proof is formalized: you check whether it's interesting and relevant, the computer checks the correctness.

Sci-fi utopia: replace LaTeX as a submission method to the arXiv with a language like Lean.

Practical application 2: reasoning

By translating mathematics to a computer language, computers can help us in doing mathematical reasoning.

I can ask Lean whether I used all hypotheses in my proof, how sensitive the conclusion is to tweaking the hypothesis, which intermediate results a theorem depends on.

Practical application 2: reasoning

Working with a definition teaches us about this definition. With no excuses, you need to learn these lessons.

If you want to reason “analogously”, you have to point out exactly which analogy you use. Is $\mathbb{Q}(\alpha)$ generated by a single element in the same way any finite separable field extension is?

Practical application 2: reasoning

Working with a definition teaches us about this definition. With no excuses, you need to learn these lessons.

If you want to reason "analogously", you have to point out exactly which analogy you use. Is $\mathbb{Q}(\alpha)$ generated by a single element in the same way any finite separable field extension is?

Here, it helps that the computer is a pedant.

Practical application 2: reasoning

In programmer terms, Lean is a more perfect rubber duck.

Practical application 2: reasoning

In programmer terms, Lean is a more perfect rubber duck.

Sci-fi utopia: replace Sage as a quick tool for checking whether some property is true, with a computer program that does computation with numbers and reasoning with proofs.

Practical application 3: learning

Teaching works better if students get instant feedback. Let them practice writing proofs and have Lean grade them in a second, not week.

Formal mathematics is more interactive than a paper textbook.

Practical application 3: learning

Teaching works better if students get instant feedback. Let them practice writing proofs and have Lean grade them in a second, not week.

Formal mathematics is more interactive than a paper textbook.

Computer science logic courses have been using proof assistants for years, and maths courses are starting to use them too.

Practical application 3: learning

Libraries of formal mathematics also let computers learn: searching through the library for examples or counterexamples to theorems, or teaching a neural network how to write correct proofs.

Practical application 3: learning

Libraries of formal mathematics also let computers learn: searching through the library for examples or counterexamples to theorems, or teaching a neural network how to write correct proofs.

Sci-fi utopia: the computer does grading, explaining and writing textbooks for you. If you need to check a fact, you can search through a huge library of mathematics instantly.

Non-practical application

To me, doing formal mathematics can be just as fun and beautiful as pen and paper. Formal proofs can look just as elegant as proofs on paper with the right eye.

Interactivity makes proving like playing a video game.

Non-practical application

To me, doing formal mathematics can be just as fun and beautiful as pen and paper. Formal proofs can look just as elegant as proofs on paper with the right eye.

Interactivity makes proving like playing a video game.

Sci-fi utopia: the newspaper crossword is replaced with an unformalized theorem.

Our formal future

Increasing awareness of formal mathematics

Formalization is not just an obscure computer scientist / logician hobby.
An increasing number of mathematicians are getting interested.

Increasing awareness of formal mathematics

Formalization is not just an obscure computer scientist / logician hobby.
An increasing number of mathematicians are getting interested.

In the near future, formalizing each paper is still not feasible.
I expect we'll see people regularly formalizing tricky parts of their proofs.

Automating away the boring parts

People are writing ever more useful automated tactics, and machine learning is making ever better proof suggestions.

Automating away the boring parts

People are writing ever more useful automated tactics, and machine learning is making ever better proof suggestions.

These techniques are complementary: if you are lucky the proof is automatic, and if you are unlucky you can tell exactly which tactics to use at each point.

Automating away the boring parts

People are writing ever more useful automated tactics, and machine learning is making ever better proof suggestions.

These techniques are complementary: if you are lucky the proof is automatic, and if you are unlucky you can tell exactly which tactics to use at each point.

I am not aware of much automation focussed on creating new definitions and theorems, just proving existing ones. Human mathematicians will still be needed to judge the value of new results.

Replacing human mathematicians

If the future is formal, it means we need more mathematicians, since formalizing mathematics opens up new areas of research and at best helping with the boring work, at worst creating new boring work.

Replacing human mathematicians

If the future is formal, it means we need more mathematicians, since formalizing mathematics opens up new areas of research and at best helping with the boring work, at worst creating new boring work.

The job of mathematician may change, it will not disappear yet.

Conclusion

The best way to predict the future is to build it.

This is the future I want to build: by translating our mathematics into a computer language, we unlock a new, interesting field of study and we gain a powerful ally for computation and for reasoning.