Computing with or despite the computer

Anne Baanen

Vrije Universiteit Amsterdam

Machine Assisted Proofs, 2023-02-14

Computing the class number

I am formalizing algebraic number theory in Lean as part of mathlib.
With Alex J. Best, Nirvana Coppola and Sander Dahmen,
I worked on computing the class number of some rings of integers.

Computing the class number

I am formalizing algebraic number theory in Lean as part of mathlib.
With Alex J. Best, Nirvana Coppola and Sander Dahmen,
I worked on computing the class number of some rings of integers.

Class number: the (finite) cardinality of the ideal class group.
Class group: the quotient of the invertible fractional ideals by principal fractional ideals.
Fractional ideal of R: an R-submodule of $\mathrm{Frac}(R)$ such that an R-multiple is contained in R.

Computing the class number

I am formalizing algebraic number theory in Lean as part of mathlib.
With Alex J. Best, Nirvana Coppola and Sander Dahmen,
I worked on computing the class number of some rings of integers.

Class number: the (finite) cardinality of the ideal class group.
Class group: the quotient of the invertible fractional ideals by principal fractional ideals.
Fractional ideal of R: an R-submodule of $\mathsf{Frac}(R)$ such that an R-multiple is contained in R.
Ring of integers: the integral closure of $\mathbb{Z}$ in a number field.
Number field: a finite field extension of $\mathbb{Q}$.

...

Why the class number?

We compute the class number for a few reasons:

Why the class number?

We compute the class number for a few reasons:

Useful: we use the class number to learn something about the solutions to Diophantine equations.

Why the class number?

We compute the class number for a few reasons:

Useful: we use the class number to learn something about the solutions to Diophantine equations.

Testable: verify a relatively involved definition can fit together to produce a concrete natural number.

Why the class number?

We compute the class number for a few reasons:

Useful: we use the class number to learn something about the solutions to Diophantine equations.

Testable: verify a relatively involved definition can fit together to produce a concrete natural number.

Doable: class numbers are known for over a century, and computer algebra systems can do it in milliseconds.

Why compute it in Lean?

If you can find the class number by hand in a few minutes, and Sage answers in less than a second, why did we spend months on it in Lean?

Why compute it in Lean?

If you can find the class number by hand in a few minutes, and Sage answers in less than a second, why did we spend months on it in Lean?

Because we had to spend months on it in Lean!
We want to identify the barriers that make the Lean computation so hard.

Caveat

We did not actually spend months only for a few computations, most of our time was spent:

- Setting up the definitions
- Filling in missing theory
- Figuring out the right level of generality
- Understanding Lean's limitations
- Polishing the result

Views of computation

What do we mean by computing?

Computing the class number on paper is not the same thing as computing the class number in Sage, or in Lean.

What do we mean by computing?

Computing the class number on paper is not the same thing as computing the class number in Sage, or in Lean.

Mathematics: computation proves equalities without needing creative insight.

What do we mean by computing?

Computing the class number on paper is not the same thing as computing the class number in Sage, or in Lean.

Mathematics: computation proves equalities without needing creative insight.

Computer science: a computation is a fixed process mapping input data to output.

What do we mean by computing?

Computing the class number on paper is not the same thing as computing the class number in Sage, or in Lean.

Mathematics: computation proves equalities without needing creative insight.

Computer science: a computation is a fixed process mapping input data to output.

Formalizing: a computation is a process showing the output is the correct answer to the problem posed in the input.

Examples of computation

The prototypical examples agree with all three notions:

$$37 + 5 = 6 * 7$$

Addition and multiplication are well-defined processes mapping input to output, so to show this equality, we compute and verify the output matches our expectation.

Examples of computation

For mathematical computation, I think diagram chasing:
there are just a few operations to do at each stage until we prove the desired property.

Examples of computation

For mathematical computation, I think diagram chasing:
there are just a few operations to do at each stage until we prove the desired property.

Not a CS computation: the input and output are properties, not data.

Examples of computation

For mathematical computation, I think diagram chasing:
there are just a few operations to do at each stage until we prove the desired property.

Not a CS computation: the input and output are properties, not data.

With the right tools, it can be a formalized computation:
the input problem is "is this map zero", output is yes/no + correctness proof.
Lean's simplifier is good at these jobs.

Examples of computation

A mathematical computation shows the squares in $\mathbb{Z}/4\mathbb{Z}$ are 0 and 1: simply consider all 4 possibilities.

Examples of computation

A mathematical computation shows the squares in $\mathbb{Z}/4\mathbb{Z}$ are 0 and 1: simply consider all 4 possibilities.

This maps directly to a formalized computation in Lean:

```
theorem zmod4.square_iff :
  ∀ d : zmod 4, -- Let d ∈ ℤ/4ℤ. Then
  (∃ x, x^2 = d) ↔ -- d is a square, iff
  (d ∈ {0, 1}) := -- d is either 0 or 1
begin -- Proof:
  dec_trivial -- consider all possibilities.
end -- QED
```

Examples of computation

A mathematical computation shows the squares in $\mathbb{Z}/4\mathbb{Z}$ are 0 and 1: simply consider all 4 possibilities.

This maps directly to a formalized computation in Lean:

```
theorem zmod4.square_iff :
  ∀ d : zmod 4, -- Let d ∈ ℤ/4ℤ. Then
  (∃ x, x^2 = d) ↔ -- d is a square, iff
  (d ∈ {0, 1}) := -- d is either 0 or 1
begin -- Proof:
  dec_trivial -- consider all possibilities.
end -- QED
```

In fact, `dec_trivial` invokes the computer science notion of computation!

Definitional equality

A bit of dependent type theory

Lean is based on the Calculus of Constructions, a constructive dependent type theory closely related to Martin-Löf type theory and Homotopy type theory.

A bit of dependent type theory

Lean is based on the Calculus of Constructions, a constructive dependent type theory closely related to Martin-Löf type theory and Homotopy type theory.

Martin-Löf wanted to unify the theory of programming languages (read: computer computations) with the logic and objects of mathematics.

A bit of dependent type theory

So, we assign a *computational* interpretation to our logic and objects, following Brouwer, Heyting, Kolmogorov, Curry and Howard:

A bit of dependent type theory

So, we assign a *computational* interpretation to our logic and objects, following Brouwer, Heyting, Kolmogorov, Curry and Howard:

- A proof of a conjunction $P \wedge Q$ is a proof of P together with a proof of Q.

A bit of dependent type theory

So, we assign a *computational* interpretation to our logic and objects, following Brouwer, Heyting, Kolmogorov, Curry and Howard:

- A proof of a conjunction $P \wedge Q$ is a proof of P together with a proof of Q.
- A proof of an implication $P \to Q$ is a procedure turning a proof of P into a proof of Q.

A bit of dependent type theory

So, we assign a *computational* interpretation to our logic and objects, following Brouwer, Heyting, Kolmogorov, Curry and Howard:

- A proof of a conjunction $P \wedge Q$ is a proof of P together with a proof of Q.
- A proof of an implication $P \to Q$ is a procedure turning a proof of P into a proof of Q.

A proof that $(P \wedge Q) \to P$ is a procedure taking the first element p of a pair (p, q). In MLTT this is one of the primitive operators fst that we define as part of the axioms.

We identify logical propositions with the set (type) of their proofs:

A bit of dependent type theory

We identify logical propositions with the set (type) of their proofs:
$P \wedge Q$ is the cartesian product of P and Q,
$P \to Q$ is the set of functions with domain P and codomain Q.

A bit of dependent type theory

We identify logical propositions with the set (type) of their proofs:
$P \wedge Q$ is the cartesian product of P and Q,
$P \to Q$ is the set of functions with domain P and codomain Q.

To deal with quantifiers, our type theory becomes dependent:
a proof of $\exists x, P(x)$ consists of a witness t and a proof of $P(t)$.
P is a type that depends on another object t.

A bit of dependent type theory

We identify logical propositions with the set (type) of their proofs:
$P \wedge Q$ is the cartesian product of P and Q,
$P \rightarrow Q$ is the set of functions with domain P and codomain Q.

To deal with quantifiers, our type theory becomes dependent:
a proof of $\exists x, P(x)$ consists of a witness t and a proof of $P(t)$.
P is a type that depends on another object t.

So we identify $\exists x, P(x)$ with the disjoint sum $\bigsqcup_x P(x)$.

A bit of dependent type theory

To express mathematical statements, we also need equality.
The trick we use is that equality is the smallest reflexive relation:
every element of the identity type $a = b$ is actually $\mathsf{refl}\, a : a = a$.

A bit of dependent type theory

To express mathematical statements, we also need equality.
The trick we use is that equality is the smallest reflexive relation:
every element of the identity type $a = b$ is actually $\mathsf{refl}\, a : a = a$.

Note that homotopy type theory has a more subtle notion of equality:
the above summary is not outright wrong,
but needs to be phrased more carefully.

Computing with dependent type theory

What are a and b when we describe $a = b$?
They are not just strings of symbols: if a is "$1 + 1$" and b is "2",
then those strings of symbols are distinct,
but clearly we want to be able to prove $1 + 1 = 2$.

Computing with dependent type theory

What are a and b when we describe $a = b$?
They are not just strings of symbols: if a is "$1 + 1$" and b is "2", then those strings of symbols are distinct,
but clearly we want to be able to prove $1 + 1 = 2$.

Here the computational interpretation comes back:
$1 + 1 = 2$ because a program that evaluates $1 + 1$ returns 2.

Computing with dependent type theory

What are a and b when we describe $a = b$?
They are not just strings of symbols: if a is "$1 + 1$" and b is "2", then those strings of symbols are distinct, but clearly we want to be able to prove $1 + 1 = 2$.

Here the computational interpretation comes back:
$1 + 1 = 2$ because a program that evaluates $1 + 1$ returns 2.

To capture this notion, we introduce a second equality relation: *definitional equality* (defeq).

Definitional equality

Defeq says $a \equiv b$ whenever the computation a has the same result ("normal form") as b.

Definitional equality

Defeq says $a \equiv b$ whenever the computation a has the same result ("normal form") as b.
Whenever $a \equiv b$, we can substitute occurrences of a for b or vice versa in our proofs that $t : T$.
For example, we substitute $1 + 1 \equiv 2$ in $\mathsf{refl}\, 2 : 2 = 2$ to prove $\mathsf{refl}\, 2 : 1 + 1 = 2$.

Definitional equality

Defeq says $a \equiv b$ whenever the computation a
has the same result ("normal form") as b.
Whenever $a \equiv b$, we can substitute occurrences of a for b or vice versa in our proofs that $t : T$.
For example, we substitute $1 + 1 \equiv 2$ in $\mathsf{refl}\, 2 : 2 = 2$ to prove $\mathsf{refl}\, 2 : 1 + 1 = 2$.

In (most!) theorem provers, this substitution is automatic.

Computing with definitional equality

For each primitive operation, we introduce computation rules.
For example, fst $(p, q) \equiv p$.

Defining new operations consists of two steps, giving their type and giving their definitional equalities:

```
(a : ℕ) + (b : ℕ) : ℕ
 0      +  b      ≡ b
(suc a) +  b      ≡ suc (a + b)
```

Definitional and propositional equality

Crucially, we keep the type $a = b$. $\equiv$ is not a type so it cannot be passed around as a hypothesis.

Definitional and propositional equality

Crucially, we keep the type $a = b$. $\equiv$ is not a type so it cannot be passed around as a hypothesis.

The issue lies not just in notation:
checking defeq is possible because of computation.
Checking an arbitrary set of equations implies equality of two strings ("the Word Problem") is not possible automatically.

Crucially, we keep the type $a = b$. $\equiv$ is not a type so it cannot be passed around as a hypothesis.

The issue lies not just in notation:
checking defeq is possible because of computation.
Checking an arbitrary set of equations implies equality of two strings ("the Word Problem") is not possible automatically.

Extensional type theory (e.g. Nuprl) has richer judgmental equalities, in exchange for requiring the user to supply proofs.

Definitional equality and structures

Definitional equality is extremely useful for multiple structures on the same object:
viewing $\mathbb{Z}$ as multiplicative semigroup, as monoid, as ring, ...
The definition of `int.ring` is `int.monoid` extended with some extra fields. A theorem about monoid structure underlying a ring uses `ring.to_monoid`, projecting away those extra fields.
Works perfectly, by ensuring `int.ring.to_monoid ≡ int.monoid`.

Definitional equality and structures

Definitional equality is extremely useful for multiple structures on the same object:
viewing $\mathbb{Z}$ as multiplicative semigroup, as monoid, as ring, ...
The definition of `int.ring` is `int.monoid` extended with some extra fields. A theorem about monoid structure underlying a ring uses `ring.to_monoid`, projecting away those extra fields.
Works perfectly, by ensuring `int.ring.to_monoid ≡ int.monoid`.

The rule to never create new fields in inheritance, "forgetful inheritance", will ensure our hierarchy, including diamond inheritance, works automatically.

Fragility of definitional equality

Definitional equality is sensitive to the form of definitions:
our definition of $+$ satisfies $0 + n \equiv n$,
but proving $n + 0 = n$ requires induction on n.

Fragility of definitional equality

Definitional equality is sensitive to the form of definitions:
our definition of $+$ satisfies $0 + n \equiv n$,
but proving $n + 0 = n$ requires induction on n.

This is not a theoretical inconvenience: let $\{A_i \mid i \in \mathbb{Z}\}$ be a family of groups, with homomorphisms $f_i : A_{i-1} \to A_i$.
Definition: This family is *exact* at A_i if $\operatorname{im} f_i = \ker f_{i+1}$.

Fragility of definitional equality

Definitional equality is sensitive to the form of definitions:
our definition of $+$ satisfies $0 + n \equiv n$,
but proving $n + 0 = n$ requires induction on n.

This is not a theoretical inconvenience: let $\{A_i \mid i \in \mathbb{Z}\}$ be a family of groups, with homomorphisms $f_i : A_{i-1} \to A_i$.
Definition: This family is *exact* at A_i if $\operatorname{im} f_i = \ker f_{i+1}$.
Type error! $\operatorname{im} f_i \subseteq A_i$ but $\ker f_{i+1} \subseteq A_{i+1-1}$.

Fragility of definitional equality

Definitional equality is sensitive to the form of definitions:
our definition of $+$ satisfies $0 + n \equiv n$,
but proving $n + 0 = n$ requires induction on n.

This is not a theoretical inconvenience: let $\{A_i \mid i \in \mathbb{Z}\}$ be a family of groups, with homomorphisms $f_i : A_{i-1} \to A_i$.
Definition: This family is *exact* at A_i if $\operatorname{im} f_i = \ker f_{i+1}$.
Type error! $\operatorname{im} f_i \subseteq A_i$ but $\ker f_{i+1} \subseteq A_{i+1-1}$. $(1 + i) - 1 \equiv i$ but $(i + 1) - 1 = i$.

Fragility of definitional equality

Definitional equality is sensitive to the form of definitions:
our definition of $+$ satisfies $0 + n \equiv n$,
but proving $n + 0 = n$ requires induction on n.

This is not a theoretical inconvenience: let $\{A_i \mid i \in \mathbb{Z}\}$ be a family of groups, with homomorphisms $f_i : A_{i-1} \to A_i$.
Definition: This family is *exact* at A_i if $\operatorname{im} f_i = \ker f_{i+1}$.
Type error! $\operatorname{im} f_i \subseteq A_i$ but $\ker f_{i+1} \subseteq A_{i+1-1}$. $(1 + i) - 1 \equiv i$ but $(i + 1) - 1 = i$.

Lean's mathlib has to be carefully built to avoid defeq issues.

Computational proofs

Proof by direct computation

Now that we have an idea of definitional equality checking through computation, we see that Lean can prove $37 + 5 = 6 * 7$ by evaluating both sides and checking the result matches.

Proof by direct computation

Now that we have an idea of definitional equality checking through computation, we see that Lean can prove $37 + 5 = 6 * 7$ by evaluating both sides and checking the result matches.

Showing all squares in $\mathbb{Z}/4\mathbb{Z}$ are either 0 or 1 uses a more clever technique.
Lean records which propositions are decidable: for which we can tell if they are true or false.

- If $x, y : \mathbb{Z}$ then $x = y$ is decidable.
- If T is a finite type, and P is decidable, then $\forall x : T, P(x)$ is decidable.
- ...

Proof by direct computation

So, to prove a decidable proposition, we run the decision procedure, and if it outputs `true` we succeed.

Can we trust the decision procedure?

Proof by direct computation

So, to prove a decidable proposition, we run the decision procedure, and if it outputs `true` we succeed.
Can we trust the decision procedure?

Yes, because `decidable` consists of a decision procedure plus a proof showing the procedure produces the correct output.

Proof by direct computation

So, to prove a decidable proposition, we run the decision procedure, and if it outputs `true` we succeed.
Can we trust the decision procedure?

Yes, because `decidable` consists of a decision procedure plus a proof showing the procedure produces the correct output.

Proof by reflection: use an algorithm to check the condition, and prove that the condition is true if(f) the algorithm succeeds.

Trusting proofs by computation

Running the computation will prove our theorem if we trust the logic.
And if we trust the compiler turning our algorithm into machine code.

Trusting proofs by computation

Running the computation will prove our theorem if we trust the logic.
And if we trust the compiler turning our algorithm into machine code.
And if we trust the CPU running our machine code.

Trusting proofs by computation

Running the computation will prove our theorem if we trust the logic.
And if we trust the compiler turning our algorithm into machine code.
And if we trust the CPU running our machine code.
And if we trust our eyes to accurately interpret the "Proof succeeded!" message.

Trusting proofs by computation

Running the computation will prove our theorem if we trust the logic.
And if we trust the compiler turning our algorithm into machine code.
And if we trust the CPU running our machine code.
And if we trust our eyes to accurately interpret the “Proof succeeded!” message.

Typically we choose a more practical level of paranoia:
Coq has a complicated and fast evaluator in the proof-checking kernel.
Lean has a simpler but slower kernel.

Trusting proofs by computation

Running the computation will prove our theorem if we trust the logic.
And if we trust the compiler turning our algorithm into machine code.
And if we trust the CPU running our machine code.
And if we trust our eyes to accurately interpret the "Proof succeeded!" message.

Typically we choose a more practical level of paranoia:
Coq has a complicated and fast evaluator in the proof-checking kernel.
Lean has a simpler but slower kernel.

Definitional equality is the force that drives kernel computation.
(Recall that we can check definitional equality by evaluating terms.)

Checking execution traces

Lean has a relatively fast evaluator and a slow kernel.
So we run the algorithm in the evaluator and construct a trace, then use the kernel to verify the trace corresponds to a successful execution.

Checking execution traces

Lean has a relatively fast evaluator and a slow kernel.
So we run the algorithm in the evaluator and construct a trace, then use the kernel to verify the trace corresponds to a successful execution.

Compare this to the *Elfstedentocht*, the long distance ice skating race where competitors race to visit all eleven cities in Frisia.
Participants collect a stamp at each city, and the judge verifies the successful completion of the tour by checking the book is fully stamped.

Checking execution traces

This approach is common in Lean, for example in the `norm_num` tactic:

```
example : 37 | 999999 :=
by dec_trivial -- times out
example : 37 | 999999 :=
by norm_num -- finishes almost instantly
```

Natural numbers in Lean are unary Peano numbers by definition, but `norm_num` can use much more efficient binary numbers.

Taking execution traces further

We don't need to write the verification algorithm in Lean either: Sage already has a fast algorithm for many computations.

Taking execution traces further

We don't need to write the verification algorithm in Lean either: Sage already has a fast algorithm for many computations.

We end up with a multiple-layered construction:

- The Lean kernel verifies the proof that...
- a Lean tactic generated from...
- a Sage computation that calls...
- Pari/GP implementations.

Despite the many programs, we still only need to trust the kernel.

Proof certificates

Sage does not have a notion of predicates like Lean does.
So we have to communicate in lower-level concepts: numbers, polynomials, matrices, ...
In other words: Sage sends proof certificates to Lean.

Proof certificates

Sage does not have a notion of predicates like Lean does.
So we have to communicate in lower-level concepts: numbers, polynomials, matrices, ...
In other words: Sage sends proof certificates to Lean.

For example, performing LU decomposition in Sage can prove in Lean a matrix M is of full rank.
Lean only needs to check $LU = M$ and that L and U have no zeroes on the diagonal.

Proof certificates

Sage does not have a notion of predicates like Lean does.
So we have to communicate in lower-level concepts: numbers, polynomials, matrices, ...
In other words: Sage sends proof certificates to Lean.

For example, performing LU decomposition in Sage can prove in Lean a matrix M is of full rank.
Lean only needs to check $LU = M$ and that L and U have no zeroes on the diagonal.

To show a Diophantine equation has no solutions, a certificate can be n such that there are no solutions mod n.
Lean can "quickly" check finitely many solutions mod n.

Proof certificates

Proof certificates we used for the class number:

- For finitely generated ideals $I = \langle s \rangle$, $J = \langle t \rangle$, certify $I \subseteq J$ by writing each $x \in s$ as a linear combination of t.
- For an ideal I, prove it is not principal by computing the ideal norm, which is not a norm of an element $x \in I$.
- Show 2 is not a prime ideal in $\mathbb{Z}[\sqrt{d}]$ by computing its square root ($d \in \{2, 3\} \mod 4$).

Proof certificates

Proof certificates we used for the class number:

- For finitely generated ideals $I = \langle s \rangle$, $J = \langle t \rangle$, certify $I \subseteq J$ by writing each $x \in s$ as a linear combination of t.
- For an ideal I, prove it is not principal by computing the ideal norm, which is not a norm of an element $x \in I$.
- Show 2 is not a prime ideal in $\mathbb{Z}[\sqrt{d}]$ by computing its square root ($d \in \{2, 3\} \mod 4$).
- ... can you think of others?

Other ways to compute

Simplifying and normalizing

We don't need the one computational interpretation to compute: the Lean simplifier computes symbolically, at the level of expressions. It repeatedly tries to rewrite using all lemmas it knows, until no more lemma applies: the expression is in simp-normal form.

Simplifying and normalizing

We don't need the one computational interpretation to compute: the Lean simplifier computes symbolically, at the level of expressions. It repeatedly tries to rewrite using all lemmas it knows, until no more lemma applies: the expression is in simp-normal form.

The simplifier can easily be made to compute: make every definitional equality a simp lemma.

Simplifying and normalizing

We don't need the one computational interpretation to compute: the Lean simplifier computes symbolically, at the level of expressions. It repeatedly tries to rewrite using all lemmas it knows, until no more lemma applies: the expression is in simp-normal form.

The simplifier can easily be made to compute: make every definitional equality a simp lemma.

Mathlib includes `norm_num` which extends the simplifier with procedures for numeric computations: $37 + 5, 6 * 7, \ldots$.
`norm_num` is itself extensible: I wrote a procedure for $\mathbb{Z}[\sqrt{d}]$.

Instance synthesis

Lean infers algebraic structures using typeclass instances:
to show $\mathbb{Z}^\times$ is a group under multiplication, consider each `group` instance in turn, until we find one whose type matches `group (units ℤ)`.

Instance synthesis

Lean infers algebraic structures using typeclass instances: to show $\mathbb{Z}^\times$ is a group under multiplication, consider each `group` instance in turn, until we find one whose type matches `group (units ℤ)`.

Instances can depend on other instances: to show `group (units ℤ)`, we apply `units.group : ∀ M, [monoid M] → group (units M)` and it remains to show `monoid ℤ`.

Instance synthesis

Instance synthesis is performed recursively (when instances depend on other instances),
with multiple possible rules (multiple instances of the same class),
with backtracking (when the dependent instance could not be found).

Conclusion: we can program in Prolog using instance synthesis.
Or maybe create an awkward proof search system.

Instance synthesis

Instance synthesis is performed recursively (when instances depend on other instances),
with multiple possible rules (multiple instances of the same class),
with backtracking (when the dependent instance could not be found).

Conclusion: we can program in Prolog using instance synthesis.
Or maybe create an awkward proof search system.
We used this to evaluate `algebra_map ℤ ℚ (-5) = (-5)` when including $\mathbb{Z}[\sqrt{-5}]$ into $\mathbb{Q}(\sqrt{-5})$.

Conclusion

Conclusion

When formalizing maths on the computer, computations might become much harder than they first appear: $n + 0$ is not n, A_{i+1-1} is not A_i, we have to explicitly name every rewrite rule, etc.

On the other hand, computations occur all the time automatically, on purpose (with MLTT or computer algebra) or more accidentally (with the simplifier).

Clinging to one notion of computation causes tons of frustration. Instead we should bridge the gaps flexibly.