Talking About Mathematics in a Programming Language

Donnacha Oisín Kidney

November 21, 2018

Formalised Mathematics

Programming is Proving

A Polynomial Solver

Formalised Mathematics

Formalised mathematics is an attempt to find a core set of axioms and consistent rules from which all mathematical truths can be derived.

Hilbert's program: "dispose of the foundational questions in mathematics once and for all."

We're trying to address the foundational crisis of mathematics from the early 20th century

But didn't Hilbert's program fail?

Lawrence C Paulson. The Future of Formalised Mathematics, 2016

But didn't Hilbert's program fail?

A. N. Whitehead and B. Russell.
Principia Mathematica. Vol. I.
1910 p. 379

Lawrence C Paulson. The Future of Formalised Mathematics, 2016

This is the citation for Whitehead and Russell's proof of the fact that 1+1=2.

It was the first real crack at the foundational problem.

If it was *that tedious* to figure out something so simple, what chance do we have of more complex proofs—*exactly* the kinds of things we *want* to formalise?

But didn't Hilbert's program fail?

A. N. Whitehead and B. Russell.
Principia Mathematica. Vol. I.
1910 p. 379

Gödel showed that universal
formal systems are incomplete

Lawrence C Paulson. The Future of Formalised Mathematics, 2016

Any "good enough" system (formal, effectively axiomatized, consistent) isn't going to be complete! There are statements we can't prove, in other words.

But didn't Hilbert's program fail?

A. N. Whitehead and B. Russell.
Principia Mathematica. Vol. I.
1910 p. 379

Gödel showed that universal
formal systems are incomplete

Church Proved the
Entscheidungsproblem is
unsolvable!

Lawrence C Paulson. The Future of Formalised Mathematics, 2016

Church showed that the decision problem is undecidable in general. This is really the final nail in the coffin for Hilbert's program.

There is no program which, given a decision problem, can solve it automatically.

But didn't Hilbert's program fail?

A. N. Whitehead and B. Russell.
Principia Mathematica. Vol. I.
1910 p. 379

Formal systems have improved

Gödel showed that universal formal systems are incomplete

Church Proved the Entscheidungsproblem is unsolvable!

Lawrence C Paulson. The Future of Formalised Mathematics, 2016

We have much better formalisms now.

Although they're still tedious, they're nowhere near the verbosity of principia.

But didn't Hilbert's program fail?

A. N. Whitehead and B. Russell.
Principia Mathematica. Vol. I.
1910 p. 379

Formal systems have improved

Gödel showed that universal formal systems are incomplete

We don't need universal systems!

Church Proved the Entscheidungsproblem is unsolvable!

Lawrence C Paulson. The Future of Formalised Mathematics, 2016

Turns out we don't need all of the power of the "good enough".
We get very far with just *slightly* less.

But didn't Hilbert's program fail?

A. N. Whitehead and B. Russell.
Principia Mathematica. Vol. I.
1910 p. 379

Formal systems have improved

Gödel showed that universal formal systems are incomplete

We don't need universal systems!

Church Proved the Entscheidungsproblem is unsolvable!

We don't automate everything

Lawrence C Paulson. The Future of Formalised Mathematics, 2016

Again, we give up the full goal, but we can get *close*: automation and decidability assist our construction of proofs.

What Does our System Look Like?

So what have we given up since Hilbert's program?

What Does our System Look Like?

Constructivist

To show something exists, you have to *construct* it.

We're going to work in a constructivist setting.
This means that to prove something exists you actually have to construct the thing.

In that sense, it's proof "by example", although you can indeed prove things with a "forall" quantifier before them, as we'll see in a minute.

What Does our System Look Like?

Constructivist

To show something exists, you have to *construct* it.

Law of the Excluded Middle ×

$$p \vee \neg p$$

Proof By Contradiction ×

$$\neg\neg p \to p$$

Principle of Explosion ✓

$$\neg p \wedge p \to q$$

In practice, it means you basically have to throw out a couple familiar big-ticket axioms like proof by contradiction and the law of the excluded middle.

There's some confusion floating around about what exactly you lose when you discard these axioms—you keep the principle of explosion, for instance. Furthermore, within the constructivist framework, you can model classical logic, if all of your proofs are just double negatives. In other words, you can prove the double negation of classical logic statements.

It's important to note that we *want* to give up these two axioms: they don't make sense in a constructivist context. The law of the excluded middle is effectively the Entscheidungsproblem—given a statement, it will provide a proof of its truth, or of its negation.

There are other axioms we may want to have, but that we lose in the particular framework we're going to choose. The design space for these theories is quite large, and especially around the area of "equality" people haven't nailed down the perfect base yet.

What Does our System Look Like?

Constructivist

To show something exists, you have to *construct* it.

Law of the Excluded Middle ×

$$p \vee \neg p$$

Proof By Contradiction ×

$$\neg\neg p \to p$$

Principle of Explosion ✓

$$\neg p \wedge p \to q$$

***Partially* Automated**

While our system can't solve arbitrary problems, we can write certain solvers for specific domains—that's the purpose of this project.

It's similar to the goal of machine learning and AI today, in this sense. While we probably can't build something to solve *everything*, we can build a system that assists us in solving "everything".

What do Programming Languages Have to do with it?

For one thing, it would be nice if the proofs we wrote were *checkable* by a computer.

Kenneth Appel and Wolfgang Haken. The Solution of the Four-Color-Map Problem.

***Scientific American*, 237(4):108–121, 1977**

In particular, it would be great if we could check *computer-assisted* proofs.

The famous example is the four-colour map theorem.

First major mathematical proof which relied heavily on computer assistance.

The problem is thus: can you colour a map, with only four colours, so that every border has two different colours?

The proof effectively relied on checking a large number of different cases—a computer program was used to check each one.

A computer is a perfect candidate for checking computer programs.

Kenneth Appel and Wolfgang Haken. The Solution of the Four-Color-Map Problem.

***Scientific American*, 237(4):108–121, 1977**

Did contain bugs!

Unfortunately, this is still difficult to do

The formalized version of the four-colour theorem came out a full 29 years later!

Kenneth Appel and Wolfgang Haken. The Solution of the Four-Color-Map Problem.

***Scientific American*, 237(4):108–121, 1977**

Did contain bugs! Georges Gonthier. Formal Proof—The Four-Color Theorem.

***Notices of the AMS*, 55(11):12, 2008**

Programming Proofs

Per Martin-Löf. *Intuitionistic Type Theory*.

Padua, June 1980

Beyond just checking proofs, or automating proofs, it turns out that programming languages—and in particular *type systems*—are pretty decent formalisms for writing proofs.

Why Would a Programmer Want to Use this Language?

Suppose I convince you that this formalism is good enough to do maths—is it good enough to do *programming*? Surely the two aims are orthogonal? While most languages for "proving" these days are indeed not suitable for general-purpose programming, ideas from them are leaking into mainstream languages.

And, of course, Idris is a general-purpose language which can prove as good as anything!

Why Would a Programmer Want to Use this Language?

- *Prove* things about code

```
assert(list(reversed([1,2,3])) == [3,2,1])
```

vs

reverse-involution : ∀ *xs* → reverse (reverse *xs*) ≡ *xs*

Not just test!

Why Would a Programmer Want to Use this Language?

- *Prove* things about code
- Use ideas and concepts from maths—why reinvent them?

Mathematics and formal language has existed for thousands of years; programming has existed for only 60!

Why Would a Programmer Want to Use this Language?

- *Prove* things about code
- Use ideas and concepts from maths—why reinvent them?
- Provide coherent *justification* for language features

Programming is Proving

The Curry-Howard Correspondence

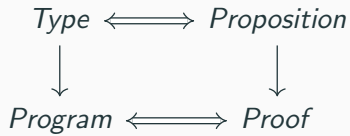
Philip Wadler. Propositions As Types.

***Commun. ACM*, 58(12):75–84, November 2015**

To use a programming language as a proof language, we'll need to see how programming constructs map on to constructs in logic.

This "mapping" is known as the curry-howard correspondence (or isomorphism).

The Curry-Howard Correspondence



Philip Wadler. Propositions As Types.

***Commun. ACM*, 58(12):75–84, November 2015**

Here's the high-level overview.

"Program" here just means anything with a type, basically. In $x = 2$, x is a program, and 2 is a program, and so on. Functions are programs, etc. We could have also said "value" or something, but program is the word used in the literature.

Types are Propositions

Types are (usually):

- Int
- String
- ...

How are these propositions?

Propositions are things like "there are infinite primes", etc. Int certainly doesn't *look* like a proposition.

Existential Proofs

We use a trick to translate: put a "there exists" before the type. Remember, we're constructivist!

Existential Proofs

So when you see:

$$x : \mathbb{N}$$

Existential Proofs

So when you see:

$x : \mathbb{N}$

Think:

$\exists.\mathbb{N}$

Existential Proofs

So when you see:	Think:
$x : \mathbb{N}$	$\exists.\mathbb{N}$

NB

We'll see a more powerful and precise version of $\exists$ later.

Existential Proofs

So when you see:	Think:
$x : \mathbb{N}$	$\exists.\mathbb{N}$

NB

We'll see a more powerful and precise version of $\exists$ later.

Proof is "by example":

Existential Proofs

So when you see:	Think:
$x : \mathbb{N}$	$\exists.\mathbb{N}$

NB

We'll see a more powerful and precise version of $\exists$ later.

Proof is "by example":

$$x = 1$$

Programs are Proofs

Let's start working with a function as if it were a proof. The function we'll choose gets the first element from a list. It's commonly called "head" in functional programming.

Programs are Proofs

```
>>> head [1,2,3]
1
```

Programs are Proofs

```
>>> head [1,2,3]
1
```

Here's the type:

$$\mathsf{head} : \{A : \mathsf{Set}\} \to \mathsf{List}\ A \to A$$

head is what would be called a "generic" function in languages like Java. In other words, the type A is not specified in the implementation of the function.

Basic Agda Syntax

Equivalent in other languages:

Haskell

```
head :: [a] -> a
```

Swift

```
func head<A>(xs : [A]) -> A {
```

Basic Agda Syntax

Equivalent in other languages:

Haskell
```
head :: [a] -> a
```

Swift
```
func head<A>(xs : [A]) -> A {
```

$$\mathsf{head} : \{A : \mathsf{Set}\} \to \mathsf{List}\ A \to A$$

In Agda, you must supply the type to the function: the curly brackets mean the argument is implicit.

Basic Agda Syntax

Equivalent in other languages:

Haskell
```
head :: [a] -> a
```

Swift
```
func head<A>(xs : [A]) -> A {
```

$\mathsf{head} : \{A : \mathsf{Set}\} \to \mathsf{List}\ A \to A$ "Takes a list of things, and returns one of those things".

The Proposition is False!

```
>>> head []
error "head: empty list"
```

head isn't defined on the empty list, so the function *doesn't* exist. In other words, its type is a false proposition.

The Proposition is False!

```
>>> head []
error "head: empty list"
```

$$\mathsf{head} : \{A : \mathsf{Set}\} \to \mathsf{List}\ A \to A$$

The Proposition is False!

```
>>> head []
error "head: empty list"
```

$\mathsf{head} : \{A : \mathsf{Set}\} \to \mathsf{List}\ A \to A$

False

If Agda is correct (as a formal logic):

If Agda is correct (as a formal logic):

We shouldn't be able to prove this using Agda

If Agda is correct (as a formal logic):

We shouldn't be able write this function in Agda

But Let's Try Anyway!

```
length : {A : Set} → List A → ℕ
length [] = 0
length (x :: xs) = 1 + length xs
```

For lists, we also have two patterns: the empty list, and the head element followed by the rest of the list.

But Let's Try Anyway!

$$\mathsf{head}\ (x :: xs) = x$$

Here's a candidate definition for head.

Remember, we shouldn't be able to write it, so if this definition is accepted by Agda, then Agda isn't correct.

So how do we disallow it?

But Let's Try Anyway!

$$\mathsf{head}\ (x :: xs) = x$$

Rule 1

No partial functions

We disallow it because it doesn't match all patterns.

Agda will only accept functions which are defined for *all* of their inputs.

But Let's Try Anyway!

$$\text{head}\ (x :: xs) = x$$

Rule 1

No partial functions

So we need something to write for the second clause, the empty list.

It seems like we can't, but people familiar with Haskell may have spotted a way to do it.

But Let's Try Anyway!

$$\text{head}\ (x :: xs) = x$$

$$\text{head}\ [] = \text{head}\ []$$

Rule 1

No partial functions

In Haskell, a definition like this is perfectly acceptable: it's just recursive. Here, though, we've obviously proved a falsehood, so we need some way to disallow it.

If we were to run this program, it would just loop forever: disallowing that turns out to be enough to keep the logic consistent.

But Let's Try Anyway!

$$\text{head}\ (x :: xs) = x$$

$$\text{head}\ [] = \text{head}\ []$$

Rule 1

No partial functions

Rule 2

All programs are total

Bear in mind that even if we don't obey the rules the program can still be a valid proof—we just have to run it first.

Obeying these rules ensures that the proofs are valid if they typecheck.

What does "total" mean? Well, it's something like terminating...

Turing Completeness

Have we just thrown out Turing completeness?

If we're not allowed infinite loops, then we're not turing complete, right?

Well, no...

Turing Completeness

The dual to termination is *productivity*

Consider a program like a webserver, or a clock on your computer.

Neither of these things should "terminate", but we don't want them to contain infinite loops, either.

The property we want them to posses is called *productivity*: they always produce another step of computation in finite time, even if there are infinitely many steps.

Agda can check for productivity, too.

Turing Completeness

The dual to termination is *productivity*

```
record Stream (A : Set) : Set where
  coinductive
  field
    head : A
    tail : Stream A
```


The definition of this type (and the coinductive keyword) change the behaviour of the termination-checker. We can now construct infinite structures.

Using types like this, we can (for instance), simulate a turing machine, or write a lambda-calculus interpreter.

What we *can't* do is lie about the types of those programs: we won't be able to write a function like "run" which produces a finite result. We could write a function that runs for some finite number of steps, and produces a finite result, or a function which produces an infinite result, though.

The limitation is that there will be terminating programs that we can't prove are terminating. In this case, we can default to saying "maybe terminating".

Turing Completeness

The dual to termination is *productivity*

```
record Stream (A : Set) : Set where
  coinductive
  field
    head : A
    tail : Stream A
```

You can write terminating and non-terminating programs: *you just have to say so*

Enough Restrictions!

That's a lot of things we *can't* prove.

How about something that we can?

How about the converse?

Can we *prove* that head doesn't exist?

After all, all we have so far is "proof by trying really hard".

Falsehood

First we'll need a notion of "False". Often it's said that you can't prove negatives in dependently typed programming: not true! We'll use the principle of explosion: "A false thing is one that can be used to prove anything".

Falsehood

Principle of Explosion

"Ex falso quodlibet"

If you stand for nothing, you'll fall for anything.

Falsehood

$$\neg : \forall \{\ell\} \to \mathsf{Set}\ \ell \to \mathsf{Set}\ \_$$
$$\neg\ A = A \to \{B : \mathsf{Set}\} \to B$$

Principle of Explosion

"Ex falso quodlibet"

If you stand for nothing, you'll fall for anything.

```
head-doesn't-exist : ¬ ({A : Set} → List A → A)
head-doesn't-exist head = head []
```

Here's how the proof works: for falsehood, we need to prove the supplied proposition, no matter what it is. If head exists, this is no problem! Just get the head of a list of proofs of the proposition, which can be empty.

Proofs are Programs

So that was an attempt to show that programs are proofs, if you look at them funny.

Now let's go the other direction: let's see what some constructs in proof theory look like when translated into programming.

Proofs are Programs

Types/Propositions are *sets*

```
data Bool : Set where
  true  : Bool
  false : Bool
```

Proofs are Programs

Types/Propositions are *sets*

```
data Bool : Set where
  true  : Bool
  false : Bool
```

Inhabited by *proofs*

Bool	Proposition
true, false	Proof

Implication

Just a function arrow

Implication

$A \to B$

Implication

$A \to B$

A implies B

Implication

$A \to B$

A implies B

Constructivist/Intuitionistic

Give me a proof of a, I'll give you a proof of b

Booleans?

We *don't* use bools to express truth and falsehood.

Bool is just a set with two values: nothing "true" or "false" about either of them!

This is the difference between using a computer to do maths and *doing maths in a programming language*

Booleans?

```
data ⊥ : Set where
```

Contradiction

Falsehood (contradiction) is the proposition with no proofs.

It's equivalent to what we had previously.

Booleans?

```
data ⊥ : Set where                    Contradiction

  law-of-non-contradiction : ∀ {a} {A : Set a} → ¬ A → A → ⊥
  law-of-non-contradiction f x = f x
```

In fact, we can convert from what we had previously

Booleans?

```
data ⊥ : Set where                    Contradiction

  law-of-non-contradiction : ∀ {a} {A : Set a} → ¬ A → A → ⊥
  law-of-non-contradiction f x = f x

  not-false : ¬ ⊥
  not-false ()
```


And *to* what we had previously.

Here, we use an impossible pattern.

Booleans?

```
data ⊥ : Set where
```

Contradiction

```
data ⊤ : Set where
  tt : ⊤
```

Tautology

Tautology is kind of the “boring” type.

Dependent Types

Dependent Types

Everything so far has been non-dependent

In other words, lots of modern languages support it. (Haskell)

Dependent Types

Everything so far has been non-dependent

Proving things using this bare-bones toolbox is difficult (though possible)

The proof that head doesn't exists, for instance, could be written in vanilla Haskell.

It's difficult to prove more complex statements using this pretty bare-bones toolbox, though, so we're going to introduce some extra handy features.

Dependent Types

Everything so far has been non-dependent

Proving things using this bare-bones toolbox is difficult (though possible)

To make things easier, we're going to add some things to our types

Per Martin-Löf. *Intuitionistic Type Theory.*

Padua, June 1980

Russell's Paradox

"The Set of all Sets which do not contain themselves"

One of the things that tripped up early logicians is Russell's paradox.

Russell's Paradox

"The Set of all Sets which do not contain themselves"

Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*.

PhD Thesis, PhD thesis, Université Paris VII, 1972

In type theory, it's called Girard's paradox.

Russell's Paradox

"The Set of all Sets which do not contain themselves"

Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur.*

PhD Thesis, PhD thesis, Université Paris VII, 1972

```
not : Bool → Bool
not true  = false
not false = true
```


Remember that types are defined as sets. Bool is a set, int is a set, etc. Values have types, and types are sets. Bool -> Bool, for instance, is the type of not. Bool -> Bool is a Set.

Russell's Paradox

"The Set of all Sets which do not contain themselves"

Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur.*

PhD Thesis, PhD thesis, Université Paris VII, 1972

```
not : Bool → Bool
not true  = false
not false = true
```

```
¬_ : Set → Set
¬ A = A → ⊥
```


However, we've already broken this boundary: The type of negation was Set -> Set. Is Set -> Set a type?

There are other examples: List is a function of type Set -> Set.

Fine. So "neg" has the type Set -> Set. Here's the question, though: is Set -> Set a Set?

We've allowed a set to be a member of itself, opening the door to russell's paradox.

There are a number of different ways to avoid it; in Agda, all types are "Set"s. Set -> Set, though, is a Set1. Set1 -> Set1 is a Set2. And so on.

Function Extensionality

```
postulate function-extensionality
          : {A B : Set} {f g : A → B}
          → (∀ x → f x ≡ g x)
          → f ≡ g
```

In contrast to LEM etc., this is a law that it would be nice to have.
In fact, certain type theories *do* have it.
But all of these small rules around equality are very much in flux: if you grant constructor Injectivity (a very similar rule to this one), you can prove a contradiction!

Other systems, such as Homotopy type theory, observational equality, and so on, have very different ideas about equality.

A Polynomial Solver

Project I've been working on for the past few months.
As mentioned previously, while we can't automate all of mathematics, there are some restricted areas where we can.
This area is that of equivalence relations between rings of commutative polynomials.
Doing this by hand usually consists of long chains of rewrites, of the style "apply commutativity of $+$, then associativity of $+$, then at this position apply distributivity of $*$ over $+$" and so on, when really the programmer wants to say "rearrange the expression into this form, checking it's correct".

Monoids

Before we see the full polynomial solver, we're going to look at a simpler algebra (monoids) to illustrate the technique the solver is going to use.

Monoids

Monoid

A monoid is a set equipped with a binary operation, $\bullet$, and a distinguished element ϵ, such that the following equations hold:

$$x \bullet (y \bullet z) = (x \bullet y) \bullet z \qquad \text{(Associativity)}$$

$$x \bullet \epsilon = x \qquad \text{(Left Identity)}$$

$$\epsilon \bullet x = x \qquad \text{(Right Identity)}$$

Addition and multiplication (with 0 and 1 being the respective identity elements) are perhaps the most obvious instances of the algebra. In computer science, monoids have proved a useful abstraction for formalizing concurrency (in a sense, an associative operator is one which can be evaluated in any order).

A Boring Proof

$$\begin{aligned}&\mathsf{ident} : \forall\ w\ x\ y\ z \\ &\quad \to ((w \bullet \varepsilon) \bullet (x \bullet y)) \bullet z \approx (w \bullet x) \bullet (y \bullet z)\end{aligned}$$

And this is the kind of proposition we want to automatically prove.

We have two expressions, each with some free variables, and we want to construct a proof that they're equivalent, relying only on the monoid laws.

To a human, the fact that the identity holds may well be obvious: $\bullet$ is associative, so we can scrub out all the parentheses, and ε is the identity element, so scrub it out too. After that, both sides are equal, so voilà!

A Boring Proof

```
ident : ∀ w x y z
  → ((w • ε) • (x • y)) • z ≈ (w • x) • (y • z)


ident w x y z =
  begin
    ((w • ε) • (x • y)) • z
  ≈⟨ assoc (w • ε) (x • y) z ⟩
    (w • ε) • ((x • y) • z)
  ≈⟨ identityʳ w ⟨ •-cong ⟩ assoc x y z ⟩
    w • (x • (y • z))
  ≈⟨ sym (assoc w x (y • z)) ⟩
    (w • x) • (y • z)
  ■
```


Unfortunately, the proof is tedious. We have to specify every rewrite.

The syntax is designed to mimic that of a handwritten proof: line 3 is the expression on the left-hand side of ≈ in the type, and line 9 the right-hand-side. In between, the expression is repeatedly rewritten into equivalent forms, with justification provided inside the angle brackets. For instance, to translate the expression from the form on line 3 to that on line 5, the associative property of • is used on line 4.

Interestingly, this syntax isn't built-in: it's defined in the standard library.

What we're *actually* doing here is constructing a relation between the left-hand-side expression and the right, using composition of smaller relations. Each of the "smaller" relations is some monoid law. And, of course, each of these relations is an equivalence relation, so it obeys the usual laws (composition is transitivity, and we can see symmetry in the last composition there)

Automation

So our goal is obvious: write a procedure (or function or whatever) which automates the construction of these relations.

Automation

Goals

Automation

Goals

Decidable Should be total, and terminating.

Since we're writing this in a total language, this goal is somewhat thrust upon us.

Nonetheless, a nontotal solver isn't really a solver at all!

This will be something we have to prove, alongside everything else.

Automation

Goals

Decidable Should be total, and terminating.

General Should work in as many settings as possible.

While our solver will indeed deal only with a subset of propositions, we want that subset to be as large as possible. Therefore, we will constrain the domain as *little* as possible.

Automation

Goals

Decidable	Should be total, and terminating.
General	Should work in as many settings as possible.
Efficient	Should actually (as well as theoretically) terminate.

The solver will run at compile-time, so the performance isn't *super* important, but ideally we'll stay away from any nastier complexity classes.

Automation

Goals

Decidable Should be total, and terminating.

General Should work in as many settings as possible.

Efficient Should actually (as well as theoretically) terminate.

Approaches

Automation

Goals

Decidable Should be total, and terminating.

General Should work in as many settings as possible.

Efficient Should actually (as well as theoretically) terminate.

Approaches

Presburger Arithmetic Decidable, first-order theory of natural numbers.

Presburger arithmetic is a subset of arithmetic which is consistent, complete, and decidable.

However, it is *very* constrained.

Its running-time is also doubly-exponential.

Automation

Goals

Decidable Should be total, and terminating.

General Should work in as many settings as possible.

Efficient Should actually (as well as theoretically) terminate.

Approaches

Presburger Arithmetic Decidable, first-order theory of natural numbers.

External solvers (Z3, etc) We don't trust them!

While external solvers are fast and powerful, they're not verified. The objective here is to keep within the bounds of the formal system.

Automation

Goals

Decidable Should be total, and terminating.

General Should work in as many settings as possible.

Efficient Should actually (as well as theoretically) terminate.

Approaches

Presburger Arithmetic Decidable, first-order theory of natural numbers.

External solvers (Z3, etc) We don't trust them!

Canonical forms Our approach.

Canonical Forms

```
infixr 5 _::_
data List (i : ℕ) : Set where
  [] : List i
  _::_ : Fin i → List i → List i
```

This approach basically consists of converting each side of the equation into a canonical form, and checking those two forms equal.

It's not complete, but it's pretty good.

We do need to prove that the conversion preserves homomorphism, and so on, which we do later.

Not every algebra has a canonical form: monoids do, though, and it's the simple list.

Canonical Forms

```
infixr 5 _::_
data List (i : ℕ) : Set where
  [] : List i
  _::_ : Fin i → List i → List i


infixr 5 _++_
_++_ : ∀ {i} → List i → List i → List i
[] ++ ys = ys
(x :: xs) ++ ys = x :: xs ++ ys
```

We're going to treat this type like an AST for a simple "language of lists". This language supports two functions: the empty list, and concatenation.

Canonical Forms

```
infixr 5 _::_
data List (i : ℕ) : Set where
  [] : List i
  _::_ : Fin i → List i → List i


infixr 5 _⧺_
_⧺_ : ∀ {i} → List i → List i → List i
[] ⧺ ys = ys
(x :: xs) ⧺ ys = x :: xs ⧺ ys


_μ_ : ∀ {i} → List i → Vec Carrier i → Carrier
[] μ ρ = ε
(x :: xs) μ ρ = lookup x ρ • xs μ ρ
```


The type itself parameterized by the number of variables it contains. Users can refer to variables by their index.

And we can interpret this language with values for each variable supplied in a vector:

```
obvious
  : (List 4 ∋
  ((η # 0 ++ []) ++ (η # 1 ++ η # 2)) ++ η # 3)
  ≡ (η # 0 ++ η # 1) ++ (η # 2 ++ η # 3)
obvious = ≡.refl
```

Compare this language to the language of monoid expressions that Figure??? uses: both have identity elements and a binary operator, and both refer to variables. Our language of lists, however, has one significant advantage: the monoid operations don't depend on the contents of the lists, only the structure. In other words, an expression in the language of lists will reduce to a flat list even if it has elements which are abstract variables. As a result, the identity from Figure is *definitionally* true when written in the language of lists

Extracting Evidence

```
data Expr (i : ℕ) : Set c where
  _⊕_ : Expr i → Expr i → Expr i
  e   : Expr i
  ν_  : Fin i → Expr i
```

While this is beginning to look like a solver, it's still not entirely clear how we're going to join up the pieces. The first step is to get a concrete representation of expressions which we can manipulate and pattern-match on.

Extracting Evidence

```
data Expr (i : ℕ) : Set c where
  _⊕_ : Expr i → Expr i → Expr i
  e   : Expr i
  v_  : Fin i → Expr i
```

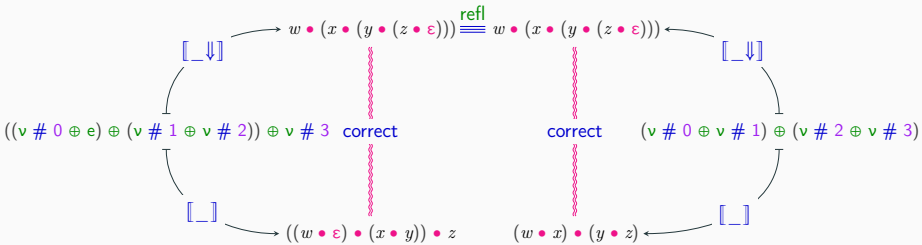
This can be converted to an expression (evaluated) in one of two ways: the straightforward way

Extracting Evidence

```
data Expr (i : ℕ) : Set c where
  _⊕_ : Expr i → Expr i → Expr i
  e   : Expr i
  ν_  : Fin i → Expr i
```

```
⟦_⟧ : ∀ {i} → Expr i → Vec Carrier i → Carrier
⟦ x ⊕ y ⟧ ρ = ⟦ x ⟧ ρ • ⟦ y ⟧ ρ
⟦ e ⟧ ρ     = ε
⟦ ν i ⟧ ρ   = lookup i ρ
```

This can be converted to an expression (evaluated) in one of two ways: the straightforward way Or we could convert it to the normal form (lists) first, and *then* evaluate it. This finally gives us a link between the normalised and non-normalised forms.



The goal is to construct a proof of equivalence between the two expressions at the bottom: to do this, we first construct the AST which represents the two expressions (for now, we'll assume the user constructs this AST themselves. Later we'll see how too construct it automatically from the provided expressions). Then, we can evaluate it into either the normalized form, or the unnormalized form. Since the normalized forms are syntactically equal, all we need is refl to prove their equality. The only missing part now is correct, which is the task of the next section.

Moving on to Polynomials

Benjamin Grégoire and Assia Mahboubi. Proving Equalities in a Commutative Ring Done Right in Coq.

In *Theorem Proving in Higher Order Logics*, volume 3603 of *Lecture Notes in Computer Science*, pages 98–113, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg

We now know the components required for an automatic solver for some algebra: a canonical form, a concrete representation of expressions, and a proof of correctness. We now turn our focus to polynomials.

The state-of-the art approach is presented in this paper.

Canonical Form

```
Poly : Set ℓ
Poly = List Carrier

_⊞_ : Poly → Poly → Poly
[] ⊞ ys = ys
(x :: xs) ⊞ [] = x :: xs
(x :: xs) ⊞ (y :: ys) = x + y :: xs ⊞ ys

_⊠_ : Poly → Poly → Poly
_⊠_ [] _ = []
_⊠_ (x :: xs) =
  foldr (λ y ys → x * y :: map (_* y) xs ⊞ ys) []
```

The canonical representation of polynomials is a list of coefficients, least significant first ("Horner Normal Form"). Our initial attempt at encoding this representation is as follows.

Horner's Rule

$$
\begin{aligned}
p(x) &= a_0 x^0 + a_1 x^1 + a_2 x^2 + \ldots a_n x^n \\
&= a_0 + x(a_1 + x(a_2 + x(\ldots a_n + x(0))))
\end{aligned}
$$

Horner's rule is a method for evaluating polynomials which lets you avoid repeatedly exponentiating the input variable.

Horner's Rule

$$
\begin{aligned}
p(x) &= a_0 x^0 + a_1 x^1 + a_2 x^2 + \ldots a_n x^n \\
&= a_0 + x(a_1 + x(a_2 + x(\ldots a_n + x(0))))
\end{aligned}
$$

$[\![\_]\!] : \text{Poly} \to \text{Carrier} \to \text{Carrier}$
$[\![\ x\ ]\!]\ \rho = \text{foldr}\ (\lambda\ y\ ys \to y + \rho\ *\ ys)\ 0\#\ x$

Haskellers will recognize it as a foldr operation.

Problems

As it stands, the above representation has two problems

Redundancy

$$
\begin{aligned}
2x = &\ 0, 2 \\
&\ 0, 2, 0 \\
&\ 0, 2, 0, 0 \\
&\ 0, 2, 0, 0, 0, 0, 0
\end{aligned}
$$

The representation suffers from the problem of trailing zeroes. In other words, the polynomial $2x$ could be represented by any of the following:

This is a problem for a solver: the whole *point* is that equivalent expressions are represented the same way.

Problems

Redundancy

$$
\begin{aligned}
2x &= 0, 2 \\
&\quad 0, 2, 0 \\
&\quad 0, 2, 0, 0 \\
&\quad 0, 2, 0, 0, 0, 0, 0
\end{aligned}
$$

Inefficiency

Expressions will tend to have large gaps, full only of zeroes. Something like x^5 will be represented as a list with 6 elements, only the last one being of interest. Since addition is linear in the length of the list, and multiplication quadratic, this is a major concern.

A Sparse Encoding

$3 + 2x^2 + 4x^5 + 2x^7$

The solution usually used is a "power index". A representation of the gap between adjacent nonzero coefficients.

We rewrite the following equation as so:

A Sparse Encoding

$$3 + 2x^2 + 4x^5 + 2x^7 = x^0(3 + xx^1(2 + xx^2 * (4 + xx^1(2 + x0))))$$

```
[(3,0),(2,1),(4,2),(2,1)]
```


And then we can represent it in a list like so:

```
infixl 6 _≠0
record Coeff : Set (a ⊔ ℓ) where
  inductive
  constructor _≠0
  field
    coeff : Carrier
    .{coeff≠0} : ¬ Zero coeff
open Coeff

Poly : Set (a ⊔ ℓ)
Poly = List (Coeff × ℕ)
```

We actually go further than the previous approach, because we *prove* that the coefficients are in normal form.

Termination

```
fib : ℕ → ℕ
fib 0 = 0
fib 1 = 1
fib (1+ (1+ n)) = fib (1+ n) + fib n
```

```
fib : ℕ → ℕ
fib 0 = 0
fib 1 = 1
fib n = fib (n − 1) + fib (n − 2)
```


Ever-present in Agda is the need to prove termination.

Our conversion to normal form, and the operations on that normal form, are complex, and not trivially terminating.

To pass the termination checker, a recursive call has to have structurally smaller arguments. In other words, the arguments must be subterms of what was given.

Comparing these two recursive functions, only the first passes the termination checker. The first n is a subterm of the argument passed in; in the second example, it's not.

Well-Founded Recursion

Well-Founded Relation

Contains no infinite descending chains.

"Less than" on $\mathbb{N}$

$$1 < 2 < 4 < 8$$

Bengt Nordström. Terminating general recursion.

***BIT*, 28(3):605–619, September 1987**

Is this consistent?

The majority of programs will be structurally recursive.
However, some (as in the second fib above) will not be, and we will need some extra information to prove termination.
In Agda, we do this using well-founded relations.

Well-Founded Recursion

```
data Acc {A : Set} (_R_ : A → A → Set) (x : A) : Set where
  acc : (∀ y → y R x → Acc _R_ y) → Acc _R_ x

data _<_ (m : ℕ) : ℕ → Set where
  0<1 : m < suc m
  m<s : ∀ {n} → m < n → m < suc n

<-wellFounded : ∀ m → Acc _<_ m
<-wellFounded = acc ∘ go
  where
  go : ∀ m n → n < m → Acc _<_ n
  go zero n ()
  go (suc m) .m 0<1 = <-wellFounded m
  go (suc m) n (m<s n<m) = go m n n<m
```


We don't add well-founded relations to the theory, as each individual relation may not be consistent with other notions of termination.

Instead, for any relation, we *reformulate* it in terms of structural recursion.

Homomorphism

Richard Bird and Oege de Moor. *Algebra of Programming*.

Prentice-Hall international series in computer science. Prentice Hall, London ; New York, 1997

Shin-Cheng Mu, Hsiang-Shang Ko, and Patrik Jansson. Algebra of programming in Agda: Dependent types for relational program derivation.

***Journal of Functional Programming*, 19(5):545–579, September 2009**

After termination is proven, we need to prove that the canonical form corresponds to the underlying carrier. This is proving that the operations we have defined are a ring homomorphism.

The only last thing I'll mention is with regards to the nature of the proofs. They use a technique called the "algebra of programming" to shorten them up: because most of our functions are defined using higher-order folds and so on, we can express the proofs in very abstract, terse terms.

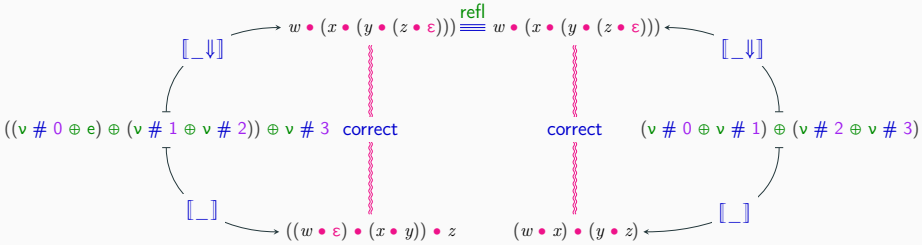
The proofs are still roughly 1000 lines long, though.

Interface

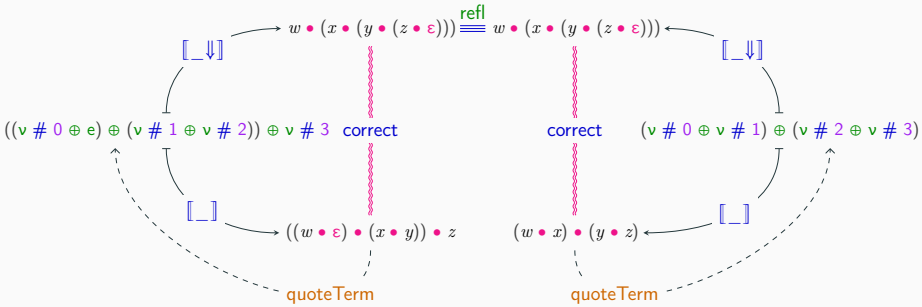
```
ident′ : ∀ w x y z
       → ((w • ε) • (x • y)) • z
       ≈ (w • x) • (y • z)
ident′ = solve 4
  ( λ w x y z
    → ((w ⊕ e) ⊕ (x ⊕ y)) ⊕ z
    ⊜ (w ⊕ x) ⊕ (y ⊕ z))
  refl
```


Though what we have works, it's still awkward to use, because we ask the user to construct the AST themselves. This means they write the type twice, and it is often tedious to do so.

This is the "automatic" proof for the monoid solver we had previously.



What we'd like to do is provide the ASTs here automatically, from the two expressions at the bottom.



What can we use for that? *Reflection*

Reflection is often thought to be the purview of unsafe or dynamic languages, but it actually fits really well into the dependently typed setting. As it happens, Agda has a decent (type-directed) macro system. The end result is a very clean system

The Finished Solver

```
lemma : ∀ x y
      → x + y * 1 + 3 ≈ 2 + 1 + x + y
lemma = solve NatRing
```

Even more, because this relies on the *type* of the hole (the inferred obligation) if you need to prove two things line up, you just pinpoint where they son't match, and say "solve, please!"

Setoids

We used a setoid throughout the solver, which really made things difficult for us.

Usually, The purpose of this particular hair shirt is flexibility: users can still use the solver even if their type only satisfies the monoid laws modulo some equivalence relation (perhaps they are have an implementation of finite, mergeable sets as balanced trees, and want to treat two sets as equivalent if their elements are equal, even if their internal structures are not).

It also frees up some very interesting applications, though.

Setoids

Pedagogical Solutions

```
data Traced {A : Set} (x : A) : A → Set where
  refl     : Traced x x
  ⟨_⟩≡_ : ∀ {y z}
        → (reason : String)
        → Traced y z
        → Traced x z
```

All you need a setoid to be is something with symmetry, reflexivity, and transitivity. One example would be a list of rewrite rules! Now, the output is a proof, along with a *step-by-step* solution to how you got from one equation to the other. We get wolfram alpha for free! And it's verified!

Reflexivity is the empty list.

Transitivity is concatenation.

Symmetry is reverse.

Isomorphisms

```
record _⇌_ (x y : Set) : Set where
  field ← : x → y; → : y → x
open _⇌_

sym : ∀ {x y} → x ⇌ y → y ⇌ x
sym x⇌y .← x = x⇌y .→ x
sym x⇌y .→ y = x⇌y .← y

trans : ∀ {x y z} → x ⇌ y → y ⇌ z → x ⇌ z
trans x⇌y y⇌z .← x = y⇌z .← (x⇌y .← x)
trans x⇌y y⇌z .→ z = x⇌y .→ (y⇌z .→ z)

refl : ∀ {x} → x ⇌ x
refl .← x = x; refl .→ x = x
```


Some of the type operations we mentioned earlier were ring-like: sum types, product types, the empty type, the singleton type, etc.

Turns out these can be directly translated into polynomials!

And the equivalence relation? An isomorphism! So now we can automatically construct isomorphisms between equivalent types.

The Correct-by-Construction Approach

The Agda and Coq communities exhibit something of a cultural difference when it comes to proving things. Coq users seem to prefer writing simpler, almost non-dependent code and algorithms, to separately prove properties about that code in auxiliary lemmas. Agda users, on the other hand, seem to prefer baking the properties into the definition of the types themselves, and writing the functions in such a way that they prove those properties as they go (the "correct-by-construction" approach).

There are advantages and disadvantages to each approach. The Coq approach, for instance, allows you to reuse the same functions in different settings, verifying different properties about them depending on what's required. In Agda, this is more difficult: you usually need a new type for every invariant you maintain (lists, and then length-indexed lists, and then sorted lists, etc.). On the other hand, the proofs themselves often contain a lot of duplication of the logic in the implementation: in the Agda style, you avoid this duplication, by doing both at once. Also worth noting is that occasionally attempting to write a function that is correct by construction will lead to a much more elegant formulation of the original algorithm, or expose symmetries between the proof and implementation that would have

Benjamin Grégoire and Assia Mahboubi. Proving Equalities in a Commutative Ring Done Right in Coq.

In *Theorem Proving in Higher Order Logics*, volume 3603 of *Lecture Notes in Computer Science*, pages 98–113, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg

```
Inductive Pol (C:Set) : Set :=
  | Pc : C -> Pol C
  | Pinj : positive -> Pol C -> Pol C
  | PX : Pol C -> positive -> Pol C -> Pol C.
```


The gregoire version as an example, is very much in the Coq style: the definition of the polynomial type has no type indices, and makes no requirements on its internal structure:

The implementation presented here straddles both camps: we verify homomorphism in separate lemmas, but the type itself does carry information: it's indexed by the number of variables it contains, for instance, and it statically ensures it's always in canonical form.

The Correct-by-Construction Approach

Franck Slama and Edwin Brady. Automatically Proving Equivalence by Type-Safe Reflection.

In Herman Geuvers, Matthew England, Osman Hasan, Florian Rabe, and Olaf Teschke, editors, *Intelligent Computer Mathematics*, volume 10383, pages 40–55. Springer International Publishing, Cham, 2017

The correct-by-construction approach is explored in Idris, but they don't employ the same level of reflection or optimisation as we do.

The Correct-by-Construction Approach

```
data Poly : Carrier → Set (a ⊔ ℓ) where
  ⟦⟧ : Poly 0#
  ⟦_::_⟧
    : ∀ x {xs}
    → Poly xs
    → Poly (λ ρ → x Coeff.+ ρ Coeff.* xs ρ)

infixr 0 _⇐_
record Expr (expr : Carrier) : Set (a ⊔ ℓ) where
  constructor _⇐_
  field
    {norm} : Carrier
    poly   : Poly norm
    proof  : expr ≋ norm
```


Nonetheless, we do provide an implementation of this version, for comparison.