Talking About Mathematics in a Programming Language

Donnacha Oisín Kidney

October 15, 2018

What do Programming Languages Have to do with Mathematics?

Programming is Proving

A Polynomial Solver

The p-Adics

What do Programming Languages Have to do with Mathematics?

Languages for proofs and languages for programs have a lot of the same requirements.

Languages for proofs and languages for programs have a lot of the same requirements.

A *Syntax* that is

- Readable
- Precise
- Terse

Languages for proofs and languages for programs have a lot of the same requirements.

A *Syntax* that is

- Readable
- Precise
- Terse

***Semantics* that are**

- Small
- Powerful
- Consistent

Semantics/axiomatic core Some of these are conflicting!

Why not use a programming language as our proof language?

Benefits For Programmers

Benefits For Programmers

- *Prove* things about code

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

vs

$$\text{reverse-involution} : \forall\ xs \rightarrow \text{reverse}\ (\text{reverse}\ xs) \equiv xs$$

Not just test!

Benefits For Programmers

- *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!

Benefits For Programmers

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

Benefits For Mathematicians

Benefits For Mathematicians

- Have a machine check your proofs

Currently, though, this is *tedious*

Benefits For Mathematicians

- Have a machine check your proofs
- Run your proofs

Benefits For Mathematicians

- Have a machine check your proofs
- Run your proofs
- Develop a consistent foundation for maths

Benefits For Mathematicians

- Have a machine check your proofs
- Run your proofs
- Develop a consistent foundation for maths

Wait—isn't this impossible?

Formalizing Mathematics

Lawrence C Paulson. The Future of Formalised Mathematics, 2016

Formalizing Mathematics

Whitehead and Russell took
hundreds of pages to prove
$1 + 1 = 2$

Lawrence C Paulson. The Future of Formalised Mathematics, 2016

Formalizing Mathematics

Whitehead and Russell took *hundreds* of pages to prove $1 + 1 = 2$

Gödel showed that universal formal systems are incomplete

Lawrence C Paulson. The Future of Formalised Mathematics, 2016

Formalizing Mathematics

Whitehead and Russell took *hundreds* of pages to prove $1 + 1 = 2$

Formal systems have improved

Gödel showed that universal formal systems are incomplete

Lawrence C Paulson. The Future of Formalised Mathematics, 2016

Formalizing Mathematics

Whitehead and Russell took *hundreds* of pages to prove $1 + 1 = 2$

Formal systems have improved

Gödel showed that universal formal systems are incomplete

We don't need universal systems

Lawrence C Paulson. The Future of Formalised Mathematics, 2016

What About Automated Theorem Provers?

Use a combination of heuristics and exhaustive search to check some proposition.

We have to trust the implementation.

What About Automated Theorem Provers?

Generally regarded as:

What About Automated Theorem Provers?

Generally regarded as:

- Inelegant

What About Automated Theorem Provers?

Generally regarded as:

- Inelegant
- Lacking Rigour

What About Automated Theorem Provers?

Generally regarded as:

- Inelegant
- Lacking Rigour
- Not Insightful

What About Automated Theorem Provers?

Generally regarded as:

- Inelegant
- Lacking Rigour
- Not Insightful

Require trust

Non Surveyable

The Four-Colour Theorem

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

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

Did contain bugs!

But what if our formal language is executable?

But what if our formal language is executable?

Can we write *verified* automated theorem provers?

Prove things about programs, and prove things about maths

But what if our formal language is executable?

Can we write *verified* automated theorem provers?

Georges Gonthier. Formal Proof—The Four-Color Theorem.

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

Programming is Proving

The Curry-Howard Correspondence

$$\begin{tikzcd} \mathit{Type} \arrow[r, Leftrightarrow] \arrow[d] & \mathit{Proposition} \arrow[d] \\ \mathit{Program} \arrow[r, Leftrightarrow] & \mathit{Proof} \end{tikzcd}$$

Philip Wadler. Propositions As Types.

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

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.

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.

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 {`

```
head : {A : Set} → List A → 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!

Function definition syntax

$$
\begin{aligned}
&\mathsf{fib} : \mathbb{N} \to \mathbb{N} \\
&\mathsf{fib}\ 0 && = 0 \\
&\mathsf{fib}\ (1+\ 0) && = 1+\ 0 \\
&\mathsf{fib}\ (1+\ (1+\ n)) && = \mathsf{fib}\ (1+\ n) + \mathsf{fib}\ n
\end{aligned}
$$

Agda functions are defined (usually) with *pattern-matching*. For the natural numbers, we use the Peano numbers, which gives us 2 patterns: zero, and successor.

But Let's Try Anyway!

$$\mathsf{length} : \{A : \mathsf{Set}\} \to \mathsf{List}\ A \to \mathbb{N}$$
$$\mathsf{length}\ [] = 0$$
$$\mathsf{length}\ (x :: xs) = 1 + \mathsf{length}\ xs$$

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!

Here's a definition for head:

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

No!
For correct proofs, partial functions aren't allowed

We need to disallow functions which don't match all patterns. Array access out-of-bounds, etc., also not allowed.

But Let's Try Anyway!

We're not out of the woods yet:

head [] = head []

No!
For correct proofs, all functions must be total

To disallow *this* kind of thing, we must ensure all functions are *total*. For now, assume this means "terminating".

Correctness

For the proofs to be correct, we have two extra conditions that you usually don't have in programming:

- No partial programs
- Only total programs

Without these conditions, your proofs are still correct *if they run*.

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

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

$A \rightarrow B$

Implication

$A \to B$

A implies B

Implication

$A \rightarrow B$

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 $\bot$: Set where

Contradiction

$\text{ptb} : \forall\ \{a\}\ \{A : \text{Set}\ a\} \to \neg\ A \to A \to \bot$
$\text{ptb}\ f\ x = f\ x$

In fact, we can convert from what we had previously

Booleans?

Contradiction

```
data ⊥ : Set where

ptb : ∀ {a} {A : Set a} → ¬ A → A → ⊥
ptb f x = f x

Inc : ¬ ⊥
Inc ()
```


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.

Conjunction

Conjunction (“and”) is represented as a data type.

Conjunction

```
record _×_ (A B : Set) : Set where
  constructor _,_
  field
    fst  : A
    snd : B
```


It has two type parameters, and two fields.

Conjunction

```
record _×_ (A B : Set) : Set where
  constructor _,_
  field
    fst  : A
    snd : B
```

Swift

```swift
struct Pair<A,B>{
  let fst: A
  let snd: B
}
```

Python

```python
class Pair:
    def __init__(self, x, y):
        self.fst = x
        self.snd = y
```


Syntax-wise, it's equivalent to a *class* in other languages.

Conjunction

```
record _×_ (A B : Set) : Set where
  constructor _,_
  field
    fst  : A
    snd : B


data _×_ (A B : Set) : Set where
  _,_ : A → B → A × B
```


We could also have written it like this. (Haskell-style)

The definition is basically equivalent, but we don't get two field accessors (we'd have to define them manually) and some of the syntax is better suited to the record form.

It does show the type of the constructor, though (which is the same in both).

It's curried, which you don't need to understand: just think of it as taking two arguments.

"If you have a proof of A, and a proof of B, you have a proof of A *and* B"

Conjunction

```
record _×_ (A B : Set) : Set where
  constructor _,_
  field
    fst  : A
    snd : B
```

Type Theory
2-Tuple

Conjunction

```
record _×_ (A B : Set) : Set where
  constructor _,_
  field
    fst  : A
    snd : B
```

Set Theory
Cartesian Product

$$\{t, f\} \times \{1, 2, 3\} = \{(t, 1), (f, 1), (t, 2), (f, 2), (t, 3), (f, 3)\}$$

Conjunction

```
record _×_ (A B : Set) : Set where
  constructor _,_
  field
    fst  : A
    snd : B
```

Familiar identities: conjunction-elimination

```
cnj-elim : ∀ {A B} → A × B → A
cnj-elim = fst
```

$$A \wedge B \implies A$$

Currying

Just a short note on currying.

People familiar with Haskell will know what it is, I won't explain it in its entirety here, though. Just a little interesting thing on how it translates into logic.

Currying

$$\text{curry} : \{A\ B\ C : \text{Set}\} \to (A \times B \to C) \to A \to (B \to C)$$
$$\text{curry}\ f\ x\ y = f\ (x\ ,\ y)$$

Just a short note on currying.

People familiar with Haskell will know what it is, I won't explain it in its entirety here, though. Just a little interesting thing on how it translates into logic.

Currying

The type:
$A, B \to C$

Just a short note on currying.

People familiar with Haskell will know what it is, I won't explain it in its entirety here, though. Just a little interesting thing on how it translates into logic.

Currying

The type:

$A, B \to C$

Is isomorphic to:

$A \to (B \to C)$

Just a short note on currying.

People familiar with Haskell will know what it is, I won't explain it in its entirety here, though. Just a little interesting thing on how it translates into logic.

Currying

The type:
$A, B \to C$

Is isomorphic to:
$A \to (B \to C)$

Because the statement:
"A and B implies C"

Just a short note on currying.

People familiar with Haskell will know what it is, I won't explain it in its entirety here, though. Just a little interesting thing on how it translates into logic.

Currying

The type:
$A, B \rightarrow C$

Is isomorphic to:
$A \rightarrow (B \rightarrow C)$

Because the statement:
"A and B implies C"

Is the same as saying:
"A implies B implies C"

Just a short note on currying.

People familiar with Haskell will know what it is, I won't explain it in its entirety here, though. Just a little interesting thing on how it translates into logic.

Currying

"If I'm outside and it's raining, I'm going to get wet"

$$\mathit{Outside} \wedge \mathit{Raining} \implies \mathit{Wet}$$

Just a short note on currying.

People familiar with Haskell will know what it is, I won't explain it in its entirety here, though. Just a little interesting thing on how it translates into logic.

Currying

"If I'm outside and it's raining, I'm going to get wet"

$$\mathit{Outside} \wedge \mathit{Raining} \implies \mathit{Wet}$$

"When I'm outside, if it's raining I'm going to get wet"

$$\mathit{Outside} \implies \mathit{Raining} \implies \mathit{Wet}$$

Just a short note on currying.

People familiar with Haskell will know what it is, I won't explain it in its entirety here, though. Just a little interesting thing on how it translates into logic.

Disjunction

```
data _∪_ (A B : Set) : Set where
  inl : A → A ∪ B
  inr : B → A ∪ B
```

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.

NOTE: when you prove things in non-total languages, the proofs only hold *if they terminate*. That doesn't *really* mean that they're "invalid", it just means that you have to run it for every case you want to check.

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

The Π Type

The Π Type

Upgrade the *function arrow*

First, we upgrade the function arrow, so the right-hand-side can talk about the value on the left.

The Π Type

Upgrade the *function arrow*

$$\text{prop} : (x : \mathbb{N}) \to 0 \leq x$$

This lets us easily express *properties*

The Π Type

Upgrade the *function arrow*

$$\text{prop} : (x : \mathbb{N}) \to 0 \leq x$$

Now we have a proper ∀

The Σ Type

The Σ Type

Upgrade *product types*

The Σ Type

Upgrade *product types*

```
record NonZero : Set where
  field
    n     : ℕ
    proof : 0 < n
```


Later fields can refer to earlier ones.

The Σ Type

Upgrade *product types*

```
record NonZero : Set where
  field
    n     : ℕ
    proof : 0 < n
```

Now we have a proper ∃

The Equality Type

```
infix 4 _≡_
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```


Final piece of the puzzle.

The type of this type has 2 parameters.

But the only way to construct the type is if the two parameters are the same.

You then get evidence of their sameness when you pattern-match on that constructor.

Equality

$\_+\_ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$
$0 + y = y$
$\mathsf{suc}\ x + y = \mathsf{suc}\ (x + y)$

$\mathsf{obvious} : \forall\ x \to 0 + x \equiv x$
$\mathsf{obvious}\ \_ = \mathsf{refl}$

Agda uses propositional equality

You can construct the equality proof when it's obvious.

Equality

```
_+_ : ℕ → ℕ → ℕ
0 + y = y
suc x + y = suc (x + y)

obvious : ∀ x → 0 + x ≡ x
obvious _ = refl


cong : ∀ {A B} → (f : A → B) → ∀ {x y} → x ≡ y → f x ≡ f y
cong _ refl = refl

not-obvious : ∀ x → x + 0 ≡ x
not-obvious zero = refl
not-obvious (suc x) = cong suc (not-obvious x)
```


you need to supply the proof yourself when it's not obvious.

Open Areas and Weirdness

- Law of Excluded Middle?
- Russell's Paradox
- Function Extensionality
- Data Constructor Injectivity
- Observational Equality
- Homotopy Type Theory

A Polynomial Solver

The p-Adics