Talking About Mathematics in a Programming Language

Donnacha Oisín Kidney

October 14, 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

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

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

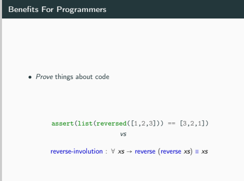
Benefits For Programmers

- *Prove* things about code

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

vs

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



Not just test!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?

- *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 Programmers

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

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

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?

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?

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

$$\begin{array}{ccc} \textit{Type} & \Longleftrightarrow & \textit{Proposition} \\ \downarrow & & \downarrow \\ \textit{Program} & \Longleftrightarrow & \textit{Proof} \end{array}$$

Philip Wadler. Propositions As Types.

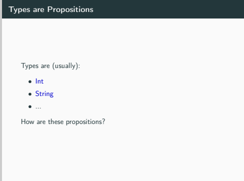
***Commun. ACM*, 58(12):75–84, November 2015**

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:	Think:
$x : \mathbb{N}$	$\exists . \mathbb{N}$

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:

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

Basic Agda Syntax

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 {
```

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

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

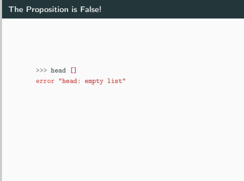
Equivalent in other languages:

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

$\text{head} : \{A : \text{Set}\} \to \text{List}\ A \to A$ "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

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

But Let's Try Anyway!

Function definition syntax

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

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!

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

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

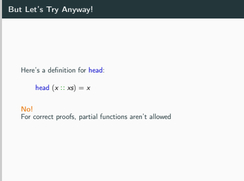
For lists, we also have two patterns: the empty list, and the head element followed by the rest of the list.

But Let's Try Anyway!

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:

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

No!
For correct proofs, all functions must be total

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

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?

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

```
¬ : ∀ {ℓ} → Set ℓ → Set _
¬ A = A → {B : Set} → B
```

Principle of Explosion
"Ex falso quodlibet"

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

$\text{head-doesn't-exist} : \neg\, (\{A : \mathsf{Set}\} \to \mathsf{List}\ A \to A)$
$\text{head-doesn't-exist}\ \mathit{head} = \mathit{head}\ []$

```
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

$A \to B$

A implies B

Constructivist/Intuitionistic

Implication

$A \to B$

A implies B

Constructivist/Intuitionistic

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

Booleans?

<1> 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?

<1> 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*

```
data ⊥ : Set where
```

Contradiction

```
Inc : ⊥ → {A : Set} → A
Inc ()
```

Booleans?

<1> 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*

```
data ⊥ : Set where          Contradiction

  Inc : ⊥ → {A : Set} → A
  Inc ()
```

Falsehood (contradiction) is the proposition with no proofs.

It's equivalent to what we had previously.

Booleans?

<1> 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*

```
data ⊥ : Set where

  Inc : ⊥ → {A : Set} → A
  Inc ()
```

Contradiction

```
data ⊤ : Set where
  tt : ⊤
```

Tautology

A Polynomial Solver

The p-Adics