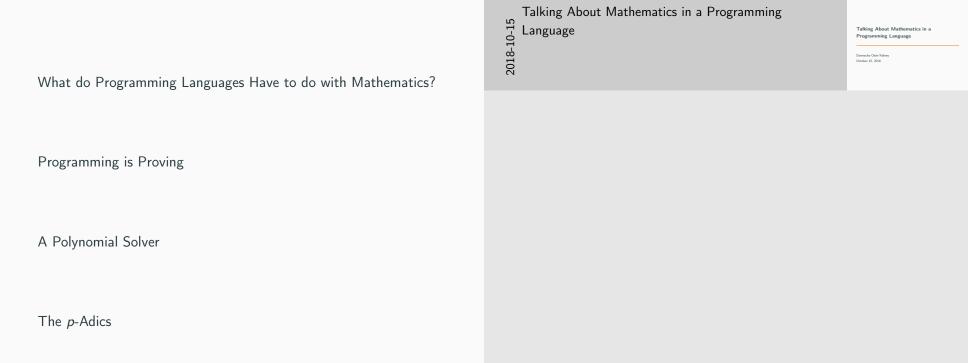
Donnacha Oisín Kidney October 15, 2018

Talking About Mathematics in a Programming Language

Talking About Mathematics in a Programming Language

Donnacha Oisin Kidney October 15, 2018



Talking About Mathematics in a Programming

Language

What do Programming Languages Have to do with

Mathematics?

What do Programming Languages Have to do with Mathematics?

What do Programming Languages Have to do with Mathematics?

Talking About Mathematics in a Programming Language —What do Programming Languages Have to do with Mathematics?

Languages for proofs and languages for programs have a lot of the

Talking About Mathematics in a Programming Language -What do Programming Languages Have to do with Mathematics?

Languages for proofs and languages for programs have a lot of the

 Precise • Terse

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.

Α	Sı	ntax	that	i
$\overline{}$	\sim 1	<i>HILUA</i>	LIIGL	

- Readable
- Precise
- Terse

- Semantics that are
 - Small
 - Powerful
 - Consistent

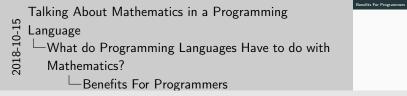
Talking About Mathematics in a Programming Language -What do Programming Languages Have to do with Mathematics?

Semantics/axiomatic core Some of these are conflicting!

Language

What do Programming Languages Have to do with Mathematics?

Talking About Mathematics in a Programming



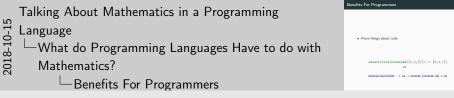
Mathematics and formal language has existed for thousands of years; programming has existed for only 60!

• Prove things about code

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

vs
```

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



Not just test! Mathematics and formal language has existed for thousands of years; programming has existed for only 60!

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

Talking About Mathematics in a Programming
Language

What do Programming Languages Have to do with
Mathematics?

Benefits For Programmers

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

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

Talking About Mathematics in a Programming

Language

What do Programming Languages Have to do with

Mathematics?

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!

Talking About Mathematics in a Programming

Language

What do Programming Languages Have to do with

Mathematics?

Benefits For Mathematicians

• Have a machine check your proofs

Currently, though, this is *tedious*

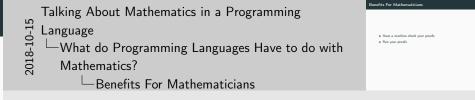
Talking About Mathematics in a Programming
Language
What do Programming Languages Have to do with
Mathematics?
Benefits For Mathematicians

Benefits For Mathematicians

Have a muchine check your proofs

Currently, though, this is indices

- Have a machine check your proofs
- Run your proofs



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

Talking About Mathematics in a Programming

Language

What do Programming Languages Have to do with

Mathematics?

Benefits For Mathematicians

- Benefits For Mathematicians
- · Have a machine check your proofs
- Run your proofs
 Develop a consistent foundation for maths

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

Wait—isn't this impossible?

Talking About Mathematics in a Programming

Language

What do Programming Languages Have to do with

Mathematics?

Benefits For Mathematicians



Talking About Mathematics in a Programming Language -What do Programming Languages Have to do with Mathematics? Formalizing Mathematics

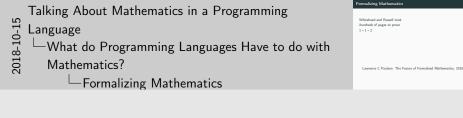
Formalizing Mathematics

Lawrence C Paulson. The Future of Formalised Mathematics, 2016

Lawrence C Paulson. The Future of Formalised Mathematics, 2016

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

Lawrence C Paulson. The Future of Formalised Mathematics, 2016



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

hundreds of pages to prove Language -What do Programming Languages Have to do with formal systems are incomplete Mathematics? Lawrence C Paulson. The Future of Formalised Mathematics, 2016 -Formalizing Mathematics

Talking About Mathematics in a Programming

Formalizing Mathematics

Gödel showed that universal

Whitehead and Russell took hundreds of pages to prove Formal systems have improved 1+1=2

formal systems are incomplete

Lawrence C Paulson. The Future of Formalised Mathematics. 2016

Talking About Mathematics in a Programming

Language

What do Programming Languages Have to do with

Mathematics?

Formalizing Mathematics

Formalizing Mathematics

4

1 + 1 = 2

Whitehead and Russell took

hundreds of pages to prove

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

Talking About Mathematics in a Programming

Language

What do Programming Languages Have to do with

Mathematics?

Formalizing Mathematics

Formalizing Mathematics

4

Talking About Mathematics in a Programming

Language

What do Programming Languages Have to do with

Mathematics?

What About Automated Theorem Provers?

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

What About Automated Theorem Provers?

We have to trust the implementation.

Generally regarded as:

Talking About Mathematics in a Programming

Language

What do Programming Languages Have to do with

Mathematics?

What About Automated Theorem Provers?

Generally regarded as:

Inelegant

Talking About Mathematics in a Programming

Language

What do Programming Languages Have to do with

Mathematics?

What About Automated Theorem Provers?

Generally regarded as:

- Inelegant
- Lacking Rigour

Talking About Mathematics in a Programming

Language

What do Programming Languages Have to do with

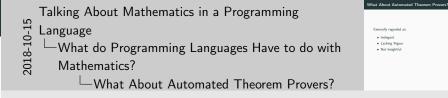
Mathematics?

What About Automated Theorem Provers?

5

Generally regarded as:

- Inelegant
- Lacking Rigour
- Not Insightful



2018-10-15

Talking About Mathematics in a Programming Language Mathematics?

-What do Programming Languages Have to do with

-What About Automated Theorem Provers?

What About Automated Theorem Provers? Generally regarded as: · Lacking Rigour

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!

The Four-Colour Theorem Talking About Mathematics in a Programming Language -What do Programming Languages Have to do with Mathematics? The Four-Colour Theorem

Kenneth Appel and Wolfgang Haken. The Solution of th

Mathematics?

—What do Programming Languages Have to do with

But what if our formal language is executable?

Mathematics?

But what if our formal language is executable? Can we write *verified* automated theorem provers? Prove things about programs, and prove things about maths

-What do Programming Languages Have to do with

Talking About Mathematics in a Programming Language -What do Programming Languages Have to do with Mathematics?

But what if our formal language is executable? Georges Gonthier, Formal Proof-The Four-Color Theorem. Notices of the AMS, 55(11):12, 2008

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

Talking About Mathematics in a Programming
Language
Programming is Proving

Programming is Proving

The Curry-Howard Correspondence

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

Philip Wadler. Propositions As Types.

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

Talking About Mathematics in a Programming
Language
Programming is Proving
The Curry-Howard Correspondence

The Curry Housed Correspondence

7pp ← → Preparation

Prepara ← → Preparation

Preparation At Types

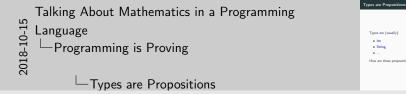
ACM # \$55(7) VIE. The Managember \$105.

Types are Propositions

Types are (usually):

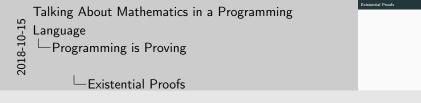
- Int
- String
- ...

How are these propositions?



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

Existential Proofs

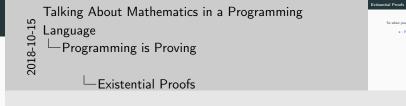


We use a trick to translate: put a "there exists" before the type.

Existential Proofs

So when you see:

 $\mathsf{x}:\mathbb{N}$



So when you see: Think:

 $\mathsf{x}:\mathbb{N}$

 $\mathbb{A}.\mathbb{E}$

Talking About Mathematics in a Programming

Language

Programming is Proving

Existential Proofs

e: T

Existential Proofs

Think: 3.N

10

So when you see: Think:

 $\mathsf{x}:\mathbb{N}$ 3.N

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

Talking About Mathematics in a Programming

Language

Programming is Proving

Existential Proofs

So when you see: Think: $x:\mathbb{N} \qquad 3.\mathbb{N}$ NB $W_W||_{See \ a \ more \ powerful \ and \ precise \ version \ of \ 3 \ Later.}$

Existential Proofs

So when you see: Think:

 $\mathsf{x}:\mathbb{N}$ 3.N

 $\ensuremath{\mathsf{NB}}$ We'll see a more powerful and precise version of \exists later.

Proof is "by example":

Talking About Mathematics in a Programming

Language
Programming is Proving

Existential Proofs

So when you see:

Think:

 $\mathsf{x}:\mathbb{N}$

 $\mathbb{A}.\mathbb{E}$

NB

We'll see a more powerful and precise version of ∃ later.

Proof is "by example":

$$x = 1$$

Talking About Mathematics in a Programming
Language
Programming is Proving
LExistential Proofs

Extracted Proofs $So \ when you use Think: $x:N$ 2.N$ NB Will use a more posseful and precise version of 3 later. Proof is "by example": $x=1$$

Programs are Proofs

Talking About Mathematics in a Programming

Language

Programming is Proving

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

Programs are Proofs

>>> head [1,2,3]

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

Programs are Proofs

Programs are Proofs

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

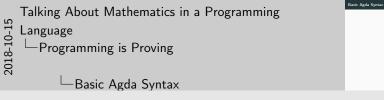
Here's the type:

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

Talking About Mathematics in a Programming Language
Programming is Proving
Programs are Proofs

>>> head $\{1,2,3\}$. $\label{eq:lemma:problem} 1$ Heav's the type: $\label{eq:lemma:lemma:problem} \text{head}: \{A: \mathsf{Set}\} \to \mathsf{Lite}\: A \to A$

Programs are Proofs



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 {

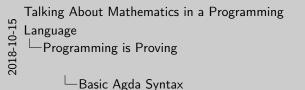
Talking About Mathematics in a Programming
Language
Programming is Proving
Basic Agda Syntax

Equivalent in other languages:

Haskell head :: [a] -> a

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

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





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

Equivalent in other languages:

Haskell head :: [a] -> a

Swift func $head < A > (xs : [A]) \rightarrow A$ {

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

Talking About Mathematics in a Programming Language -Programming is Proving

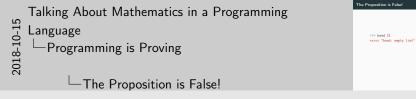
-Basic Agda Syntax

Basic Agda Syntax

12

The Proposition is False!

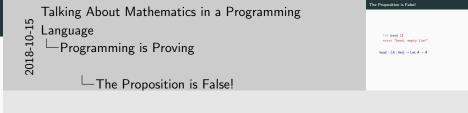
```
>>> head []
error "head: empty list"
```



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" head: \{A : Set\} \rightarrow List A \rightarrow A
```



The Proposition is False!

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

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

False

Talking About Mathematics in a Programming

Language

Programming is Proving

The Proposition is 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

-Programming is Proving

If Agda is correct (as a formal logic):

We shouldn't be able write this function in Agda

Function definition syntax

fib:
$$\mathbb{N} \to \mathbb{N}$$

fib 0 = 0
fib (1+0) = 1+0
fib (1+ (1+ n)) = fib (1+ n) + fib n

Talking About Mathematics in a Programming Language

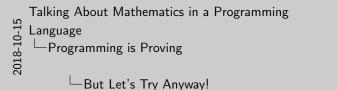
—Programming is Proving

But Let's Try Aryway! $Function definition system <math display="block">\begin{cases} 6: \ N-N \\ 6: \ 0 \end{cases} = 0 \\ 6: \ (1+0) \\ = 1+0 \\ 6: \ (1+n)+6: \ n \end{cases}$

∟But Let's Try Anyway!

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.

```
length : \{A : \mathsf{Set}\} \to \mathsf{List}\ A \to \mathbb{N}
length [] = 0
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.

Here's a definition for head:

head
$$(x :: xs) = x$$

No!

For correct proofs, partial functions aren't allowed

Talking About Mathematics in a Programming

Language

Programming is Proving

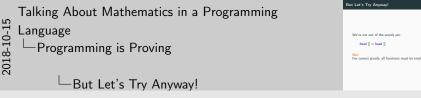
But Let's Try Anyway!

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

We're not out of the woods yet:

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

Talking About Mathematics in a Programming Language

—Programming is Proving

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

Only total programs

└ Correctness

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

Talking About Mathematics in a Programming Language

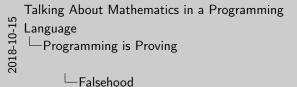
☐ Programming is Proving

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



Falsahood

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.

Talking About Mathematics in a Programming

Language

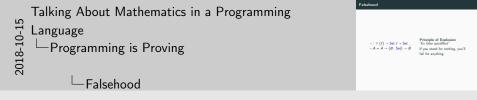
Programming is Proving

Falsehood

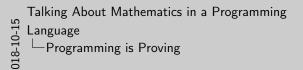
$$\neg: \forall \{\ell\} \to \mathsf{Set} \ \ell \to \mathsf{Set} \ _$$
$$\neg \ A = A \to \{B : \mathsf{Set}\} \to B$$

Principle of Explosion
"Ex falso quodlibet"

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



head-doesn't-exist : \neg ({A : Set} \rightarrow List $A \rightarrow A$) head-doesn't-exist head = head []



doesn't-exist : $\neg (\{A : Set\} \rightarrow List A \rightarrow A)$

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

Talking About Mathematics in a Programming
Language
Programming is Proving
Proofs are Programs

Proofs are Programs

Proofs are Programs

Types/Propositions are sets

data Bool : Set where

true : Bool false : Bool

Talking About Mathematics in a Programming

Language

Programming is Proving

Proofs are Programs

Types/Propositions are sets data Bool : Set where true : Bool false : Bool

Proofs are Programs

Proofs are Programs

Types/Propositions are sets

data Bool : Set where

true : Bool false : Bool

Inhabited by *proofs*

Bool Proposition true, false Proof

Talking About Mathematics in a Programming

Language

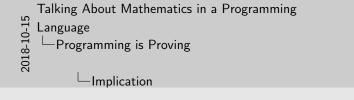
Programming is Proving

Proofs are Programs

Proofs are Programs

Types/Propositions are sets
data Bool Set where
true Bool
false Proposition
fram, false Proof

Implication



Just a function arrow



 $A \rightarrow B$

Talking About Mathematics in a Programming

Language

Programming is Proving

Implication



 $A \rightarrow B$ A implies B

Talking About Mathematics in a Programming
Language
Programming is Proving
Implication

A -- B A Implies B



Talking About Mathematics in a Programming Language —Programming is Proving

 $A \rightarrow B$

A implies B

Constructivist/Intuitionistic

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

☐ Implication

Booleans?

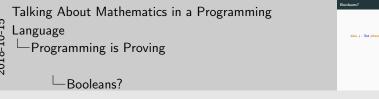
Talking About Mathematics in a Programming Language -Programming is Proving -Booleans?

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

data 1 : Set where Contradiction

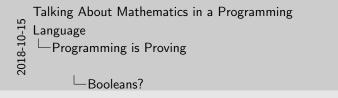


Falsehood (contradiction) is the proposition with no proofs. It's equivalent to what we had previously.

data ⊥ : Set where

Contradiction

```
ptb : \forall \{a\} \{A : Set a\} \rightarrow \neg A \rightarrow A \rightarrow \bot
ptb fx = fx
```



onlines ? $\frac{\text{data }_{1}:\text{Set where}}{\text{pib }_{1}:\text{Set where}} \qquad \text{Contradiction}$ $\text{pib }_{2}:\text{pib }_{2}:\text{pib }_{3}:\text{pib }_{4}:\text{pib }_{4}:\text{$

In fact, we can convert from what we had previously

```
data ⊥ : Set where
```

Contradiction

```
ptb : \forall {a} {A : Set a} \rightarrow \neg A \rightarrow A \rightarrow \bot ptb f x = f x lnc : \neg \bot lnc ()
```

Talking About Mathematics in a Programming
Language
Programming is Proving



And to what we had previously.

└─Booleans?

Here, we use an impossible pattern.

data ⊥ : Set where Contradiction

data ⊤ : Set where Tautology

tt : ⊤

Talking About Mathematics in a Programming Language
Programming is Proving
Booleans?

data 1: Set where Controdiction
data 1: Set where Tautology
11: 17

Tautology is kind of the "boring" type.

Talking About Mathematics in a Programming
Language
Programming is Proving
Conjunction

Conjunction ("and") is represented as a data type.

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

Talking About Mathematics in a Programming
Language
Programming is Proving
Conjunction

Conjunction

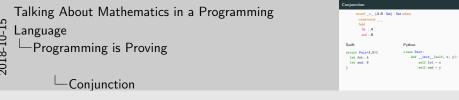
recod _ _ (A B : Set) : Set where
constructor ____
field

fir : A

set : B

It has two type parameters, and two fields.

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



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

```
record \times (A B : Set) : Set where
  constructor ,
  field
    fst: A
    snd: B
data \times (A B : Set) : Set where
  A \rightarrow B \rightarrow A \times B
```



—Conjunction

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"

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

Type Theory 2-Tuple

Talking About Mathematics in a Programming
Language
Programming is Proving
Conjunction



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

Set Theory Cartesian Product

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

Talking About Mathematics in a Programming

Language

Programming is Proving

Conjunction

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

Familiar identities: conjunction-elimination

cnj-elim :
$$\forall \{A B\} \rightarrow A \times B \rightarrow A$$

cnj-elim = fst $A \land B \implies A$

Talking About Mathematics in a Programming
Language
Programming is Proving
Conjunction

Conjunction $\max_{x \in A} (A \mid S \mid Sa) : Sat where constructive __ : fad \\ fad \\ fat : A \\ set : S \\ set : S \\ set : S \\ set : S \\ set : A \\ A \cap S \\ set : A \\ A \cap S \\ set : A \\ A \cap S \\$

Talking About Mathematics in a Programming

Language

Programming is Proving

Currying

Just a short note on currying.

curry :
$$\{A \ B \ C : \mathsf{Set}\} \to (A \times B \to C) \to A \to (B \to C)$$

curry $f \times y = f(x, y)$

Talking About Mathematics in a Programming

Language

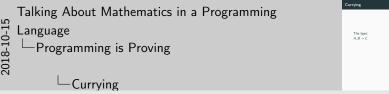
Programming is Proving

Currying

Just a short note on currying.

The type:

 $A, B \rightarrow C$



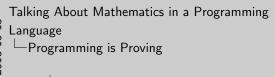
Just a short note on currying.

The type:

Is isomorphic to:

$$A, B \rightarrow C$$

$$A \rightarrow (B \rightarrow C)$$



is isomorphic to $A \rightarrow (B \rightarrow C)$

└─ Currying

Just a short note on currying.

The type: $A, B \rightarrow C$

Is isomorphic to:

$$A \rightarrow (B \rightarrow C)$$

Because the statement:

"A and B implies C"

Talking About Mathematics in a Programming
Language
Programming is Proving
Currying



Just a short note on currying.

The type: $A, B \rightarrow C$

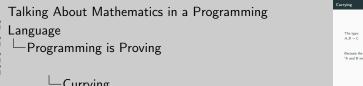
Is isomorphic to:

$$A \rightarrow (B \rightarrow C)$$

Because the statement:

"A and B implies C"

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



-Currying

 $A \rightarrow (B \rightarrow C)$

Just a short note on currying.

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

 $Outside \land Raining \implies Wet$

Talking About Mathematics in a Programming
Language
Programming is Proving
Currying

I'm outside and it's raining, I'm going to get wet" $\text{Outside} \wedge \text{Raining} \longrightarrow \text{Wet}$

Just a short note on currying.

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

$$Outside \land Raining \implies Wet$$

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

$$Outside \implies Raining \implies Wet$$

Talking About Mathematics in a Programming Language

—Programming is Proving

I'm outside and it's raining. I'm going to get w

Outside ∧ Raining → Wet

hen I'm outside, if it's raining I'm going to get

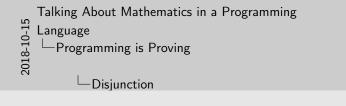
Outside → Raining → We

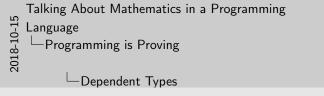
└ Currying

Just a short note on currying.

Disjunction

```
data \_\cup\_ (A B : Set) : Set where
inl : A \to A \cup B
inr : B \to A \cup B
```





Dependent Types

Everything so far has been non-dependent

Talking About Mathematics in a Programming

Language

— Programming is Proving

— Dependent Types

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

Everything so far has been non-dependent

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

Talking About Mathematics in a Programming Language

└─Programming is Proving

Everything so far has been non-dependent.
Proving things using this barra-bones toolbox is difficult (though possible)

-Dependent Types

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.

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

Talking About Mathematics in a Programming

Language

Programming is Proving

Dependent Types

The **□** Type

Talking About Mathematics in a Programming Language —Programming is Proving └─The Π Type

The II Type

The ∏ Type

Upgrade the function arrow

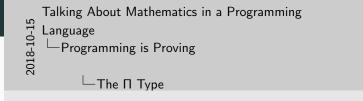
	TH: AL . M. I D	The II Type
0-15	Talking About Mathematics in a Programming	
	Language	Upgrade the function arrow
18-10	Programming is Proving	
201	└─The П Type	
	The TT Type	

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

The ∏ Type

Upgrade the function arrow

$$\mathsf{prop}: (x: \mathbb{N}) \to 0 \le x$$



The O Type $Upgrads the function arrow <math display="block">prop: \{x: T\} \to 0 \le x$

This lets us easily express *properties*



Talking About Mathematics in a Programming Language —Programming is Proving └─The Π Type

Upgrade the function arrow prop: $(x: \mathbb{N}) \to 0 \le x$ Now we have a proper ∀

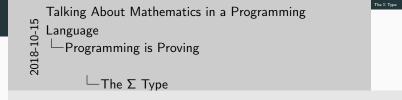
The IT Type

Upgrade the function arrow

$$\mathsf{prop}: (x: \mathbb{N}) \to 0 \le x$$

Now we have a proper \forall

The Σ Type



The Σ Type

Upgrade *product types*

Talking About Mathematics in a Programming

Language

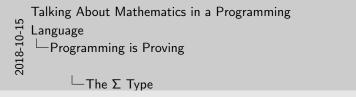
— Programming is Proving

— The Σ Type

The Σ Type

Upgrade product types

```
record NonZero : Set where field \begin{array}{ccc} n & : \mathbb{N} \\ proof : 0 < n \end{array}
```



The X Type

Upgrade product types

record Nucleon Set where
fall : N

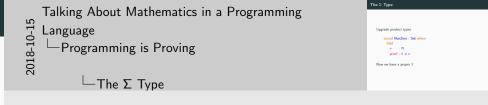
proof: 0 < 0

Later fields can refer to earlier ones.

The Σ Type

Upgrade product types

Now we have a proper \exists



The Equality Type

```
infix 4 = 
data =  \{A : Set\} (x : A) : A \rightarrow Set where
refl : x = x
```

Talking About Mathematics in a Programming Language

—Programming is Proving



Final piece of the puzzle.

The type of this type has 2 parameters.

The Equality Type

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

+ :
$$\mathbb{N} \to \mathbb{N} \to \mathbb{N}$$

 $0 + y = y$
 $\text{suc } x + y = \text{suc } (x + y)$
obvious : $\forall x \to 0 + x \equiv x$
obvious _ = refl

Talking About Mathematics in a Programming Language

—Programming is Proving



☐ Equality

Agda uses propositional equality

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

```
+ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}
0 + y = y
suc x + y = suc (x + y)
obvious : \forall x \rightarrow 0 + x \equiv x
obvious = refl
cong: \forall \{A B\} \rightarrow (f: A \rightarrow B) \rightarrow \forall \{x y\} \rightarrow x \equiv y \rightarrow f x \equiv f y
cong refl = refl
not-obvious : \forall x \rightarrow x + 0 \equiv x
not-obvious zero = refl
not-obvious (suc x) = cong suc (not-obvious x)
```

Talking About Mathematics in a Programming
Language
—Programming is Proving
—Equality

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

Talking About Mathematics in a Programming

Language

Programming is Proving

Open Areas and Weirdness

A Polynomial Solver

The *p*-Adics

Talking About Mathematics in a Programming Language The p-Adics \sqsubseteq The *p*-Adics