

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

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 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

2018-10-15

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!

Benefits For Programmers

- *Prove* things about code

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

vs

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

2018-10-15

Talking About Mathematics in a Programming Language

└ What do Programming Languages Have to do with Mathematics?

└ Benefits For Programmers

Benefits For Programmers

- Prove things about code

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

```
reverse-involution :  $\forall xs \rightarrow reverse (reverse xs) \approx xs$ 
```

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?

2018-10-15

Talking About Mathematics in a Programming Language

- └ What do Programming Languages Have to do with Mathematics?
- └ Benefits For Programmers

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

2018-10-15

Talking About Mathematics in a Programming Language

- └ What do Programming Languages Have to do with Mathematics?
 - └ Benefits For Programmers

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?

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

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?

2018-10-15

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.

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?

2018-10-15

Talking About Mathematics in a Programming Language

└ What do Programming Languages Have to do with 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

But what if our formal language is executable?

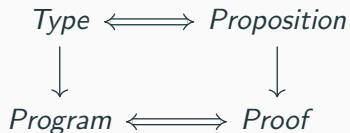
Can we write *verified* automated theorem provers?

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

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

Programming is Proving

The Curry-Howard Correspondence



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?

2018-10-15

Talking About Mathematics in a Programming Language

└ Programming is Proving

└ Types are Propositions

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.

2018-10-15

Talking About Mathematics in a Programming Language

└ Programming is Proving

└ Existential Proofs

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

So when you see:

$$x : \mathbb{N}$$

So when you see:

$$x : \mathbb{N}$$

Think:

$$\exists . \mathbb{N}$$

So when you see:

$x : \mathbb{N}$

Think:

$\exists.\mathbb{N}$

NB

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

So when you see:

$x : \mathbb{N}$

Think:

$\exists.\mathbb{N}$

NB

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

Proof is “by example”:

So when you see:

$$x : \mathbb{N}$$

Think:

$$\exists.\mathbb{N}$$

NB

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

Proof is “by example”:

$$x = 1$$

2018-10-15

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.

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

```
1
```

Programs are Proofs

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

Here's the type:

`head` : $\{A : \text{Set}\} \rightarrow \text{List } A \rightarrow A$

2018-10-15

Talking About Mathematics in a Programming Language

- └ Programming is Proving

- └ 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.

Equivalent in other languages:

Haskell

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

Swift

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


Equivalent in other languages:

Haskell

`head :: [a] -> a`

Swift

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

`head : {A : Set} → List 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 : Set} → List A → 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]) -> A {`

`head : {A : Set} → List A → A` “Takes a list of things, and
returns one of those things”.

The Proposition is False!

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

2018-10-15

Talking About Mathematics in a Programming Language

└ Programming is Proving

└ The Proposition is False!

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"
```

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

The Proposition is False!

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

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

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

$\text{fib} : \mathbb{N} \rightarrow \mathbb{N}$

$\text{fib } 0 = 0$

$\text{fib } (1 + 0) = 1 + 0$

$\text{fib } (1 + (1 + n)) = \text{fib } (1 + n) + \text{fib } n$

Talking About Mathematics in a Programming Language

└ Programming is Proving

└ But Let's Try Anyway!

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!

$\text{length} : \{A : \text{Set}\} \rightarrow \text{List } A \rightarrow \mathbb{N}$

$\text{length } [] = 0$

$\text{length } (x :: xs) = 1 + \text{length } xs$

2018-10-15

Talking About Mathematics in a Programming Language

└ Programming is Proving

└ But Let's Try Anyway!

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!

Here's a definition for `head`:

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

No!

For correct proofs, partial functions aren't allowed

2018-10-15

Talking About Mathematics in a Programming Language

└ Programming is Proving

└ But Let's Try Anyway!

But Let's Try Anyway!

Here's a definition for `head`:

`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

2018-10-15

Talking About Mathematics in a Programming Language

└ Programming is Proving

└ But Let's Try Anyway!

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”.

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

2018-10-15

Talking About Mathematics in a Programming Language

└ Programming is Proving

└ Correctness

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*.

2018-10-15

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?

2018-10-15

Talking About Mathematics in a Programming Language

└ Programming is Proving

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

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

Talking About Mathematics in a Programming Language

└ Programming is Proving

└ 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".

Principle of Explosion

“Ex falso quodlibet”

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

$\neg : \forall \{l\} \rightarrow \text{Set } l \rightarrow \text{Set } _$
 $\neg A = A \rightarrow \{B : \text{Set}\} \rightarrow B$

Principle of Explosion

“Ex falso quodlibet”

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

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

head-doesn't-exist *head* = *head* []

Talking About Mathematics in a Programming Language

└ Programming is Proving

```
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

2018-10-15

Talking About Mathematics in a Programming Language

└ Programming is Proving

└ Implication

Just a function arrow

Implication

$$A \rightarrow B$$

Implication

$A \rightarrow B$

A implies B

Implication

$A \rightarrow B$

A implies B

Constructivist/Intuitionistic

2018-10-15

Talking About Mathematics in a Programming Language

└ Programming is Proving

└ Implication

Implication

$A \rightarrow B$

$A \text{ implies } B$

Constructivist/Intuitionistic

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

Booleans?

2018-10-15

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*

Booleans?

data \perp : Set where

Contradiction

2018-10-15

Talking About Mathematics in a Programming Language

└ Programming is Proving

└ Booleans?

Booleans?

`data A : Set where`

`Contradiction`

Falsehood (contradiction) is the proposition with no proofs.

It's equivalent to what we had previously.

Booleans?

data \perp : Set where

Contradiction

ptb : $\forall \{a\} \{A : \text{Set } a\} \rightarrow \neg A \rightarrow A \rightarrow \perp$

ptb $f\ x = f\ x$

2018-10-15

Talking About Mathematics in a Programming Language

└ Programming is Proving

└ Booleans?

Booleans?

data \perp : Set where

Contradiction

pth : $\forall (a) (A : \text{Set } a) \rightarrow \neg A \rightarrow A \rightarrow \perp$

pth $f\ x = f\ x$

In fact, we can convert from what we had previously

Booleans?

data \perp : Set where

Contradiction

ptb : $\forall \{a\} \{A : \text{Set } a\} \rightarrow \neg A \rightarrow A \rightarrow \perp$

ptb $f\ x = f\ x$

Inc : $\neg \perp$

Inc ()

2018-10-15

Talking About Mathematics in a Programming Language

└ Programming is Proving

└ Booleans?

And *to* what we had previously.

Here, we use an impossible pattern.

Booleans?

data \perp : Set where

Contradiction

pth : $\forall \{a\} (A : \text{Set } a) \rightarrow \neg A \rightarrow A \rightarrow \perp$
pth $f x = f x$

inc : $\neg \perp$
inc ()

Booleans?

data \perp : Set where

Contradiction

data \top : Set where

tt : \top

Tautology

2018-10-15

Talking About Mathematics in a Programming Language

└ Programming is Proving

└ Booleans?

Booleans?

`data J : Set where`

Contradiction

`data T : Set where`

Tautology

`tt : T`

Tautology is kind of the “boring” type.

Conjunction

2018-10-15

Talking About Mathematics in a Programming Language

- └ Programming is Proving

- └ Conjunction

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

Conjunction

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

2018-10-15

Talking About Mathematics in a Programming Language

└ Programming is Proving

└ Conjunction

It has two type parameters, and two fields.

Conjunction

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

Conjunction

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

Swift

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

Python

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

2018-10-15

Talking About Mathematics in a Programming Language

└ Programming is Proving

└ Conjunction

Conjunction

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

Swift

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

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  $\_ \times \_$  (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
```


Talking About Mathematics in a Programming Language

└ Programming is Proving

└ Conjunction

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  $\times$  (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 :  $\forall \{A B\} \rightarrow A \times B \rightarrow A$ 
cnj-elim = fst
```

$$A \wedge B \implies A$$

Talking About Mathematics in a Programming Language

- └ Programming is Proving

- └ 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.

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

Talking About Mathematics in a Programming Language

└ Programming is Proving

└ Currying

curry : $\{A\ B\ C : \text{Set}\} \rightarrow (A \times B \rightarrow C) \rightarrow A \rightarrow (B \rightarrow C)$
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.

The type:

$$A, B \rightarrow C$$

2018-10-15

Talking About Mathematics in a Programming Language

└ Programming is Proving

└ Currying

Currying

The type
 $A, B \rightarrow C$

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.

The type:

$$A, B \rightarrow C$$

Is isomorphic to:

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

Talking About Mathematics in a Programming Language

└ Programming is Proving

└ Currying

The type
 $A, B \rightarrow C$

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

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.

The type:

$A, B \rightarrow C$

Is isomorphic to:

$A \rightarrow (B \rightarrow C)$

Because the statement:

“A and B implies C”

2018-10-15

Talking About Mathematics in a Programming Language

└ Programming is Proving

└ Currying

Currying

The type:
 $A, B \rightarrow C$

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

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.

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”

2018-10-15

Talking About Mathematics in a Programming Language

└ Programming is Proving

└ Currying

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.

“If I’m outside and it’s raining, I’m going to get wet”

$$Outside \wedge Raining \implies Wet$$

2018-10-15

Talking About Mathematics in a Programming Language

└ Programming is Proving

└ Currying

Currying

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

$Outside \wedge Raining \implies Wet$

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.

“If I’m outside and it’s raining, I’m going to get wet”

$$Outside \wedge Raining \implies Wet$$

“When I’m outside, if it’s raining I’m going to get wet”

$$Outside \implies Raining \implies Wet$$

2018-10-15

Talking About Mathematics in a Programming Language

└ Programming is Proving

└ Currying

Currying

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

$Outside \wedge Raining \implies Wet$

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

$Outside \implies Raining \implies Wet$

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
```


Everything so far has been non-dependent

2018-10-15

Talking About Mathematics in a Programming Language

└ Programming is Proving

└ Dependent Types

Dependent Types

Everything so far has been non-dependent

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

└ 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 exist, 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

The Π Type

The Π Type

Upgrade the *function arrow*

2018-10-15

Talking About Mathematics in a Programming Language

└ Programming is Proving

└ 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*

`prop : (x : \mathbb{N}) \rightarrow $0 \leq x$`

2018-10-15

Talking About Mathematics in a Programming Language

└ Programming is Proving

└ The Π Type

This lets us easily express *properties*

The Π Type

Upgrade the *function arrow*

`prop : (x : \mathbb{N}) \rightarrow $0 \leq x$`

The Π Type

Upgrade the *function arrow*

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

Now we have a proper \forall

The Σ Type

```
record NonZero : Set where  
  field  
    n :  $\mathbb{N}$   
    proof :  $0 < n$ 
```

2018-10-15

Talking About Mathematics in a Programming Language

└ Programming is Proving

└ The Σ Type

Now, we upgrade our product types.

Later fields can refer to earlier ones.

The Σ Type

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

The Equality Type

A Polynomial Solver

The p -Adics
