Talking About Mathematics in a Programming Language

Donnacha Oisín Kidney

October 17, 2018

What do we want from a Language for Mathematics?

Programming is Proving

A Polynomial Solver

The *p*-Adics

What do we want from a Language for Mathematics?

What do we want from a Language for Mathematics?

As it turns out, the languages we use for maths already look a little like programming languages.

In designing them we encounter a lot of the same goals.

What do we want from a Language for Mathematics?

A *Syntax* that is

- Readable
- Precise
- Terse

What do we want from a Language for Mathematics?

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?

Reason 1: Because Computer-Assisted Proofs are “Bad”

Aren’t computer-assisted proofs bad? Inelegant? Less rigorous?

Reason 1: Because Computer-Assisted Proofs are “Bad”

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

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

The famous example is the four-colour map theorem.

First major mathematical proof which relied heavily on computer assistance.

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

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

Reason 1: Because Computer-Assisted Proofs are “Bad”

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

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

- Non-Surveyable

The proof is too large for another mathematician to check its work! (that is, after all, why a computer was used)

Reason 1: Because Computer-Assisted Proofs are “Bad”

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

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

- Non-Surveyable
- Doesn’t Provide Insight

This is maybe an aesthetic concern, but the prevailing attitude is that a non computer-assisted proof would provide a deeper understanding of the problem, and more general tools to be used later, rather than a simple statement “yes the proposition is true” or “no it’s false”.

Of course, in practice, working a proof to a level where it becomes solvable via a computer requires insight in and of itself, but perhaps less insight than another method.

Reason 1: Because Computer-Assisted Proofs are "Bad"

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

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

- Non-Surveyable
- Doesn't Provide Insight
- Requires Trust

We have to believe that the program used to prove the proposition doesn't contain bugs!

Reason 1: Because Computer-Assisted Proofs are "Bad"

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

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

- Non-Surveyable
- Doesn't Provide Insight
- Requires Trust

Did contain bugs!

Although they weren't critical to the correctness.

This is Different

This is Different

We're looking for a core set of axioms/semantics and a syntax to talk about mathematics

We're going to use PL theory to help get us there

This is Different

We’re looking for a core set of axioms/semantics and a syntax to talk about mathematics

If we do it right, it should be so simple that “even a computer could understand it”

But this is incidental!

The real work is in finding the language that works.

Even now, most compilers for these languages are grad students!

This is Different

We're looking for a core set of axioms/semantics and a syntax to talk about mathematics

If we do it right, it should be so simple that "even a computer could understand it"

Why would we want a computer to understand our language?

This is Different

We're looking for a core set of axioms/semantics and a syntax to talk about mathematics

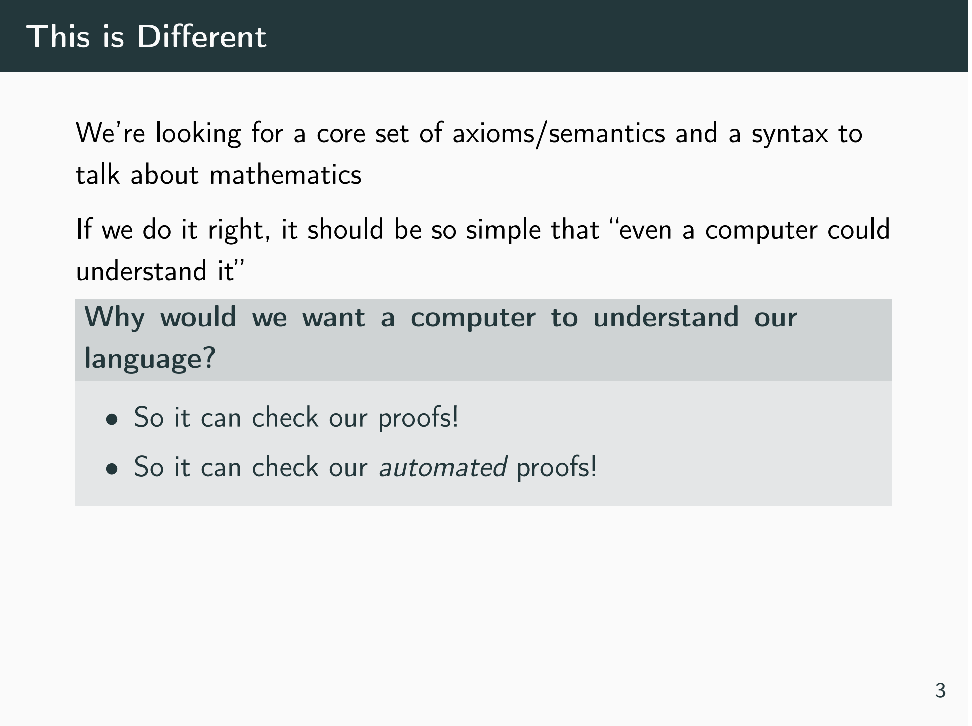
If we do it right, it should be so simple that "even a computer could understand it"

Why would we want a computer to understand our language?

- So it can check our proofs!

If a machine can read your proofs, then it can *check* your proofs.

This adds a level of rigour that you just don't get with handwritten proofs.



This is Different

We're looking for a core set of axioms/semantics and a syntax to talk about mathematics

If we do it right, it should be so simple that "even a computer could understand it"

Why would we want a computer to understand our language?

- So it can check our proofs!
- So it can check our *automated* proofs!

A perfect candidate for the kinds of proofs we'd like a machine to check *are* computer-assisted proofs.

Remember, our language is a programming language: write the automated theorem prover in it, and then *verify* the theorem prover in it!

This is Different

We're looking for a core set of axioms/semantics and a syntax to talk about mathematics

If we do it right, it should be so simple that "even a computer could understand it"

Why would we want a computer to understand our language?

- So it can check our proofs!
- So it can check our *automated* proofs!

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

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

Unfortunately, this is still difficult to do

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

Reason 2: Haven't We Tried This Before?

Lawrence C Paulson. The Future of Formalised Mathematics, 2016

Fully formalizing mathematics from the ground-up has long been a goal. (Hilbert)

Haven't other attempts failed?

Reason 2: Haven’t We Tried This Before?

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

Lawrence C Paulson. The Future of Formalised Mathematics, 2016

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

Is a formalization really going to be *this* tedious?

Reason 2: Haven’t We Tried This Before?

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

Gödel showed that universal formal systems are incomplete

Lawrence C Paulson. The Future of Formalised Mathematics, 2016

Besides—hasn’t it been shown to be impossible, anyway?

Reason 2: Haven't We Tried This Before?

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

Formal systems have improved

Gödel showed that universal formal systems are incomplete

Lawrence C Paulson. The Future of Formalised Mathematics, 2016

We have much better formalisms now.

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

Reason 2: Haven't We Tried This Before?

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

Formal systems have improved

Gödel showed that universal formal systems are incomplete

We don't need universal systems

Lawrence C Paulson. The Future of Formalised Mathematics, 2016

A universal system is too powerful—we can get by with less.

Law of the Excluded Middle

Why Would a Programmer Want to Use this Language?

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

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

Why Would a Programmer Want to Use this Language?

- *Prove* things about code

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

vs

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

Not just test!

Why Would a Programmer Want to Use this Language?

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

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

Why Would a Programmer Want to Use this Language?

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

Programming is Proving

The Curry-Howard Correspondence

Philip Wadler. Propositions As Types.

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

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

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

The Curry-Howard Correspondence

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

Philip Wadler. Propositions As Types.

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

Here's the high-level overview.

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

Types are Propositions

Types are (usually):

- Int
- String
- ...

How are these propositions?

Propositions are things like “there are infinite primes”, etc. Int certainly doesn’t *look* like a proposition.

Existential Proofs

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

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 : 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"
```

```
head : {A : Set} → List A → A
```

The Proposition is False!

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

head : {A : Set} → List A → 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

```
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!

```
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!

```
head (x :: xs) = x
```

Here's a candidate definition for head.

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

So how do we disallow it?

But Let's Try Anyway!

```
head (x :: xs) = x
```

Rule 1

No partial functions

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

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

But Let's Try Anyway!

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

Rule 1

No partial functions

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

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

But Let's Try Anyway!

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

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

Rule 1

No partial functions

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

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

But Let's Try Anyway!

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

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

Rule 1

No partial functions

Rule 2

All programs are total

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

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

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

Turing Completeness

Have we just thrown out Turing completeness?

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

Well, no...

Turing Completeness

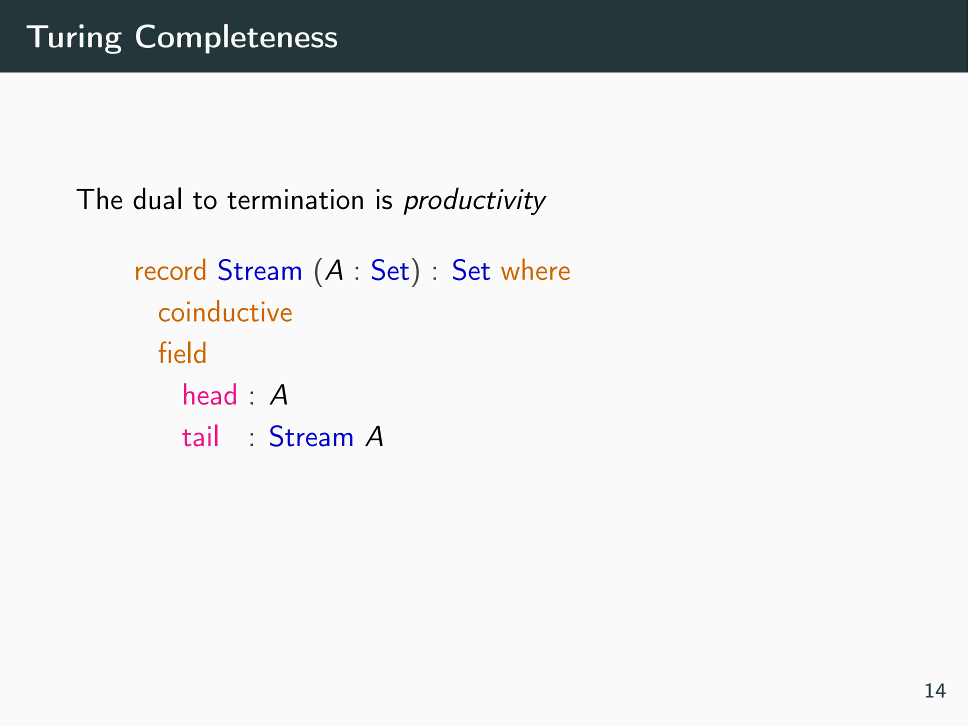
The dual to termination is *productivity*

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

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

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

Agda can check for productivity, too.



Turing Completeness

The dual to termination is *productivity*

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

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

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

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

Turing Completeness

The dual to termination is *productivity*

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

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

Enough Restrictions!

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

How about something that we can?

How about the converse?

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

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

Falsehood

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

Falsehood

Principle of Explosion

“Ex falso quodlibet”

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

Falsehood

```
¬ : ∀ {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.

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

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

Proofs are Programs

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

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

Proofs are Programs

Types/Propositions are *sets*

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

Proofs are Programs

Types/Propositions are *sets*

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

Inhabited by *proofs*

Bool	Proposition
true, false	Proof

Implication

Just a function arrow

Implication

$A \rightarrow B$

Implication

$A \to B$

A implies B

Implication

$A \to B$

A implies B

Constructivist/Intuitionistic

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

Booleans?

We *don't* use bools to express truth and falsehood.
Bool is just a set with two values: nothing “true” or “false” about either of them!

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

Booleans?

```
data ⊥ : Set where
```

Contradiction

Falsehood (contradiction) is the proposition with no proofs.

It's equivalent to what we had previously.

Booleans?

```
data ⊥ : Set where
```

Contradiction

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

In fact, we can convert from what we had previously

Booleans?

Contradiction

```
data ⊥ : Set where


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


  not-false : ¬ ⊥
  not-false ()
```

And *to* what we had previously.

Here, we use an impossible pattern.

Booleans?

```
data ⊥ : Set where
```

Contradiction

```
data ⊤ : Set where
  tt : ⊤
```

Tautology

Tautology is kind of the “boring” type.

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

```
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 _×_ (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

```
curry : {A B C : Set} → (A × B → C) → A → (B → 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.

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:
$A, B \to C$

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

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"

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

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*

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

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

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

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