Programming Mathematics in Agda

Donnacha Oisín Kidney

October 13, 2018

Programming is Proving

A Polynomial Solver

The p-Adics

Programming is Proving

The Curry-Howard Correspondence

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

Philip Wadler. Propositions As Types.

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

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

Central idea of dependent types for proofs.We start by showing that the left-hand-side translates to the right.

Types are Propositions

Types are (usually):

- Int
- String
- ...

How are these 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.

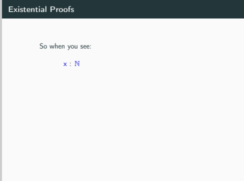
Existential Proofs

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

Existential Proofs

So when you see:

$$x : \mathbb{N}$$



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

Existential Proofs

So when you see:

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

Think:

$\exists.\mathbb{N}$

So when you see:	Think:
x : ℕ	∃.ℕ

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

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 : ℕ	∃.ℕ

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

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

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 : ℕ	∃.ℕ

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

Proof is "by example":

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

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":

$$x = 1$$

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
```

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

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
```

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

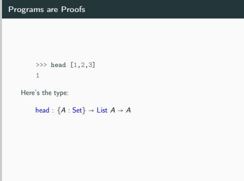
Let's start working with a function as if it were a proof. The function we'll choose gets the first element from a list. It's commonly called "head" in functional programming.

Programs are Proofs

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

Here's the type:

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



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.

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

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.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 {`

Equivalent in other languages:

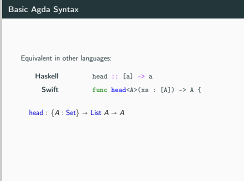
Haskell	`head :: [a] -> a`
Swift	`func head<A>(xs : [A]) -> A {`

head is what would be called a "generic" function in languages like Java.In other words, the type A is not specified in the implementation of the function.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 {`

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



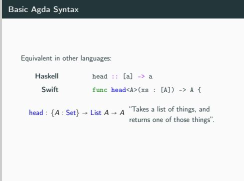
head is what would be called a “generic” function in languages like Java.In other words, the type A is not specified in the implementation of the function.In Agda, you must supply the type to the function: the curly brackets mean the argument is implicit.

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”.



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

The Proposition is False!

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

The Proposition is False!

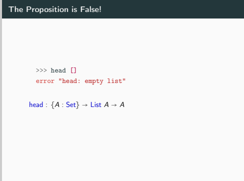
```
>>> head []
error "head: empty list"
```

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"
```

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



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"
```

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

False

The Proposition is False!

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

```
head : {A : Set} → List A → A
```

False

head isn't defined on the empty list, so the function *doesn't* exist. In other words, its type is a false proposition.

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! i

Function definition syntax

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

But Let's Try Anyway! i

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.For lists, we also have two patterns: the empty list, and the head element followed by the rest of the list.We need to disallow functions which don't match all patterns. Array access out-of-bounds, etc., also not allowed.To disallow *this* kind of thing, we must ensure all functions are *total*. For now, assume this means "terminating".

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

Here's a definition for head:

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

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

But Let's Try Anyway! iv

We're not out of the woods yet:

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

No!
For correct proofs, all functions must be total

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

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

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

Enough Restrictions!

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

How about something that we can?

How about the converse? 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"

Principle of Explosion
"Ex falso quodlibet"

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

Falsehood

Principle of Explosion
"Ex falso quodlibet"
If you stand for nothing, you'll fall for anything.

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

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

Principle of Explosion
"Ex falso quodlibet"

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

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.

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"

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

```
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.

A Polynomial Solver

The p-Adics