Dependently Typed Programming

The Curry-Howard Correspondence

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

Philip Wadler. Propositions As Types.

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

Types are (usually):

- Int
- String
- ...

How are these propositions?

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"

$x = 1$

Example "Proof"

Let's start working with a function as if it were a proof.

The example function we'll choose gets the first element of a list and returns it (commonly called head in functional programming languages).

Here's the type:

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

Basic 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: it just “takes a list of things, and returns one of those things”.

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

The Proposition is False!

What happens if we call head on an empty list? In this case, head isn't defined.

In other words, the proposition:

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

Is *False*.

We shouldn't be able to prove this using Agda.

But Let's Try Anyway i

Agda functions are defined (usually) with *pattern-matching*.

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

For the natural numbers, we use the Peano numbers, which gives us 2 patterns: zero, and successor.

But Let's Try Anyway ii

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

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

But Let's Try Anyway iii

For head, then, we can just write the following:

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

No!
Partial functions aren't allowed!

But Let's Try Anyway iv

It might seem like we can't write this function, then, but there is one more way we could get around it.

```
head [] = head []
```

To disallow *this* kind of thing, we must ensure all functions are *total*. For now, assume this means "terminating".

So that's all well and good: there's a proposition which we know isn't true, and we've tried to prove it, and failed (as well we should).

Let's go a little further, though: can we *prove* that head doesn't exist?

Falsehood

Often it's said that you can't prove negatives in dependently typed programming: not true!

In our case, we'll use the principle of explosion.

Principle of Explosion
"Ex falso quodlibet": from falsehood, anything.

In Agda:

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

So let's supply a proof of that fact!

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

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