Introduction to Dependent Types

Eagan Technology Unconference

Joseph Ching

September 22, 2015

Table of Contents

Table of Contents

Table of Contents

Quick Question

How many are familiar with this topic?

A Joke

This is not a `m-` tutorial.

A Joke

This is not a `m-` tutorial.

Nor is it a `lens` tutorial

A Joke

This is not a `m-` tutorial.
Nor is it a `lens` tutorial (aka the new new `m-` tutorial...

A Joke

This is not a `m-` tutorial.
Nor is it a `lens` tutorial (aka the new new `m-` tutorial. . .
. . . because `arrows` *were* the new `m-` tutorials).

About This Talk

`Agda`, `Idris`, `Coq` and co* have full support for dependent types.

About This Talk

`Agda`, `Idris`, `Coq` and co* have full support for dependent types. Because of that, it's harder to see the build up, so we won't be directly using them in this talk.

About This Talk

`Agda`, `Idris`, `Coq` and co* have full support for dependent types. Because of that, it's harder to see the build up, so we won't be directly using them in this talk.

Honestly though, it's because they're way over my head :(

(*) There was another mini joke here. . .

About This Talk

But we will be using `Haskell` though :)

About This Talk

But we will be using `Haskell` though :)
It's not truely dependent, but we can do more and more with each `language extension` that comes along.
For the examples, there also will be loose translation to imperative/OOP; though please keep in mind that they are not the same thing at all.

Test

Syntax highlighting test reference, to be removed later.

```
-- Comment
data Maybe a = Nothing | Just a
               deriving (Show, Eq)

fmap :: Functor f => (a -> b) -> f a -> f b
map _ []     = []
map f (x:xs) = f x : map f xs

type family TF a :: *
type instance TF Int = Bool
```

Test

Couldn't quite yet get listing to work with overlay yet.

```
{- block comment -}
foo :: Bool -> Int -> String
foo False 0 = "Bad"
foo True  0 = "Questionable"
foo False n = "Fake"
foo True  n = "Read"
```

Test

Pausing within listing is ok?

```
{-# LANGUAGE KitchenSink #-}
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

Test

Pausing within listing is ok?

```
{-# LANGUAGE KitchenSink #-}
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith _ []       _       = []
```

Test

Pausing within listing is ok?

```
{-# LANGUAGE KitchenSink #-}
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith _ []       _        = []
zipWith _ _        []       = []
```

Test

Pausing within listing is ok?

```
{-# LANGUAGE KitchenSink #-}
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith _ []     _      = []
zipWith _ _      []     = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

better yet

```
{-# LANGUAGE KitchenSink #-}
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith _ _      _      = []
```

Values and Types

Values has Types, or Values are classified by Types.

```
..., -1, 0, 1, 2, 3, ... :: Int
```

Values and Types

Values has Types, or Values are classified by Types.

```
..., -1, 0, 1, 2, 3, ... :: Int
True, False :: Bool
```

Values and Types

Values has Types, or Values are classified by Types.

```
..., -1, 0, 1, 2, 3, ... :: Int
True, False :: Bool
'a', 'b', 'c' :: Char
```

Values and Types

Values has Types, or Values are classified by Types.

```
..., -1, 0, 1, 2, 3, ... :: Int
True, False :: Bool
'a', 'b', 'c' :: Char
"abc" :: String ~ [Char]
```

Values are also called Terms

About Types

How are the types defined?

- Some are built in magic: Int, Char, functions

About Types

How are the types defined?

- Some are built in magic: Int, Char, functions
- Some are built in sugar: list, tuples
  - We can still define these ourselves without the sugar

About Types

How are the types defined?

- Some are built in magic: Int, Char, functions
- Some are built in sugar: list, tuples
    - We can still define these ourselves without the sugar
- Rest can be user defined: Bool, String, Maybe

Empty and Unit Types

Define new data type with data.

Empty and Unit Types

Define new data type with data.

- Left hand side (LHS) - Type constructor

Empty and Unit Types

Define new data type with data.

- Left hand side (LHS) - Type constructor
- Right hand side (RHS) - Value constructor

Empty and Unit Types

Define new data type with data.

- Left hand side (LHS) - Type constructor
- Right hand side (RHS) - Value constructor

```
data Bool = False | True
```

Here, Bool is the Type constructor, True and False are Value constructors.

Does this remind you of anything?

Empty and Unit Types

Define new data type with data.

- Left hand side (LHS) - Type constructor
- Right hand side (RHS) - Value constructor

```
data Bool = False | True
```

A loose translation:

```
enum Bool { False, True }
```

Sum Types

Simply, Types with more than one constructors.

Sum Types

Simply, Types with more than one constructors.

```
data Bool = False | True
data Weekdays = Sunday | Monday | Tuesday | Wednesday
              | Thursday | Friday | Saturday
```

Sum Types

Simply, Types with more than one constructors.

```
data Bool = False | True
data Weekdays = Sunday | Monday | Tuesday | Wednesday
              | Thursday | Friday | Saturday
```

Can parametrize over another type:

Sum Types

Simply, Types with more than one constructors.

```
data Bool = False | True
data Weekdays = Sunday | Monday | Tuesday | Wednesday
              | Thursday | Friday | Saturday
```

Can parametrize over another type:

```
data Identity a = Identity a
```

Sum Types

Simply, Types with more than one constructors.

```
data Bool = False | True
data Weekdays = Sunday | Monday | Tuesday | Wednesday
              | Thursday | Friday | Saturday
```

Can parametrize over another type:

```
data Identity a = Identity a
```

A *very* loose translation (assuming capitalization implies constructor):

```
enum Identity<T> {
  Identity(T t)
}
```

Sum Types

Simply, Types with more than one constructors.

```
data Bool = False | True
data Weekdays = Sunday | Monday | Tuesday | Wednesday
              | Thursday | Friday | Saturday
```

Can parametrize over another type:

```
data Identity a = Identity a
```

And its Type:

```
Identity :: a -> Identity t
```

Sum Types

Another example:

```
data Maybe a = Nothing | Just a
```

Sum Types

Another example:

```
data Maybe a = Nothing | Just a
```

The *very* loose translation:

```
enum Maybe<T> {
  Nothing,
  Just(T t)
}
```

Sum Types

Another example:

```
data Maybe a = Nothing | Just a
```

The *very* loose translation:

```
enum Maybe<T> {
  Nothing,
  Just(T t)
}
```

The Types of the two Value constructors:

```
Nothing :: Maybe a
Just    :: a -> Maybe a
```

Sum Types

A more involved example with Either:

```
data Either a b = Left a | Right b
```

Sum Types

A more involved example with Either:

```
data Either a b = Left a | Right b
```

Another *very* loose translation:

```
enum Either<T1, T2> {
  Left(T1 t1),
  Right(T2 t2)
}
```

Sum Types

A more involved example with Either:

```
data Either a b = Left a | Right b
```

Another *very* loose translation:

```
enum Either<T1, T2> {
  Left(T1 t1),
  Right(T2 t2)
}
```

The two Value constructors have Types:

```
Left  :: a -> Either a b
Right :: b -> Either a b
```

Product Types

Types with Recursion

Phantom Types

Language Extension - GADTs

Type Synonyms

Functions

Higher-order Functions

Questions?