Introduction to Dependent Types

Eagan Technology Unconference

Joseph Ching

September 22, 2015

Outline

Outline

Outline

Outline

Outline

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 `mtl` tutorial.
Nor is it a `lens` tutorial (aka the new new `mtl` 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.

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 will also be *very* loose translation to imperative/OOP. Though please keep in mind that these are merely syntax translations, the actual concepts can differ vastly.

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 data types defined?

About Types

How are data types defined?

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

About Types

How are data types defined?

- Some are built in magic: Int, Char, functions
- Some are built in sugar: list, tuples
  - We can define equivalent non-sugar-ed version ourselves

About Types

How are data types defined?

- Some are built in magic: Int, Char, functions
- Some are built in sugar: list, tuples
  - We can define equivalent non-sugar-ed version ourselves
- Rest can be user defined: Bool, String, Maybe

About Types

What does the data structure looks like?

About Types

What does the data structure looks like?

- Simple

About Types

What does the data structure looks like?

- Simple
- Multiple Value constructors

About Types

What does the data structure looks like?

- Simple
- Multiple Value constructors
- Paremetrize over another type

About Types

What does the data structure looks like?

- Simple
- Multiple Value constructors
- Paremetrize over another type
- Recursive

Defining Data Types

Define new data type with data.

Defining Data Types

Define new data type with data.

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

Defining Data Types

Define new data type with data.

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

Type constructors and Value constructors are capticalized.

Our First Example!

Define a person:

```
-- | params for firstname, lastname, age respectively
data Person = Person String String Int
```

Our First Example!

Define a person:

```
-- | params for firstname, lastname, age respectively
data Person = Person String String Int
```

A loose translation:

```
enum Person {
  Person(String firstname, String lastname, Int age)
}
```

Our First Example!

Define a person:

```
-- | params for firstname, lastname, age respectively
data Person = Person String String Int
```

A loose translation:

```
enum Person {
  Person(String firstname, String lastname, Int age)
}
```

In this example, the Type and Value constructor have the same name. The Type of the Person constructor:

```
Person :: String -> String -> Int -> Person

bobby :: Person
bobby = Person "Bobby" "Smith" 23
```

Multiple Value Constructors

Data can have multiple Value constructors:

```
data Bool = False | True

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

Does this remind you of anything?

Multiple Value Constructors

Data can have multiple Value constructors:

```
data Bool = False | True

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

Does this remind you of anything?

A loose translation:

```
enum Bool { False, True }

enum Weekdays {
  Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,
      Saturday
}
```

Multiple Value Constructor

You can do type aliasing with type:

```
type Side = Double
type Radius = Double
```

Multiple Value Constructor

You can do type aliasing with type:

```
type Side = Double
type Radius = Double
```

For example:

```
data Shape = Triangle Side Side Side
           | Rectangle Side Side
           | Circle Radius
```

Multiple Value Constructor

You can do type aliasing with type:

```
type Side = Double
type Radius = Double
```

For example:

```
data Shape = Triangle Side Side Side
           | Rectangle Side Side
           | Circle Radius
```

A loose translation:

```
enum Shape {
  Triangle(Double side1, Double side2, Double side3),
  Rectangle(Double length, Double width),
  Circle(Double radius)
}
```

Multiple Value Constructor

Recall Side $\sim$ Radius $\sim$ Double:

```
data Shape = Triangle Side Side Side
           | Rectangle Side Side
           | Circle Radius
```

Multiple Value Constructor

Recall Side ∼ Radius ∼ Double:

```
data Shape = Triangle Side Side Side
           | Rectangle Side Side
           | Circle Radius
```

Types of the Value constructors:

```
Triangle  :: Side -> Side -> Side -> Shape
Rectangle :: Side -> Side -> Shape
Circle    :: Radius -> Shape
```

Multiple Value Constructor

Recall Side ∼ Radius ∼ Double:

```
data Shape = Triangle Side Side Side
           | Rectangle Side Side
           | Circle Radius
```

Types of the Value constructors:

```
Triangle  :: Side -> Side -> Side -> Shape
Rectangle :: Side -> Side -> Shape
Circle    :: Radius -> Shape
```

Example Shapes:

```
myTri, myRect, myCir :: Shape
myTri  = Triangle 2.1 3.2 5
myRect = Rectangle 4 4
myCir  = Circle 7.2
```

Parametrization

Types can parametrize over another type:

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

Parametrization

Types can parametrize over another type:

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

A loose translation:

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

Parametrization

Types can parametrize over another type:

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

A loose translation:

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

The Type of the Identity constructor:

```
Identity :: a -> Identity a

intIdwrtSum :: Indentity Int
intIdwrtSum = Identity 0
```

Tuple

Parametrize over 2 types - 2-tuple!

```haskell
data Tuple a b = Tuple a b
-- Actual built-in sugared version is something like:
-- data (,) a b = (a, b)
```

Tuple

Parametrize over 2 types - 2-tuple!

```
data Tuple a b = Tuple a b
-- Actual built-in sugared version is something like:
-- data (,) a b = (a, b)
```

A loose translation:

```
enum Tuple<T1, T2> {
  Tuple(T1 t1, T2 t2)
}
```

Tuple

Parametrize over 2 types - 2-tuple!

```haskell
data Tuple a b = Tuple a b
-- Actual built-in sugared version is something like:
-- data (,) a b = (a, b)
```

A loose translation:

```
enum Tuple<T1, T2> {
  Tuple(T1 t1, T2 t2)
}
```

With:

```haskell
Tuple :: a -> b -> Tuple a b
```

Tuple

Use the default sugared version:

```
data (,) a b = (a, b)
```

Tuple

Use the default sugared version:

```
data (,) a b = (a, b)
```

An example:

```
type Employed = Bool

barbara, chet, luffy :: (Person, Employed)
barbara = (Person "Barbara" "Sakura" 30, True)
chet    = (Person "Chet" "Awesome-Laser" 2, False)
luffy   = (Person "Luff D." "Monkey" 19, False)
```

Maybe

Like Bool, but parametrize an *a* over the True part:

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

Maybe

Like Bool, but parametrize an *a* over the True part:

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

A loose translation:

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

Maybe

Like Bool, but parametrize an *a* over the True part:

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

A loose translation:

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

The Types of the two Value constructors:

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

Maybe

From previous slide:

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

Maybe

From previous slide:

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

Say more with the occupation:

```
type Occupation = Maybe String

barbara, chet, luffy :: (Person, Occupation)
barbara = (Person "Barbara" "Sakura" 30, Just "dancer")
chet    = (Person "Chet" "Awesome-Laser" 2, Nothing)
luffy   = (Person "Luff D." "Monkey" 19, Just "pirate")
```

Either

Like Bool, but parametrize over both True and False:

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

Either

Like Bool, but parametrize over both True and False:

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

A loose translation:

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

Either

Like Bool, but parametrize over both True and False:

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

A 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
```

Either

From previous slide:

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

Either

From previous slide:

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

Refine with more details:

```
type Earning = Either String Int

barbara, chet, luffy :: (Person, Earning)
barbara = (Person "Barbara" "Sakura" 30,
           Right 100000)
chet    = (Person "Chet" "Awesome-Laser" 2,
           Left "Is a baby")
luffy   = (Person "Luff D." "Monkey" 19,
           Right 2000000)
```

Types with Recursion

Phantom Types

Functions

Higher-order Functions

Lambda Cube

Kinds

GADTs

KindSignatures

ConstraintKinds

Type Operators

DataKinds

Type Families

Questions?