Haskell's Types

Mihai Maruseac

8.07.2013

What's in a Type?

- *syntactic-wise*: label associated with variable
- *value space*: **set** of values with **same** properties
  - maximal set of values.

Basic Types

```
3 :: Int
120894192912541294198294982 :: Integer
'a' :: Char
"Winter is Coming" :: String
True :: Bool
```

Compound Types

```
[3, 4, 5] :: [Int]
['a', 'b'] :: [Char] -- String
[[3,4], []] :: [[Int]]
() :: ()
(3, "Winter is Coming") :: (Int, String)
(3, True, 3.4, "It's July") :: (Int, Bool, Float, [Char])
```

Functions and Type Variables (1)

```
f x = x
```

```
f :: Int -> Int
```

Why not:

```
f True
```

Functions and Type Variables (2)

```
f x = x

f :: a -> a
```

- capital letter -> proper type
- small letter -> type variable (stands for multiple types)

Functions and Type Variables (3)

- remember that functions are curry

```
a -> b -> c
a -> (b -> c) -- same as above
(a -> b) -> c -- different type
```

Types as documentation (1)

- Which functions have type `a -> a`?
  - excluding dummy cases
    - `f x = error "undefined"`
    - `f x = undefined`
    - `f x = f x`
  - want the simplest possible expression
    - `f x = head [...]`
  - a single solution:

```
f x = x
```

Types as documentation (2)

- Which functions have type `(a, b) -> a`?
  - a single solution:

```
f (x, y) = x
```

Hayoo, Hoogle

- hoogle
- hayoo

Seeing Types in GHCI

- :t expression

Programmer Defined Types (1)

- how is a String a synonym of [Char]?

```
type String = [Char]

type Point = (Double, Double)
type Size = (Double, Double)
type Vector = (Double, Double)

area :: Size -> Double
area (x, y) = x * y

p :: Point
p = (3.14, 4.2)

area p
```

Programmer Defined Types (2)

- type synonyms help only at lexical level
- compiler sees the base type and works with it

Programmer Defined Types (3)

- how do we build our own types?

```
data Colour = Red | Green | Blue |
    Pink | Gray | ...

data Bool = True | False
```

- a type is a collection of constructors
- constructors start with capital letters

Programmer Defined Types (4)

- constructors with fields

```
data Person
    = Male String Int -- name and age
    | Female String String Int -- name, maiden name and age
```

- constructors are functions

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

Programmer Defined Types (5)

- generics (type variables in constructors, `[a]`)

```
data Container a = Empty | Holding a
```

Interesting Haskell Types

```haskell
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b
```

Trees (Recursive Data Types)

```
data BinaryLeafTree a = Leaf a
    | Node (BinaryLeafTree a) (BinaryLeafTree a)
data BinaryTree a = Leaf' a
    | Node' (BinaryTree a) a (BinaryTree a)
data RoseTree a = Leaf'' a
    | Node'' a [RoseTree a]
```

Working with Programmer Defined Types

```haskell
data Container a = Empty | Holding a

isEmpty :: Container a -> Bool
isEmpty Empty = True
isEmpty _ = False
-- isEmpty (Holding _ ) = False

place :: Container a -> a -> Container a
place Empty x = Holding x
place _ _ = error "Container already full"
```

Space Optimization

- my type has one constructor with one field
- `data` needs too much extra-memory (constructor tags, thunk tags)
- `type` doesn't offer type safety
- have the cookie and eat it too

```haskell
newtype OneFieldOnly a = Constructor a
```

Record Types (1)

```
data Person = P String String String
    Int Int Person Person
```

- which one is name of father? (can use type synonims)
- change order by mistake
- add another parameter

Record Types (2)

```
data Person = P
    { name :: String
    , address :: String
    , nationality :: String
    , age :: Int
    , number_of_children :: Int
    , father :: Person
    , mother :: Person
    }
```

- each field is a function

```
name :: Person -> String
P :: String -> String -> String -> Int -> Int ->
    Person -> Person -> Person
```

Polymorphism (1)

- want to convert type to `String`
- `toString` in Java
- `show :: a -> String`

```
class Show a where
    show :: a -> String

data Color = Red | Green | Blue deriving Show

data Container a = Empty | Holding a

instance Show (Container Int) where
    show (Holding x) = "[ " ++ show x ++ " ]"
    show _ = "[ ]"
```

Polymorphism (2)

```
class Num a where
    (+), (-), (*) :: a -> a -> a
    negate :: a -> a
    abs :: a -> a
    signum :: a -> a
    fromInteger :: Integer -> a

    x - y = x + negate y
    negate x = 0 - x
```

- define all but one of `(-)` and `negate`

Polymorphism (3)

```
class Eq a => Ord a where
    compare :: a -> a -> Ordering
    (<), (<=), (>), (>=) :: a -> a -> Bool
    max, min :: a -> a -> a

data Ordering = LT | EQ | GT
```

Polymorphism (4)

- we want a `map`-like operation on trees

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b

instance Functor [] where
    fmap = map
```

Polymorphism (4)

- we want a map-like operation on trees
- we want a map-like operation on containers

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b

instance Functor [] where
    fmap = map
```

Polymorphism (4)

- we want a `map`-like operation on trees
- we want a `map`-like operation on containers
- we want a function `f :: (a -> b) -> c a -> c b` where `c` is a container-type

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b

instance Functor [] where
    fmap = map
```

Typeclasses

- capture common operations (polymorphism)
- capture common patterns (map, errors, etc.)

Hands-On

- Eq instance for `Person`

```
instance Eq Person where
  p1 == p2 =
    name p1 == name p2 &&
    age p1 == age p2
```