COMP3258

Functional Programming

Tutorial Session 5: Datatypes

Table of Contents

- Datatypes (Data, Types, Kinds)
- ~~Classes~~
- Folding Over Datatypes

Review

- Type Declarations
- Data Declarations
- Newtype Declarations

Type Declarations

```
type String    = [Char]
type Pos       = (Int, Int)
type Trans     = Pos → Pos

type Pair a    = (a, a)
type Assoc k v = [(k, v)]
```

- Use type keyword to declare a new type
- Use type constructor to construct a type
    - 0-argument type constructor
    - n-argument type constructor

Type Declarations (Kinds)

- Types have kinds (the type of types)
- Use :k or :kind to ask for it

```
type String    = [Char]
type Pos       = (Int, Int)
type Trans     = Pos -> Pos

type Pair a    = (a, a)
type Assoc k v = [(k, v)]
```

What's the kind of Assoc?

```
>>> :k String
String :: *

>>> :k []
[] :: * -> *

>>> :k (,)
(,) :: * -> * -> *

>>> :k (->)
(->) :: * -> * -> *
```

Data Declarations (Bool)

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

We're introducing a new type constructor Bool

```
>>> :k Bool
Bool :: *
```

We're introducing two new data constructors False and True

```
>>> :t True
True :: Bool

>>> :t False
False :: Bool
```

Data Declarations

- Use keyword data to declare a new datatype.
- When we're defining a new datatype by data, we're actually
  - Introducing a new type constructor
  - Introducing some new data constructors
    - only way to construct the inhabitant of this type.

Data Declarations (Maybe)

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

We're introducing a new type constructor Maybe

```
>>> :k Maybe
Maybe :: * -> *

>>> :k Maybe Int
Maybe Int :: *
```

We're introducing two new data constructors Nothing and Just

```
>>> :t Nothing
Nothing :: Maybe a

>>> :t Just
Just :: a -> Maybe a

>>> :t Just True
Just True :: Maybe Int
```

Pattern matching is the only way to eliminate/destruct constructors.

Newtype Declaration

```
newtype Nat = N Int
data    Nat = N Int
type    Nat = Int
```

- For a new type with a single constructor, it can be declared by a newtype
- newtype (vs. data) brings an efficiency benefit

Folding Over Datatypes

Folding Over Expressions

```haskell
data Expr = Val Int
          | Add Expr Expr
          | Mul Expr Expr

size :: Expr → Int
size (Val n) = 1
size (Add x y) = size x + size y
size (Mul x y) = size x + size y

eval :: Expr → Int
eval (Val n) = n
eval (Add x y) = eval x + eval y
eval (Mul x y) = eval x * eval y
```

Folding Over Expressions

```
foldExpr :: (Int → a) → (a → a → a) → (a → a → a) → Expr → a
foldExpr v _ _ (Val n) = v n
foldExpr v a m (Add x y) = a (foldExpr v a m x) (foldExpr v a m y)
foldExpr v a m (Mul x y) = m (foldExpr v a m x) (foldExpr v a m y)

size' = foldExpr (\_ → 1) (+) (+)
eval' = foldExpr (\x → x) (+) (*)
```

Question: Printing Expressions

Implement printExpr function using foldExpr and binary.

```
import Text.Printf
foldExpr :: (Int → a) → (a → a → a) → (a → a → a) → Expr → a

binary :: String → String → String → String
binary op x y = printf "(%s %s %s)" x op y

> printExpr (Add (Val 1) (Mul (Val 2) (Val 3)))
"(1 + (2 * 3))"
```

Question: Printing Expressions

```
import Text.Printf

binary :: String → String → String → String
binary op x y = printf "(%s %s %s)" x op y

printExpr :: Expr → String
printExpr (Val n) = show n
printExpr (Add x y) = binary "+" (printExpr x) (printExpr y)
printExpr (Mul x y) = binary "*" (printExpr x) (printExpr y)

> printExpr (Add (Val 1) (Mul (Val 2) (Val 3)))
"(1 + (2 * 3))"

printExpr = foldExpr show (binary "+") (binary "*")
```

Question: Collect

Implement a function that collects with foldExpr, which collects all the numbers (in the Val case) in an expression.

```
collect :: Expr → [Int]
```

```
collect :: Expr → [Int]
collect = foldExpr (\x → [x]) (++) (++)
```

Further Reading: Catamorphism

12

Recently I've finally started to feel like I understand catamorphisms. I wrote some about them in [a recent answer](), but briefly I would say a catamorphism for a type abstracts over the process of recursively traversing a value of that type, with the pattern matches on that type reified into one function for each constructor the type has. While I would welcome any corrections on this point or on the longer version in the answer of mine linked above, I think I have this more or less down and that is not the subject of this question, just some background.

Once I realized that the functions you pass to a catamorphism correspond exactly to the type's constructors, and the arguments of those functions likewise correspond to the types of those constructors' fields, it all suddenly feels quite mechanical and I don't see where there is any wiggle room for alternate implementations.

For example, I just made up this silly type, with no real concept of what its structure "means", and derived a catamorphism for it. I don't see any other way I could define a general-purpose fold over this type:

```
data X a b f = A Int b
             | B
             | C (f a) (X a b f)
             | D a

xCata :: (Int -> b -> r)
      -> r
      -> (f a -> r -> r)
      -> (a -> r)
      -> X a b f
      -> r
xCata a b c d v = case v of
  A i x -> a i x
  B -> b
  C f x -> c f (xCata a b c d x)
  D x -> d x
```

https://stackoverflow.com/questions/46561125/does-each-type-have-a-unique-catamorphism

Folding Over Trees

```
data Tree a = Leaf
            | Node (Tree a) a (Tree a)
```

- It parameterizes the data.
- It contains two recursive structures.
  - multiple choices to traverse the structure
  - Pre-order, In-order and Post-order

Folding Over Trees (Cheat)

```haskell
{-# LANGUAGE DeriveFoldable #-}
data Tree a = Leaf
            | Node (Tree a) a (Tree a) deriving (Show, Eq, Foldable)

foldTree = foldr

foldTree :: (a → b → b) → b → Tree a → b
foldTree _ base Leaf = base
foldTree fn base (Node left a right) = foldTree fn base' left
   where
   base'  = fn a base''
   base'' = foldTree fn base right
```

Folding Over Trees (General)

```
data Tree a = Leaf
            | Node (Tree a) a (Tree a)

foldr :: (a → b → b) → b → [a] → b
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)

foldTree :: (b → a → b → b) → b → Tree a → b
foldTree f z Leaf = z
foldTree f z (Node l a r) = f (foldTree f z l) a (foldTree f z r)
```

Folding Over Trees (Ad-hoc)

```
foldTreePos :: (a → b → b) → b → Tree a → b
foldTreePos f z Leaf = z
foldTreePos f z (Node l a r) = f a (foldTreePos f (foldTreePos f z l) r)

foldTreePre :: (a → b → b) → b → Tree a → b
foldTreePre f z Leaf = z
foldTreePre f z (Node l a r) = foldTreePre f (foldTreePre f (f a z) l) r

foldTreeIn :: (a → b → b) → b → Tree a → b
foldTreeIn f z Leaf = z
foldTreeIn f z (Node l a r) = foldTreeIn f (f a (foldTreeIn f z l)) r
```

Thank you!