Types in Haskell

Antoni Diller

1 November 2010

1 Enumerated Types

```
{- These definitions can be found in file:
   ~ard/teaching/workshop/public/types.hs -}

data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
           deriving (Eq,Ord,Enum,Show)

fromDay :: Day -> Int
fromDay = fromEnum

toDay :: Int -> Day
toDay = toEnum

weekend :: Day -> Bool
weekend Sun = True
weekend Sat = True
weekend _   = False
```

2 Composite Types

```
data Tag = Tagi Int | Tagb Bool
           deriving (Eq,Ord,Show)

isInt :: Tag -> Bool
isInt (Tagi _ ) = True
isInt (Tagb _ ) = False
```

3 Recursive Types

```
data Btree a = Tip a | Bin (Btree a) (Btree a)
               deriving (Eq,Ord,Show)

tree1 = Bin (Tip 1) (Bin (Tip 6) (Tip 5))

size :: Btree a -> Int
size (Tip _)   = 1
size (Bin s t) = size s + size t

depth :: Btree a -> Int
depth (Tip _)   = 0
depth (Bin s t) = 1 + max (depth s) (depth t)

flatten :: Btree a -> [a]
flatten (Tip x)     = [x]
flatten (Bin xt yt) = flatten xt ++ flatten yt



inodes :: Btree a -> Int
inodes (Tip _)     = 0
inodes (Bin xt yt) = 1 + inodes xt + inodes yt

maxBtree :: Ord a => Btree a -> a
maxBtree (Tip x)     = x
maxBtree (Bin xt yt) = max (maxBtree xt) ( maxBtree yt)

mapBtree :: (a -> b) -> Btree a -> Btree b
mapBtree f (Tip x)     = Tip (f x)
mapBtree f (Bin xt yt) = Bin (mapBtree f xt) (mapBtree f yt)

foldBtree :: (a -> b) -> (b -> b -> b) -> Btree a -> b
foldBtree f g (Tip x)     = f x
foldBtree f g (Bin xt yt) = g (foldBtree f g xt) (foldBtree f g yt)
```

```
size1 :: Btree a -> Integer
size1 = foldBtree (const 1) (+) where const x y = x

depth1 :: Btree a -> Integer
depth1 = foldBtree (const 0) op
         where op m n = 1 + max m n; const x y = x

flatten1 :: Btree a -> [a]
flatten1 = foldBtree wrap (++) where wrap x = [x]

maxBtree1 :: Ord a => Btree a -> a
maxBtree1 = foldBtree id max

mapBtree1 :: (a -> b) -> Btree a -> Btree b
mapBtree1 f = foldBtree (Tip . f) Bin
```

4 Another Recursive Type

```
data Tree a = Empty | Bin2 a (Tree a) (Tree a)
              deriving (Eq, Ord, Show)

insert :: Ord a => a -> Tree a -> Tree a
insert i Empty = Bin2 i Empty Empty
insert i (Bin2 j left right)
  | i <= j = Bin2 j (insert i left) right
  | i > j  = Bin2 j left (insert i right)

mktree :: Ord a => [a] -> Tree a
mktree = foldr insert Empty

listify :: Tree a -> [a]
listify Empty = []
listify (Bin2 j left right) = (listify left) ++ [j] ++ (listify right)

sort :: Ord a => [a] -> [a]
sort = listify . mktree
```