Introduction to Functional Programming

Complex numbers, Bag, Type classes

Department of Programming Languages and Compilers
Faculty of Informatics
Eötvös Loránd University
Budapest, Hungary
zsv@elte.hu

Overview

1. C set number
2. Bag as ADT
3. Type classes

Complex nr

```
:: C = { re :: Real
       , im :: Real
       }
mkC n d = { re = n, im = d }
Start = mkC 1.0 10.0 // (C 1 10)

instance + C
where
    (+) x y = mkC (x.re+y.re) (x.im+y.im)
Start = mkC 2.2 4.1 + mkC 1.5 6.4 // (C 3.7 10.5)

instance - C
where
   (-) x y = mkC (x.re-y.re) (x.im-y.im)
Start = mkC 2.2 4.1 - mkC 1.5 6.4 // (C 0.7 -2.3)
```

Complex nr

```
instance * C
where
    (*) x y = mkC (x.re*y.re - x.im*y.im) (x.re*y.im + x.im*y.re)

Start = mkC 2.0 4.0 * mkC 3.0 2.0 // (C -2 16)

// for simplicity only division by a real nr is defined
instance / C
where
    (/) x y
    | y.im == 0.0 = mkC (x.re/y.re) (x.im/y.re)
    = abort "division not defined"

Start = (mkC 2.0 4.0) / (mkC 2.0 0.0) // (C 1 2)
```

Complex nr

```
instance fromReal C
where
    fromReal r = mkC r 0.0

Start :: C
Start = fromReal 3.0 // (C 3 0)

instance toReal C
where
    toReal x
    | x.im == 0.0 = x.re
    = abort "x has imaginary part"

Start = toReal (mkC 3.0 0.0) // 3
```

Complex nr

```
instance zero C
where
    zero = fromReal 0.0

Start :: C
Start = zero // (C 0 0)

instance one C
where
    one = fromReal 1.0

Start :: C
Start = one // (C 1 0)
```

Complex nr

```
instance abs C
where
    abs x = fromReal (sqrt (x.re*x.re + x.im*x.im))

Start = abs (mkC 3.0 4.0) // (C 5 0)

//conjugate of a complex x+yi is x-yi
instance ¬ C
where
    (¬) x = mkC x.re (¬x.im)

Start = ¬ (mkC 2.0 3.0) // (C 2 -3)
```

Complex nr

```
instance toString C
where
    toString x
        | x.im == 0.0 = toString x.re
        | otherwise = toString x.re +++ "+"
                    +++ toString x.im +++ "i"

Start = toString (mkC 3.0 4.0) // "3+4i"

instance == C
where
    (==) x y = x.re == y.re && x.im==y.im

Start = mkC 1.0 2.0 == mkC 1.0 2.0  // True
```

Complex nr

```
// test whether the complex number represents a real nr
isRealC :: C → Bool
isRealC x
| x.im == 0.0 = True
= False

Start = isRealC (mkC 2.0 0.0) // True

re :: C → Real
re x = x.re
Start = re (mkC 1.0 2.0) // 1

im :: C → Real
im x = x.im
Start = im (mkC 1.0 2.0) // 2
```

Bag

```
definition module Bag
import StdEnv

:: Bag a

newB     ::    (Bag a)                       // empty bag
isempty  ::    (Bag a) → Bool
insertB  ::  a (Bag a) → Bag a  | Eq a   // insert an element
removeB  ::  a (Bag a) → Bag a  | Eq a   // remove an element
sizeB    ::    (Bag a) → Int              // return all nr elements
```

Bag

```
implementation module Bag
import StdEnv

:: Bag a :== [(Int,a)]

newB :: Bag a
newB = []

isempty  :: (Bag a) -> Bool
isempty  [] = True
isempty  x  = False
```

Bag

```
insertB :: a (Bag a) → Bag a | Eq a
insertB e [] = [(1,e)]
insertB e [(m,x):t]
| e == x = [(m+1,x):t]
= [(m,x)] ++ insertB e t

removeB :: a (Bag a) → Bag a | Eq a
removeB e [] = []
removeB e [(m,x):t]
| e == x && (m-1) == 0 = t
| e == x = [(m-1,x):t]
= [(m,x)] ++ removeB e t
```

Bag

```
sizeB :: (Bag a) → Int
sizeB [] = 0
sizeB [(m,x):t] = m + sizeB t

// tests of implementations:
Start = ( "s0 = newB = ",           s0,'\n'
   , "s1 = insertB 1 s0 = ",s1,'\n'
   , "s2 = insertB 1 s1 = ",s2,'\n'
   , "s3 = insertB 2 s2 = ",s3,'\n'
   , "s4 = removeB 1 s3 = ",s4,'\n'
   , "s5 = sizeB      s3 = ",s5,'\n'
   , "test = isempty s3 = ",test,'\n')
```

Bag

where

```
s0 = newB
s1 = insertB   1      s0
s2 = insertB   1      s1
s3 = insertB   2      s2
s4 = removeB   1      s3
s5 = sizeB          s3
test              = isempty        s3
```

/ ("s0 = newB = ",[],'*
',"s1 = insertB 1 s0 = ",[(1,1)],'
',"s2 = insertB 1 s1 = ",[(2,1)],'
',"s3 = insertB 2 s2 = ",[(2,1),(1,2)],'
',"s4 = removeB 1 s3 = ",[(1,1),(1,2)],'
',"s5 = sizeB s3 = ",3,'
',"test = isempty s3 = ",False,'
*') */*

Map

```
module Map
import StdEnv

// The (Maybe a) type represents a collection of at most one element
:: Maybe a =  Just a
            | Nothing

// Binary trees
::  Tree a = Leaf | Node a (Tree a) (Tree a)

// Single tree
:: Tree1 a = Node1 a [Tree1 a]
```

Map

// the type constructor class Map such that the all instances bellow can be created.

```
class Map t :: (a → b) (t a) → t b

instance Map []
where Map f xs = map1 f xs

instance Map Maybe
where Map f mb = mapMaybe f mb

instance Map Tree
where Map f tr = mapTree  f tr
```

Map

```
instance Map Tree1
where Map f tr = mapTree1  f tr

instance Map ((,) a)
where
   Map :: (a → b) (c,a) → (c,b)
   Map f (x,y) = (x,f y)
```

Map

// given function, for lists:

```
map1 :: (a → b) [a] → [b]
map1 f [] = []
map1 f [x:xs] = [f x : map1 f xs]
```

// given function, for Maybe:

```
mapMaybe              :: (a → b) (Maybe a) → Maybe b
mapMaybe f Nothing     = Nothing
mapMaybe f (Just x)    = Just (f x)
```

Map

// given function, for Tree:

```
mapTree                :: (a → b) (Tree a) → Tree b
mapTree f Leaf         = Leaf
mapTree f (Node x le ri) = Node (f x) (mapTree f le) (mapTree f ri)
```

// given function, for Tree1:

```
mapTree1                    :: (a → b) (Tree1 a) → Tree1 b
mapTree1 f (Node1 elem ls) = Node1 (f elem) (map (mapTree1 f) ls)
```

Map

```
t1 :: Tree Int
t1 = Node 1 Leaf (Node 2 Leaf (Node 3 Leaf (Node 4 Leaf Leaf)))

a1 :: Tree1 Int
a1 = Node1 1 [Node1 2 [Node1 3 [], Node1 4 [], Node1 5
     [Node1 6 []]]]

Start = Map inc [1..10]

Start :: Maybe Int
Start = Map inc (Just 4)
Start = Map inc Nothing
```

Map

```
Start = t1
Start = Map inc t1

Start = a1
Start = Map inc a1

Start = Map inc (True, 4)
Start = Map inc (1.5, 2)
```