Introduction to Functional Programming

Zsók Viktória

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

Overview

Records

```
:: Person = { name :: String
            , birthdate :: (Int,Int,Int)
            , fpprogramer :: Bool
            }

IsfpUser :: Person -> String
IsfpUser {fpprogramer = True} = "Yes"
IsfpUser _                    = "No"

Start = IsfpUser { name = "Me"
                 , birthdate = (1,1,1999)
                 , fpprogramer = True}     // "Yes"
```

Records

```
:: Person = { name :: String
            , birthdate :: (Int,Int,Int)
            , fpprogramer :: Bool
            }

GetName :: Person → String
GetName p = p.name

GetName2 :: Person → String
GetName2 {name} = name

ChangeN :: Person String → Person
ChangeN p s = {p & name = s}

Start = ChangeN {name = "XY", birthdate = (1,1,2000),
                 fpprogramer = True} "Alex"
```

Records

```
:: Point = { x       :: Real
           , y       :: Real
           , visible :: Bool
           }

:: Vector = { dx      :: Real
            , dy      :: Real
            }

Origo :: Point
Origo = { x = 0.0
        , y = 0.0
        , visible = True
        }
Dist :: Vector
Dist = { dx = 1.0
       , dy = 2.0
       }
```

Records

```
IsVisible :: Point → Bool
IsVisible {visible = True} = True
IsVisible _                = False

xcoordinate :: Point → Real
xcoordinate p = p.x

hide :: Point → Point
hide p = { p & visible = False }

Move :: Point Vector → Point
Move p v = { p & x = p.x + v.dx, y = p.y + v.dy }

Start = Move (hide Origo) Dist
```

Records

```
:: Q = { nom :: Int
       , den :: Int
       }
QZero = { nom = 0, den = 1 }
QOne = { nom = 1, den = 1 }

simplify {nom=n,den=d}
  | d == 0 = abort " denominator is 0"
  | d < 0  = { nom = ~n/g, den = ~d/g}
  | otherwise = { nom = n/g, den = d/g}
  where g = gcdm n d

gcdm x y = gcdnat (abs x) (abs y)
  where gcdnat x 0 = x
        gcdnat x y = gcdnat y (x rem y)

mkQ n d = simplify { nom = n, den = d }
Start = mkQ 81 90
```

Arrays

```
MyArray :: {Int}
MyArray = {1,3,5,7,9}

Start = MyArray.[2] // 5

MapArray1 f a = {f e \\ e <-: a}

Start :: {Int}
Start = MapArray1 inc MyArray

// Comprehension transformations:
Array = {elem \\ elem <- List}
List = [elem \\ elem <-: Array]
```

Algebraic types

```
:: Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun

:: Tree a = Node a (Tree a) (Tree a)
          | Leaf

sizeT :: (Tree a) → Int
sizeT Leaf = 0
sizeT (Node x l r) = 1 + sizeT l + sizeT r

Start = sizeT (Node 4 (Node 2 (Node 1 Leaf Leaf)
              (Node 3 Leaf Leaf)) Leaf)          // 4
```

Algebraic types

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

atree = Node 2 (Node 1 Leaf Leaf) (Node 3 Leaf Leaf)

depth :: (Tree a) → Int
depth Leaf = 0
depth (Node _ l r) = (max (depth l) (depth r)) + 1

Start = depth atree // 2
```

Algebraic types

```
treesort :: ([a]→ [a]) | Eq, Ord a
treesort = collect o listtoTree

listtoTree :: [a] → Tree a | Ord, Eq a
listtoTree [] = Leaf
listtoTree [x:xs] = insertTree x (listtoTree xs)

insertTree :: a (Tree a) → Tree a | Ord a
insertTree e Leaf = Node e Leaf Leaf
insertTree e (Node x le ri)
    | e<=x = Node x (insertTree e le) ri
    | e>x  = Node x le (insertTree e ri)

collect :: (Tree a) → [a]
collect Leaf = []
collect (Node x le ri) = collect le ++ [x] ++ collect ri

Start = treesort [3, 1, 5, 9, 2, 7, 0] // [0, 1, 2, 3, 5, 7, 9]
```

Algebraic types

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

atree = Node 2 (Node 1 Leaf Leaf) (Node 3 Leaf Leaf)

:: Tree2 a = Node2 a (Tree2 a) (Tree2 a)
           | Leaf2 a
```

More trees

```
nrNodes :: (Tree2 a) → Int
nrNodes (Leaf2 y) = 1
nrNodes (Node2 x l r) = 1 + nrNodes l + nrNodes r

aTree2 :: Tree2 Int

aTree2 = Node2 4 (Node2 2 (Node2 1 (Leaf2 1) (Leaf2 1))
                 (Node2 3 (Leaf2 3) (Leaf2 3))) (Leaf2 5)

Start = nrNodes aTree2      // 9
```

More trees

```
:: Tree3 a b = Node3 a (Tree3 a b) (Tree3 a b)
             | Leaf3 b

aTree3 :: Tree3 Int Real

aTree3 = Node3 2 (Node3 1 (Leaf3 1.1) (Leaf3 2.5))
                 (Node3 3 (Leaf3 3.0) (Leaf3 6.9))

sumLeaves :: (Tree3 Int Real) -> Real
sumLeaves (Leaf3 y) = y
sumLeaves (Node3 x le ri) = sumLeaves le + sumLeaves ri

Start = sumLeaves aTree3 // 13.5
```

Algebraic types

```
// Triple branches
:: Tree4 a = Node4 a (Tree4 a) (Tree4 a) (Tree4 a)
           | Leaf4

// Rose-tree - tree with variable multiple branches
// No leaf constructor, node with no branches
:: Tree5 a = Node5 a [Tree5 a]

// Every node has one branch = list
:: Tree6 a = Node6 a (Tree6 a)
           | Leaf6

// Tree with different types
:: Tree7 a b = Node7a Int (Tree7 a b) (Tree7 a b)
             | Node7b b (Tree7 a b)
             | Leaf7a b
             | Leaf7b Int
```

Map, foldr on trees

```
:: BTree a                  = Bin (BTree a) (BTree a)
                            | Tip a

mapbtree                    :: (a → b) (BTree a) → BTree b
mapbtree f (Tip x)          = Tip (f x)
mapbtree f (Bin t1 t2)      = Bin (mapbtree f t1) (mapbtree f t2)

foldbtree                   :: (a a → a) (BTree a) → a
foldbtree f (Tip x)         = x
foldbtree f (Bin t1 t2)     = f (foldbtree f t1) (foldbtree f t2)

aBTree = Bin (Bin (Bin (Tip 1) (Tip 1))
                  (Bin (Tip 3) (Tip 3))) (Tip 5)

Start = mapbtree inc aBTree
Start = foldbtree (+) aBTree // 13
```

Abstract Data Types

```
definition module Stack

:: Stack a

newStack :: (Stack a)               // Creates empty stack
empty    :: (Stack a) → Bool        // Checks if a stack is empty
push     :: a (Stack a) → Stack a   // push new element on top of the stack
pop      :: (Stack a) → Stack a     // Remove the top element from the stack
top      :: (Stack a) → a           // Return the top element from the stack
```

Abstract Data Types

```
implementation module Stack
import StdEnv
:: Stack a :== [a]

newStack :: Stack a
newStack = []

empty :: (Stack a) -> Bool
empty [] = True
empty x  = False

push :: a (Stack a) -> Stack a
push e s = [e : s]
pop :: (Stack a) -> Stack a
pop [e : s] = s
top :: (Stack a) -> a
top [e : s] = e
```

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,'
') */
```