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
            }

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"
```

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
```