Introduction to Functional Programming

Records, Arrays, Algebraic Types

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

```
:: 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]
```