Esempi di Programmazione Haskell

Marco Comini

29 ottobre 2013

In questo scritto intendo mostrare alcuni semplici esempi di programmi in Haskell per iniziare ad impratichirsi con il linguaggio più alcuni esempi completi di Strutture Dati Astratte sviluppate cercando di sfruttare appieno le caratteristiche di polimorfismo implicito e del meccanismo delle classi.

Questo è un work in progress che non pretende ne di essere esauriente ne tantomeno esaustivo.

1 Esempietti

1.1 Numeri e Liste

1. In questo esempio generiamo la lista dei primi numeri primi minori di n, con un algoritmo non troppo efficiente ma molto semplice da scrivere.

```haskell
eratostene n = 1:era [2..n]
  where
    era [] = []
    era (x:xs)
      | x*x < n   = x:era ys
      | otherwise = x:ys
      where
        ys = filter (\y->y `rem` x /= 0) xs
```

   si noti come nello scope del where è visibile la n del pattern di definizione.

2. In questo esempio voglio mostrare una variante dell'algoritmo di *counting sort* che non ha bisogno di sapere a priori il massimo e il minimo dei valori da ordinare. Come nell'algoritmo originale abbiamo una fase in cui vogliamo accumulare il numero di occorrenze di ogni numero da ordinare. Invece di usare un vettore utilizziamo una funzione che dato un certo valore ci restituirà il numero di occorrenze di detto valore. Man mano leggiamo la lista aggiorniamo tale funzione nonchè il massimo e il minimo valore incontrato in modo da poter ricostruire poi la lista ordinata.

```haskell
countsort :: (Enum a, Ord a) => [a] -> [a]
countsort []       = []
countsort xs@(x:_) = cnt2list [] list2cnt
  where
    list2cnt = foldl aggr (x,x,\_->0) xs
      where
        aggr (a,b,f) n = (min a n,max b n,
                          \i -> if i == n then 1+f n else f i)

    cnt2list xs (a,b,f)
```

```
    | a > b      = xs
    | otherwise = cnt2list (put (f b) xs) (a,pred b,f)
  where
    put 0 ys = ys
    put n ys = b : put (n-1) ys
```

chiaramente questo algoritmo sarà tanto più inefficiente quanto meno è "densa" la distribuzione dei valori all'interno del range minimo–massimo.

1.2 Alberi

Si definiscano gli Alberi Binari di Ricerca col seguente tipo di dato astratto (polimorfo)

```
data (Ord a, Show a, Read a) => BST a = Void | Node a (BST a) (BST a)
    deriving (Eq, Ord, Read, Show)
```

e si usi (per comodità) lo stesso tipo di dato anche per Alberi Binari normali.

1. Iniziamo con un esempio facile per calcolare l'altezza di un albero (dove non uso volutamente nessun tipo di fold, per quello si vedano gli esercizi).

   ```
   treeheight Void = 0
   treeheight (Node y l r) = 1 + max (treeheight l) (treeheight r)
   ```

2. Ora vediamo come "potare" un albero ad una certa profondità.

   ```
   takedepth 0 _            = Void
   takedepth _ Void         = Void
   takedepth n (Node y l r) = Node y (takedepth (n-1) l) (takedepth (n-1) r)
   ```

1.3 Matrici

Implementiamo le matrici si implementano come liste di liste, per righe. Quindi

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

verrà codificata come

```
[ [1, 2, 3, 4],
  [5, 6, 7, 8],
  [9, 10, 11, 12]]
```

2 Binary (Search) Trees

2.1 BinaryTrees.hs

Qui l'interessante è la definizione delle istanze di **Show** e **Read** oltre che la definizione di una `fold` su alberi che può essere usata, come per le fold su liste, per fare moltissime cose.

Si veda la `toList`.

```haskell
module BinaryTrees(Tree(..),
  empty,
  fold,
  isEmpty,
  new,
  toList) where

data Tree a = Void | Node a (Tree a) (Tree a) -- deriving (Eq)

instance Show a => Show (Tree a)
  where
    showsPrec _ = showsTree
      where
        -- showsTree          :: Show a => Tree a -> ShowS
        showsTree Void         = ('.':)
        showsTree (Node x l r) = ('<':) . showsTree l .
                                 (' ':) . shows x .
                                 (' ':) . showsTree r .
                                 ('>':)

instance Read a => Read (Tree a)
  where
    readsPrec _ = readsTree
      where
        readsTree  :: Read a => ReadS (Tree a)
        readsTree ts = [(Node x l r, zs++ys) | (('<':rs), us) <- lex ts,
                                               (l, vs) <- readsTree (rs++us),
                                               (x,ws) <- reads vs,
                                               (r, xs) <- readsTree ws,
                                               (('>':zs),ys) <- lex xs ]
                       ++
                       [(Void,rs++us) | (('.':rs),us) <- lex ts ]

-- Exported functions

empty :: Tree a
empty = Void

isEmpty      :: Tree a -> Bool
isEmpty Void = True
isEmpty _    = False
-- if Eq a then isEmpty = (Void==)

new      :: a -> Tree a -> Tree a -> Tree a
-- new x l r = (Node x l r)
new = Node

fold                   :: (a -> b -> b -> b) -> b -> Tree a -> b
fold _ z Void          = z
fold f z (Node x l r)  = f x (fold f z l) (fold f z r)

toList   :: Tree a -> [a]
toList t = fold aggr id t []
  where
    -- aggr :: a -> ([a]->[a]) -> ([a]->[a]) -> ([a] -> [a])
    aggr x lacc racc = lacc . (x:) . racc

--- Not exported functions

sum = fold (\x y z->x+y+z) 0

filteredSum p = fold filteredsum 0
```

```haskell
  where
    filteredsum x y z
      | p x       = x + y + z
      | otherwise = y + z
```

2.2 BinarySearchTrees.hs

Qui si mostra come riutilizzare un altro ADT (Abstract Data Type) per implementarne uno nuovo, esportando metodi del vecchio (eventualmente cambiando i nomi) e/o aggiungendo nuovi metodi.

```haskell
module BinarySearchTrees(BST,
  elem,
  empty,
  fold,
  fromList,
  fromListUnique,
  insert,
  insertUnique,
  isEmpty,
  join,
  joinUnique,
  meet,
  sort,
  toList) where

import Prelude hiding (elem)
import BinaryTrees(Tree(..))
import qualified BinaryTrees as BT

type BST a = Tree a

-- Exported functions

empty :: Ord a => BST a
empty = BT.empty

isEmpty :: Ord a => BST a -> Bool
isEmpty = BT.isEmpty

fold :: Ord a => (a -> b -> b -> b) -> b -> BST a -> b
fold = BT.fold

toList :: Ord a => BST a -> [a]
toList = BT.toList



elem          :: (Ord a) => a -> BST a -> Bool
elem x Void = False
elem x (Node y l r)
  | x == y      = True
  | x < y       = elem x l
  | otherwise = elem x r

insert          :: (Ord a) => a -> BST a -> BST a
insert x Void = Node x Void Void
insert x (Node y l r)
  | x<=y        = Node y (insert x l) r
  | otherwise = Node y l (insert x r)
```

```
insertUnique        :: (Ord a) => a -> BST a -> BST a
insertUnique x Void = Node x Void Void
insertUnique x n@(Node y l r)
  | x==y       = n
  | x<y        = Node y (insertUnique x l) r
  | otherwise = Node y l (insertUnique x r)

fromList :: Ord a => [a] -> BST a
fromList = foldl (flip insert) Void

fromListUnique :: Ord a => [a] -> BST a
fromListUnique = foldl (flip insertUnique) Void

sort :: Ord a => [a] -> [a]
sort = toList . fromList

join       :: (Ord a) => BST a -> BST a -> BST a
join t1 t2 = fold aggr id t2 t1
  where
    -- aggr :: (Ord a) => a -> (BST a -> BST a) -> (BST a -> BST a) -> BST a
        -> BST a
    aggr x lacc racc = lacc . insert x . racc

joinUnique       :: (Ord a) => BST a -> BST a -> BST a
joinUnique t1 t2 = fold aggr id t2 t1
  where
    aggr x lacc racc = lacc . insertUnique x . racc

meet       :: (Ord a) => BST a -> BST a -> BST a
meet t1 t2 = fromList $ meetlist (toList t1) (toList t2)
  where
    meetlist [] _ = []
    meetlist _ [] = []
    meetlist as@(x:xs) bs@(y:ys)
      | x==y       = x : meetlist xs ys
      | x<y        = meetlist xs bs
      | otherwise = meetlist as ys
```

3 Red Black Trees

Invece di implementare i Red Black Trees scegliendo un qualche metodo di default su quando e come inserire elementi, in questa implementazione si va a estendere la classe `Ord` con dei metodi che vadano a specificare cosa fare nei vari casi. In questo modo, con opportune istanze dei metodi si possono ottenere comportamenti molto diversi mantenendo lo stesso codice. Tutti gli esempi su Insiemi, Multinsiemi e Tabelle Associative nel seguito sono proprio istanze opportune di Red Black Trees.

Qui l'interessante è la definizione di due tipi di `map`. Se la funzione f che vogliamo applicare agli elementi di un RBT è monotona (rispetto agli ordini delle istanze di `Ord`) allora possiamo andare a rimpiazzare le immagini via f in place, altrimenti il nuovo albero va interamente ricostruito.

La funzione di ricerca qui potrebbe dover restituire diversi elementi e quindi restituisce una lista.

```
module RedBlackTrees(RBT(),
  Ord'(insEq,valJoin,valMeet),
  empty,
```

```haskell
  isEmpty,
  fold,
  lookup,
  toList,
  update,
  join,
  fromList,
  sort,
  meet,
  mapMonotone,
  map,
  delete,
  diff,
  findMin,
  findMax) where

import Prelude hiding (lookup,map)

--- The colors of a node in a red-black tree.
data Color = Red | Black | DoublyBlack deriving (Eq, Ord)

instance Show Color where
  show Red    = "r"
  show Black = "b"
  show DoublyBlack = "B"

instance Read Color where
  readsPrec _ = readsColor
    where
      readsColor (' ':xs)  = readsColor xs
      readsColor ('\t':xs) = readsColor xs
      readsColor ('\n':xs) = readsColor xs
      readsColor ('b':xs) = [(Black,xs)]
      readsColor ('B':xs) = [(DoublyBlack,xs)]
      readsColor ('r':xs) = [(Red,xs)]
      readsColor (x:xs) =   [(error ("undefined color '"++x:"'."),xs)]
      readsColor [] = []

class (Ord a) => Ord' a
  where
    insEq :: a -> a -> Bool
    valJoin, valMeet :: a -> a -> a

    -- we assume that `insEq` implies (==)

    insEq  = (==)

    valJoin x _ = x

    valMeet x _ = x


--- The structure of red-black trees.
data (Ord' a) => RBT a = Node Color a (RBT a) (RBT a) | Void
  deriving (Eq, Ord)

instance (Show a, Ord' a) => Show (RBT a)
  where
    showsPrec _ = showsTree
      where
        -- showsTree          :: (Show a, Ord' a) => RBT a -> ShowS
        showsTree Void            = ('.':)
```

```haskell
        showsTree (Node c x l r) = ('(':) .
                                   showsTree l . (' ':) .
                                   shows x . (':':) . shows c . (' ':) .
                                   showsTree r .
                                   (')':)

instance (Read a, Ord' a) => Read (RBT a)
  where
    readsPrec _ = readsTree
      where
        -- readsTree :: (Read a, Ord' a) => ReadS (RBT a)
        readsTree ts = [(Node c x l r, qs) | ("(", us) <- lex ts,
                                            (l, vs) <- readsTree us,
                                            (x,ws) <- reads vs,
                                            (":",xs) <- lex ws,
                                            (c,ys) <- reads xs,
                                            (r, zs) <- readsTree ys,
                                            (")",qs) <- lex zs ]
                       ++
                       [(Void,rs++us) | (('.':rs),us) <- lex ts ]


empty :: (Ord' a) => RBT a
empty = Void

isEmpty      :: (Ord' a) => RBT a -> Bool
isEmpty Void = True
isEmpty _    = False

fold                    :: (Ord' a) => (a -> b -> b -> b) -> b -> RBT a -> b
fold _ z Void           = z
fold f z (Node _ x l r) = f x (fold f z l) (fold f z r)

-- in the following function we can see how the potential differences
-- between == and `insEq` do interact, since we cannot assume that in case
-- of equality we can just return (y:xs)

-- there can be more than one match
lookup     :: (Ord' a) => a -> RBT a -> [a]
lookup x t = fold aggr id t []
  where
    aggr y lacc racc
      | y == x    = lacc . (y:) . racc
      | x < y     = lacc
      | y < x     = racc
     -- | otherwise = error "RedBlackTrees.lookup: internal error" -- lacc (
         racc xs)

toList   :: (Ord' a) => RBT a -> [a]
toList t = fold aggr id t []
  where
    aggr x lacc racc = lacc . (x:) . racc

update     :: (Ord' a) => a -> RBT a -> RBT a
update x t = let (Node _ x2 l r) = upd t
             in Node Black x2 l r
  where
    upd Void = Node Red x Void Void
    upd (Node c x2 l r)
      | x `insEq` x2 = (Node c (x `valJoin` x2) l r)
      | x <  x2      = balanceL (Node c x2 (upd l) r)
      | otherwise    = balanceR (Node c x2 l (upd r))
```

```haskell
join        :: (Ord' a) => RBT a -> RBT a -> RBT a
join t1 t2 = fold aggr id t2 t1
  where
    -- aggr :: (Ord' a) => a -> (RBT a -> RBT a) -> (RBT a -> RBT a) -> RBT a
        -> RBT a
    aggr x lacc racc = lacc . update x . racc

fromList :: (Ord' a) => [a] -> RBT a
fromList = foldl (flip update) Void

sort :: (Ord' a) => [a] -> [a]
sort = toList . fromList

meet        :: (Ord' a) => RBT a -> RBT a -> RBT a
meet t1 t2 = fromList$meetlist (toList t1) (toList t2)
  where
    meetlist [] _ = []
    meetlist _ [] = []
    meetlist as@(x:xs) bs@(y:ys)
      | x == y     = (x `valMeet` y):meetlist xs ys
      | x < y      = meetlist xs bs
      | otherwise = meetlist as ys

-- if function is monotone then the tree structure is preserved thus we can
-- apply the function directly in node values (cannot use fold cause it looses
    color)

mapMonotone        :: (Ord' a, Ord' b) => (a -> b) -> RBT a -> RBT b
mapMonotone _ Void = Void
mapMonotone f (Node c x l r) = Node c (f x) (map f l) (map f r)

-- otherwise we have to reconstruct a new tree entirly

map              :: (Ord' a, Ord' b) => (a -> b) -> RBT a -> RBT b
map f t = fold aggr id t Void
  where
    aggr x lacc racc = racc . update (f x) . lacc

delete :: (Ord' a) => a -> RBT a -> RBT a
delete x t = blackenRoot $ deleteTree x t
  where
    blackenRoot Void = Void
    blackenRoot (Node _ x l r) = Node Black x l r

    deleteTree _ Void = Void  -- no error for non existence
    deleteTree e (Node c e2 l r)
          | e == e2 = if l==Void then addColor c r else
                      if r==Void then addColor c l
                                 else let el = rightMost l
                                        in delBalanceL (Node c el (deleteTree
                                             el l) r)
          | e < e2    = delBalanceL (Node c e2 (deleteTree e l) r)
          | otherwise = delBalanceR (Node c e2 l (deleteTree e r))
        where
          addColor Red tree = tree
          addColor Black Void = Void
          addColor Black (Node Red x lx rx)   = Node Black x lx rx
          addColor Black (Node Black x lx rx) = Node DoublyBlack x lx rx

          rightMost (Node _ x _ rx) = if rx==Void then x else rightMost rx
          rightMost Void            = error "internal error on function delete"
```

```
diff        :: (Ord' a) => RBT a -> RBT a -> RBT a
diff t1 t2 = fold aggr id t1 t2
  where
    aggr x lacc racc = lacc . delete x . racc

{-
 - findMin                 :: (Ord' a) => RBT a -> a
 - findMin (Node _ x Void _) = x
 - findMin (Node _ _ l _)    = findMin l
 - findMin Void              = error "RedBlackTrees.findMin: empty tree has no
     minimal element"
 -
 - findMax                 :: (Ord' a) => RBT a -> a
 - findMax (Node _ x _ Void) = x
 - findMax (Node _ _ _ r)    = findMax r
 - findMax Void              = error "RedBlackTrees.findMax: empty tree has no
     maximal element"
 -}

findMin :: (Ord' a) => RBT a -> Maybe a
findMin = fold aggr Nothing
  where
    aggr x Nothing _ = Just x
    aggr _ l       _ = l

findMax :: (Ord' a) => RBT a -> Maybe a
findMax = fold aggr Nothing
  where
    aggr x _ Nothing = Just x
    aggr _ _ l       = l

-- Not exported

isBlack                :: (Ord' a) => RBT a -> Bool
isBlack Void            = True
isBlack (Node c _ _ _) = c==Black

isRed                :: (Ord' a) => RBT a -> Bool
isRed Void            = False
isRed (Node c _ _ _) = c==Red

isDoublyBlack                :: (Ord' a) => RBT a -> Bool
isDoublyBlack Void            = True
isDoublyBlack (Node c _ _ _) = c==DoublyBlack

left                :: (Ord' a) => RBT a -> RBT a
left (Node _ _ l _) = l
left _              = error "RedBlackTrees.left: empty tree has no left son"

right                :: (Ord' a) => RBT a -> RBT a
right (Node _ _ _ r) = r
right _              = error "RedBlackTrees.right: empty tree has no right son
   "

singleBlack                           :: (Ord' a) => RBT a -> RBT a
singleBlack Void                       = Void
singleBlack (Node DoublyBlack x l r) = Node Black x l r
singleBlack _                          = error "RedBlackTrees.singleBlack:
    internal error"

-- for the implementation of balanceL and balanceR refer to picture 3.5, page
```

```haskell
    27,
-- Okasaki "Purely Functional Data Structures"
balanceL :: (Ord' a) => RBT a -> RBT a
balanceL tree
  | isRed leftTree && isRed (left leftTree)
   = let Node _ z (Node _ y (Node _ x a b) c) d = tree
     in Node Red y (Node Black x a b) (Node Black z c d)

  | isRed leftTree && isRed (right leftTree)
   = let Node _ z (Node _ x a (Node _ y b c)) d = tree
     in Node Red y (Node Black x a b) (Node Black z c d)

  | otherwise
   = tree

  where
    leftTree = left tree

balanceR :: (Ord' a) => RBT a -> RBT a
balanceR tree
  | isRed rightTree && isRed (right rightTree)
   = let Node _ x a (Node _ y b (Node _ z c d)) = tree
     in Node Red y (Node Black x a b) (Node Black z c d)

  | isRed rightTree && isRed (left rightTree)
   = let Node _ x a (Node _ z (Node _ y b c) d) = tree
     in Node Red y (Node Black x a b) (Node Black z c d)

  | otherwise
   = tree

  where
    rightTree = right tree


-- balancing after deletion

delBalanceL :: (Ord' a) => RBT a -> RBT a
delBalanceL tree
  | isDoublyBlack (left tree) = reviseLeft tree
  | otherwise                 = tree
  where
    reviseLeft tree
      | r==Void
       = tree
      | isblackr && isRed (left r)
       = let Node col x a (Node _ z (Node _ y b c) d) = tree
         in Node col y (Node Black x (singleBlack a) b) (Node Black z c d)
      | isblackr && isRed (right r)
       = let Node col x a (Node _ y b (Node _ z c d)) = tree
         in Node col y (Node Black x (singleBlack a) b) (Node Black z c d)
      | isblackr
       = let Node col x a (Node _ y b c) = tree
         in Node (if col==Red then Black else DoublyBlack) x (singleBlack a) (
             Node Red y b c)
      | otherwise
       = let Node _ x a (Node _ y b c) = tree
         in Node Black y (reviseLeft (Node Red x a b)) c
     where
       r = right tree
       isblackr = isBlack r
```

```haskell
delBalanceR :: (Ord' a) => RBT a  ->  RBT a
delBalanceR tree
  | isDoublyBlack (right tree) = reviseRight tree
  | otherwise                  = tree
  where
    reviseRight tree
      | l==Void = tree
      | isblackl && isRed (left l)
      = let Node col x (Node _ y (Node _ z d c) b) a = tree
         in Node col y (Node Black z d c) (Node Black x b (singleBlack a))
      | isblackl && isRed (right l)
      = let Node col x (Node _ z d (Node _ y c b)) a = tree
         in Node col y (Node Black z d c) (Node Black x b (singleBlack a))
      | isblackl
      = let Node col x (Node _ y c b) a = tree
         in Node (if col==Red then Black else DoublyBlack) x (Node Red y c b)
              (singleBlack a)
      | otherwise
      = let Node _ x (Node _ y c b) a = tree
         in Node Black y c (reviseRight (Node Red x b a))
      where
        l = left tree
        isblackl = isBlack l
```

4 Sets e MultiSets

4.1 Sets.hs

Qui l'interessante per prima cosa è la definizione del tipo `Set a` con le varie istanze per usare opportunamente RBT, oltre che l'istanza **Eq** (Set a).

Inoltre è interessante la ri-definizione delle istanze di **Show** e **Read** che andremmo a ereditare da RBT. Una nota d'attenzione merita l'accorgimento per mettere il giusto numero di **,** fra gli elementi di un insieme.

Svariati metodi di RBT vengono nascosti mentre molte operazioini tipiche degli insiemi vengono introdotte mediante opportune `fold`.

```haskell
module Sets(Set,
  (\\),
  delete,
  difference,
  elem,
  empty,
  filter,
  findMax,
  findMin,
  fold,
  fromList,
  insert,
  intersection,
  intersections,
  isEmpty,
  isProperSubsetOf,
  isSubsetOf,
  map,
  mapMonotone,
  singleton,
  size,
  toList,
  union,
```

```
  unions) where

import qualified RedBlackTrees as RBT
import RedBlackTrees(RBT,Ord')
import Prelude hiding (
  elem,
  filter,
  lookup,
  map,
  sum)


newtype (Ord a) => Val a = SetValue a deriving (Eq,Ord)

instance (Ord a) => Ord' (Val a)

instance (Ord a, Show a) => Show (Val a)
  where
    showsPrec _ (SetValue x) = shows x

instance (Ord a, Read a) => Read (Val a)
  where
    readsPrec _ xs = [ (SetValue x, ys) | (x,ys)<-reads xs ]


newtype Set a = MakeSet (RBT (Val a))

instance (Ord a) => Eq (Set a)
  where
    x == y = toList x == toList y

instance (Ord a, Show a) => Show (Set a)
  where
    showsPrec _ (MakeSet t) = ('{':) . (RBT.fold aggr id t) . (end:)
      where
        end = '}'

        -- aggr :: Show a => a -> ShowS -> ShowS -> ShowS
        aggr x lacc racc = lacc . shows x . comma racc

        -- comma :: ShowS -> ShowS
        comma shws xs = if head ys == end then ys else (',':ys)
          where ys = shws xs

instance (Ord a, Read a) => Read (Set a)
  where
    readsPrec _ = readsSet
      where
        readsSet ts = [ (MakeSet t, vs) | ("{", us) <- lex ts,
                                          (t, vs) <- readl us]

        readl ts = [ (RBT.empty, us) | ("}", us) <- lex ts ] ++
                   [ (RBT.update x s, vs) | (x, us) <- reads ts,
                                             (s,vs) <- readl' us]

        readl' ts = [ (RBT.empty, us) | ("}", us) <- lex ts ] ++
                    [ (RBT.update x s, ws) | (",", us) <- lex ts,
                                              (x,vs) <- reads us,
                                              (s,ws) <- readl' vs]


-- Exported functions
```

```haskell
empty :: Ord a => Set a
empty = MakeSet RBT.empty

isEmpty :: (Ord a) => Set a -> Bool
isEmpty (MakeSet t) = RBT.isEmpty t

elem :: (Ord a) => a -> Set a -> Bool
x `elem` (MakeSet t) = RBT.lookup (SetValue x) t /= []

size :: Ord a => Set a -> Integer
size (MakeSet t) = RBT.fold count 0 t
  where
    count _ y z = 1+y+z


insert :: (Ord a) => a -> Set a -> Set a
insert x (MakeSet t) = MakeSet (RBT.update (SetValue x) t)

singleton :: (Ord a) => a -> Set a
singleton x = insert x empty

delete :: Ord a => a -> Set a -> Set a
delete x (MakeSet t) = MakeSet (RBT.delete (SetValue x) t)

fromList :: (Ord a) => [a] -> Set a
fromList = MakeSet . (foldl update' RBT.empty)
  where
    update' t x = RBT.update (SetValue x) t

fold :: (Ord a) => (a -> b -> b) -> b -> Set a -> b
fold f z (MakeSet t) = RBT.fold aggr id t z
  where
    aggr (SetValue x) lacc racc = lacc . f x . racc

toList :: (Ord a) => Set a -> [a]
toList = fold (:) []

mapMonotone :: (Ord a, Ord b) => (a -> b) -> Set a -> Set b
mapMonotone f (MakeSet t) = MakeSet (RBT.mapMonotone f' t)
  where
    f' (SetValue x) = SetValue (f x)

map :: (Ord a, Ord b) => (a -> b) -> Set a -> Set b
map f (MakeSet t) = MakeSet (RBT.map f' t)
  where
    f' (SetValue x) = SetValue (f x)

union :: (Ord a) => Set a -> Set a -> Set a
union (MakeSet t1) (MakeSet t2) = MakeSet (RBT.join t2 t1)

unions :: (Ord a) => [Set a] -> Set a
unions = foldr union empty

intersection :: (Ord a) => Set a -> Set a -> Set a
intersection (MakeSet t1) (MakeSet t2) = MakeSet (RBT.meet t1 t2)

intersections :: (Ord a) => [Set a] -> Set a
intersections [] = empty
intersections xs = foldr1 intersection xs

difference :: (Ord a) => Set a -> Set a -> Set a
```

```haskell
difference (MakeSet t1) (MakeSet t2) = MakeSet (RBT.diff t2 t1)

infixl 9 \\

(\\) :: (Ord a) => Set a -> Set a -> Set a
(\\) = difference

filter :: Ord a => (a -> Bool) -> Set a -> Set a
filter p = fold insert' empty
  where
    insert' x = if p x then insert x else id

findMin :: Ord a => Set a -> Maybe a
findMin (MakeSet t) = fmap cnv $ RBT.findMin t
  where cnv (SetValue x) = x

findMax :: Ord a => Set a -> Maybe a
findMax (MakeSet t) = fmap cnv $ RBT.findMax t
  where cnv (SetValue x) = x

isSubsetOf :: Ord a => Set a -> Set a -> Bool
x `isSubsetOf` y = isEmpty (x\\y)

isProperSubsetOf :: Ord a => Set a -> Set a -> Bool
x `isProperSubsetOf` y = x `isSubsetOf` y && (not $ isEmpty $ y\\x)

---- Not exported functions

sum :: (Num a, Ord a) => Set a -> a
sum = fold (+) 0

{- TO DO
partition :: Ord a => (a -> Bool) -> Set a -> (Set a, Set a)

split :: Ord a => a -> Set a -> (Set a, Set a)

splitMember :: Ord a => a -> Set a -> (Set a, Bool, Set a)

deleteMin :: Set a -> Set a

deleteMax :: Set a -> Set a

deleteFindMin :: Set a -> (a, Set a)

deleteFindMax :: Set a -> (a, Set a)
-}
```

4.2 MultiSets.hs

Questo modulo è praticamente identico al precedente a parte l'istanza **Ord**' (Val a) che ci obbliga a inserire elementi uguali più volte generando quindi multinsiemi. Ovviamente vale solo come esempio visto che tenere i duplicati è sicuramente il metodo più inefficiante per impelmentare multinsiemi.

```haskell
module MultiSets(MultiSet,
  (\\),
  delete,
  difference,
  elem,
  empty,
  filter,
```

```haskell
  findMax,
  findMin,
  fold,
  fromList,
  insert,
  intersection,
  intersections,
  isEmpty,
  isProperSubsetOf,
  isSubsetOf,
  map,
  mapMonotone,
  singleton,
  size,
  toList,
  union,
  unions) where

import qualified RedBlackTrees as RBT
import RedBlackTrees(RBT,Ord')
import Prelude hiding (
  elem,
  filter,
  lookup,
  map,
  sum)


newtype (Ord a) => Val a = SetValue a deriving (Eq,Ord)

instance (Ord a) => Ord' (Val a)
  where
    insEq _ _ = False

instance (Ord a, Show a) => Show (Val a)
  where
    showsPrec _ (SetValue x) = shows x

instance (Ord a, Read a) => Read (Val a)
  where
    readsPrec _ xs = [ (SetValue x, ys) | (x,ys)<-reads xs ]


newtype MultiSet a = MakeSet (RBT (Val a))

instance (Ord a) => Eq (MultiSet a)
  where
    x == y = toList x == toList y

instance (Ord a, Show a) => Show (MultiSet a)
  where
    showsPrec _ (MakeSet t) = ('{':) . (RBT.fold aggr id t) . (end:)
      where
        end = '}'

        aggr x lacc racc = lacc . shows x . comma racc

        comma shws xs = if head ys == end then ys else (',':ys)
          where ys = shws xs

instance (Ord a, Read a) => Read (MultiSet a)
  where
```

```
    readsPrec _ = readsSet
      where
        readsSet ts = [ (MakeSet t, vs) | ("{", us) <- lex ts,
                                           (t, vs) <- readl us]

        readl ts = [ (RBT.empty, us) | ("}", us) <- lex ts ] ++
                   [ (RBT.update x s, vs) | (x, us) <- reads ts,
                                              (s,vs) <- readl' us]

        readl' ts = [ (RBT.empty, us) | ("}", us) <- lex ts ] ++
                    [ (RBT.update x s, ws) | (",", us) <- lex ts,
                                               (x,vs) <- reads us,
                                               (s,ws) <- readl' vs]

type Set a = MultiSet a


-- Exported functions

empty :: Ord a => Set a
empty = MakeSet RBT.empty

isEmpty :: (Ord a) => Set a -> Bool
isEmpty (MakeSet t) = RBT.isEmpty t

elem :: (Ord a) => a -> Set a -> Bool
x `elem` (MakeSet t) = RBT.lookup (SetValue x) t /= []

size :: Ord a => Set a -> Integer
size (MakeSet t) = RBT.fold count 0 t
  where
    count _ y z = 1+y+z


insert :: (Ord a) => a -> Set a -> Set a
insert x (MakeSet t) = MakeSet (RBT.update (SetValue x) t)

singleton :: (Ord a) => a -> Set a
singleton x = insert x empty

delete :: Ord a => a -> Set a -> Set a
delete x (MakeSet t) = MakeSet (RBT.delete (SetValue x) t)

fromList :: (Ord a) => [a] -> Set a
fromList = MakeSet . (foldl update' RBT.empty)
  where
    update' t x = RBT.update (SetValue x) t

fold :: (Ord a) => (a -> b -> b) -> b -> Set a -> b
fold f z (MakeSet t) = RBT.fold aggr id t z
  where
    aggr (SetValue x) lacc racc = lacc . f x . racc

toList :: (Ord a) => Set a -> [a]
toList = fold (:) []

mapMonotone :: (Ord a, Ord b) => (a -> b) -> Set a -> Set b
mapMonotone f (MakeSet t) = MakeSet (RBT.mapMonotone f' t)
  where
    f' (SetValue x) = SetValue (f x)

map :: (Ord a, Ord b) => (a -> b) -> Set a -> Set b
```

```haskell
map f (MakeSet t) = MakeSet (RBT.map f' t)
  where
    f' (SetValue x) = SetValue (f x)

union :: (Ord a) => Set a -> Set a -> Set a
union (MakeSet t1) (MakeSet t2) = MakeSet (RBT.join t2 t1)

unions :: (Ord a) => [Set a] -> Set a
unions = foldr union empty

intersection :: (Ord a) => Set a -> Set a -> Set a
intersection (MakeSet t1) (MakeSet t2) = MakeSet (RBT.meet t1 t2)

intersections :: (Ord a) => [Set a] -> Set a
intersections [] = empty
intersections xs = foldr1 intersection xs

difference :: (Ord a) => Set a -> Set a -> Set a
difference (MakeSet t1) (MakeSet t2) = MakeSet (RBT.diff t2 t1)

infixl 9 \\

(\\) :: (Ord a) => Set a -> Set a -> Set a
(\\) = difference

filter :: Ord a => (a -> Bool) -> Set a -> Set a
filter p = fold insert' empty
  where
    insert' x = if p x then insert x else id

findMin :: Ord a => Set a -> Maybe a
findMin (MakeSet t) = fmap cnv $ RBT.findMin t
  where cnv (SetValue x) = x

findMax :: Ord a => Set a -> Maybe a
findMax (MakeSet t) = fmap cnv $ RBT.findMax t
  where cnv (SetValue x) = x

isSubsetOf :: Ord a => Set a -> Set a -> Bool
x `isSubsetOf` y = isEmpty (x\\y)

isProperSubsetOf :: Ord a => Set a -> Set a -> Bool
x `isProperSubsetOf` y = x `isSubsetOf` y && (not $ isEmpty $ y\\x)

-- Not exported functions

sum :: (Num a, Ord a) => Set a -> a
sum = fold (+) 0
```

4.3 MultiSetsCompact.hs

Qui implementiamo multinsiemi più astutamente come coppie (valore,numero di ripetizioni). Si vedano le dichiarazioni delle istanze (specialmente **Ord**' (Val a), **Ord** (Val a) e **Show** (Val a)).

```haskell
module MultiSetsCompact(MultiSet,
  (\\),
  delete,
  difference,
  elem,
  empty,
```

```
  filter,
  findMax,
  findMin,
  fold,
  fromList,
  insert,
  intersection,
  intersections,
  isEmpty,
  isProperSubsetOf,
  isSubsetOf,
  map,
  mapMonotone,
  singleton,
  size,
  toList,
  union,
  unions) where

import qualified RedBlackTrees as RBT
import RedBlackTrees(RBT,Ord')
import Prelude hiding (
  elem,
  filter,
  lookup,
  map,
  sum)


data (Ord a) => Val a = MSetVal a Int

instance (Ord a) => Eq (Val a)
  where
    (MSetVal x _) == (MSetVal y _) = x == y

instance (Ord a) => Ord (Val a)
  where
    compare (MSetVal x _) (MSetVal y _) = compare x y

instance (Ord a) => Ord' (Val a)
  where
    valJoin (MSetVal x n) (MSetVal y m)
      | x == y      = MSetVal x (max (n+m) 0)
      | otherwise = error "MultiSets.valJoin: unjoinable items"

    valMeet (MSetVal x n) (MSetVal y m)
      | x == y      = MSetVal x (min n m)
      | otherwise = error "MultiSets.valMeet: unmeetable items"

instance (Ord a, Show a) => Show (Val a)
  where
    showsPrec _ (MSetVal x n) = shows x . if n==1 then id else (':':) . shows
        n

instance (Ord a, Read a) => Read (Val a)
  where
    readsPrec _ us = [ (MSetVal x n, xs) | (x,vs)   <- reads us,
                                           (":",ws) <- lex vs,
                                           (n,xs)   <- reads ws] ++
                     [ (MSetVal x 1, vs) | (x,vs)   <- reads us]
```

```
newtype MultiSet a = MakeSet (RBT (Val a))

instance (Ord a) => Eq (MultiSet a)
  where
    x == y = toList x == toList y

instance (Ord a, Show a) => Show (MultiSet a)
  where
    showsPrec _ (MakeSet t) = ('{':) . (RBT.fold aggr id t) . (end:)
      where
        end = '}'

        aggr x lacc racc = lacc . shows x . comma racc

        comma shws xs = if head ys == end then ys else (',':ys)
          where ys = shws xs

instance (Ord a, Read a) => Read (MultiSet a)
  where
    readsPrec _ = readsMultiSet
      where
        readsMultiSet ts = [ (MakeSet t, vs) | ("{", us) <- lex ts,
                                               (t, vs) <- readl us]

        readl ts = [ (RBT.empty, us) | ("}", us) <- lex ts ] ++
                   [ (RBT.update x s, vs) | (x, us) <- reads ts,
                                             (s,vs) <- readl' us]

        readl' ts = [ (RBT.empty, us) | ("}", us) <- lex ts ] ++
                    [ (RBT.update x s, ws) | (",", us) <- lex ts,
                                              (x,vs) <- reads us,
                                              (s,ws) <- readl' vs]


type Set a = MultiSet a


-- Exported functions

empty :: Ord a => Set a
empty = MakeSet RBT.empty

isEmpty :: (Ord a) => Set a -> Bool
isEmpty (MakeSet t) = RBT.isEmpty t

elem :: (Ord a) => a -> Set a -> Bool
x `elem` (MakeSet t) = RBT.lookup (MSetVal x (error "MultiSetsCompact.lookup: 
    internal error")) t /= []

size :: Ord a => Set a -> Integer
size (MakeSet t) = RBT.fold count 0 t
  where
    count _ y z = 1+y+z


insert :: (Ord a) => Int -> a -> Set a -> Set a
insert n x (MakeSet t) = MakeSet (RBT.update (MSetVal x n) t)

singleton :: (Ord a) => a -> Set a
singleton x = insert 1 x empty

fromList :: (Ord a) => [a] -> Set a
```

```haskell
fromList = MakeSet . (foldl update' RBT.empty)
  where
    update' t x = RBT.update (MSetVal x 1) t

fold :: (Ord a) => (Int -> a -> b -> b) -> b -> Set a -> b
fold f z (MakeSet t) = RBT.fold aggr id t z
  where
    aggr (MSetVal x n) lacc racc = lacc . f n x . racc

toList :: (Ord a) => Set a -> [a]
toList = fold (ntimes (:)) []

ntimes :: (a -> b -> b) -> Int -> a -> b -> b
ntimes f n x y
  | n > 0      = f x (ntimes f (n-1) x y)
  | n == 0     = y
  | otherwise  = error "MultiSets.ntimes: negative repetitions"

mapMonotone :: (Ord a, Ord b) => (a -> b) -> Set a -> Set b
mapMonotone f (MakeSet t) = MakeSet (RBT.mapMonotone f' t)
  where
    f' (MSetVal x n) = MSetVal (f x) n

map :: (Ord a, Ord b) => (a -> b) -> Set a -> Set b
map f (MakeSet t) = MakeSet (RBT.map f' t)
  where
    f' (MSetVal x n) = MSetVal (f x) n

union :: (Ord a) => Set a -> Set a -> Set a
union (MakeSet t1) (MakeSet t2) = MakeSet (RBT.join t2 t1)

unions :: (Ord a) => [Set a] -> Set a
unions = foldr union empty

intersection :: (Ord a) => Set a -> Set a -> Set a
intersection (MakeSet t1) (MakeSet t2) = MakeSet (RBT.meet t1 t2)

intersections :: (Ord a) => [Set a] -> Set a
intersections [] = empty
intersections xs = foldr1 intersection xs


delete :: Ord a => Int -> a -> Set a -> Set a
delete n x (MakeSet t) = MakeSet (del n x t)

del :: (Ord a) => Int -> a -> RBT (Val a) -> RBT (Val a)
del n x t
  | m > 0     = t1
  | otherwise = t2
  where
    t1 = RBT.update (MSetVal x (-n)) t
    (MSetVal _ m) = head (RBT.lookup (MSetVal x (error "MultiSetsCompact.del: 
        internal error")) t1)
    t2 = RBT.delete (MSetVal x m) t1

difference :: (Ord a) => Set a -> Set a -> Set a
difference (MakeSet t1) (MakeSet t2) = MakeSet (RBT.fold aggr id t2 t1)
 where
    -- aggr :: (Ord a) => Val a -> (RBT (Val a) -> RBT (Val a)) -> (RBT (Val a
        ) -> RBT (Val a)) -> (RBT (Val a) -> RBT (Val a))
    aggr (MSetVal x n) lacc racc = lacc . del n x . racc
```

```
infixl 9 \\

(\\) :: (Ord a) => Set a -> Set a -> Set a
(\\) = difference

filter :: Ord a => (a -> Bool) -> Set a -> Set a
filter p = fold insert' empty
  where
    insert' n x set@(MakeSet t) = if p x then MakeSet (RBT.update (MSetVal x n
        ) t) else set

findMin :: Ord a => Set a -> Maybe a
findMin (MakeSet t) = fmap cnv $ RBT.findMin t
  where cnv (MSetVal x _) = x

findMax :: Ord a => Set a -> Maybe a
findMax (MakeSet t) = fmap cnv $ RBT.findMax t
  where cnv (MSetVal x _) = x

isSubsetOf :: Ord a => Set a -> Set a -> Bool
x `isSubsetOf` y = isEmpty (x\\y)

isProperSubsetOf :: Ord a => Set a -> Set a -> Bool
x `isProperSubsetOf` y = x `isSubsetOf` y && (not $ isEmpty $ y\\x)

---- Not exported functions

sum :: (Num a, Ord a) => Set a -> a
sum = fold (ntimes (+)) 0
```

4.4 Tables.hs

Con poco sforzo si possono implementare Tabelle Associative da `a` a `b`, che son poi funzioni da `a` in `b`, che andiamo a mantenere estensionalmente. Qui la cosa interessante è la `lookup` che deve restituire **Maybe** b invece di [b].

```
module Tables(Table,
  empty,
  filter,
  findMax,
  findMin,
  fold,
  fromList,
  isEmpty,
  lookup,
  map,
  mapMonotone,
  merge,
  merges,
  remove,
  size,
  toList,
  update) where

import qualified RedBlackTrees as RBT
import RedBlackTrees(RBT,Ord')
import Prelude hiding (
  filter,
  lookup,
  map)
```

```haskell
data (Ord a) => Val a b = TableValue a b

instance (Ord a) => Eq (Val a b)
  where
    (TableValue k1 _) == (TableValue k2 _) = k1 == k2

instance (Ord a) => Ord (Val a b)
  where
    compare (TableValue k1 _) (TableValue k2 _) = compare k1 k2

instance (Ord a) => Ord' (Val a b)

instance (Ord a, Show a, Show b) => Show (Val a b)
  where
    showsPrec p (TableValue k v) = shows k . ("->" ++) . shows v

instance (Ord a, Read a, Read b) => Read (Val a b)
  where
    readsPrec p ts = [ (TableValue k v, ws) | (k,us) <- reads ts,
                                              ("->",vs) <- lex us,
                                              (v,ws) <- reads vs ]


newtype Table a b = Table (RBT (Val a b))

instance (Ord a, Eq b) => Eq (Table a b)
  where
    x == y = toList x == toList y

instance (Ord a, Show a, Show b) => Show (Table a b)
  where
    showsPrec _ (Table t) = ('<':) . (RBT.fold aggr id t) . (end:)
      where
        end = '>'

        -- aggr :: Show a => a -> ShowS -> ShowS -> ShowS
        aggr x lacc racc = lacc . shows x . comma racc

        -- comma :: ShowS -> ShowS
        comma shws xs = if head ys == end then ys else (',':ys)
          where ys = shws xs


instance (Ord a, Read a, Read b) => Read (Table a b)
  where
    readsPrec _ = readsTable
      where
        readsTable ts = [ (Table t, vs) | ("<", us) <- lex ts,
                                          (t, vs) <- readl us]

        readl ts = [ (RBT.empty, us) | (">", us) <- lex ts ] ++
                   [ (RBT.update x s, vs) | (x, us) <- reads ts,
                                            (s,vs) <- readl' us]

        readl' ts = [ (RBT.empty, us) | (">", us) <- lex ts ] ++
                    [ (RBT.update x s, ws) | (",", us) <- lex ts,
                                             (x,vs) <- reads us,
                                             (s,ws) <- readl' vs]


-- Exported functions
```

```haskell
empty :: Ord a => Table a b
empty = Table RBT.empty

isEmpty :: (Ord a) => Table a b -> Bool
isEmpty (Table t) = RBT.isEmpty t

lookup :: (Ord a) => a -> Table a b -> Maybe b
lookup k (Table t) = cnv asslist
  where
    asslist = RBT.lookup (TableValue k (error "Table.lookup: RBT.lookup internal error")) t

    cnv [] = Nothing
    cnv [TableValue k' v]
      | k == k'   = Just v
      | otherwise = error "Table.lookup: lookup mismatch"
    cnv _ =  error "Table.lookup: non-deterministic lookup"


size :: Ord a => Table a b -> Integer
size (Table t) = RBT.fold count 0 t
  where
    count _ y z = 1+y+z

update :: (Ord a) => (a,b) -> Table a b -> Table a b
update (k,v) (Table t) = Table (RBT.update (TableValue k v) t)

fromList :: (Ord a) => [(a,b)] -> Table a b
fromList = Table . (foldl update' RBT.empty)
  where
    update' t (k,v) = RBT.update (TableValue k v) t

fold :: (Ord a) => ((a,b) -> c -> c) -> c -> Table a b -> c
fold f z (Table t) = RBT.fold aggr id t z
  where
    aggr (TableValue k v) lacc racc = lacc . f (k,v) . racc

toList :: (Ord a) => Table a b -> [(a,b)]
toList = fold (:) []

mapMonotone :: (Ord a, Ord c) => ((a,b) -> (c,d)) -> Table a b -> Table c d
mapMonotone f (Table t) = Table (RBT.mapMonotone f' t)
  where
    f' (TableValue k v) = TableValue k' v'
      where
        (k',v') = f (k,v)

map :: (Ord a, Ord c) => ((a,b) -> (c,d)) -> Table a b -> Table c d
map f (Table t) = Table (RBT.map f' t)
  where
    f' (TableValue k v) = TableValue k' v'
      where
        (k',v') = f (k,v)

merge :: (Ord a) => Table a b -> Table a b -> Table a b
merge (Table t1) (Table t2) = Table (RBT.join t1 t2)

merges :: (Ord a) => [Table a b] -> Table a b
merges = foldr merge empty
```

```
remove :: Ord a => a -> Table a b -> Table a b
remove k (Table t) = Table (RBT.delete (TableValue k (error "Table.del: 
    internal error")) t)

filter :: Ord a => (a -> Bool) -> Table a b -> Table a b
filter p = fold insert' empty
  where
    insert' (k,v) set@(Table t) = if p k then Table (RBT.update (TableValue k
        v) t) else set

findMin :: Ord a => Table a b -> Maybe a
findMin (Table t) = case RBT.findMin t of
  Just (TableValue k _) -> Just k
  Nothing               -> Nothing

findMax :: Ord a => Table a b -> Maybe a
findMax (Table t) = case RBT.findMax t of
  Just (TableValue k _) -> Just k
  Nothing               -> Nothing

-- Not exported functions
```