HW 11 **Prof. Caldwell**
Due: 16 October 2011 **COSC 3015**
g

1 Avl Trees

In class we presented an implementation of Avl trees. I have included the code on the web-page together with this assignment. For completeness I have included it here as well.

```
module Avl (Avl, show, fmap, foldr, all, find, fork, isEmpty, empty, ht, insert, delete, join) where

  import Prelude hiding (foldr,all)
  import Data.Foldable
  import Data.Functor

  data Avl a = Null | Fork Int (Avl a) a (Avl a)

  instance (Show a) => Show (Avl a) where
    show Null = "[]"
    show (Fork h Null y Null) = "["  ++ show y ++ "]"
    show (Fork h xt y zt) = "[" ++ show xt ++ show y ++ show zt ++ "]"

  empty = Null

  instance Functor Avl where
    fmap f Null = Null
    fmap f (Fork h xt y zt) = Fork h (fmap f xt) (f y) (fmap f zt)

  instance Foldable Avl where
    foldr f id  Null = id
    foldr f id (Fork h xt y zt) = foldr f (f y (foldr f id zt)) xt

  fork :: (Avl a) -> a -> (Avl a) -> (Avl a)
  fork xt y zt = Fork h xt y zt
     where h = 1 + max (ht xt) (ht zt)

  isEmpty Null = True
  isEmpty _ = False

  ht :: (Avl a) -> Int
  ht Null = 0
  ht (Fork h _ _ _) = h

  insert :: (Ord a) => a -> Avl a -> Avl a
  insert x Null = fork Null x Null
  insert x (Fork h xt y zt)
    | (x < y) = rebalance (insert x xt) y zt
    | (x == y) = Fork h xt x zt
    | (x > y) = rebalance xt y (insert x zt)
```

```
delete :: (Ord a) ⇒ a → Avl a → Avl a
delete x Null = Null
delete x (Fork h xt y zt)
  | (x < y) = rebalance (delete x xt) y zt
  | (x == y) = join xt zt
  | (x > y) = rebalance xt y (delete x zt)

join :: Avl a → Avl a → Avl a
join xt yt = if isEmpty yt then xt else rebalance xt y zt
                  where (y,zt) = splitTree yt

splitTree :: Avl a → (a, Avl a)
splitTree (Fork h xt y zt) =
    if isEmpty xt then (y,zt) else (u,rebalance vt y zt)
       where (u,vt) = splitTree xt

bias :: Avl a → Int
bias (Fork h xt y zt) = ht xt - ht zt

rotr :: Avl a → Avl a
rotr (Fork m (Fork n ut v wt) y zt) = fork ut v (fork wt y zt)

rotl :: Avl a → Avl a
rotl (Fork m ut v (Fork n rt s tt)) = fork (fork ut v rt) s tt

rebalance :: Avl a → a → Avl a → Avl a
rebalance xt y zt
  | (hz+1 < hx) && (bias xt < 0) = rotr (fork (rotl xt) y zt)
  | (hz+1 < hx)                  = rotr (fork xt y zt)
  | (hx+1 < hz) && (0 < bias zt) = rotl (fork xt y (rotr zt))
  | (hx + 1 < hz)                = rotl (fork xt y zt)
  | otherwise                    = fork xt y zt
      where hx = ht xt
            hz = ht zt
```

The assignment here is to build a type of finite functions where the underlying representation is the *Avl* tree. In class we discussed how the *Avl* trees are like sets - if you add an element to the set that is already there, it does not chance the set.

A finite function is a set of functional pairs. Recall the definition of functionality for a function f from a to b.

$$\forall x : a.\ \forall y, z : b.\ (\langle x, y\rangle \in f \wedge \langle x, z\rangle \in f) \Rightarrow y = z$$

That is, no two distinct elements can be paired with the same first element. Thus, to model functions with *Avl* trees, you will need to be careful never to add a pair (x, y) to the tree when there is already a pair in the tree with x as its first element. Also, you will want to order the pairs in the tree by their first elements. There is an elegant way to do this in Haskell by creating a special type of *FPair*. You can instantiate the type *FPair* in the *Eq* type class by ignoring the second element and you can instantiate *FPair* in the *Ord* type class by ordering on the first element. Then, an *Avl* tree of FPairs will satisfy the constraints just mentioned. Here is a module of *FPair*.

```
module FPair where
data FPair a b = P a b
instance (Show a, Show b) ⇒ Show (FPair a b) where
   show (P x y) = show x ++ " := " ++ show
instance (Eq a) ⇒ Eq (FPair a b)
   (P x y) == (P z w) = x ==
instance (Ord a) ⇒ Ord (FPair a b) where
  (P x y) le (P z w) = x <=
```

Now a type *FinFun* of finite functions can be defined where the implementation is defined as follows:

```
data FinFun a b = FinFun (Avl (FPair a b))
instance (Show a, Show b) ⇒ Show (FinFun a b) where
   show (FinFun f) = show (foldr (:) [] f)
```

Exercise 1.1. Based on this implementation idea - you need to implement the following constants and operations on finite functions. I have included some test cases with this assignment.

```
empty :: FinFun a b
apply :: (Eq a) ⇒ (FinFun a b) → a → b
update :: (Ord a) ⇒ FinFun a b → (a, b) → FinFun a b
dom :: FinFun a b → [a]
range :: FinFun a b → [b]
injection :: FinFun a b → Bool
```

We briefly describe each.

```
empty :: FinFun a b
```

The empty *FinFun* behaves as a function with an empty domain and range.

```
apply :: (Eq a) ⇒ (FinFun a b) → a → b
```

apply f x applies the *FinFun* f to the argument x. If there is no *FPair* in f with first element x, raise an exception. If x is the first element of some element $P\ x\ y$ return y.

```
update :: (Ord a) ⇒ FinFun a b → (a, b) → FinFun a b
```

A call of the form *update f (x,y)* inserts the *FPair*, $P\ x\ y$ into the *FinFun* f. You might study the *Avl* tree code for insert to see why this might work. Consider the equality and order relation for `FPairs`.

dom :: FinFun a b → [a]
range :: FinFun a b → [b]

The functions return lists of the domain elements and the range elements for a *FinFun*. Note that there is a *foldr* operator defined on *Avl* trees which you may find useful.

injection :: (Eq a, Show a, Eq b) =¿ FinFun a b -¿ Bool

A function is an injection of every distinct element of the domain gets mapped to a distinct element of the range *i.e.* no two elements x and y ($x \neq y$) of the domain get mapped to the same element of the range. Write a predicate to check if a *FinFun* is an injection.