L28: Advanced functional programming

Exercise 2

Due on 25th February 2016

Submission instructions

Your solutions for this exericse should be handed in to the Graduate Education Office by 4pm on the due date. Additionally, please email the completed file `exercise2.ml` to `jeremy.yallop@cl.cam.ac.uk`.

AA trees

The following definition (adapted from Lecture 8) defines a type `btree` of perfectly-balanced binary trees:

```
type z = Z : z
type 'n s = S : 'n -> 'n s

type (_, _) btree =
| Leaf : ('a, z) btree
| Branch : ('a, 'n) btree * 'a * ('a, 'n) btree -> ('a, 'n s) btree
```

The second parameter of the `btree` type is an index representing the height of the tree. A node `Branch(l, v, r)` is constrained to have a height one greater than `l`, and `r` is constrained to have the same height as `l`.

AA trees (named after their developer, Arne Andersson), are another type of self-balancing binary tree. An AA tree contains two kinds of `Branch` node:

Vertical nodes are ordinary balanced tree nodes – the right sub-tree must have the same height as the left sub-tree, and the node has a height one greater than the sub-trees.

Horizontal nodes have a height one greater than their left sub-tree, but the same height as their right sub-tree.

There is one additional constraint: the right sub-tree of a horizontal node must be a vertical node. This constraint ensures that the longest path down the right-hand side of a tree is at most twice as long as the longest path down the left-hand side. Leaf nodes are considered horizontal for the purposes of this restriction.

Figure 1 shows an example of an AA tree. The nodes `B` and `H` are horizontal nodes and the other non-leaf nodes are vertical.

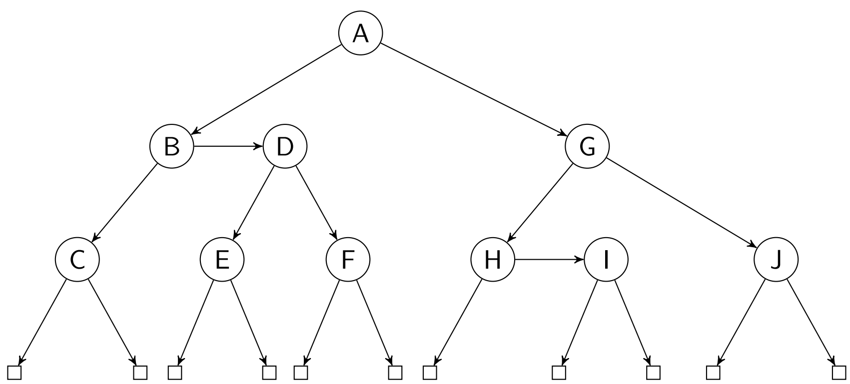


Figure 1: An example of an AA tree

1. Fill in the `?` in the following OCaml code to create a type `atree` which represents an AA tree, such that:

- The types `z` and `s` are used in indices to represent the heights of the nodes.
- The types `vert` and `horiz` are used in indices to represent the kinds of nodes.
- Each `Branch` node contains a `dir` value which indicates what kind of node it is: `H` for horizontal and `V` for vertical.
- The four type parameters of the `dir` type are indices representing respectively:
  (i) The kind of the node
  (ii) The height of the node
  (iii) The kind of the node's right sub-tree
  (iv) The height of the node's right sub-tree
- The first two type parameters of the `atree` type are indices representing respectively:
  (i) The kind of the tree
  (ii) The height of the tree

  The third type parameter of the `atree` type is the type of elements in the tree.

```
type z = Z : z
type _ s = S : 'n -> 'n s

type vert = V and horiz = H

type ('o,'n,'ro,'rn) dir =
  | H : (?, ?, ?, ?) dir
  | V : (?, ?, ?, ?) dir

type ('o,'n,'a) atree =
  | Leaf : (horiz, z, 'a) atree
  | Branch : (?, ?, ?, ?) dir
           * (? , ?, 'a) atree * 'a * (?, ?, 'a) atree
           -> (?, ?, 'a) atree
```

(5 marks)

2. The following sum type represents the results of comparing two values:

```
type compare = LessThan | Equal | GreaterThan
```

A comparison function of type `'a -> 'a -> compare` returns

- `LessThan` if the first argument is less than the second
- `Equal` if the first argument is equal to the second,
- `GreaterThan` if the first argument is greater than the second

Implement the membership function `member` of type:

```
val member : ('a -> 'a -> compare) -> 'a -> ('o, 'n, 'a) atree
               -> bool
```

such that `member cmp x t` returns `true` iff the value `x` is present in the AA tree `t` *assuming that the elements of the tree are in order according to the comparison function `cmp`.* ("In order" means that the elements in the left sub-tree of a node are less than the element in the node and the elements in the right sub-tree of a node are greater than the element in the node.)

(2 marks)

Insertion

Inserting an element into an AA tree may change the height and kind of the tree. A horizontal node may become a vertical node with an increased height, or a vertical node may become horizontal. The type `inserted` represents the result of inserting an element:

```
type (_,_,_) inserted =
  | Up : (vert, 'n s, 'a) atree -> (horiz, 'n, 'a) inserted
  | VertToHoriz : (horiz, 'n, 'a) atree -> (vert, 'n, 'a) inserted
  | Same : ('o, 'n, 'a) atree -> ('o, 'n, 'a) inserted
```

3. Given

   - an AA tree `l` of height n
   - a value `v`
   - an AA tree `r` of height $n+1$

   then the AA tree `Branch(H, l, v, r)` is not necessarily a valid AA tree because `r` may be a horizontal node.

   However, if `r` is horizontal then it is possible to construct a valid AA tree by performing the rotation shown in Figure 2, resulting in a tree whose height is one greater than the original.

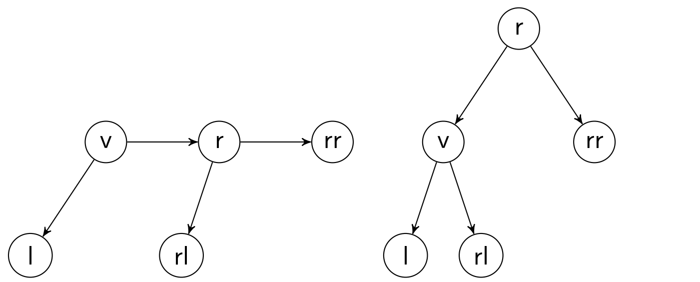
   Implement a function `split` of type:

   ```
   val split : (_,'n,'a) atree -> 'a -> ('o,'n s,'a) atree
                 -> ('o,'n s,'a) inserted
   ```

   such that `split l v r` contains a valid AA tree built from `l`, `v` and `r`. If `r` is horizontal then `split` should perform the rotation shown in Figure 2 to create the tree.

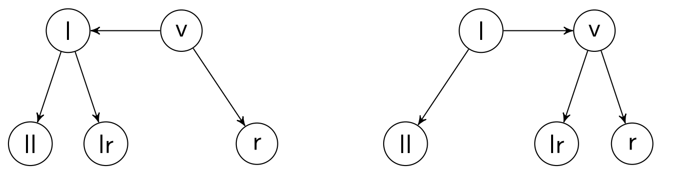
   Take care to ensure that your implementation maintains the order of elements!

(3 marks)



(a) Before: an invalid AA tree (b) After: a valid AA tree

Figure 2: The *split* rotation



(a) Before: an invalid AA tree (b) After: a valid AA tree

Figure 3: The *skew* rotation

4. Given

   - a node kind `k`
   - a vertical AA tree `l` of height $n + 1$
   - a value `v`
   - an AA tree `r` that is
     (i) vertical and of height $n + 1$ if `k` is `H`
     (ii) of height n if `k` is `V`

   then the AA tree `Branch(k, l, v, r)` is not a valid AA tree because the height of `l` is too great. However, it is possible to construct a valid AA tree from `l`, `v` and `r` by performing the rotation shown in Figure 3 followed by a call to `split`.

   Implement a function `skew` of type:

```
val skew : ('o, 'n s, 'ro, 'r) dir
           -> (vert, 'n s, 'a) atree -> 'a -> ('ro, 'r, 'a) atree
           -> ('o, 'n s, 'a) inserted
```

   such that `skew k l v r` contains a valid AA tree built from `l`, `v` and `r`, either by performing the rotation shown in Figure 2 and then calling `split` to create the tree, or otherwise.

   Take care to ensure that your implementation maintains the order of elements!

(2 marks)

5. Insertion of an element into an *ordered* AA tree is very similar to insertion of an element into an ordered binary tree, except that in some cases `split` and `skew` are needed to maintain the constraints on node heights and kinds.

   Implement the insertion function `insert` of type:

```
val insert : ('a -> 'a -> compare) -> 'a -> ('o, 'n, 'a) atree
               -> ('o, 'n, 'a) inserted
```

   such that `insert cmp x t` returns an ordered AA tree that contains the value `x` and all elements of the ordered AA tree `t`. *The elements of `t` are assumed to be in order according to the comparison function `cmp`, and the elements of the resulting tree must also be in order according to the comparison function `cmp`.* You can assume that the input tree contains no duplicates and should ensure that the result contains no duplicates.

(4 marks)

Deletion

Deleting an element from an AA tree may change the height and kind of the tree. A vertical node may decrease in height, or a horizontal node may become vertical. The type `deleted` represents the result of deleting an element:

```
type (_,_,_) deleted =
  | Down : (_, 'n, 'a) atree -> (vert, 'n s, 'a) deleted
  | HorizToVert : (vert, 'n, 'a) atree -> (horiz, 'n, 'a) deleted
  | Same : ('o, 'n, 'a) atree -> ('o, 'n, 'a) deleted
```

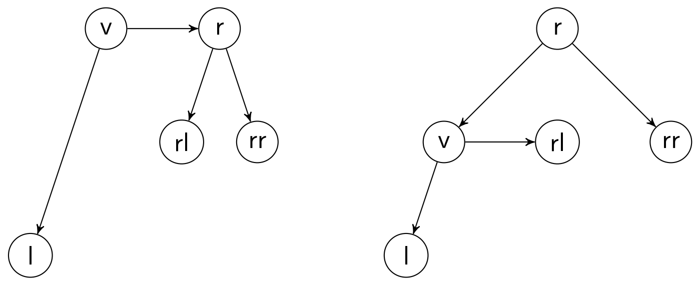
6. Given

   - a node kind `k`
   - an AA tree `l` of height n
   - a value `v`
   - an AA tree `r` that is
     (i) vertical and of height $n+2$ if `k` is `H`
     (ii) of height $n+1$ if `k` is `V`

   then the AA tree `Branch(k, l, v, r)` is not a valid AA tree because the height of `l` is too low. However, it is possible to create a valid AA tree from `k`, `l`, `v` and `r` as follows:

   (i) If `k` is `V` then use `split`.

   (ii) If `k` is `H` then perform the rotation shown in Figure 4. After this rotation a call to `split` may be required to re-balance the left-hand side of the tree, which may then cause the root of the tree to become unbalanced and require a call to `skew` to re-balance it.



(a) Before (b) After

Figure 4: Split 2

Implement a function `split2` of type:

```
val split2 : ('o,'n s s,'ro,'r) dir
                -> (_,'n,'a) atree -> 'a -> ('ro,'r,'a) atree
                -> ('o, 'n s s, 'a) deleted
```

such that `split2 k l v r` contains a valid AA tree built from `l`, `v` and `r`.

Take care to ensure that your implementation maintains the order of elements!

(4 marks)

7. Given

   - an AA tree `l` of height $n+1$
   - a value `v`
   - an AA tree `r` of height n

   then the AA tree `Branch(V, l, v, r)` is not a valid AA tree because the height of `r` is too low. However, it is possible to build a valid AA tree from `l`, `v` and `r` by applying `split2` to the result of the rotation shown in Figure 5. In some cases an application of `skew` may be necessary before the application of `split2`.

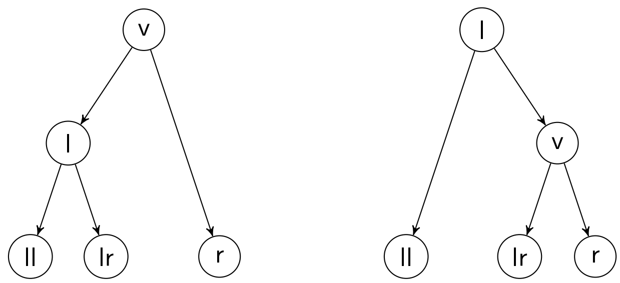
   Implement a function `skew2` of type:

```
val skew2: (_,'n s,'a) atree -> 'a -> (_,'n,'a) atree
             -> (vert, 'n s s, 'a) deleted =
```

   such that `skew2 l v r` contains a valid AA tree built from `l`, `v` and `r`.

   Take care to ensure that your implementation maintains the order of elements!

   (3 marks)



(a) Before (b) After

Figure 5: Skew 2

8. The left-most element of an AA tree can be removed by descending the left branches of the tree, removing the lowest element, then using `split2` and `skew2` to maintain the constraints on heights and kinds of nodes whilst ascending back up the tree.

   Implement the function `pop` of type:

```
val pop : ('o, 'n s, 'a) atree -> 'a * ('o, 'n s, 'a) deleted
```

   such that `pop t` returns the least element of the ordered AA tree `t` and an ordered AA tree containing all the elements of `t` except the least element. You can assume that the input trees contain no duplicates and should ensure that the result contains no duplicates.

   (4 marks)

9. Deletion of an element from an *ordered* AA tree is very similar to deletion of an element into an ordered binary tree, except that in some cases `split2` and `skew2` are needed to maintain the constraints on node heights and kinds. First the element to be deleted is located in the tree, then its *in-order successor* is removed from its place in the tree (using `pop` on the right child of the deleted element) and used to replace the deleted element.

   Implement the deletion function `delete` of type:

```
val delete : ('a -> 'a -> cmp) -> 'a -> ('o, 'n, 'a) atree
               -> ('o, 'n, 'a) deleted
```

   such that `delete cmp x t` returns an ordered AA tree that contains all elements of the ordered AA tree `t` except elements that are equal to `x` according to `cmp`. *The elements of `t` are assumed to be in order according to the comparison function `cmp`, and the elements of the resulting tree must also be in order according to the comparison function `cmp`.* You can assume that the input trees contain no duplicates and should ensure that the result contains no duplicates.

(5 marks)

10. AA trees are a good data-structure for implementing sets. Implement a functor `Set` of the following module type:

```
module Set :
  functor (X : sig type t val compare : t -> t -> compare end) ->
    sig
      type t
      val empty : t
      val member : X.t -> t -> bool
      val add : X.t -> t -> t
      val remove : X.t -> t -> t
    end
```

which implements sets using `atree`.

(3 marks)