Programming in Haskell
Aug–Nov 2015

LECTURE 19

OCTOBER 27, 2015

S P Suresh
Chennai Mathematical Institute

More on Set

- **Previous lecture**: Implementation of `Set` supporting
  - `insert`, `delete`, `search`, `empty` (creating an empty set) and `isempty` (checking if a set is empty)
  - O(log N) time for each operation
  - These are **dictionary operations**
- **This lecture**
  - `union, intersect, setdiff`
  - **Set operations**

Set implementations

- `newtype Set a = Setof [a]`

- ```
  empty = Setof []
  isempty (Setof l) = null l
  search (Setof l) x = elem x l
  insert (Setof l) x = Setof (x:l)
  delete (Setof l) x = Setof (filter (/= x) l)
  ```

- `empty`, `isempty`, `insert` take $O(1)$ time, while `search` and `delete` take $O(N)$ time, where the set has N elements
  ```

Set implementations

- ```
  newtype Set a = Setof [a]
  ```
- ```
  union :: Set a -> Set a -> Set a
  union (Setof xs) (Setof ys) = (Setof (xs++ys))
  ```
- ```
  intersect :: Eq a => Set a -> Set a -> Set a
  intersect (Setof xs) (Setof ys) = Setof [y | y <- ys,
                                           elem y xs]
  ```
- ```
  setdiff :: Eq a => Set a -> Set a -> Set a
  setdiff (Setof xs) (Setof ys) = Setof [x | x <- xs,
                                         not (elem x ys)]
  ```
- $O(M)$, $O(MN)$, and $O(MN)$ time, where M and N are the set sizes

type, data, newtype

- `type Set a = [a]`
  - `Set a` is a **synonym** for `[a]`, internal structure visible
    `tail s` is legal, for a set `s`
- `data Set a = Setof [a]`
  - Wrapper around `[a]`, internal structure not accessible
    Haskell spends a lot of time wrapping and unwrapping
- `newtype Set a = Setof [a]`
  - Internal structure not visible, but efficiency like `type`
    Only for data types with a single constructor

Set implementations

- We could maintain a set of **distinct elements**
- `newtype Set a = Setof [a]`
-
```
empty = Setof []
isempty (Setof l) = null l
search (Setof l) x = elem x l
insert (Setof l) x = Setof (x:filter (/= x) l)
delete (Setof l) x = Setof (filter (/= x) l)
```
- `empty`, `isempty` take O(1) time, while `insert`, `search` and `delete` take O(N) time, where the set has N elements

Set implementations

* ```
  union :: Eq a => Set a -> Set a -> Set a
  union (Setof xs) (Setof [])     = Setof xs
  union (Setof xs) (Setof (y:ys) =
                        union (insert (Setof xs) y)
                              (Setof ys)
  ```

* O(MN) time, where M and N are the set sizes
  ```

Set implementations

- ```
  intersect :: Eq a => Set a -> Set a -> Set a
  intersect (Setof xs) (Setof ys) =
                     Setof [y | y <- ys, elem y xs]
  ```

- ```
  setdiff :: Eq a => Set a -> Set a -> Set a
  setdiff (Setof xs) (Setof ys) =
               Setof [x | x <- xs, not (elem x ys)]
  ```

- O(MN) time, where M and N are the set sizes

Set implementations

- If the elements have an order, we could maintain a **sorted list**
- `newtype Set a = Setof [a]`
- `empty = Setof []`
- `isempty (Setof l) = null l`
- `search (Setof l) x = elem x l`
- `empty`, `isempty` take O(1) time, while `search` takes O(N) time, where the set has N elements

Set implementations

- ```
  insert (Setof l) x = Setof (insertaux x l)
      where
          insertaux x []  = [x]
          insertaux x (y:ys)
              | x == y    = y:ys
              | x < y     = x:y:ys
              | otherwise = y: insertaux x ys
  ```

- `delete (Setof l) x = Setof (filter (/= x) l)`

- Both take O(N) time, where the set has N elements

- The dictionary operations are implemented the same way as before – no gains
  ```

Set implementations

- For sorted lists, the set operations can be based on **merge**
- ```
  union (Setof xs) (Setof ys) =
                            Setof (unionmerge xs ys)
    where
      unionmerge [] ys    = ys
      unionmerge xs []    = xs
      unionmerge (x:xs) (y:ys)
              | x < y     = x:(unionmerge xs (y:ys))
              | y < x     = y:(unionmerge (x:xs) ys)
              | otherwise = x:(unionmerge xs ys)
  ```
- O(M+N) time
  ```

Set implementations

- For sorted lists, the set operations can be based on **merge**
- ```
  intersect (Setof xs) (Setof ys) =
                      Setof (intersectmerge xs ys)
    where
      intersectmerge [] ys     = []
      intersectmerge xs []     = []
      intersectmerge (x:xs) (y:ys)
          | x < y     = intersectmerge xs (y:ys)
          | y < x     = intersectmerge (x:xs) ys
          | otherwise = x:(intersectmerge xs ys)
  ```
- O(M+N) time
  ```

Set implementations

- For sorted lists, the set operations can be based on **merge**
- ```
setdiff (Setof xs) (Setof ys) =
                      Setof (setdiffmerge xs ys)
  where
    setdiffmerge [] ys   = []
    setdiffmerge xs []   = xs
    setdiffmerge (x:xs) (y:ys)
        | x < y   = x:(setdiffmerge xs (y:ys))
        | y < x   = setdiffmerge (x:xs) ys
        | otherwise = setdiffmerge xs ys
```
- O(M+N) time
```

Set implementations

- If the elements have an order, we could use an **AVL tree**
- `newtype Set a = Setof (AVLTree a)`
- ```
  empty                 = Setof Nil
  isempty (Setof t)     = t == Nil
  search (Setof t) x    = AVLTree.search t x
  insert (Setof t) x    = Setof (AVLTree.insert t x)
  delete (Setof t) x    = Setof (AVLTree.delete t x)
  ```
- All operations take O(logN) time, where the set has N elements
  ```

Set implementations

- If the elements have an order, we could use an **AVL tree**
- How do we implement the set operations?
  - Convert the trees to sorted lists and use the merge-based operations
  - Convert the resulting sorted list back to a tree
- Converting a tree to sorted list – **inorder**
- Converting sorted list to an AVL tree – **mkAVLTree**

inorder

- ```
  inorder :: Ord a => AVLTree a -> [a]
  inorder Nil = []
  inorder (Node tl x h tr) = inorder tl ++
                                      [x] ++
                             inorder tr
  ```

- If the tree is balanced and has N nodes, the time complexity of `inorder` is
  $T(N) = 2\,T(N/2) + O(N/2)$

- $T(N) = O(N \log N)$
  ```

More efficient inorder

- ```
  inorderaux :: Ord a => AVLTree a -> [a] -> [a]
  inorderaux Nil l = l
  inorderaux (Node tl x h tr) l =
              inorderaux tl (x:inorderaux tr l)
  ```
- `inorder t = inorderaux t []`
- If the tree is balanced and has N nodes, the time complexity of `inorderaux` is
  $T(N) = 2\,T(N/2) + O(1)$
- $T(N) = O(N)$
  ```

mkAVLTree

- If `l` is sorted, we want `mkAVLTree` to be a balanced binary search tree
- **Naive method**: split down the middle, and recursively form the left and right subtrees

mkAVLTree

- ```
  mkAVLTree :: Ord a => [a] -> AVLTree a
  mkAVLtree [] = Nil
  mkAVLtree [x] = Node Nil x 1 Nil
  mkAVLtree l =  Node tl root h tr
      where
          m = (length l) `div` 2
          root == l!!m
          tl = mkAVLTree (take m l)
          tr = mkAVLTree (drop (m+1) l)
          h = 1 + max (height tl) (height tr)
  ```
  ```

Complexity of mkAVLTree

- If there are N elements, we need
  - O(N) time to compute length, take, drop, access the middle etc.
  - 2T(N/2) to recursively build the left and right subtrees
  - T(N) = 2T(N/2) + O(N)
  - T(N) = O(N log N)

More efficient mkAVLTree

- ```
  mkAVLTreeaux :: Ord a => [a] -> Int -> (AVLTree a, [a])
  mkAVLTreeaux l n = (mkAVLTree (take n l), drop n l)
  ```
- So `mkAVLTree l = fst (mkAVLTreeaux l (length l))`
- ```
  mkAVLTreeaux [] n = (Nil, [])
  mkAVLTreeaux l  0 = (Nil, l)
  mkAVLTreeaux l  n = (Node t1 root h t2, l2)
      where
          m = n `div` 2
          (t1, root:rest) = mkAVLTreeaux l m
          (t2, l2) = mkAVLTreeaux rest (n-(m+1))
  ```
- $T(N) = 2T(N/2) + O(1)$. $T(N) = O(N)$.

Set operations

- ```
  union (Setof t1) (Setof t2) = Setof (mkAVLTree l)
    where
      l = unionmerge (inorder t1) (inorder t2)
  ```
- ```
  intersect (Setof t1) (Setof t2) = Setof (mkAVLTree l)
    where
      l = intersectmerge (inorder t1) (inorder t2)
  ```
- ```
  setdiff (Setof t1) (Setof t2) = Setof (mkAVLTree l)
    where
      l = setdiffmerge (inorder t1) (inorder t2)
  ```
- $O(M+N)$ time, where M and N are the sizes of the two sets
  ```

Summary

- Set operations **union**, **intersect**, and **setdiff**
- Linear time implementations with the aid of smart **inorder**, **mkAVLTree** and **merge**