Haskell Tutorial: Sets

September 27, 2019

[1]:
```
:opt no-lint
```

0.1 Ordered Sets

Haskell has support for typed ordered sets with fast insert, delete, and lookup operations. It is highly recommended to use the set implementation instead of abusing lists to represent ordered sets. To use the set functions, you have to import `Data.Set`. It is a recommended to use a qualified import in order to handle name clashes with functions defined by the prelude. And using a qualified import may also help with readability.

[3]:
```
import qualified Data.Set as Set
```

A common way to create sets is to create them from a list. But there are also convenient ways to create an empty set or a set has a single member, called a *singleton*.

[4]:
```
s0 = Set.empty
s1 = Set.singleton 'c'
s2 = Set.fromList "Hello World"
s3 = Set.fromList ['a'..'z']
```

The `null` function can be used to test whether a set is empty while the `size` function returns the number of elements of a set.

[5]:
```
map Set.null [s0,s1,s2,s3]
map Set.size [s0,s1,s2,s3]
```

```
[True,False,False,False]
```

```
[0,1,8,26]
```

The `toAscList` function returns the elements of a set as a list in ascending order. Similarly, `toDescList` returns the elements of a set as a list in descending order.

[6]:
```
Set.toAscList s3
Set.toDescList s3
```

```
"abcdefghijklmnopqrstuvwxyz"
```

```
"zyxwvutsrqponmlkjihgfedcba"
```

Set `member` function can be used to test whether a value is a member of a given list and the `isSubsetOf` function can be used to test whether the first set is a subset of the second set.

[8]:
```
map (Set.member 'o') [s0,s1,s2,s3]
map (Set.isSubsetOf s2) [s0,s1,s2,s3]
```

```
[False,False,True,True]
```

```
[False,False,True,False]
```

The `insert` and `delete` functions can be used to add and delete elements. There is also a `filter` function that can be used to create a set from all members of a set where a given predicate is true.

[8]:
```
Set.null $ Set.delete 42 $ Set.insert 42 $ Set.insert 42 s0
Set.toAscList $ Set.filter (< 'g') s3
```

```
True
```

```
"abcdef"
```

You can use the standard set operations `union`, `intersection`, and `difference`. The `isSubsetOf` function tests whether the first argument is a subset of the second argument and the `member` function tests whether a member is in a set. The `disjoint` function tests whether two sets are disjoint, i.e., they have no common elements.

[9]:
```
Set.union s2 s3
Set.intersection s2 s3
Set.difference s2 s3
```

```
fromList " HWabcdefghijklmnopqrstuvwxyz"
```

```
fromList "delor"
```

```
fromList " HW"
```

Since the sets have an order, it is possible to split sets at a certain elem. Note that the splitting element is not required to be a member of the set.

[11]:
```
sp = Set.split 'm' s3
Set.elems $ fst sp
```

```
Set.elems $ snd sp
```

```
"abcdefghijkl"


"nopqrstuvwxyz"
```

[12]:
```
Set.split 32 $ Set.fromList [0,10..60]
```

```
(fromList [0,10,20,30],fromList [40,50,60])
```

Many other funtions like `map`, `filter`, `take`, `drop`, and various folds are implemented for sets as well.

Sets are internally represented as balanced ordered binary trees. It is possible to look at the internal representation by using the `showTree` function. Note that there may be leaf nodes without a value.

[7]:
```
putStrLn $ Set.showTree s3
```

```
'h'
+--'d'
|  +--'b'
|  |  +--'a'
|  |  +--'c'
|  +--'f'
|     +--'e'
|     +--'g'
+--'p'
   +--'l'
   |  +--'j'
   |  |  +--'i'
   |  |  +--'k'
   |  +--'n'
   |     +--'m'
   |     +--'o'
   +--'t'
      +--'r'
      |  +--'q'
      |  +--'s'
      +--'x'
         +--'v'
         |  +--'u'
         |  +--'w'
         +--'y'
            +--|
            +--'z'
```