Data.Set

Portability	portable
Stability	provisional
Maintainer	libraries@haskell.org

Contents

Description

An efficient implementation of sets.

This module is intended to be imported `qualified`, to avoid name clashes with Prelude functions. eg.

```
import Data.Set as Set
```

The implementation of `Set` is based on *size balanced* binary trees (or trees of *bounded balance*) as described by:

- Stephen Adams, "*Efficient sets: a balancing act*", Journal of Functional Programming 3(4):553-562, October 1993, http://www.swiss.ai.mit.edu/~adams/BB.
- J. Nievergelt and E.M. Reingold, "*Binary search trees of bounded balance*", SIAM journal of computing 2(1), March 1973.

Note that the implementation is *left-biased* -- the elements of a first argument are always perferred to the second, for example in `union` or `insert`. Of course, left-biasing can only be observed when equality is an equivalence relation instead of structural equality.

Synopsis

```
data Set a
```

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

```haskell
null :: Set a -> Bool

size :: Set a -> Int

member :: Ord a => a -> Set a -> Bool

isSubsetOf :: Ord a => Set a -> Set a -> Bool

isProperSubsetOf :: Ord a => Set a -> Set a -> Bool

empty :: Set a

singleton :: a -> Set a

insert :: Ord a => a -> Set a -> Set a

delete :: Ord a => a -> Set a -> Set a

union :: Ord a => Set a -> Set a -> Set a

unions :: Ord a => [Set a] -> Set a

difference :: Ord a => Set a -> Set a -> Set a

intersection :: Ord a => Set a -> Set a -> Set a

filter :: Ord a => (a -> Bool) -> Set a -> Set a

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)

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

mapMonotonic :: (a -> b) -> Set a -> Set b

fold :: (a -> b -> b) -> b -> Set a -> b

findMin :: Set a -> a

findMax :: Set a -> a

deleteMin :: Set a -> Set a

deleteMax :: Set a -> Set a

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

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

elems :: Set a -> [a]

toList :: Set a -> [a]

fromList :: Ord a => [a] -> Set a

toAscList :: Set a -> [a]

fromAscList :: Eq a => [a] -> Set a

fromDistinctAscList :: [a] -> Set a

showTree :: Show a => Set a -> String

showTreeWith :: Show a => Bool -> Bool -> Set a -> String

valid :: Ord a => Set a -> Bool

emptySet :: Set a
```

```haskell
mkSet :: Ord a => [a] -> Set a
setToList :: Set a -> [a]
unitSet :: a -> Set a
elementOf :: Ord a => a -> Set a -> Bool
isEmptySet :: Set a -> Bool
cardinality :: Set a -> Int
unionManySets :: Ord a => [Set a] -> Set a
minusSet :: Ord a => Set a -> Set a -> Set a
mapSet :: (Ord a, Ord b) => (b -> a) -> Set b -> Set a
intersect :: Ord a => Set a -> Set a -> Set a
addToSet :: Ord a => Set a -> a -> Set a
delFromSet :: Ord a => Set a -> a -> Set a
```

Set type

`data Set a`

A set of values `a`.

Instances

```haskell
Typeable1 Set
(Data a, Ord a) => Data (Set a)
Eq a => Eq (Set a)
Ord a => Monoid (Set a)
Ord a => Ord (Set a)
Show a => Show (Set a)
```

Operators

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

O(n+m). See `difference`.

Query

`null :: Set a -> Bool`

O(1). Is this the empty set?

`size :: Set a -> Int`

O(1). The number of elements in the set.

`member :: Ord a => a -> Set a -> Bool`

O(log n). Is the element in the set?

`isSubsetOf :: Ord a => Set a -> Set a -> Bool`

O(n+m). Is this a subset? `(s1 isSubsetOf s2)` tells whether `s1` is a subset of `s2`.

`isProperSubsetOf :: Ord a => Set a -> Set a -> Bool`

O(n+m). Is this a proper subset? (ie. a subset but not equal).

Construction

`empty :: Set a`

O(1). The empty set.

`singleton :: a -> Set a`

O(1). Create a singleton set.

`insert :: Ord a => a -> Set a -> Set a`

O(log n). Insert an element in a set.

`delete :: Ord a => a -> Set a -> Set a`

O(log n). Delete an element from a set.

Combine

`union :: Ord a => Set a -> Set a -> Set a`

O(n+m). The union of two sets. Uses the efficient *hedge-union* algorithm. Hedge-union is more efficient on (bigset `union` smallset).

`unions :: Ord a => [Set a] -> Set a`

The union of a list of sets: (`unions == foldl union empty`).

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

O(n+m). Difference of two sets. The implementation uses an efficient *hedge* algorithm comparable with *hedge-union*.

`intersection :: Ord a => Set a -> Set a -> Set a`

O(n+m). The intersection of two sets. Intersection is more efficient on (bigset `intersection` smallset).

Filter

`filter :: Ord a => (a -> Bool) -> Set a -> Set a`

O(n). Filter all elements that satisfy the predicate.

`partition :: Ord a => (a -> Bool) -> Set a -> (Set a, Set a)`

O(n). Partition the set into two sets, one with all elements that satisfy the predicate and one with all elements that don't satisfy the predicate. See also `split`.

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

O(log n). The expression (`split x set`) is a pair `(set1,set2)` where all elements in `set1` are lower than `x` and all elements in `set2` larger than `x`. `x` is not found in neither `set1` nor `set2`.

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

O(log n). Performs a `split` but also returns whether the pivot element was found in the original set.

Map

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

*O(n*log n)*. `map f s` is the set obtained by applying `f` to each element of `s`.

It's worth noting that the size of the result may be smaller if, for some `(x,y)`, `x /= y && f x == f y`

mapMonotonic :: (a -> b) -> Set a -> Set b

O(n). The

`mapMonotonic f s == map f s`, but works only when `f` is monotonic. *The precondition is not checked.* Semi-formally, we have:

```
and [x < y ==> f x < f y | x <- ls, y <- ls]
                    ==> mapMonotonic f s == map f s
    where ls = toList s
```

Fold

fold :: (a -> b -> b) -> b -> Set a -> b

O(n). Fold over the elements of a set in an unspecified order.

Min/Max

findMin :: Set a -> a

O(log n). The minimal element of a set.

findMax :: Set a -> a

O(log n). The maximal element of a set.

deleteMin :: Set a -> Set a

O(log n). Delete the minimal element.

deleteMax :: Set a -> Set a

O(log n). Delete the maximal element.

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

O(log n). Delete and find the minimal element.

```
deleteFindMin set = (findMin set, deleteMin set)
```

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

O(log n). Delete and find the maximal element.

```
deleteFindMax set = (findMax set, deleteMax set)
```

Conversion

List

elems :: Set a -> [a]

O(n). The elements of a set.

toList :: Set a -> [a]

O(n). Convert the set to a list of elements.

fromList :: Ord a => [a] -> Set a

*O(n*log n)*. Create a set from a list of elements.

Ordered list

toAscList :: Set a -> [a]

O(n). Convert the set to an ascending list of elements.

fromAscList :: Eq a => [a] -> Set a

O(n). Build a set from an ascending list in linear time. *The precondition (input list is ascending) is not checked.*

fromDistinctAscList :: [a] -> Set a

O(n). Build a set from an ascending list of distinct elements in linear time. *The precondition (input list is strictly ascending) is not checked.*

Debugging

showTree :: Show a => Set a -> String

O(n). Show the tree that implements the set. The tree is shown in a compressed, hanging format.

showTreeWith :: Show a => Bool -> Bool -> Set a -> String

O(n). The expression (`showTreeWith hang wide map`) shows the tree that implements the set. If `hang` is `True`, a *hanging* tree is shown otherwise a rotated tree is shown. If `wide` is `True`, an extra wide version is shown.

```
Set> putStrLn $ showTreeWith True False $ fromDistinctAscList [1..5]
4
+--2
|  +--1
|  +--3
+--5

Set> putStrLn $ showTreeWith True True $ fromDistinctAscList [1..5]
4
|
+--2
|  |
```

```
|   +--1
|   |
|   +--3
|
+--5

Set> putStrLn $ showTreeWith False True $ fromDistinctAscList [1..5]
+--5
|
4
|
|   +--3
|   |
+--2
    |
    +--1
```

valid :: Ord a => Set a -> Bool

O(n). Test if the internal set structure is valid.

Old interface, DEPRECATED

emptySet :: Set a

Obsolete equivalent of `empty`.

mkSet :: Ord a => [a] -> Set a

Obsolete equivalent of `fromList`.

setToList :: Set a -> [a]

Obsolete equivalent of `elems`.

unitSet :: a -> Set a

Obsolete equivalent of `singleton`.

elementOf :: Ord a => a -> Set a -> Bool

Obsolete equivalent of `member`.

isEmptySet :: Set a -> Bool

Obsolete equivalent of `null`.

cardinality :: Set a -> Int

Obsolete equivalent of `size`.

unionManySets :: Ord a => [Set a] -> Set a

Obsolete equivalent of `unions`.

minusSet :: Ord a => Set a -> Set a -> Set a

Obsolete equivalent of `difference`.

mapSet :: (Ord a, Ord b) => (b -> a) -> Set b -> Set a

Obsolete equivalent of `map`.

intersect :: Ord a => Set a -> Set a -> Set a

Obsolete equivalent of `intersection`.

`addToSet :: Ord a => Set a -> a -> Set a`

Obsolete equivalent of `flip insert`.

`delFromSet :: Ord a => Set a -> a -> Set a`

Obsolete equivalent of `flip delete`.

Data.SetExtras

Portability	experimental
Stability	experimental
Maintainer	João Ferreira, Alexandra Mendes

Contents

Description

Extra functions to use with Sets

Synopsis

```
filterSet :: Ord a => (a -> Bool) -> Set a -> Set a
dunion :: Ord a => Set (Set a) -> Set a
readFile_Set :: (Read a, Ord a, Show c) => FilePath -> (Set a -> c) -> IO c
interact_Set :: (Read a, Ord a, Show c) => FilePath -> FilePath -> (Set a -> c) -> IO ()
```

Sets' basic functions

`filterSet` `:: Ord a => (a -> Bool) -> Set a -> Set a`

Given a predicate p and a set, yields a set whose elements validate p.

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

Given a set of sets ss, the resulting set is the union of all the elements (these are sets themselves) of ss, i.e. it contains all the elements of all the sets of ss.

File IO

`readFile_Set` `:: (Read a, Ord a, Show c) => FilePath -> (Set a -> c) -> IO c`

Applies a given function to a set read from a given file.

`interact_Set` `:: (Read a, Ord a, Show c) => FilePath -> FilePath -> (Set a -> c) -> IO ()`

Applies readFile_Set and writes the result in a given file.

Data.Map

Portability	portable
Stability	provisional
Maintainer	libraries@haskell.org

Contents

Description

An efficient implementation of maps from keys to values (dictionaries).

This module is intended to be imported `qualified`, to avoid name clashes with Prelude functions. eg.

```
import Data.Map as Map
```

The implementation of `Map` is based on *size balanced* binary trees (or trees of *bounded balance*) as described by:

- Stephen Adams, "*Efficient sets: a balancing act*", Journal of Functional Programming 3(4):553-562, October 1993, http://www.swiss.ai.mit.edu/~adams/BB.
- J. Nievergelt and E.M. Reingold, "*Binary search trees of bounded balance*", SIAM

journal of computing 2(1), March 1973.

Synopsis

```
data Map k a
(!) :: Ord k => Map k a -> k -> a
(\\) :: Ord k => Map k a -> Map k b -> Map k a
null :: Map k a -> Bool
size :: Map k a -> Int
member :: Ord k => k -> Map k a -> Bool
lookup :: (Monad m, Ord k) => k -> Map k a -> m a
findWithDefault :: Ord k => a -> k -> Map k a -> a
empty :: Map k a
singleton :: k -> a -> Map k a
insert :: Ord k => k -> a -> Map k a -> Map k a
insertWith :: Ord k => (a -> a -> a) -> k -> a -> Map k a -> Map k a
insertWithKey :: Ord k => (k -> a -> a -> a) -> k -> a -> Map k a -> Map k a
insertLookupWithKey :: Ord k => (k -> a -> a -> a) -> k -> a -> Map k a -> (Maybe
a, Map k a)
delete :: Ord k => k -> Map k a -> Map k a
adjust :: Ord k => (a -> a) -> k -> Map k a -> Map k a
adjustWithKey :: Ord k => (k -> a -> a) -> k -> Map k a -> Map k a
update :: Ord k => (a -> Maybe a) -> k -> Map k a -> Map k a
updateWithKey :: Ord k => (k -> a -> Maybe a) -> k -> Map k a -> Map k a
updateLookupWithKey :: Ord k => (k -> a -> Maybe a) -> k -> Map k a -> (Maybe a,
Map k a)
union :: Ord k => Map k a -> Map k a -> Map k a
unionWith :: Ord k => (a -> a -> a) -> Map k a -> Map k a -> Map k a
unionWithKey :: Ord k => (k -> a -> a -> a) -> Map k a -> Map k a -> Map k a
unions :: Ord k => [Map k a] -> Map k a
unionsWith :: Ord k => (a -> a -> a) -> [Map k a] -> Map k a
difference :: Ord k => Map k a -> Map k b -> Map k a
differenceWith :: Ord k => (a -> b -> Maybe a) -> Map k a -> Map k b -> Map k a
differenceWithKey :: Ord k => (k -> a -> b -> Maybe a) -> Map k a -> Map k b -> Map
k a
intersection :: Ord k => Map k a -> Map k b -> Map k a
intersectionWith :: Ord k => (a -> b -> c) -> Map k a -> Map k b -> Map k c
intersectionWithKey :: Ord k => (k -> a -> b -> c) -> Map k a -> Map k b -> Map k c
```

```haskell
map :: (a -> b) -> Map k a -> Map k b

mapWithKey :: (k -> a -> b) -> Map k a -> Map k b

mapAccum :: (a -> b -> (a, c)) -> a -> Map k b -> (a, Map k c)

mapAccumWithKey :: (a -> k -> b -> (a, c)) -> a -> Map k b -> (a, Map k c)

mapKeys :: Ord k2 => (k1 -> k2) -> Map k1 a -> Map k2 a

mapKeysWith :: Ord k2 => (a -> a -> a) -> (k1 -> k2) -> Map k1 a -> Map k2 a

mapKeysMonotonic :: (k1 -> k2) -> Map k1 a -> Map k2 a

fold :: (a -> b -> b) -> b -> Map k a -> b

foldWithKey :: (k -> a -> b -> b) -> b -> Map k a -> b

elems :: Map k a -> [a]

keys :: Map k a -> [k]

keysSet :: Map k a -> Set k

assocs :: Map k a -> [(k, a)]

toList :: Map k a -> [(k, a)]

fromList :: Ord k => [(k, a)] -> Map k a

fromListWith :: Ord k => (a -> a -> a) -> [(k, a)] -> Map k a

fromListWithKey :: Ord k => (k -> a -> a -> a) -> [(k, a)] -> Map k a

toAscList :: Map k a -> [(k, a)]

fromAscList :: Eq k => [(k, a)] -> Map k a

fromAscListWith :: Eq k => (a -> a -> a) -> [(k, a)] -> Map k a

fromAscListWithKey :: Eq k => (k -> a -> a -> a) -> [(k, a)] -> Map k a

fromDistinctAscList :: [(k, a)] -> Map k a

filter :: Ord k => (a -> Bool) -> Map k a -> Map k a

filterWithKey :: Ord k => (k -> a -> Bool) -> Map k a -> Map k a

partition :: Ord k => (a -> Bool) -> Map k a -> (Map k a, Map k a)

partitionWithKey :: Ord k => (k -> a -> Bool) -> Map k a -> (Map k a, Map k a)

split :: Ord k => k -> Map k a -> (Map k a, Map k a)

splitLookup :: Ord k => k -> Map k a -> (Map k a, Maybe a, Map k a)

isSubmapOf :: (Ord k, Eq a) => Map k a -> Map k a -> Bool

isSubmapOfBy :: Ord k => (a -> b -> Bool) -> Map k a -> Map k b -> Bool

isProperSubmapOf :: (Ord k, Eq a) => Map k a -> Map k a -> Bool

isProperSubmapOfBy :: Ord k => (a -> b -> Bool) -> Map k a -> Map k b -> Bool

lookupIndex :: (Monad m, Ord k) => k -> Map k a -> m Int

findIndex :: Ord k => k -> Map k a -> Int

elemAt :: Int -> Map k a -> (k, a)

updateAt :: (k -> a -> Maybe a) -> Int -> Map k a -> Map k a
```

deleteAt :: Int -> Map k a -> Map k a

findMin :: Map k a -> (k, a)

findMax :: Map k a -> (k, a)

deleteMin :: Map k a -> Map k a

deleteMax :: Map k a -> Map k a

deleteFindMin :: Map k a -> ((k, a), Map k a)

deleteFindMax :: Map k a -> ((k, a), Map k a)

updateMin :: (a -> Maybe a) -> Map k a -> Map k a

updateMax :: (a -> Maybe a) -> Map k a -> Map k a

updateMinWithKey :: (k -> a -> Maybe a) -> Map k a -> Map k a

updateMaxWithKey :: (k -> a -> Maybe a) -> Map k a -> Map k a

showTree :: (Show k, Show a) => Map k a -> String

showTreeWith :: (k -> a -> String) -> Bool -> Bool -> Map k a -> String

valid :: Ord k => Map k a -> Bool

Map type

data **Map** k a

A Map from keys `k` to values `a`.

Instances

Typeable2 Map
Functor (Map k)
(Data k, Data a, Ord k) => Data (Map k a)
(Eq k, Eq a) => Eq (Map k a)
Ord k => Monoid (Map k v)
(Ord k, Ord v) => Ord (Map k v)
(Show k, Show a) => Show (Map k a)

Operators

(!) :: Ord k => Map k a -> k -> a

O(log n). Find the value at a key. Calls `error` when the element can not be found.

(\\) :: Ord k => Map k a -> Map k b -> Map k a

O(n+m). See `difference`.

Query

null :: Map k a -> Bool

O(1). Is the map empty?

size :: Map k a -> Int

O(1). The number of elements in the map.

member :: Ord k => k -> Map k a -> Bool
O(log n). Is the key a member of the map?

lookup :: (Monad m, Ord k) => k -> Map k a -> m a
O(log n). Lookup the value at a key in the map.

findWithDefault :: Ord k => a -> k -> Map k a -> a
O(log n). The expression `(findWithDefault def k map)` returns the value at key `k` or returns `def` when the key is not in the map.

Construction

empty :: Map k a
O(1). The empty map.

singleton :: k -> a -> Map k a
O(1). A map with a single element.

Insertion

insert :: Ord k => k -> a -> Map k a -> Map k a
O(log n). Insert a new key and value in the map.

insertWith :: Ord k => (a -> a -> a) -> k -> a -> Map k a -> Map k a
O(log n). Insert with a combining function.

insertWithKey :: Ord k => (k -> a -> a -> a) -> k -> a -> Map k a -> Map k a
O(log n). Insert with a combining function.

insertLookupWithKey :: Ord k => (k -> a -> a -> a) -> k -> a -> Map k a -> (Maybe a, Map k a)
O(log n). The expression (`insertLookupWithKey f k x map`) is a pair where the first element is equal to (`lookup k map`) and the second element equal to (`insertWithKey f k x map`).

Delete/Update

delete :: Ord k => k -> Map k a -> Map k a
O(log n). Delete a key and its value from the map. When the key is not a member of the map, the original map is returned.

adjust :: Ord k => (a -> a) -> k -> Map k a -> Map k a
O(log n). Adjust a value at a specific key. When the key is not a member of the map, the original map is returned.

adjustWithKey :: Ord k => (k -> a -> a) -> k -> Map k a -> Map k a
O(log n). Adjust a value at a specific key. When the key is not a member of the map, the original map is returned.

update :: Ord k => (a -> Maybe a) -> k -> Map k a -> Map k a

O(log n). The expression (`update f k map`) updates the value `x` at `k` (if it is in the map). If (`f x`) is `Nothing`, the element is deleted. If it is (`Just y`), the key `k` is bound to the new value `y`.

updateWithKey `:: Ord k => (k -> a -> Maybe a) -> k -> Map k a -> Map k a`

O(log n). The expression (`updateWithKey f k map`) updates the value `x` at `k` (if it is in the map). If (`f k x`) is `Nothing`, the element is deleted. If it is (`Just y`), the key `k` is bound to the new value `y`.

updateLookupWithKey `:: Ord k => (k -> a -> Maybe a) -> k -> Map k a -> (Maybe a, Map k a)`

O(log n). Lookup and update.

Combine

Union

union `:: Ord k => Map k a -> Map k a -> Map k a`

O(n+m). The expression (`union t1 t2`) takes the left-biased union of `t1` and `t2`. It prefers `t1` when duplicate keys are encountered, i.e. (`union == unionWith const`). The implementation uses the efficient *hedge-union* algorithm. Hedge-union is more efficient on (bigset `union` smallset)?

unionWith `:: Ord k => (a -> a -> a) -> Map k a -> Map k a -> Map k a`

O(n+m). Union with a combining function. The implementation uses the efficient *hedge-union* algorithm.

unionWithKey `:: Ord k => (k -> a -> a -> a) -> Map k a -> Map k a -> Map k a`

O(n+m). Union with a combining function. The implementation uses the efficient *hedge-union* algorithm. Hedge-union is more efficient on (bigset `union` smallset).

unions `:: Ord k => [Map k a] -> Map k a`

The union of a list of maps: (`unions == foldl union empty`).

unionsWith `:: Ord k => (a -> a -> a) -> [Map k a] -> Map k a`

The union of a list of maps, with a combining operation: (`unionsWith f == foldl (unionWith f) empty`).

Difference

difference `:: Ord k => Map k a -> Map k b -> Map k a`

O(n+m). Difference of two maps. The implementation uses an efficient *hedge* algorithm comparable with *hedge-union*.

differenceWith `:: Ord k => (a -> b -> Maybe a) -> Map k a -> Map k b -> Map k a`

O(n+m). Difference with a combining function. The implementation uses an efficient *hedge* algorithm comparable with *hedge-union*.

differenceWithKey `:: Ord k => (k -> a -> b -> Maybe a) -> Map k a -> Map k b -> Map k a`

O(n+m). Difference with a combining function. When two equal keys are encountered, the combining function is applied to the key and both values. If it returns `Nothing`, the element is discarded (proper set difference). If it returns (`Just y`), the element is updated with a new value `y`. The implementation uses an efficient *hedge* algorithm comparable with *hedge-union*.

Intersection

`intersection` `:: Ord k => Map k a -> Map k b -> Map k a`

O(n+m). Intersection of two maps. The values in the first map are returned, i.e. (`intersection m1 m2 == intersectionWith const m1 m2`).

`intersectionWith` `:: Ord k => (a -> b -> c) -> Map k a -> Map k b -> Map k c`

O(n+m). Intersection with a combining function.

`intersectionWithKey` `:: Ord k => (k -> a -> b -> c) -> Map k a -> Map k b -> Map k c`

O(n+m). Intersection with a combining function. Intersection is more efficient on (bigset `intersection` smallset)

Traversal

Map

`map` `:: (a -> b) -> Map k a -> Map k b`

O(n). Map a function over all values in the map.

`mapWithKey` `:: (k -> a -> b) -> Map k a -> Map k b`

O(n). Map a function over all values in the map.

`mapAccum` `:: (a -> b -> (a, c)) -> a -> Map k b -> (a, Map k c)`

O(n). The function `mapAccum` threads an accumulating argument through the map in ascending order of keys.

`mapAccumWithKey` `:: (a -> k -> b -> (a, c)) -> a -> Map k b -> (a, Map k c)`

O(n). The function `mapAccumWithKey` threads an accumulating argument through the map in ascending order of keys.

`mapKeys` `:: Ord k2 => (k1 -> k2) -> Map k1 a -> Map k2 a`

*O(n*log n)*. `mapKeys f s` is the map obtained by applying `f` to each key of `s`.

The size of the result may be smaller if `f` maps two or more distinct keys to the same new key. In this case the value at the smallest of these keys is retained.

`mapKeysWith` `:: Ord k2 => (a -> a -> a) -> (k1 -> k2) -> Map k1 a -> Map k2 a`

*O(n*log n)*. `mapKeysWith c f s` is the map obtained by applying `f` to each key of `s`.

The size of the result may be smaller if `f` maps two or more distinct keys to the same new key. In this case the associated values will be combined using `c`.

`mapKeysMonotonic` `:: (k1 -> k2) -> Map k1 a -> Map k2 a`

O(n). `mapKeysMonotonic f s == mapKeys f s`, but works only when `f` is strictly monotonic. *The* *precondition is not checked.* Semi-formally, we have:

```
and [x < y ==> f x < f y | x <- ls, y <- ls]
                    ==> mapKeysMonotonic f s == mapKeys f s
    where ls = keys s
```

Fold

fold :: (a -> b -> b) -> b -> Map k a -> b

O(n). Fold the values in the map, such that `fold f z == foldr f z . elems`. For example,

```
elems map = fold (:) [] map
```

foldWithKey :: (k -> a -> b -> b) -> b -> Map k a -> b

O(n). Fold the keys and values in the map, such that `foldWithKey f z == foldr (uncurry f) z . toAscList`. For example,

```
keys map = foldWithKey (\k x ks -> k:ks) [] map
```

Conversion

elems :: Map k a -> [a]

O(n). Return all elements of the map in the ascending order of their keys.

keys :: Map k a -> [k]

O(n). Return all keys of the map in ascending order.

keysSet :: Map k a -> Set k

O(n). The set of all keys of the map.

assocs :: Map k a -> [(k, a)]

O(n). Return all key/value pairs in the map in ascending key order.

Lists

toList :: Map k a -> [(k, a)]

O(n). Convert to a list of key/value pairs.

fromList :: Ord k => [(k, a)] -> Map k a

*O(n*log n)*. Build a map from a list of key/value pairs. See also `fromAscList`.

fromListWith :: Ord k => (a -> a -> a) -> [(k, a)] -> Map k a

*O(n*log n)*. Build a map from a list of key/value pairs with a combining function. See also `fromAscListWith`.

fromListWithKey :: Ord k => (k -> a -> a -> a) -> [(k, a)] -> Map k a

*O(n*log n)*. Build a map from a list of key/value pairs with a combining function. See also `fromAscListWithKey`.

Ordered lists

toAscList `:: Map k a -> [(k, a)]`

O(n). Convert to an ascending list.

fromAscList `:: Eq k => [(k, a)] -> Map k a`

O(n). Build a map from an ascending list in linear time. *The precondition (input list is ascending) is not checked.*

fromAscListWith `:: Eq k => (a -> a -> a) -> [(k, a)] -> Map k a`

O(n). Build a map from an ascending list in linear time with a combining function for equal keys. *The precondition (input list is ascending) is not checked.*

fromAscListWithKey `:: Eq k => (k -> a -> a -> a) -> [(k, a)] -> Map k a`

O(n). Build a map from an ascending list in linear time with a combining function for equal keys. *The precondition (input list is ascending) is not checked.*

fromDistinctAscList `:: [(k, a)] -> Map k a`

O(n). Build a map from an ascending list of distinct elements in linear time. *The precondition is not checked.*

Filter

filter `:: Ord k => (a -> Bool) -> Map k a -> Map k a`

O(n). Filter all values that satisfy the predicate.

filterWithKey `:: Ord k => (k -> a -> Bool) -> Map k a -> Map k a`

O(n). Filter all keys/values that satisfy the predicate.

partition `:: Ord k => (a -> Bool) -> Map k a -> (Map k a, Map k a)`

O(n). partition the map according to a predicate. The first map contains all elements that satisfy the predicate, the second all elements that fail the predicate. See also `split`.

partitionWithKey `:: Ord k => (k -> a -> Bool) -> Map k a -> (Map k a, Map k a)`

O(n). partition the map according to a predicate. The first map contains all elements that satisfy the predicate, the second all elements that fail the predicate. See also `split`.

split `:: Ord k => k -> Map k a -> (Map k a, Map k a)`

O(log n). The expression (`split k map`) is a pair `(map1,map2)` where the keys in `map1` are smaller than `k` and the keys in `map2` larger than `k`. Any key equal to `k` is found in neither `map1` nor `map2`.

splitLookup `:: Ord k => k -> Map k a -> (Map k a, Maybe a, Map k a)`

O(log n). The expression (`splitLookup k map`) splits a map just like `split` but also returns `lookup k map`.

Submap

isSubmapOf `:: (Ord k, Eq a) => Map k a -> Map k a -> Bool`

O(n+m). This function is defined as (`isSubmapOf = isSubmapOfBy (==)`).

isSubmapOfBy :: Ord k => (a -> b -> Bool) -> Map k a -> Map k b -> Bool

O(n+m). The expression (`isSubmapOfBy f t1 t2`) returns `True` if all keys in `t1` are in tree `t2`, and when `f` returns `True` when applied to their respective values. For example, the following expressions are all `True`:

```
isSubmapOfBy (==) (fromList [('a',1)]) (fromList [('a',1),('b',2)])
isSubmapOfBy (<=) (fromList [('a',1)]) (fromList [('a',1),('b',2)])
isSubmapOfBy (==) (fromList [('a',1),('b',2)]) (fromList [('a',1),('b',2)])
```

But the following are all `False`:

```
isSubmapOfBy (==) (fromList [('a',2)]) (fromList [('a',1),('b',2)])
isSubmapOfBy (<)  (fromList [('a',1)]) (fromList [('a',1),('b',2)])
isSubmapOfBy (==) (fromList [('a',1),('b',2)]) (fromList [('a',1)])
```

isProperSubmapOf :: (Ord k, Eq a) => Map k a -> Map k a -> Bool

O(n+m). Is this a proper submap? (ie. a submap but not equal). Defined as (`isProperSubmapOf = isProperSubmapOfBy (==)`).

isProperSubmapOfBy :: Ord k => (a -> b -> Bool) -> Map k a -> Map k b -> Bool

O(n+m). Is this a proper submap? (ie. a submap but not equal). The expression (`isProperSubmapOfBy f m1 m2`) returns `True` when `m1` and `m2` are not equal, all keys in `m1` are in `m2`, and when `f` returns `True` when applied to their respective values. For example, the following expressions are all `True`:

```
isProperSubmapOfBy (==) (fromList [(1,1)]) (fromList [(1,1),(2,2)])
isProperSubmapOfBy (<=) (fromList [(1,1)]) (fromList [(1,1),(2,2)])
```

But the following are all `False`:

```
isProperSubmapOfBy (==) (fromList [(1,1),(2,2)]) (fromList [(1,1),(2,2)])
isProperSubmapOfBy (==) (fromList [(1,1),(2,2)]) (fromList [(1,1)])
isProperSubmapOfBy (<)  (fromList [(1,1)])       (fromList [(1,1),(2,2)])
```

Indexed

lookupIndex :: (Monad m, Ord k) => k -> Map k a -> m Int

O(log n). Lookup the *index* of a key. The index is a number from *0* up to, but not including, the `size` of the map.

findIndex :: Ord k => k -> Map k a -> Int

O(log n). Return the *index* of a key. The index is a number from *0* up to, but not including, the `size` of the map. Calls `error` when the key is not a `member` of the map.

elemAt :: Int -> Map k a -> (k, a)

O(log n). Retrieve an element by *index*. Calls `error` when an invalid index is used.

updateAt :: (k -> a -> Maybe a) -> Int -> Map k a -> Map k a

O(log n). Update the element at *index*. Calls `error` when an invalid index is used.

deleteAt :: Int -> Map k a -> Map k a

O(log n). Delete the element at *index*. Defined as (`deleteAt i map = updateAt (k x ->`

`Nothing) i map`

Min/Max

findMin :: `Map k a -> (k, a)`

O(log n). The minimal key of the map.

findMax :: `Map k a -> (k, a)`

O(log n). The maximal key of the map.

deleteMin :: `Map k a -> Map k a`

O(log n). Delete the minimal key.

deleteMax :: `Map k a -> Map k a`

O(log n). Delete the maximal key.

deleteFindMin :: `Map k a -> ((k, a), Map k a)`

O(log n). Delete and find the minimal element.

deleteFindMax :: `Map k a -> ((k, a), Map k a)`

O(log n). Delete and find the maximal element.

updateMin :: `(a -> Maybe a) -> Map k a -> Map k a`

O(log n). Update the value at the minimal key.

updateMax :: `(a -> Maybe a) -> Map k a -> Map k a`

O(log n). Update the value at the maximal key.

updateMinWithKey :: `(k -> a -> Maybe a) -> Map k a -> Map k a`

O(log n). Update the value at the minimal key.

updateMaxWithKey :: `(k -> a -> Maybe a) -> Map k a -> Map k a`

O(log n). Update the value at the maximal key.

Debugging

showTree :: `(Show k, Show a) => Map k a -> String`

O(n). Show the tree that implements the map. The tree is shown in a compressed, hanging format.

showTreeWith :: `(k -> a -> String) -> Bool -> Bool -> Map k a -> String`

O(n). The expression (`showTreeWith showelem hang wide map`) shows the tree that implements the map. Elements are shown using the `showElem` function. If `hang` is `True`, a *hanging* tree is shown otherwise a rotated tree is shown. If `wide` is `True`, an extra wide version is shown.

```
Map> let t = fromDistinctAscList [(x,()) | x <- [1..5]]
Map> putStrLn $ showTreeWith (\k x -> show (k,x)) True False t
(4,())
+--(2,())
|  +--(1,())
|  +--(3,())
```

```
+--(5,())

Map> putStrLn $ showTreeWith (\k x -> show (k,x)) True True t
(4,())
|
+--(2,())
|   |
|   +--(1,())
|   |
|   +--(3,())
|
+--(5,())

Map> putStrLn $ showTreeWith (\k x -> show (k,x)) False True t
+--(5,())
|
(4,())
|
|   +--(3,())
|   |
+--(2,())
    |
    +--(1,())
```

valid **::** Ord k => Map k a -> Bool

O(n). Test if the internal map structure is valid.

Data.FiniteMapExtras

Portability	experimental
Stability	experimental
Maintainer	Joao Ferreira, Alexandra Mendes

Contents

Description

Extra functions to use with FiniteMaps (includes all VDM-SL functions)

Synopsis

```
domFM :: Ord a => FiniteMap a b -> Set a
rngFM :: (Ord a, Ord b) => FiniteMap a b -> Set b
munion :: Ord a => FiniteMap a b -> FiniteMap a b -> Maybe (FiniteMap a b)
munionRel :: (Ord a, Ord b) => FiniteMap a b -> FiniteMap a b -> Rel a b
(+++) :: Ord a => FiniteMap a b -> FiniteMap a b -> FiniteMap a b
override :: Ord a => FiniteMap a b -> FiniteMap a b -> FiniteMap a b
merge :: Ord a => Set (FiniteMap a b) -> Maybe (FiniteMap a b)
(<:) :: Ord a => Set a -> FiniteMap a b -> FiniteMap a b
(<-:) :: Ord a => Set a -> FiniteMap a b -> FiniteMap a b
(>:) :: (Ord a, Ord b) => FiniteMap a b -> Set b -> FiniteMap a b
(>-:) :: (Ord a, Ord b) => FiniteMap a b -> Set b -> FiniteMap a b
compFM :: Ord a => FiniteMap a a -> FiniteMap a a -> Maybe (FiniteMap a a)
(***) :: (Ord a, Num b) => FiniteMap a a -> b -> Maybe (FiniteMap a a)
inverse :: (Ord key, Ord elt) => FiniteMap key elt -> Maybe (FiniteMap elt key)
inverse2 :: (Ord key, Ord elt) => FiniteMap key elt -> Maybe (FiniteMap elt key)
m :: Ord key => FiniteMap key elt -> key -> Maybe elt
injective :: (Ord key, Ord elt) => FiniteMap key elt -> Bool
mkr :: (Ord key, Ord elt) => FiniteMap key elt -> Rel key elt
fmToSet :: (Ord key, Ord elt) => FiniteMap key elt -> Set (key, elt)
setOfKeysFM :: (Ord key, Ord elt) => FiniteMap key elt -> Set key
setOfEltsFM :: (Ord key, Ord elt) => FiniteMap key elt -> Set elt
readFile_FM :: (Read a, Read b, Ord a, Show c) => FilePath -> (FiniteMap a b -> c) -> IO c
interact_FM :: (Read a, Read b, Ord a, Show c) => FilePath -> FilePath -> (FiniteMap a b -> c) -> IO ()
```

FiniteMaps' basic functions

```
domFM :: Ord a => FiniteMap a b -> Set a
```

Yields the domain (the set of keys) of a map.

VDM: dom m

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

VDM: rng m

`munion :: Ord a => FiniteMap a b -> FiniteMap a b -> Maybe (FiniteMap a b)`

Yields a map combined by two other maps, such that the resulting map maps the elements of the domain of both maps. The two maps must have disjoint domains.

VDM: munion m1 m2

`munionRel :: (Ord a, Ord b) => FiniteMap a b -> FiniteMap a b -> Rel a b`

Yields a relation that has all pairs (key,elt) of the two given maps.

`(+++) :: Ord a => FiniteMap a b -> FiniteMap a b -> FiniteMap a b`

Overrides and merges two maps. It is like munion, except that both maps don't need to be compatible; the values of the second map override the ones of the first.

`override :: Ord a => FiniteMap a b -> FiniteMap a b -> FiniteMap a b`

Same as (+++).

VDM: m1 ++ m2

`merge :: Ord a => Set (FiniteMap a b) -> Maybe (FiniteMap a b)`

Given a set of maps, yields the map that is contructed by merging them all. The maps must be compatible.

VDM: merge ms

`(<:) :: Ord a => Set a -> FiniteMap a b -> FiniteMap a b`

Given a set and a map, creates the map consisting of the elements whose key is in the set. The set don't need to be a subset of the given map's domain.

VDM: s <: m

`(<-:) :: Ord a => Set a -> FiniteMap a b -> FiniteMap a b`

Given a set and a map, creates the map consisting of the elements whose key is not in the set. The set don't need to be a subset of the given map's domain.

VDM: s <-: m

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

Given a map and a set, creates the map consisting of the elements whose information value is in the set. The set don't need to be a subset of the given map's range.

VDM: m :> s

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

Given a map and a set, creates the map consisting of the elements whose information value is not in the set. The set don't need to be a subset of the given map's range.

VDM: m :-> s

`compFM :: Ord a => FiniteMap a a -> FiniteMap a a -> Maybe (FiniteMap a a)`

Given two maps m1 and m2, yields the map that is created by composing m2 elements with m1 elements. The resulting map is a map with the same domain as m2. The information value corresponding to a key is the one found by first applying m2 to the key and then applying m1 to the result. rngFM m2 must be a subset of domFM m1.

VDM: m1 comp m2

`(***) :: (Ord a, Num b) => FiniteMap a a -> b -> Maybe (FiniteMap a a)`

Given a map m and a positive integer n, yields the map where m is composed with itself n times. n=0 yields the identity map where each element of domFM m is map into itself; n=1 yields m itself. For n>1, the range of m must be a subset of domFM m.

VDM: m ** n

`inverse :: (Ord key, Ord elt) => FiniteMap key elt -> Maybe (FiniteMap elt key)`

Given a map m, yields the inverse map of m. m must be a 1-to-1 mapping.

VDM: inverse m

inverse2 :: `(Ord key, Ord elt) => FiniteMap key elt -> Maybe (FiniteMap elt key)`

Given a map m, yields the inverse map of m. m must be a 1-to-1 mapping. This is a slightly more efficient version than inverse.

VDM: inverse m

m :: `Ord key => FiniteMap key elt -> key -> Maybe elt`

Given a map and a key, yields the information value associated with that key, which must be in the domain of m.

VDM: m(d)

injective :: `(Ord key, Ord elt) => FiniteMap key elt -> Bool`

Given a map m, returns true if m is injective.

Extra functions

mkr :: `(Ord key, Ord elt) => FiniteMap key elt -> Rel key elt`

Given a map m, yields the set of pairs (key,elt) where m(key)=elt, ie, builds the relation defined by the map. mkr means 'make relation'.

fmToSet :: `(Ord key, Ord elt) => FiniteMap key elt -> Set (key, elt)`

Same as mkr.

setOfKeysFM :: `(Ord key, Ord elt) => FiniteMap key elt -> Set key`

Given a map, yields the set of keys. It is the same as domFM.

setOfEltsFM :: `(Ord key, Ord elt) => FiniteMap key elt -> Set elt`

Given a map, yields the set of elements. It is the same as rngFM.

File IO

readFile_FM :: `(Read a, Read b, Ord a, Show c) => FilePath -> (FiniteMap a b -> c) -> IO c`

Applies a given function to a map read from a given file.

interact_FM :: `(Read a, Read b, Ord a, Show c) => FilePath -> FilePath -> (FiniteMap a b -> c) -> IO ()`

Applies readFile_FM and writes the result in a given file.