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.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.