Data.Vector

A library for boxed vectors (that is, polymorphic arrays capable of holding any Haskell value). The vectors come in two flavours:

- mutable
- immutable

and support a rich interface of both list-like operations, and bulk array operations.

For unboxed arrays, use Data.Vector.Unboxed

Contents

Boxed vectors

```
data Vector a
```
Source

Boxed vectors, supporting efficient slicing.

Instances

```
Alternative Vector
Monad Vector
Functor Vector
MonadPlus Vector
Applicative Vector
Foldable Vector
Traversable Vector
Vector Vector a
IsList (Vector a)
Eq a => Eq (Vector a)
Data a => Data (Vector a)
Ord a => Ord (Vector a)
Read a => Read (Vector a)
Show a => Show (Vector a)
Monoid (Vector a)
NFData a => NFData (Vector a)
Typeable (* -> *) Vector
type Mutable Vector = MVector
type Item (Vector a) = a
```

```
data MVector s a
```
Source

Mutable boxed vectors keyed on the monad they live in (`IO` or `ST s`).

Instances

```
MVector MVector a
Typeable (* -> * -> *) MVector
```

Accessors

Length information

length :: Vector a -> Int — Source

O(1) Yield the length of the vector.

null :: Vector a -> Bool — Source

O(1) Test whether a vector if empty

Indexing

(!) :: Vector a -> Int -> a — Source

O(1) Indexing

(!?) :: Vector a -> Int -> Maybe a — Source

O(1) Safe indexing

head :: Vector a -> a — Source

O(1) First element

last :: Vector a -> a — Source

O(1) Last element

unsafeIndex :: Vector a -> Int -> a — Source

O(1) Unsafe indexing without bounds checking

unsafeHead :: Vector a -> a — Source

O(1) First element without checking if the vector is empty

unsafeLast :: Vector a -> a — Source

O(1) Last element without checking if the vector is empty

Monadic indexing

indexM :: Monad m => Vector a -> Int -> m a — Source

O(1) Indexing in a monad.

The monad allows operations to be strict in the vector when necessary. Suppose vector copying is implemented like this:

```
copy mv v = ... write mv i (v ! i) ...
```

For lazy vectors, `v ! i` would not be evaluated which means that `mv` would unnecessarily retain a reference to `v` in each element written.

With `indexM`, copying can be implemented like this instead:

```
copy mv v = ... do
                  x <- indexM v i
                  write mv i x
```

Here, no references to v are retained because indexing (but *not* the elements) is evaluated eagerly.

headM :: Monad m => Vector a -> m a Source

O(1) First element of a vector in a monad. See indexM for an explanation of why this is useful.

lastM :: Monad m => Vector a -> m a Source

O(1) Last element of a vector in a monad. See indexM for an explanation of why this is useful.

unsafeIndexM :: Monad m => Vector a -> Int -> m a Source

O(1) Indexing in a monad without bounds checks. See indexM for an explanation of why this is useful.

unsafeHeadM :: Monad m => Vector a -> m a Source

O(1) First element in a monad without checking for empty vectors. See indexM for an explanation of why this is useful.

unsafeLastM :: Monad m => Vector a -> m a Source

O(1) Last element in a monad without checking for empty vectors. See indexM for an explanation of why this is useful.

Extracting subvectors (slicing)

slice Source

```
:: Int        i starting index
-> Int        n length
-> Vector a
-> Vector a
```

O(1) Yield a slice of the vector without copying it. The vector must contain at least i+n elements.

init :: Vector a -> Vector a Source

O(1) Yield all but the last element without copying. The vector may not be empty.

tail :: Vector a -> Vector a Source

O(1) Yield all but the first element without copying. The vector may not be empty.

take :: Int -> Vector a -> Vector a Source

O(1) Yield at the first n elements without copying. The vector may contain less than n elements in which case it is returned unchanged.

drop :: Int -> Vector a -> Vector a Source

O(1) Yield all but the first n elements without copying. The vector may contain less than n elements in which case an empty vector is returned.

---

splitAt :: Int -> Vector a -> (Vector a, Vector a) Source

O(1) Yield the first n elements paired with the remainder without copying.

Note that `splitAt n v` is equivalent to `(take n v, drop n v)` but slightly more efficient.

---

unsafeSlice Source

```
:: Int          i starting index
-> Int          n length
-> Vector a
-> Vector a
```

O(1) Yield a slice of the vector without copying. The vector must contain at least `i+n` elements but this is not checked.

---

unsafeInit :: Vector a -> Vector a Source

O(1) Yield all but the last element without copying. The vector may not be empty but this is not checked.

---

unsafeTail :: Vector a -> Vector a Source

O(1) Yield all but the first element without copying. The vector may not be empty but this is not checked.

---

unsafeTake :: Int -> Vector a -> Vector a Source

O(1) Yield the first n elements without copying. The vector must contain at least n elements but this is not checked.

---

unsafeDrop :: Int -> Vector a -> Vector a Source

O(1) Yield all but the first n elements without copying. The vector must contain at least n elements but this is not checked.

Construction

Initialisation

---

empty :: Vector a Source

O(1) Empty vector

---

singleton :: a -> Vector a Source

O(1) Vector with exactly one element

---

replicate :: Int -> a -> Vector a Source

O(n) Vector of the given length with the same value in each position

---

generate :: Int -> (Int -> a) -> Vector a Source

O(n) Construct a vector of the given length by applying the function to each index

iterateN :: Int -> (a -> a) -> a -> Vector a Source

O(n) Apply function n times to value. Zeroth element is original value.

Monadic initialisation

replicateM :: Monad m => Int -> m a -> m (Vector a) Source

O(n) Execute the monadic action the given number of times and store the results in a vector.

generateM :: Monad m => Int -> (Int -> m a) -> m (Vector a) Source

O(n) Construct a vector of the given length by applying the monadic action to each index

create :: (forall s. ST s (MVector s a)) -> Vector a Source

Execute the monadic action and freeze the resulting vector.

```
create (do { v <- new 2; write v 0 'a'; write v 1 'b'; return v }) = <a,b>
```

Unfolding

unfoldr :: (b -> Maybe (a, b)) -> b -> Vector a Source

O(n) Construct a vector by repeatedly applying the generator function to a seed. The generator function yields `Just` the next element and the new seed or `Nothing` if there are no more elements.

```
unfoldr (\n -> if n == 0 then Nothing else Just (n,n-1)) 10
 = <10,9,8,7,6,5,4,3,2,1>
```

unfoldrN :: Int -> (b -> Maybe (a, b)) -> b -> Vector a Source

O(n) Construct a vector with at most n by repeatedly applying the generator function to the a seed. The generator function yields `Just` the next element and the new seed or `Nothing` if there are no more elements.

```
unfoldrN 3 (\n -> Just (n,n-1)) 10 = <10,9,8>
```

constructN :: Int -> (Vector a -> a) -> Vector a Source

O(n) Construct a vector with n elements by repeatedly applying the generator function to the already constructed part of the vector.

```
constructN 3 f = let a = f <> ; b = f <a> ; c = f <a,b> in f <a,b,c>
```

constructrN :: Int -> (Vector a -> a) -> Vector a Source

O(n) Construct a vector with n elements from right to left by repeatedly applying the generator function to the already constructed part of the vector.

```
constructrN 3 f = let a = f <> ; b = f<a> ; c = f <b,a> in f <c,b,a>
```

Enumeration

enumFromN :: Num a => a -> Int -> Vector a | Source

O(n) Yield a vector of the given length containing the values x, x+1 etc. This operation is usually more efficient than enumFromTo.

```
enumFromN 5 3 = <5,6,7>
```

enumFromStepN :: Num a => a -> a -> Int -> Vector a | Source

O(n) Yield a vector of the given length containing the values x, x+y, x+y+y etc. This operations is usually more efficient than enumFromThenTo.

```
enumFromStepN 1 0.1 5 = <1,1.1,1.2,1.3,1.4>
```

enumFromTo :: Enum a => a -> a -> Vector a | Source

O(n) Enumerate values from x to y.

WARNING: This operation can be very inefficient. If at all possible, use enumFromN instead.

enumFromThenTo :: Enum a => a -> a -> a -> Vector a | Source

O(n) Enumerate values from x to y with a specific step z.

WARNING: This operation can be very inefficient. If at all possible, use enumFromStepN instead.

Concatenation

cons :: a -> Vector a -> Vector a | Source

O(n) Prepend an element

snoc :: Vector a -> a -> Vector a | Source

O(n) Append an element

(++) :: Vector a -> Vector a -> Vector a | infixr 5 | Source

O(m+n) Concatenate two vectors

concat :: [Vector a] -> Vector a | Source

O(n) Concatenate all vectors in the list

Restricting memory usage

force :: Vector a -> Vector a | Source

O(n) Yield the argument but force it not to retain any extra memory, possibly by copying it.

This is especially useful when dealing with slices. For example:

```
force (slice 0 2 <huge vector>)
```

Here, the slice retains a reference to the huge vector. Forcing it creates a copy of just the elements that belong to the slice and allows the huge vector to be garbage collected.

Modifying vectors

Bulk updates

(//) Source

```
:: Vector a       initial vector (of length m)
-> [(Int, a)]     list of index/value pairs (of length n)
-> Vector a
```

O(m+n) For each pair `(i,a)` from the list, replace the vector element at position `i` by `a`.

```
<5,9,2,7> // [(2,1),(0,3),(2,8)] = <3,9,8,7>
```

update Source

```
:: Vector a           initial vector (of length m)
-> Vector (Int, a)    vector of index/value pairs (of length n)
-> Vector a
```

O(m+n) For each pair `(i,a)` from the vector of index/value pairs, replace the vector element at position `i` by `a`.

```
update <5,9,2,7> <(2,1),(0,3),(2,8)> = <3,9,8,7>
```

update_ Source

```
:: Vector a       initial vector (of length m)
-> Vector Int     index vector (of length n1)
-> Vector a       value vector (of length n2)
-> Vector a
```

O(m+min(n1,n2)) For each index `i` from the index vector and the corresponding value `a` from the value vector, replace the element of the initial vector at position `i` by `a`.

```
update_ <5,9,2,7>  <2,0,2> <1,3,8> = <3,9,8,7>
```

The function `update` provides the same functionality and is usually more convenient.

```
update_ xs is ys = update xs (zip is ys)
```

unsafeUpd `:: Vector a -> [(Int, a)] -> Vector a` Source

Same as (`//`) but without bounds checking.

unsafeUpdate `:: Vector a -> Vector (Int, a) -> Vector a` Source

Same as `update` but without bounds checking.

unsafeUpdate_ `:: Vector a -> Vector Int -> Vector a -> Vector a` Source

Same as `update_` but without bounds checking.

Accumulations

accum Source

```
:: (a -> b -> a)      accumulating function f
-> Vector a           initial vector (of length m)
-> [(Int, b)]         list of index/value pairs (of length n)
-> Vector a
```

O(m+n) For each pair `(i,b)` from the list, replace the vector element `a` at position `i` by `f a b`.

```
accum (+) <5,9,2> [(2,4),(1,6),(0,3),(1,7)] = <5+3, 9+6+7, 2+4>
```

accumulate Source

```
:: (a -> b -> a)      accumulating function f
-> Vector a           initial vector (of length m)
-> Vector (Int, b)    vector of index/value pairs (of length n)
-> Vector a
```

O(m+n) For each pair `(i,b)` from the vector of pairs, replace the vector element `a` at position `i` by `f a b`.

```
accumulate (+) <5,9,2> <(2,4),(1,6),(0,3),(1,7)> = <5+3, 9+6+7, 2+4>
```

accumulate_ Source

```
:: (a -> b -> a)      accumulating function f
-> Vector a           initial vector (of length m)
-> Vector Int         index vector (of length n1)
-> Vector b           value vector (of length n2)
-> Vector a
```

O(m+min(n1,n2)) For each index `i` from the index vector and the corresponding value `b` from the the value vector, replace the element of the initial vector at position `i` by `f a b`.

```
accumulate_ (+) <5,9,2> <2,1,0,1> <4,6,3,7> = <5+3, 9+6+7, 2+4>
```

The function `accumulate` provides the same functionality and is usually more convenient.

```
accumulate_ f as is bs = accumulate f as (zip is bs)
```

unsafeAccum `:: (a -> b -> a) -> Vector a -> [(Int, b)] -> Vector a` Source

Same as `accum` but without bounds checking.

unsafeAccumulate `:: (a -> b -> a) -> Vector a -> Vector (Int, b) -> Vector a` Source

Same as `accumulate` but without bounds checking.

unsafeAccumulate_ `:: (a -> b -> a) -> Vector a -> Vector Int -> Vector b -> Vector a` Source

Same as `accumulate_` but without bounds checking.

Permutations

reverse `:: Vector a -> Vector a` Source

O(n) Reverse a vector

backpermute :: Vector a -> Vector Int -> Vector a | Source

O(n) Yield the vector obtained by replacing each element `i` of the index vector by `xs!i`. This is equivalent to `map (xs!) is` but is often much more efficient.

```
backpermute <a,b,c,d> <0,3,2,3,1,0> = <a,d,c,d,b,a>
```

unsafeBackpermute :: Vector a -> Vector Int -> Vector a | Source

Same as `backpermute` but without bounds checking.

Safe destructive updates

modify :: (forall s. MVector s a -> ST s ()) -> Vector a -> Vector a | Source

Apply a destructive operation to a vector. The operation will be performed in place if it is safe to do so and will modify a copy of the vector otherwise.

```
modify (\v -> write v 0 'x') (replicate 3 'a') = <'x','a','a'>
```

Elementwise operations

Indexing

indexed :: Vector a -> Vector (Int, a) | Source

O(n) Pair each element in a vector with its index

Mapping

map :: (a -> b) -> Vector a -> Vector b | Source

O(n) Map a function over a vector

imap :: (Int -> a -> b) -> Vector a -> Vector b | Source

O(n) Apply a function to every element of a vector and its index

concatMap :: (a -> Vector b) -> Vector a -> Vector b | Source

Map a function over a vector and concatenate the results.

Monadic mapping

mapM :: Monad m => (a -> m b) -> Vector a -> m (Vector b) | Source

O(n) Apply the monadic action to all elements of the vector, yielding a vector of results

imapM :: Monad m => (Int -> a -> m b) -> Vector a -> m (Vector b) | Source

O(n) Apply the monadic action to every element of a vector and its index, yielding a vector of results

mapM_ :: Monad m => (a -> m b) -> Vector a -> m () | Source

O(n) Apply the monadic action to all elements of a vector and ignore the results

imapM_ :: Monad m => (Int -> a -> m b) -> Vector a -> m () Source

O(n) Apply the monadic action to every element of a vector and its index, ignoring the results

forM :: Monad m => Vector a -> (a -> m b) -> m (Vector b) Source

O(n) Apply the monadic action to all elements of the vector, yielding a vector of results. Equvalent to `flip mapM`.

forM_ :: Monad m => Vector a -> (a -> m b) -> m () Source

O(n) Apply the monadic action to all elements of a vector and ignore the results. Equivalent to `flip mapM_`.

Zipping

zipWith :: (a -> b -> c) -> Vector a -> Vector b -> Vector c Source

O(min(m,n)) Zip two vectors with the given function.

zipWith3 :: (a -> b -> c -> d) -> Vector a -> Vector b -> Vector c -> Vector d Source

Zip three vectors with the given function.

zipWith4 :: (a -> b -> c -> d -> e) -> Vector a -> Vector b -> Vector c -> Vector d -> Vector e Source

zipWith5 :: (a -> b -> c -> d -> e -> f) -> Vector a -> Vector b -> Vector c -> Vector d -> Vector e -> Vector f Source

zipWith6 :: (a -> b -> c -> d -> e -> f -> g) -> Vector a -> Vector b -> Vector c -> Vector d -> Vector e -> Vector f -> Vector g Source

izipWith :: (Int -> a -> b -> c) -> Vector a -> Vector b -> Vector c Source

O(min(m,n)) Zip two vectors with a function that also takes the elements' indices.

izipWith3 :: (Int -> a -> b -> c -> d) -> Vector a -> Vector b -> Vector c -> Vector d Source

Zip three vectors and their indices with the given function.

izipWith4 :: (Int -> a -> b -> c -> d -> e) -> Vector a -> Vector b -> Vector c -> Vector d -> Vector e Source

izipWith5 :: (Int -> a -> b -> c -> d -> e -> f) -> Vector a -> Vector b -> Vector c -> Vector d -> Vector e -> Vector f Source

izipWith6 :: (Int -> a -> b -> c -> d -> e -> f -> g) -> Vector a -> Vector b -> Vector c -> Vector d -> Vector e -> Vector f -> Vector g Source

```
zip :: Vector a -> Vector b -> Vector (a, b)
```
Source

Elementwise pairing of array elements.

```
zip3 :: Vector a -> Vector b -> Vector c -> Vector (a, b, c)
```
Source

zip together three vectors into a vector of triples

```
zip4 :: Vector a -> Vector b -> Vector c -> Vector d -> Vector (a, b, c, d)
```
Source

```
zip5 :: Vector a -> Vector b -> Vector c -> Vector d -> Vector e -> Vector (a, b, c, d, e)
```
Source

```
zip6 :: Vector a -> Vector b -> Vector c -> Vector d -> Vector e -> Vector f -> Vector (a, b, c, d, e, f)
```
Source

Monadic zipping

```
zipWithM :: Monad m => (a -> b -> m c) -> Vector a -> Vector b -> m (Vector c)
```
Source

O(min(m,n)) Zip the two vectors with the monadic action and yield a vector of results

```
izipWithM :: Monad m => (Int -> a -> b -> m c) -> Vector a -> Vector b -> m (Vector c)
```
Source

O(min(m,n)) Zip the two vectors with a monadic action that also takes the element index and yield a vector of results

```
zipWithM_ :: Monad m => (a -> b -> m c) -> Vector a -> Vector b -> m ()
```
Source

O(min(m,n)) Zip the two vectors with the monadic action and ignore the results

```
izipWithM_ :: Monad m => (Int -> a -> b -> m c) -> Vector a -> Vector b -> m ()
```
Source

O(min(m,n)) Zip the two vectors with a monadic action that also takes the element index and ignore the results

Unzipping

```
unzip :: Vector (a, b) -> (Vector a, Vector b)
```
Source

O(min(m,n)) Unzip a vector of pairs.

```
unzip3 :: Vector (a, b, c) -> (Vector a, Vector b, Vector c)
```
Source

```
unzip4 :: Vector (a, b, c, d) -> (Vector a, Vector b, Vector c, Vector d)
```
Source

```
unzip5 :: Vector (a, b, c, d, e) -> (Vector a, Vector b, Vector c, Vector d, Vector e)
```
Source

```
unzip6 :: Vector (a, b, c, d, e, f) -> (Vector a, Vector b, Vector c, Vector d,
Vector e, Vector f)
```
Source

Working with predicates

Filtering

```
filter :: (a -> Bool) -> Vector a -> Vector a
```
Source

O(n) Drop elements that do not satisfy the predicate

```
ifilter :: (Int -> a -> Bool) -> Vector a -> Vector a
```
Source

O(n) Drop elements that do not satisfy the predicate which is applied to values and their indices

```
filterM :: Monad m => (a -> m Bool) -> Vector a -> m (Vector a)
```
Source

O(n) Drop elements that do not satisfy the monadic predicate

```
takeWhile :: (a -> Bool) -> Vector a -> Vector a
```
Source

O(n) Yield the longest prefix of elements satisfying the predicate without copying.

```
dropWhile :: (a -> Bool) -> Vector a -> Vector a
```
Source

O(n) Drop the longest prefix of elements that satisfy the predicate without copying.

Partitioning

```
partition :: (a -> Bool) -> Vector a -> (Vector a, Vector a)
```
Source

O(n) Split the vector in two parts, the first one containing those elements that satisfy the predicate and the second one those that don't. The relative order of the elements is preserved at the cost of a sometimes reduced performance compared to `unstablePartition`.

```
unstablePartition :: (a -> Bool) -> Vector a -> (Vector a, Vector a)
```
Source

O(n) Split the vector in two parts, the first one containing those elements that satisfy the predicate and the second one those that don't. The order of the elements is not preserved but the operation is often faster than `partition`.

```
span :: (a -> Bool) -> Vector a -> (Vector a, Vector a)
```
Source

O(n) Split the vector into the longest prefix of elements that satisfy the predicate and the rest without copying.

```
break :: (a -> Bool) -> Vector a -> (Vector a, Vector a)
```
Source

O(n) Split the vector into the longest prefix of elements that do not satisfy the predicate and the rest without copying.

Searching

```
elem :: Eq a => a -> Vector a -> Bool    infix 4
```
Source

O(n) Check if the vector contains an element

---

notElem :: Eq a => a -> Vector a -> Bool | infix 4 | Source

O(n) Check if the vector does not contain an element (inverse of `elem`)

---

find :: (a -> Bool) -> Vector a -> Maybe a | Source

O(n) Yield `Just` the first element matching the predicate or `Nothing` if no such element exists.

---

findIndex :: (a -> Bool) -> Vector a -> Maybe Int | Source

O(n) Yield `Just` the index of the first element matching the predicate or `Nothing` if no such element exists.

---

findIndices :: (a -> Bool) -> Vector a -> Vector Int | Source

O(n) Yield the indices of elements satisfying the predicate in ascending order.

---

elemIndex :: Eq a => a -> Vector a -> Maybe Int | Source

O(n) Yield `Just` the index of the first occurence of the given element or `Nothing` if the vector does not contain the element. This is a specialised version of `findIndex`.

---

elemIndices :: Eq a => a -> Vector a -> Vector Int | Source

O(n) Yield the indices of all occurences of the given element in ascending order. This is a specialised version of `findIndices`.

Folding

---

foldl :: (a -> b -> a) -> a -> Vector b -> a | Source

O(n) Left fold

---

foldl1 :: (a -> a -> a) -> Vector a -> a | Source

O(n) Left fold on non-empty vectors

---

foldl' :: (a -> b -> a) -> a -> Vector b -> a | Source

O(n) Left fold with strict accumulator

---

foldl1' :: (a -> a -> a) -> Vector a -> a | Source

O(n) Left fold on non-empty vectors with strict accumulator

---

foldr :: (a -> b -> b) -> b -> Vector a -> b | Source

O(n) Right fold

---

foldr1 :: (a -> a -> a) -> Vector a -> a | Source

O(n) Right fold on non-empty vectors

foldr' :: (a -> b -> b) -> b -> Vector a -> b Source

O(n) Right fold with a strict accumulator

foldr1' :: (a -> a -> a) -> Vector a -> a Source

O(n) Right fold on non-empty vectors with strict accumulator

ifoldl :: (a -> Int -> b -> a) -> a -> Vector b -> a Source

O(n) Left fold (function applied to each element and its index)

ifoldl' :: (a -> Int -> b -> a) -> a -> Vector b -> a Source

O(n) Left fold with strict accumulator (function applied to each element and its index)

ifoldr :: (Int -> a -> b -> b) -> b -> Vector a -> b Source

O(n) Right fold (function applied to each element and its index)

ifoldr' :: (Int -> a -> b -> b) -> b -> Vector a -> b Source

O(n) Right fold with strict accumulator (function applied to each element and its index)

Specialised folds

all :: (a -> Bool) -> Vector a -> Bool Source

O(n) Check if all elements satisfy the predicate.

any :: (a -> Bool) -> Vector a -> Bool Source

O(n) Check if any element satisfies the predicate.

and :: Vector Bool -> Bool Source

O(n) Check if all elements are True

or :: Vector Bool -> Bool Source

O(n) Check if any element is True

sum :: Num a => Vector a -> a Source

O(n) Compute the sum of the elements

product :: Num a => Vector a -> a Source

O(n) Compute the produce of the elements

maximum :: Ord a => Vector a -> a Source

O(n) Yield the maximum element of the vector. The vector may not be empty.

maximumBy :: (a -> a -> Ordering) -> Vector a -> a Source

O(n) Yield the maximum element of the vector according to the given comparison function. The vector may not be empty.

minimum :: Ord a => Vector a -> a Source

O(n) Yield the minimum element of the vector. The vector may not be empty.

minimumBy :: (a -> a -> Ordering) -> Vector a -> a Source

O(n) Yield the minimum element of the vector according to the given comparison function. The vector may not be empty.

minIndex :: Ord a => Vector a -> Int Source

O(n) Yield the index of the minimum element of the vector. The vector may not be empty.

minIndexBy :: (a -> a -> Ordering) -> Vector a -> Int Source

O(n) Yield the index of the minimum element of the vector according to the given comparison function. The vector may not be empty.

maxIndex :: Ord a => Vector a -> Int Source

O(n) Yield the index of the maximum element of the vector. The vector may not be empty.

maxIndexBy :: (a -> a -> Ordering) -> Vector a -> Int Source

O(n) Yield the index of the maximum element of the vector according to the given comparison function. The vector may not be empty.

Monadic folds

foldM :: Monad m => (a -> b -> m a) -> a -> Vector b -> m a Source

O(n) Monadic fold

ifoldM :: Monad m => (a -> Int -> b -> m a) -> a -> Vector b -> m a Source

O(n) Monadic fold (action applied to each element and its index)

foldM' :: Monad m => (a -> b -> m a) -> a -> Vector b -> m a Source

O(n) Monadic fold with strict accumulator

ifoldM' :: Monad m => (a -> Int -> b -> m a) -> a -> Vector b -> m a Source

O(n) Monadic fold with strict accumulator (action applied to each element and its index)

fold1M :: Monad m => (a -> a -> m a) -> Vector a -> m a Source

O(n) Monadic fold over non-empty vectors

```
fold1M' :: Monad m => (a -> a -> m a) -> Vector a -> m a
```
Source

O(n) Monadic fold over non-empty vectors with strict accumulator

```
foldM_ :: Monad m => (a -> b -> m a) -> a -> Vector b -> m ()
```
Source

O(n) Monadic fold that discards the result

```
ifoldM_ :: Monad m => (a -> Int -> b -> m a) -> a -> Vector b -> m ()
```
Source

O(n) Monadic fold that discards the result (action applied to each element and its index)

```
foldM'_ :: Monad m => (a -> b -> m a) -> a -> Vector b -> m ()
```
Source

O(n) Monadic fold with strict accumulator that discards the result

```
ifoldM'_ :: Monad m => (a -> Int -> b -> m a) -> a -> Vector b -> m ()
```
Source

O(n) Monadic fold with strict accumulator that discards the result (action applied to each element and its index)

```
fold1M_ :: Monad m => (a -> a -> m a) -> Vector a -> m ()
```
Source

O(n) Monadic fold over non-empty vectors that discards the result

```
fold1M'_ :: Monad m => (a -> a -> m a) -> Vector a -> m ()
```
Source

O(n) Monadic fold over non-empty vectors with strict accumulator that discards the result

Monadic sequencing

```
sequence :: Monad m => Vector (m a) -> m (Vector a)
```
Source

Evaluate each action and collect the results

```
sequence_ :: Monad m => Vector (m a) -> m ()
```
Source

Evaluate each action and discard the results

Prefix sums (scans)

```
prescanl :: (a -> b -> a) -> a -> Vector b -> Vector a
```
Source

O(n) Prescan

```
prescanl f z = init . scanl f z
```

Example: `prescanl (+) 0 <1,2,3,4> = <0,1,3,6>`

```
prescanl' :: (a -> b -> a) -> a -> Vector b -> Vector a
```
Source

O(n) Prescan with strict accumulator

```
postscanl :: (a -> b -> a) -> a -> Vector b -> Vector a
```
Source

O(n) Scan

```
postscanl f z = tail . scanl f z
```

Example: `postscanl (+) 0 <1,2,3,4> = <1,3,6,10>`

---

postscanl' :: (a -> b -> a) -> a -> Vector b -> Vector a Source

O(n) Scan with strict accumulator

---

scanl :: (a -> b -> a) -> a -> Vector b -> Vector a Source

O(n) Haskell-style scan

```
scanl f z <x1,...,xn> = <y1,...,y(n+1)>
  where y1 = z
        yi = f y(i-1) x(i-1)
```

Example: `scanl (+) 0 <1,2,3,4> = <0,1,3,6,10>`

---

scanl' :: (a -> b -> a) -> a -> Vector b -> Vector a Source

O(n) Haskell-style scan with strict accumulator

---

scanl1 :: (a -> a -> a) -> Vector a -> Vector a Source

O(n) Scan over a non-empty vector

```
scanl f <x1,...,xn> = <y1,...,yn>
  where y1 = x1
        yi = f y(i-1) xi
```

---

scanl1' :: (a -> a -> a) -> Vector a -> Vector a Source

O(n) Scan over a non-empty vector with a strict accumulator

---

prescanr :: (a -> b -> b) -> b -> Vector a -> Vector b Source

O(n) Right-to-left prescan

```
prescanr f z = reverse . prescanl (flip f) z . reverse
```

---

prescanr' :: (a -> b -> b) -> b -> Vector a -> Vector b Source

O(n) Right-to-left prescan with strict accumulator

---

postscanr :: (a -> b -> b) -> b -> Vector a -> Vector b Source

O(n) Right-to-left scan

---

postscanr' :: (a -> b -> b) -> b -> Vector a -> Vector b Source

O(n) Right-to-left scan with strict accumulator

---

scanr :: (a -> b -> b) -> b -> Vector a -> Vector b Source

O(n) Right-to-left Haskell-style scan

```
scanr' :: (a -> b -> b) -> b -> Vector a -> Vector b
```
Source

O(n) Right-to-left Haskell-style scan with strict accumulator

---

```
scanr1 :: (a -> a -> a) -> Vector a -> Vector a
```
Source

O(n) Right-to-left scan over a non-empty vector

```
scanr1' :: (a -> a -> a) -> Vector a -> Vector a
```
Source

O(n) Right-to-left scan over a non-empty vector with a strict accumulator

Conversions

Lists

```
toList :: Vector a -> [a]
```
Source

O(n) Convert a vector to a list

```
fromList :: [a] -> Vector a
```
Source

O(n) Convert a list to a vector

```
fromListN :: Int -> [a] -> Vector a
```
Source

O(n) Convert the first n elements of a list to a vector

```
fromListN n xs = fromList (take n xs)
```

Other vector types

```
convert :: (Vector v a, Vector w a) => v a -> w a
```
Source

O(n) Convert different vector types

Mutable vectors

```
freeze :: PrimMonad m => MVector (PrimState m) a -> m (Vector a)
```
Source

O(n) Yield an immutable copy of the mutable vector.

```
thaw :: PrimMonad m => Vector a -> m (MVector (PrimState m) a)
```
Source

O(n) Yield a mutable copy of the immutable vector.

```
copy :: PrimMonad m => MVector (PrimState m) a -> Vector a -> m ()
```
Source

O(n) Copy an immutable vector into a mutable one. The two vectors must have the same length.

```
unsafeFreeze :: PrimMonad m => MVector (PrimState m) a -> m (Vector a)
```
Source

O(1) Unsafe convert a mutable vector to an immutable one without copying. The mutable vector may not be used after this operation.

unsafeThaw :: PrimMonad m => Vector a -> m (MVector (PrimState m) a) Source

O(1) Unsafely convert an immutable vector to a mutable one without copying. The immutable vector may not be used after this operation.

unsafeCopy :: PrimMonad m => MVector (PrimState m) a -> Vector a -> m () Source

O(n) Copy an immutable vector into a mutable one. The two vectors must have the same length. This is not checked.

Produced by Haddock version 2.15.0.2