Contents

1 Basic functions

$(++) :: [a] \to [a] \to [a]$
Append two lists, i.e.,

$$[x1, ..., xm] + +[y1, ..., yn] == [x1, ..., xm, y1, ..., yn]$$
$$[x1, ..., xm] + +[y1, ...] == [x1, ..., xm, y1, ...]$$

If the first list is not finite, the result is the first list.

$head :: [a] \to a$
Extract the first element of a list, which must be non-empty.

$last :: [a] \to a$
Extract the last element of a list, which must be finite and non-empty.

$tail :: [a] \to [a]$
Extract the elements after the head of a list, which must be non-empty.

$init :: [a] \to [a]$
Return all the elements of a list except the last one. The list must be non-empty.

$uncons :: [a] \to Maybe(a, [a])$
Decompose a list into its head and tail. If the list is empty, returns Nothing. If the list is non-empty, returns Just (x, xs), where x is the head of the list and xs its tail. Since: 4.8.0.0

$null :: Foldable\ t \Rightarrow t\ a \to Bool$
Test whether the structure is empty. The default implementation is optimized for structures that are similar to cons-lists, because there is no general way to do better.

$length :: Foldable\ t \Rightarrow t\ a \to Int$
Returns the size/length of a finite structure as an Int. The default implementation is optimized for structures that are similar to cons-lists, because there is no general way to do better.

2 List transformations

map :: $(a \to b) \to [a] \to [b]$
map f xs is the list obtained by applying f to each element of xs, i.e.,

$$map\ f\ [x1, x2, ..., xn] == [f\ x1, f\ x2, ..., f\ xn]$$
$$map\ f\ [x1, x2, ...] == [f\ x1, f\ x2, ...]$$

reverse :: $[a] \to [a]$
reverse xs returns the elements of xs in reverse order. xs must be finite.

intersperse :: $a \to [a] \to [a]$
The intersperse function takes an element and a list and 'intersperses' that element between the elements of the list. For example,

intersperse ',' "abcde" == "a,b,c,d,e"

intercalate :: $[a] \to [[a]] \to [a]$
intercalate xs xss is equivalent to (concat (intersperse xs xss)). It inserts the list xs in between the lists in xss and concatenates the result.

transpose :: $[[a]] \to [[a]]$
The transpose function transposes the rows and columns of its argument. For example,

transpose [[1,2,3],[4,5,6]] == [[1,4],[2,5],[3,6]]

If some of the rows are shorter than the following rows, their elements are skipped:

transpose [[10,11],[20],[],[30,31,32]] == [[10,20,30],[11,31],[32]]

subsequences :: $[a] \to [[a]]$
The subsequences function returns the list of all subsequences of the argument.

subsequences "abc" == ["", "a","b","ab","c","ac","bc","abc"]

permutations :: $[a] \to [[a]]$
The permutations function returns the list of all permutations of the argument.

permutations "abc" == ["abc","bac","cba","bca","cab","acb"]

3 Reducing lists (folds)

$foldl :: Foldable\ t \Rightarrow (b \to a \to b) \to b \to t\ a \to b$
Left-associative fold of a structure.
In the case of lists, foldl, when applied to a binary operator, a starting value (typically the left-identity of the operator), and a list, reduces the list using the binary operator, from left to right:

foldl f z [x1, x2, ..., xn] == (...((z `f` x1) `f` x2) `f`...) `f` xn

Note that to produce the outermost application of the operator the entire input list must be traversed. This means that foldl' will diverge if given an infinite list.
Also note that if you want an efficient left-fold, you probably want to use foldl' instead of foldl. The reason for this is that latter does not force the "inner" results (e.g. z f x1 in the above example) before applying them to the operator (e.g. to (f x2)). This results in a thunk chain O(n) elements long, which then must be evaluated from the outside-in.
For a general Foldable structure this should be semantically identical to,

foldl f z = foldl f z . toList

Bsp.:

```
Input: foldl (+) 0 [1,2,3,4]
Output: (((0+1)+2)+3)+4
```

Erklärung:
Der Startwert 0 wird als linker Operand in der innersten Klammer (ganz links) verwendet, der rechte Operand ist das erste Element der Liste (1). Das Ergebnis der innersten Klammer wird als linker Operand für die nächste Operation verwendet, der rechte Operand ist das zweite Element der Liste (2).

$foldl' :: Foldable\ t \Rightarrow (b \to a \to b) \to b \to t\ a \to b$
Left-associative fold of a structure but with strict application of the operator.
This ensures that each step of the fold is forced to weak head normal form before being applied, avoiding the collection of thunks that would otherwise occur. This is often what you want to strictly reduce a finite list to a single, monolithic result (e.g. length).
For a general Foldable structure this should be semantically identical to,

foldl f z = foldl' f z . toList

$foldl1 :: Foldable\ t \Rightarrow (a \to a \to a) \to t\ a \to a$
A variant of foldl that has no base case, and thus may only be applied to non-empty structures.

foldl1 f = foldl1 f . toList

$foldl1' :: (a \to a \to a) \to [a] \to a$
A strict version of foldl1

$foldr :: Foldable\ t \Rightarrow (a \to b \to b) \to b \to t\ a \to b$
Right-associative fold of a structure.
In the case of lists, foldr, when applied to a binary operator, a starting value (typically the right-identity of the operator), and a list, reduces the list using the binary operator, from right to left:

foldr f z [x1, x2, ..., xn] == x1 `f` (x2 `f` ... (xn `f` z)...)

Note that, since the head of the resulting expression is produced by an application of the operator to the first element of the list, foldr can produce a terminating expression from an infinite list. For a general Foldable structure this should be semantically identical to,

foldr f z = foldr f z . toList

Bsp.:

Input: **foldr** (+) 0 [1,2,3,4]
Output: (1+(2+(3+(4+0))))

Der Startwert 0 wird als rechter Operand in der innersten Klammer (ganz rechts) verwendet, der linke Operand ist das letzte Element der Liste (4). Das Ergebnis der innersten Klammer wird als rechter Operand für die nächste Operation verwendet, der linke Operand ist das zweit-letzte Element der Liste (3).

$foldr1 :: Foldable\ t \Rightarrow (a \to a \to a) \to t\ a \to a$
A variant of foldr that has no base case, and thus may only be applied to non-empty structures.

foldr1 f = foldr1 f . toList

3.1 Special folds

$concat :: Foldable\ t \Rightarrow t\ [a] \rightarrow [a]$
The concatenation of all the elements of a container of lists.
Bsp.:

```
Input: concat([1,2], [3,4])
Output: [1,2,3,4]
```

$concatMap :: Foldable\ t \Rightarrow (a \rightarrow [b]) \rightarrow t\ a \rightarrow [b]$
Map a function over all the elements of a container and concatenate the resulting lists.

$and :: Foldable\ t \Rightarrow t\ Bool \rightarrow Bool$
and returns the conjunction of a container of Bools. For the result to be True, the container must be finite; False, however, results from a False value finitely far from the left end.

$or :: Foldable\ t \Rightarrow t\ Bool \rightarrow Bool$
or returns the disjunction of a container of Bools. For the result to be False, the container must be finite; True, however, results from a True value finitely far from the left end.

$any :: Foldable\ t \Rightarrow (a \rightarrow Bool) \rightarrow t\ a \rightarrow Bool$
Determines whether any element of the structure satisfies the predicate.

$all :: Foldable\ t \Rightarrow (a \rightarrow Bool) \rightarrow t\ a \rightarrow Bool$
Determines whether all elements of the structure satisfy the predicate.

$sum :: (Foldable\ t, Num\ a) \Rightarrow t\ a \rightarrow a$
The sum function computes the sum of the numbers of a structure.

$product :: (Foldable\ t, Num\ a) \Rightarrow t\ a \rightarrow a$
The product function computes the product of the numbers of a structure.

$maximum :: forall\ a.\ (Foldable\ t,\ Ord\ a) \Rightarrow t\ a \rightarrow a$
The largest element of a non-empty structure.

$minimum :: forall\ a.\ (Foldable\ t, Ord\ a) \Rightarrow t\ a \rightarrow a$
The least element of a non-empty structure.

4 Building lists

4.1 Scans

scanl :: $(b \to a \to b) \to b \to [a] \to [b]$
scanl is similar to foldl, but returns a list of successive reduced values from the left:

scanl f z [x1, x2, ...] == [z, z `f` x1, (z `f` x1) `f` x2, ...]

Note that

last (scanl f z xs) == foldl f z xs.

scanl' :: $(b \to a \to b) \to b \to [a] \to [b]$
A strictly accumulating version of scanl

scanl1 :: $(a \to a \to a) \to [a] \to [a]$
scanl1 is a variant of scanl that has no starting value argument:

scanl1 f [x1, x2, ...] == [x1, x1 `f` x2, ...]

scanr :: $(a \to b \to b) \to b \to [a] \to [b]$
scanr is the right-to-left dual of scanl. Note that

head (scanr f z xs) == foldr f z xs.

scanr1 :: $(a \to a \to a) \to [a] \to [a]$
scanr1 is a variant of scanr that has no starting value argument.

4.2 Accumulating maps

mapAccumL :: $Traversable\ t \Rightarrow (a \to b \to (a, c)) \to a \to tb \to (a, t\ c)$
The mapAccumL function behaves like a combination of fmap and foldl; it applies a function to each element of a structure, passing an accumulating parameter from left to right, and returning a final value of this accumulator together with the new structure.

mapAccumR :: $Traversable\ t \Rightarrow (a \to b \to (a, c)) \to a \to tb \to (a, t\ c)$
The mapAccumR function behaves like a combination of fmap and foldr; it applies a function to each element of a structure, passing an accumulating parameter from right to left, and returning a final value of this accumulator together with the new structure.

4.3 Infinite lists

iterate :: $(a \to a) \to a \to [a]$
iterate f x returns an infinite list of repeated applications of f to x:

iterate f x == [x, f x, f (f x), ...]

repeat :: $a \to [a]$
repeat x is an infinite list, with x the value of every element.

replicate :: $Int \to a \to [a]$
replicate n x is a list of length n with x the value of every element. It is an instance of the more general genericReplicate, in which n may be of any integral type.

cycle :: $[a] \to [a]$
cycle ties a finite list into a circular one, or equivalently, the infinite repetition of the original list. It is the identity on infinite lists.

4.4 Unfolding

$unfoldr :: (b \to Maybe\ (a, b)) \to b \to [a]$
The unfoldr function is a 'dual' to foldr: while foldr reduces a list to a summary value, unfoldr builds a list from a seed value. The function takes the element and returns Nothing if it is done producing the list or returns Just (a,b), in which case, a is a prepended to the list and b is used as the next element in a recursive call. For example,

iterate f == unfoldr (\x ->Just (x, f x))

In some cases, unfoldr can undo a foldr operation:

unfoldr f' (foldr f z xs) == xs

if the following holds:

$$f'\ (f\ xy) = Just\ (x, y)$$
$$f'\ z = Nothing$$

A simple use of unfoldr:

$$unfoldr\ (\backslash b \to if\ b == 0\ then\ Nothing\ else\ Just\ (b, b-1))\ 10$$
$$[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]$$

5 Sublists

5.1 Extracting sublists

take :: $Int \to [a] \to [a]$
take n, applied to a list xs, returns the prefix of xs of length n, or xs itself if n >length xs:

take 5 "Hello World!" == "Hello"
take 3 [1,2,3,4,5] == [1,2,3]
take 3 [1,2] == [1,2]
take 3 [] == []
take (-1) [1,2] == []
take 0 [1,2] == []

It is an instance of the more general genericTake, in which n may be of any integral type.

drop :: $Int \to [a] \to [a]$
drop n xs returns the suffix of xs after the first n elements, or [] if n >length xs:

drop 6 "Hello World!" == "World!"
drop 3 [1,2,3,4,5] == [4,5]
drop 3 [1,2] == []
drop 3 [] == []
drop (-1) [1,2] == [1,2]
drop 0 [1,2] == [1,2]

It is an instance of the more general genericDrop, in which n may be of any integral type.

splitAt :: $Int \to [a] \to ([a], [a])$
splitAt n xs returns a tuple where first element is xs prefix of length n and second element is the remainder of the list:

splitAt 6 "Hello World!" == ("Hello ", "World!")
splitAt 3 [1,2,3,4,5] == ([1,2,3],[4,5])
splitAt 1 [1,2,3] == ([1],[2,3])
splitAt 3 [1,2,3] == ([1,2,3],[])
splitAt 4 [1,2,3] == ([1,2,3],[])
splitAt 0 [1,2,3] == ([],[1,2,3])
splitAt (-1) [1,2,3] == ([],[1,2,3])

It is equivalent to (take n xs, drop n xs) when n is not _|_ (splitAt _|_ xs = _|_). splitAt is an instance of the more general genericSplitAt, in which n may be of any integral type.

takeWhile :: $(a \to Bool) \to [a] \to [a]$
takeWhile, applied to a predicate p and a list xs, returns the longest prefix (possibly empty) of xs of elements that satisfy p:

$takeWhile\ (<3)\ [1,2,3,4,1,2,3,4] == [1,2]$
$takeWhile\ (<9)\ [1,2,3] == [1,2,3]$
$takeWhile\ (<0)\ [1,2,3] == []$

$dropWhile :: (a \to Bool) \to [a] \to [a]$
dropWhile p xs returns the suffix remaining after takeWhile p xs:

$$dropWhile\ (<3)\ [1,2,3,4,5,1,2,3] == [3,4,5,1,2,3]$$
$$dropWhile\ (<9)\ [1,2,3] == []$$
$$dropWhile\ (<0)\ [1,2,3] == [1,2,3]$$

$dropWhileEnd :: (a \to Bool) \to [a] \to [a]$
The dropWhileEnd function drops the largest suffix of a list in which the given predicate holds for all elements. For example:

dropWhileEnd isSpace “”foo\n”” == “foo”
dropWhileEnd isSpace “foo bar” == “foo bar”
dropWhileEnd isSpace (“foo\n” ++ undefined) == “foo” ++ undefined

Since: 4.5.0.0

$span :: (a \to Bool) \to [a] \to ([a], [a])$
span, applied to a predicate p and a list xs, returns a tuple where first element is longest prefix (possibly empty) of xs of elements that satisfy p and second element is the remainder of the list:

$$span\ (<3)\ [1,2,3,4,1,2,3,4] == ([1,2],[3,4,1,2,3,4])$$
$$span\ (<9)\ [1,2,3] == ([1,2,3],[])$$
$$span\ (<0)\ [1,2,3] == ([],[1,2,3])$$

span p xs is equivalent to (takeWhile p xs, dropWhile p xs)

$break :: (a \to Bool) \to [a] \to ([a], [a])$
break, applied to a predicate p and a list xs, returns a tuple where first element is longest prefix (possibly empty) of xs of elements that do not satisfy p and second element is the remainder of the list:

break (>3) [1,2,3,4,1,2,3,4] == ([1,2,3],[4,1,2,3,4])
break (<9) [1,2,3] == ([],[1,2,3])
break (>9) [1,2,3] == ([1,2,3],[])
break even [1,2,3] = ([1], [2,3])

break p is equivalent to span (not . p).

Äquivalent:

```
splitWhen pred [] = ([], [])
splitWhen pred (x:xs)
        |pred x = ([], (x:xs))
        |otherwise = let (ys, zs) = splitWhen pred xs in (x:ys, zs)
```

$stripPrefix :: Eq\ a \Rightarrow [a] \to [a] \to Maybe\ [a]$
The stripPrefix function drops the given prefix from a list. It returns Nothing if the list did not start with the prefix given, or Just the list after the prefix, if it does.

stripPrefix “foo” “foobar” == Just “bar”
stripPrefix “foo” “foo” == Just ””
stripPrefix “foo” “barfoo” == Nothing
stripPrefix “foo” “barfoobaz” == Nothing

group :: *Eq a* ⇒ [*a*] → [[*a*]]
The group function takes a list and returns a list of lists such that the concatenation of the result is equal to the argument. Moreover, each sublist in the result contains only equal elements. For example,

group "Mississippi" = ["M","i","ss","i","ss","i","pp","i"]

It is a special case of groupBy, which allows the programmer to supply their own equality test.

Äquivalent:

```
groups :: Eq t => [t] -> [[t]]
groups [] = []
groups (x:xs) = let (h, r) = break (/=x) xs in (x:h):groups r
```

inits :: [*a*] → [[*a*]]
The inits function returns all initial segments of the argument, shortest first. For example,

inits "abc" == ["","a","ab","abc"]

Note that inits has the following strictness property: inits (xs ++ _|_) = inits xs ++ _|_ In particular, inits _|_ = [] : _|_

tails :: [*a*] → [[*a*]]
The tails function returns all final segments of the argument, longest first. For example,

tails "abc" == ["abc", "bc", "c", ""]

Note that tails has the following strictness property: tails _|_ = _|_ : _|_

5.2 Predicates

isPrefixOf :: *Eq a* ⇒ [*a*] → [*a*] → *Bool*
The isPrefixOf function takes two lists and returns True iff the first list is a prefix of the second.

isSuffixOf :: *Eq a* ⇒ [*a*] → [*a*] → *Bool*
The isSuffixOf function takes two lists and returns True iff the first list is a suffix of the second. The second list must be finite.

isInfixOf :: *Eq a* ⇒ [*a*] → [*a*] → *Bool*
The isInfixOf function takes two lists and returns True iff the first list is contained, wholly and intact, anywhere within the second.

isInfixOf "Haskell" "I really like Haskell." == True
isInfixOf "Ial" "I really like Haskell." == False

isSubsequenceOf :: *Eq a* ⇒ [*a*] → [*a*] → *Bool*
The isSubsequenceOf function takes two lists and returns True if all the elements of the first list occur, in order, in the second. The elements do not have to occur consecutively. isSubsequenceOf x y is equivalent to elem x (subsequences y).

6 Searching lists

6.1 Searching by equality

$elem :: (Foldable\ t, Eq\ a) \Rightarrow a \to t\ a \to Bool$
Does the element occur in the structure?

$notElem :: (Foldable\ t, Eq\ a) \Rightarrow a \to t\ a \to Bool$
notElem is the negation of elem.

$lookup :: Eq\ a \Rightarrow a \to [(a, b)] \to Maybe\ b$
lookup key assocs looks up a key in an association list.

6.2 Searching with a predicate

$find :: Foldable\ t \Rightarrow (a \to Bool) \to t\ a \to Maybe\ a$
The find function takes a predicate and a structure and returns the leftmost element of the structure matching the predicate, or Nothing if there is no such element.

$filter :: (a \to Bool) \to [a] \to [a]$
filter, applied to a predicate and a list, returns the list of those elements that satisfy the predicate; i.e.,

```
filter p xs = [ x |x <- xs, p x]
```

$partition :: (a \to Bool) \to [a] \to ([a], [a])$
The partition function takes a predicate a list and returns the pair of lists of elements which do and do not satisfy the predicate, respectively; i.e.,

```
partition p xs == (filter p xs, filter (not . p) xs)
```

7 Indexing Lists

These functions treat a list xs as a indexed collection, with indices ranging from 0 to length xs - 1.

$(!!) :: [a] \to Int \to a$
List index (subscript) operator, starting from 0 (first element has index 0). It is an instance of the more general genericIndex, which takes an index of any integral type.

$elemIndex :: Eq\ a \Rightarrow a \to [a] \to MaybeInt$
The elemIndex function returns the index of the first element in the given list which is equal (by ==) to the query element, or Nothing if there is no such element.

$elemIndices :: Eq\ a \Rightarrow a \to [a] \to [Int]$
The elemIndices function extends elemIndex, by returning the indices of all elements equal to the query element, in ascending order.

$findIndex :: (a \to Bool) \to [a] \to MaybeInt$
The findIndex function takes a predicate and a list and returns the index of the first element in the list satisfying the predicate, or Nothing if there is no such element.

$findIndices :: (a \to Bool) \to [a] \to [Int]$
The findIndices function extends findIndex, by returning the indices of all elements satisfying the predicate, in ascending order.

8 Zipping and unzipping lists

$zip :: [a] \to [b] \to [(a, b)]$
zip takes two lists and returns a list of corresponding pairs. If one input list is short, excess elements of the longer list are discarded.
zip is right-lazy:

$$\text{zip } [] \ \_|\_ = []$$

Analog: zip3, zip4, zip5, zip6, zip7 (mit 3, 4, 5, 6, 7 Eingabelisten)

$zipWith :: (a \to b \to c) \to [a] \to [b] \to [c]$
zipWith generalises zip by zipping with the function given as the first argument, instead of a tupling function. For example, zipWith (+) is applied to two lists to produce the list of corresponding sums.
zipWith is right-lazy:

$$\text{zipWith f } [] \ \_|\_ = []$$

Analog: zipWith3, zipWith4, zipWith5, zipWith6, zipWith7 (mit 3, 4, 5, 6, 7 Eingabelisten)

$unzip :: [(a, b)] \to ([a], [b])$
unzip transforms a list of pairs into a list of first components and a list of second components.
Analog: unzip3, unzip4, unzip5, unzip6, unzip7 (mit 3, 4, 5, 6, 7 Eingabelisten)

9 Special lists

9.1 Functions on strings

lines :: *String* → [*String*]
lines breaks a string up into a list of strings at newline characters. The resulting strings do not contain newlines. Note that after splitting the string at newline characters, the last part of the string is considered a line even if it doesn't end with a newline. For example,

lines "" == []
lines "\n" == [""]
lines "one" == ["one"]
lines "one\" == ["one"]
lines "one\\" == ["one",""]
lines "one\two" == ["one","two"]
lines "one\two\" == ["one","two"]

Thus lines s contains at least as many elements as newlines in s.

words :: *String* → [*String*]
words breaks a string up into a list of words, which were delimited by white space.

unlines :: [*String*] → *String*
unlines is an inverse operation to lines. It joins lines, after appending a terminating newline to each.

unwords :: [*String*] → *String*
unwords is an inverse operation to words. It joins words with separating spaces.

9.2 "Set" operations

nub :: *Eq a* ⇒ [*a*] → [*a*]
O(n^2). The nub function removes duplicate elements from a list. In particular, it keeps only the first occurrence of each element. (The name nub means 'essence'.) It is a special case of nubBy, which allows the programmer to supply their own equality test.

delete :: *Eq a* ⇒ *a* → [*a*] → [*a*]
delete x removes the first occurrence of x from its list argument. For example,

delete 'a' "banana" == "bnana"

It is a special case of deleteBy, which allows the programmer to supply their own equality test.

(\\) :: *Eq a* ⇒ [*a*] → [*a*] → [*a*]
The \\ function is list difference (non-associative). In the result of xs
ys, the first occurrence of each element of ys in turn (if any) has been removed from xs. Thus

(xs ++ ys) \\xs == ys.

It is a special case of deleteFirstsBy, which allows the programmer to supply their own equality test.

union :: *Eq a* ⇒ [*a*] → [*a*] → [*a*]
The union function returns the list union of the two lists. For example,

"dog" `union` "cow" == "dogcw"

Duplicates, and elements of the first list, are removed from the the second list, but if the first list contains duplicates, so will the result. It is a special case of unionBy, which allows the programmer to supply their own equality test.

$intersect :: Eq\ a \Rightarrow [a] \rightarrow [a] \rightarrow [a]$
The intersect function takes the list intersection of two lists. For example,

[1,2,3,4] `intersect` [2,4,6,8] == [2,4]

If the first list contains duplicates, so will the result.

[1,2,2,3,4] `intersect` [6,4,4,2] == [2,2,4]

It is a special case of intersectBy, which allows the programmer to supply their own equality test. If the element is found in both the first and the second list, the element from the first list will be used.

9.3 Ordered Lists

$sort :: Ord\ a \Rightarrow [a] \rightarrow [a]$
The sort function implements a stable sorting algorithm. It is a special case of sortBy, which allows the programmer to supply their own comparison function.
Elements are arranged from from lowest to highest, keeping duplicates in the order they appeared in the input.

$sortOn :: Ord\ b \Rightarrow (a \rightarrow b) \rightarrow [a] \rightarrow [a]$
Sort a list by comparing the results of a key function applied to each element. sortOn f is equivalent to sortBy (comparing f), but has the performance advantage of only evaluating f once for each element in the input list. This is called the decorate-sort-undecorate paradigm, or Schwartzian transform.
Elements are arranged from from lowest to highest, keeping duplicates in the order they appeared in the input.
Since: 4.8.0.0

$insert :: Ord\ a \Rightarrow a \rightarrow [a] \rightarrow [a]$
The insert function takes an element and a list and inserts the element into the list at the first position where it is less than or equal to the next element. In particular, if the list is sorted before the call, the result will also be sorted. It is a special case of insertBy, which allows the programmer to supply their own comparison function.