# Lazy evaluation

**Exercise 9.1** (*Warm-up:* Type derivation, `PolyTypes.hs`). Derive the *most general type* of the following polymorphic functions. Also include any type class information, where needed.

```
mingle xs ys = [ e | (x,y) ← zip xs ys, e ← [x,y] ]

sumWith g xs = foldr (+) 0 (map g xs)

transform f  = concat . map f
```

Check your answers using GHCi! If you had a different answer, explain the difference.

**Exercise 9.2** (*Warm-up:* Reasoning about lazy programs, `FindMinimum.hs`). Suppose that for some unfathomable reason, in your software project *selection sort* was introduced by some programmer (who has since left your team, for obvious reasons):

```
selectionSort :: (Ord a) ⇒ [a] → [a]
selectionSort [] = []
selectionSort xs = let e = minimum xs in e : selectionSort (delete e xs)
```

In a different part of the project, another programmer, unaware of the existence of the `minimum` function, decides to implement their own version as follows:

```
leastElem :: (Ord a) ⇒ [a] → a
leastElem = head . selectionSort
```

1. In call-by-value programming languages, why would this be a terrible idea?
2. How inefficient is this choice *really* in Haskell? Compare the speed of `selectionSort` on `someNums` to the `leastElem` function defined above. How do you explain the difference?
3. If you find this question a bit too trivial, consider the case where instead of *selection sort*, *insertion sort* was being used:

   ```
   insertionSort :: (Ord a) ⇒ [a] → [a]
   insertionSort []       =  []
   insertionSort (x : xs)  =  insert x (insertionSort xs)
   ```

   Which sorting algorithm is more efficient when used as a means to implement `leastElem`?

*(Note: this is only an exercise; implementing `leastElem` this way is in either case a poor choice.)*

**Exercise 9.3** (*Warm-up*: Laziness and folds, `ShortCircuit.hs`).
*(Note: this is the same exercise as Exercise 5.9. See also the quick Exercise 9.7, and Hint 1)*

Both `&&` and the `||` inspect first the value of the first argument, and if the second argument is not needed to determine the result, it is not evaluated. So we can safely write things like:

```
infinitesimal x = x == 0 || 1 / x ≥ 1e10
```

The functions `and` and `or` 'lift' these conditional tests to lists; they can be expressed using both `foldl` and `foldr`; so let's do that:

```
andl = foldl (&&) True
andr = foldr (&&) True
orl  = foldl (||) False
orr  = foldr (||) False
```

1. Predict what will happen when `andl`, `andr`, `orl` and `orr` are applied to an *infinite list* of booleans: Do this using the following examples. Check your predictions!

   ```
   andl $ False : [True,  True  ..]
   andr $ False : [True,  True  ..]
   orl  $ True  : [False, False ..]
   orr  $ True  : [False, False ..]
   ```

2. For *finite* lists, these functions are equivalent to `and` and `or` respectively. But in both cases one definition is preferable over the other. Why?

**Exercise 9.4** (*Mandatory*: Infinite data structures, `Stream.hs`). *Note:* many of the definitions below clash with definitions in the standard prelude. The skeleton file for this exercise uses a `hiding` clause to avoid these clashes.

Haskell's list datatype comprises both finite and infinite lists. We will be working with infinite lists *only*, called *streams*, which we will therefore define very similarly to lists, but omitting the case for the empty stream:

```
data Stream a = a :> Stream a
infixr 5 :>
```

For convenience, we have provided an instance of `Show` which limits the display to the first 16 elements (feel free to adjust it).

As an example, the sequence of natural numbers is given by `from 0` with `from` defined as:

```
from :: Integer → Stream Integer
from n = n :> from (n + 1)
```

1. Define the functions:

   ```
   head :: Stream a → a
   tail :: Stream a → Stream a
   ```

   that perform the similar operations as their list counterparts.

2. Define functions

```
repeat  :: a → Stream a
map     :: (a → b) → (Stream a → Stream b)
zipWith :: (a → b → c) → (Stream a → Stream b → Stream c)
filter  :: (a → Bool) → (Stream a → Stream a)
```

that correspond to their counterparts for ordinary lists:

```
 ≫ repeat 1
(1 :> 1 :> 1 :> 1 :> 1 :> 1 :> 1 :> 1 :> 1 :> 1 :> 1 :> 1 :> 1 :> 1 :> ...)
 ≫ map (2 *) (from 0)
(0 :> 2 :> 4 :> 6 :> 8 :> 10 :> 12 :> 14 :> 16 :> 18 :> 20 :> 22 :> 24 :> 26 :> ...)
 ≫ zipWith (*) (from 0) (from 1)
(0 :> 2 :> 6 :> 12 :> 20 :> 30 :> 42 :> 56 :> 72 :> 90 :> 110 :> 132 :> 156 :> 182 :> ...)
 ≫ filter odd (from 0)
(1 :> 3 :> 5 :> 7 :> 9 :> 11 :> 13 :> 15 :> 17 :> 19 :> 21 :> 23 :> 25 :> 27 :> ...)
```

3. Explain what happens if we call `filter (\x→False) (from 0)`.

4. Define functions

```
toList :: Stream a → [a]
cycle :: [a] → Stream a
```

that converts streams into list and vice versa. If a list is finite, the stream should simply repeat it:

```
 ≫ cycle [1,2,3]
(1 :> 2 :> 3 :> 1 :> 2 :> 3 :> 1 :> 2 :> 3 :> 1 :> 2 :> 3 :> 1 :> ...)
```

5. Examples of streams are the natural numbers and the Fibonacci numbers:

```
nat, fib :: Stream Integer
nat = 0 :> zipWith (+) nat (repeat 1)
fib = 0 :> 1 :> zipWith (+) fib (tail fib)
```

Define the stream `primetwins :: Stream (Integer,Integer)` listing all the *prime twins*. A *prime twin* is pair of primes that differ by exactly two, such as $(3, 5)$ and $(5, 7)$. (You will probably need to define the stream of prime numbers as an auxilliary definition.)

6. We can concatenate lists using `(++)`, but for infinite lists this will not work: if `xs` is infinite, then `xs ++ ys` would only ever return elements from `xs`; the same would apply to streams.

Define a function `combine :: Stream a → Stream a → Stream a` that combines two streams into one, but does not have this problem; i.e. every element of either input stream is guaranteed to appear in the output stream eventually.

7. *Optional (Challenging!)* Generalizing the previous function, write a function `diag :: Stream (Stream a) → Stream a` that takes a *stream of streams* and returns a stream that is guaranteed to (eventually) produce every element that is in one of the input streams.

Tip: there are many possible choices as input, but one expression of type `Stream (Stream Integer)` is for example `map repeat nat` or `map from nat`.

**Exercise 9.5** (*Mandatory:* Memoization, `EditDistance.hs`).
One possible distance measure between two strings is the *edit distance* or *Levenshtein distance*. The Levenshtein distance is the minimum number of *edit steps* that need to be taken to transform one string into the other. An edit step is either *inserting*, *deleting*, or *replacing* a character.

There might be multiple edit sequences that achieve the same edit distance: for instance, the strings *"deity"* and *"definite"* have a Levenshtein distance of 4: from "*definite*" we can delete one *f*, *n* and *i*, and change the final *e* into a *y* to get "*deity*". But we have a choice of which *i* to delete.

A naive recursive way to define the edit distance is as follows:

```
distance :: String → String → Int
distance [] ys = length ys
distance xs [] = length xs
distance (x:xs) (y:ys) = minimum [1 + distance xs (y:ys)     -- delete x
                                 ,1 + distance (x:xs) xs     -- insert y
                                 ,cost x y + distance xs ys] -- match x with y
  where cost x y = if x==y then 0 else 1
```

However, this is extremely inefficient: intermediate results are re-computed over and over again. To prevent this, *memoization* is needed: compute each result once, and store it for later usage.[1] This idea is captured by an *edit distance matrix* where for each position $(i, j)$ the matrix lists the minimum cost to transform a suffix of one string (starting at position $i$), into the suffix of the other (starting at $j$). Then the position $(0, 0)$ lists the full edit distance. For example, the *edit distance matrix* for the strings "*deity*" and "*definite*" looks like: (here **$** denotes the end-of-string)

| | *d* | *e* | *f* | *i* | *n* | *i* | *t* | *e* | **$** |
|---|---|---|---|---|---|---|---|---|---|
| *d* | 4 | 5 | 4 | 3 | 3 | 3 | 4 | 4 | 5 |
| *e* | 5 | 4 | 4 | 3 | 2 | 2 | 3 | 3 | 4 |
| *i* | 6 | 5 | 4 | 3 | 2 | 1 | 2 | 3 | 3 |
| *t* | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 2 | 2 |
| *y* | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 1 |
| **$** | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

In strict programming languages, you have to be careful *how* to fill this matrix in the correct order; but by using *lazy functional arrays* as shown in the lecture, we can let Haskell's lazy evaluation do that work for us.

Implement the *edit distance* using memoization and a lazy `Data.Array`. We recommend a two-step approach:

- First, change the definition of `distance` given above so it uses a helper function of type `(Int,Int) → Int`, instead of `String → String → Int`, that essentially computes the values of the cells in the *edit distance matrix*. This will not be any more efficient, but it allows you to focus on one task at a time.
- Then, create a `Array (Int,Int) Int` that contains the results of your helper function, and change your helper function to use it instead of direct recursive calls to itself:

```
distArray :: Array (Int,Int) Int
distArray = array ((0,0), (max_x, max_y))
                  [(ij, compute ij) | i ← [0..max_x], j ← [0..max_y], let ij = (i,j)]
```

[1] *"Almost all programming can be viewed as an exercise in caching" (Terje Mathisen)*

**Exercise 9.6** (*Extra:* Functional magic, `Ouroboros.hs`).
A common data structure in functional programming language is the *association list*. We have seen this before in e.g. building a Huffman tree. We can create a type alias for it:

```
type AssocList k a = [(k,a)]
```

The idea here is that this list associates keys with values (hence the name). By convention for association lists *keys are unique*: each key uniquely identifies a single value.

1. Write a function:

   ```
   insertLookup :: (Eq k) ⇒ k → a → AssocList k a → (Maybe a, AssocList k a)
   ```

   That combines an element lookup and insertion; that is, `insertLookup k v` will insert the key `k` with value `v` into the provided association list, but also returns the old value for that key, if it existed:

   ```
   ⋙ insertLookup "FP" "Ralf" [("OO", "Sjaak"), ("MS", "Engelbert")]
   (Nothing,[("OO","Sjaak"),("MS","Engelbert"),("FP","Ralf")])
   ⋙ insertLookup "FP" "Sjaak" [("FP", "Ralf"), ("A+D", "Frits")]
   (Just "Ralf",[("FP","Sjaak"),("A+D","Frits")])
   ```

   Using `insertLookup`, it is easy to define `insert` and `lookup`:

   ```
   insert :: (Eq k) ⇒ k → a → AssocList k a → AssocList k a
   insert k v kvs = snd $ insertLookup k v kvs

   lookup :: (Eq k) ⇒ k → AssocList k a → Maybe a
   lookup k kvs   = fst $ insertLookup k undefined kvs
   ```

   (Note the use of the function `undefined`, which like `error`, will give a runtime error if evaluated—but due to lazy evaluation, this will not occur.)

2. Using `insertLookup` (so **without using recursion**), define a function:

   ```
   adjust :: (Eq k) ⇒ (a → a) → k → AssocList k a → AssocList k a
   ```

   Which updates the value at a specific key with the result of the provided function. When the key is not a member of the association list, the original list is returned. For example:

   ```
   ⋙ adjust ("prof."++) "H-in-C" [("CT", "prof.dr. Jacobs"), ("H-in-C", "dr. Schwabe")]
   [("CT","prof.dr. Jacobs"),("H-in-C","prof.dr. Schwabe")]
   ⋙ adjust (\x→10.0) "H-in-C" [("FP", 7.5), ("OO", 6.0)]
   [("FP", 7.5), ("OO", 6.0)]
   ```

   This only requires a little bit more code than the definitions of `insert` and `lookup` above!

*(An ouroboros is a symbol depicting a snake eating its own tail.)*

**Exercise 9.7** (*Extra:* Forcing strict evaluation).
In Exercise9.3, we have seen that when using short circuiting operators on infinite lists, there can be a big difference between `foldr` and `foldl`. During the lecture, it was discussed how `seq` can be used to force strict (i.e. non-lazy) evaluation, and that `($!)` can be used as a 'strict apply' to force call-by-value evaluation.

In the module `Data.List`, there is also a 'strict `foldl`', namely `foldl'`:

```
foldl' :: (b → a → b) → b → [a] → b
foldl' f b []     = b
foldl' f b (x:xs) = let b' = f b x
                    in seq b' $ foldl' f b' xs
```

(Compare this to regular `foldl` — see Hint 1)

This exercise will explore its reason for existence.

1. When are intermediate sums arising in the expression `foldl (+) 0 [1..10]` evaluated? What about `foldl'(+) 0 [1..10]`?

2. The following folding expressions should return `True`, but of course need to evaluate the entire list for that (which in this case consists of 100 million boolean values):

   ```
   ≫ foldr (&&) True (replicate 100000000 True)
   ```

   ```
   ≫ foldl (&&) True (replicate 100000000 True)
   ```

   Which do you think is more efficient?

   Turn on performance information in GHCi using `:set +s`, and evaluate them to see which is faster and/or uses less memory.

3. Since we know the entire list must be processed, there is no penalty in using *strict evaluation*. We can do that by using `foldl'`. To do that, we need to import `Data.List`:

   ```
   ≫ import Data.List
   ≫ foldl' (&&) True (replicate 100000000 True)
   ```

   Again, make sure `:set +s` is turned on.

   We know that all three folding expressions will return the same result (why?). However, which of `foldr`, `foldl` and `foldl'` would you prefer to use in this case?

For more about `foldr`, `foldl` and `foldl'`, see: https://wiki.haskell.org/Foldr_Foldl_Foldl%27.

**Exercise 9.8** (*Extra:* Infinite data structures, `Nat.hs`).
In mathematics (to be precise, in the *Peano axioms*) the natural numbers are defined as follows:

- The number 0 is a natural number.
- Every natural number has a successor that is a natural number.
- Distinct numbers have distinct successors, 0 is never a successor.

In Haskell, we can capture this using an algebraic data type:

```
type Nat = O | S Nat deriving (Show)
```

The value `O` corresponds to the number 0, and the constructor function `S` corresponds to the increment function. So for example, the number 3 can be represented as `S (S (S O))`.

1. Provide instances for the type classes `Eq` and `Ord` for `Nat`. Of course you can cheat by using `deriving`, as we did for `Show`, but implementing them yourself is not hard, and a good exercise!

2. Implement recursive functions `(+.), (-.) :: Nat → Nat → Nat` that add/subtracts two values of type `Nat`:

   ```
   ⋙ S (S (S O)) +. S (S O)
   S (S (S (S (S O))))
   ```

   Subtraction is, of course, problematic for natural numbers; solve this like when you first learned to count: "you cannot take away something from nothing", so for instance the difference 5 − 42 is either 0, or should result in a runtime error.

   You can of course also implement `(*.) :: Nat → Nat → Nat` to multiply two natural numbers, but we don't need that in this exercise.

3. Implement a function `natLength :: [a] → Nat` that computes the length of a list as a `Nat`. Do you expect `natLength [1..]` to terminate? Try it! If it does return, can you compare the value it produces for equality or inequality to say, `toNat 5`? Do the results make sense?

4. Consider this definition:

   ```
   infinity :: Nat
   infinity = S infinity
   ```

   What does `infinity +. infinity` or `infinite -. infinity` produce?

   Is the name `infinity` appropriate? Why (or: why not)?

**Exercise 9.9** (*Extra:* Infinite trees, `Minimax.hs`). The purpose of this exercise is to explore the AI behind "strategic games", where a human plays against a computer. For simplicity, we confine ourselves to impartial two-player games where players take alternate moves and the same moves are available to both players. As an example, consider the following game:

A position of the game is given by two positive integers, say, `m, n > 0`. A move consists of picking a number, say, `m` and dividing it into two positive natural numbers `i, j > 0` such that `i + j = m`. If a player cannot make a move, i.e. `m = n = 1`, they lose. Here is an example run of the game:

> initial position `5, 7`
> Player A selects `7` and returns `3, 4`
> Player B selects `4` and returns `1, 3`
> Player A selects `3` and returns `1, 2`
> Player B selects `2` and returns `1, 1`
> Player A loses

The position `5, 7` is actually a losing position: no matter what Player A's first move, they will always lose, unless Player B makes a mistake. This can be seen by inspecting the entire game tree spawned by the initial position, see Figure 1.

1. Implement a function

   ```
   type Position = (Integer, Integer)

   moves :: Position → [Position]
   ```

   that returns a list of all possible moves from a given position e.g. `moves (1, 1) = []` and `moves (5,7) = [(1,4),(2,3),(1,6),(2,5),(3,4)]`.

2. A game tree is an instance of a *multiway tree* or *rose tree*, which is defined by the following datatype declaration:

   ```
   data Tree elem  =  Node elem [Tree elem]
   ```

   Define a function

   ```
   gametree :: (position → [position]) → (position → Tree position)
   ```

   that constructs the entire game tree for a given initial position e.g. `gametree moves (5, 7)` should yield the tree shown in Figure 1. (Perhaps, you want to write a function that displays a game tree in a human-readable way.)

3. Implement a function

   ```
   size :: Tree elem → Integer
   ```

   that computes the size of a multiway tree. As usual, the size corresponds to the number of elements contained in the tree.

4. Define functions

   ```
   winning  :: Tree position → Bool
   losing   :: Tree position → Bool
   ```

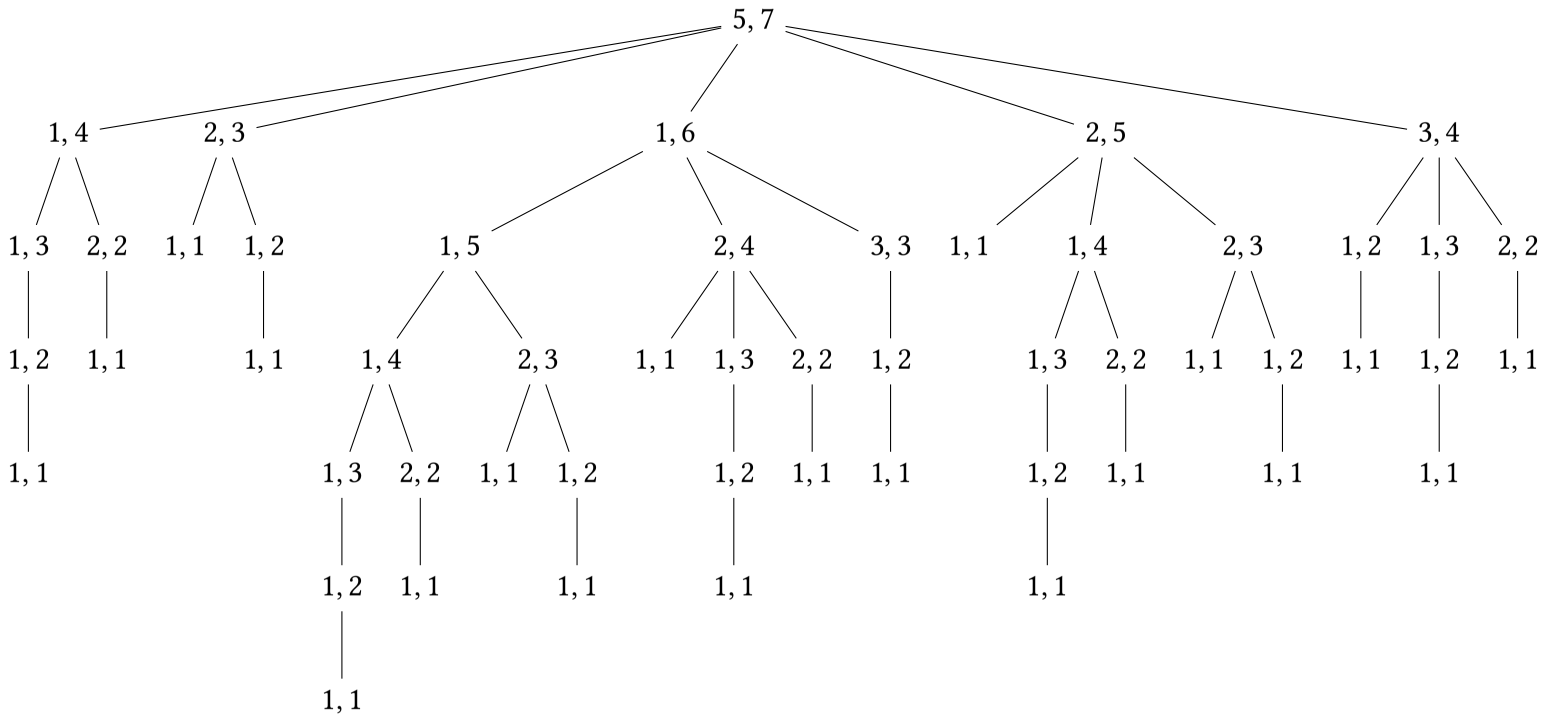


Figure 1: Game tree spawned by the initial position 5, 7 (pick'n'divide game)

that determine whether the root of a game tree is labelled with a winning or a losing position. A position is a winning position iff *some* successor position is a losing position. Dually, a position is a losing position iff *all* successor positions are winning positions. By that token, a final position is a losing position.

Apply both `size` and `winning` to some largish game tree and measure the running times (using e.g. `:set +s` within GHCi). What do you observe?

5. *Optional:* show that you lose if you don't win and vice versa:

```
not (winning gametree) = losing gametree
not (losing gametree) = winning gametree
```

6. *Optional:* in case you need some food for thought here are some additional games worth exploring.

   **mark game** A position is given by a positive integer `n > 0`; the final position is `1`; a move from a non-final position consists of replacing `n` by either `n - 1` or by `n / 2` provided `n` is even.

   **down-mark game** The set-up is identical to the mark game, except that `n` is replaced by either `n - 1` or by $\lfloor n/2 \rfloor$.

   **up-mark game** Again, the set-up is identical to the mark game, except that `n` is replaced by either `n - 1` or by $\lceil n/2 \rceil$.

Exploring an entire game tree is usually not an option, in particular, if the tree is wide and deep. To illustrate, if each position gives rise to exactly two moves, then a game tree of depth 500 has $2^{500} \approx 3 * 10^{150}$ leaves. For comparison, the number of atoms in the observable universe is estimated at $10^{78}$–$10^{82}$. Because of the exponential growth game trees are only evaluated up to some given depth e.g.

```
evaluate :: Integer → Position → Value
evaluate depth  =  maximize static . prune depth . gametree moves
```

7. Define a function

   ```
   prune :: Integer → Tree elem → Tree elem
   ```

   that prunes a multiway tree to a given depth i.e. the depth of `prune depth gametree` is at most `depth`.

8. Implement a function

   ```
   type Value = Int  -- [-100 .. 100]

   static :: Position → Value
   ```

   that statically evaluates a given position, returning some integer in the range `[-100 .. 100]`. In general, the higher the value, the higher we *estimate* our chances of winning. Thus, `-100` indicates a definite losing position, `100` a definite winning position. The other values reflect the uncertainty in our judgement.

(If you have toyed a bit with the pick'n'divide game, you may be able to come up with a static evaluation that precisely predicts the outcome. In that case, you may want to consider one of the other games suggested above.)

9. Define functions

```
maximize  :: (position → Value) → (Tree position → Value)
minimize  :: (position → Value) → (Tree position → Value)
```

that evaluate a game tree, based on the static evaluation function provided. The function `maximize` works as follows: if the node has no sub-trees (as a result of pruning or because it is a final position), then the static evaluation is used; otherwise, `minimize` is applied to each of the sub-trees and the maximal value of these is returned. Dually, `minimize` returns the negation of the static evaluation for leaf nodes; otherwise, it applies `maximize` to each of the subtrees, returning the minimal resulting value.

10. *Optional:* show that

```
negate (maximize static gametree) = minimize static gametree
negate (minimize static gametree) = maximize static gametree
```

11. *Highly Optional:* re-implement `maximize` and `minimize` using *alpha-beta pruning*. (Use Google, Wikipedia or a good algorithms bible to discover what it is.)

**Hints to practitioners 1.** *(For Exercise 9.3 and 9.7)*
Recall the definition of `foldr` and `foldl` on lists from Week 5:

```
foldr :: (a → b → b) → b → [a] → b
foldr f b [] = b
foldr f b (x:xs) = f x (foldr f b xs)

foldl :: (b → a → b) → b → [a] → b
foldl f b [] = b
foldl f b (x:xs) = foldl f (f b x) xs
```

Compare `foldl` with `foldl'` from `Data.List`:

```
foldl' :: (b → a → b) → b → [a] → b
foldl' f b []     = b
foldl' f b (x:xs) = let b' = f b x
                    in seq b' $ foldl' f b' xs
```