13 Duality: folds and unfolds

Exercise 13.1 (Warm-up: programming, `FoldUnfold.hs`). In the lectures we have re-defined `foldr` in order to exhibit the duality between folds and unfolds more clearly.

```
type LIST a = Maybe (a,[a])

out :: [a] → LIST a
out []      = Nothing
out (x:xs)  = Just (x,xs)

inn :: LIST a → [a]
inn Nothing        = []
inn (Just (x,xs)) = x:xs

foldR   :: (Maybe (a,b) → b) → ([a] → b)
foldR al   = consume where
   consume = al ∘ fmap (fmap consume) ∘ out
```

But, are the two definitions, `foldr` and `foldR`, actually equivalent?

1. Define `foldr` in terms of `foldR`.

2. Conversely, define `foldR` in terms of `foldr`.

The type `Maybe(a,b)` was used to represent one layer of the list type. The `inn` and `out` functions are conversions between the *isomorphic* types `Maybe(a,[a])` and `[a]`. Suppose we would have represented a layer by the following data type

```
data LISTB elem res = NIL | CONS elem res
```

3. Show that `Maybe(a,b)` and `LISTB a b` are isomorphic too. You can do this by defining two conversion functions:

   ```
   mbP2LB :: Maybe (elem,res) → LISTB elem res

   lB2MbP :: LISTB elem res → Maybe(elem,res)
   ```

4. Also show that `LISTB a [a]` and `[a]` are isomorphic. The *witnesses* of this isomorphism have type

   ```
   outB :: [a] → LISTB a [a]
   innB :: LISTB a [a] → [a]
   ```

   Define these functions in terms of `inn`, `out`, `mbP2LB` and `lB2MbP`

5. Make `LISTB a` an instance of class `Functor`.

6. Consider the following definition for

   ```
   foldRB al    = consume where
      consume = al ∘ fmap consume ∘ outB
   ```

What is the type of `foldRB`. Define also the dual version `unfoldRB`

7. Now, take the following alternative mapping function for type `LISTB a b`

```
emap :: (a → b) → LISTB a c → LISTB b c
emap f NIL          = NIL
emap f (CONS a b) = CONS (f a) b
```

   Now re-define the original `Prelude` function `map` in terms of `foldRB` and `emap`

8. Dually, re-define `map` in terms of `unfoldRB` and `emap`

Exercise 13.2 (Programming, `Minimax2.hs`). Reconsider the *multiway trees* of Exercise 9.4. (By the way, it's no problem if you haven't done this assignment before.)

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

1. Introduce a base functor for `Tree` and provide suitable class instances of `Functor` and `Base`.

2. Re-define the functions

```
size, depth :: Tree elem → Integer
```

   using `fold`. The function `size` computes the size of a multiway tree i.e. the number elements of type `elem`; the function `depth` computes the length of the longest path from the root to a leaf.

3. Re-implement the function

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

   that constructs a game tree in terms of `unfold`.

4. Re-implement the function

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

   using `fold`. The function determines whether the root of a game tree is labelled with a winning position.

Exercise 13.3 (Worked example: algorithmic duality, `Sorting.hs`).

⟨…⟩ we can see that sorting is worth of serious study,
as a practical matter.
Even if sorting were almost useless, there would be plenty of rewarding
reasons for studying it anyway! The ingenious algorithms that have been
discovered show that sorting is an extremely interesting topic to explore
in its own right. Many fascinating unsolved problems remain in this area,
as well as quite a few solved ones.
From a broader perspective we will find also that sorting algorithms make a valuable
case study *of how to attack computer programming problems in general.*
The Art of Computer Programming, Volume 3—Donald E. Knuth

We have used sorting as a running example to illustrate the idea of algorithmic duality. This exercise continues the journey looking at a family of sorting algorithms that are based on search trees, either implicitly or explicitly. These algorithms work in two phases:

- first phase: create a search tree from an unordered list;
- second phase: flatten the search tree to an ordered list.

In order to be able to apply the machinery of higher-order operators such as folds and unfolds, we assume the following definition of binary trees and its associated base functor.

```
data Tree elem      =  Empty | Node (Tree elem) elem (Tree elem)
data TREE elem tree =  EMPTY | NODE tree elem tree
```

During the lectures, the first phase has already been discussed. The (non-recursive) core algorithm that served as the basis for two dual variants of phase one, was defined as follows:

```
growCore :: (Ord a) ⇒ LIST a (x,TREE a x) → TREE a (Either x (LIST a x))
growCore NIL = EMPTY
growCore (CONS a (et, EMPTY)) = NODE (Left et) a (Left et)
growCore (CONS a (nt, NODE l b r))
  | a < b       =  NODE (Right (CONS a l)) b (Left r)
  | otherwise   =  NODE (Left l) b (Right (CONS a r))
```

The dual functions that create a search tree are:

```
grow1, grow2 :: (Ord elem) ⇒ [elem] → Tree elem
grow1  =  unfold  (para  (fmap (joinRight inn) ∘ growCore))
grow2  =  fold    (apo  (growCore ∘ fmap (splitRight out)))
```

1. Define functions that flatten a search tree.

   ```
   flatten1, flatten2 :: Tree elem → [elem]
   flatten1  =  fold    (apo   (flattenCore ∘ fmap (splitRight out)))
   flatten2  =  unfold  (para  (fmap (joinRight inn) ∘ flattenCore))
   ```

   Again, you only have to define the *algorithmic core* of these functions:

   ```
   flattenCore :: TREE a (x, LIST a x) → LIST a (Either x (TREE a x))
   ```

   If you're not able to find a proper definition for `flattenCore`, it might be a good idea to specify the coalgebra `flattenCore ∘ fmap (splitRight out)` directly. It might even be a better idea to do this for the complete algebra `apo (flattenCore ∘ fmap (splitRight out))`. Hence, we are looking for an algebra, say `flatalg` such that

   ```
   flatten1 :: Tree elem → [elem]
   flatten1  =  fold  flatalg
   ```

   From the type of `fold` we can infer that

   ```
   flatalg :: TREE elem [elem] → [elem]
   ```

   The definition of `flatalg` is:

```
flatalg :: TREE elem [elem] → [elem]
flatalg EMPTY = []
flatalg (NODE l e r) = l ++ [e] ++ r
```

Agree? Now let's re-define `flatalg` in term of `apo`, i.e.

```
flatalg :: TREE elem [elem] → [elem]
flatalg = apo flatalgcoalg
```

Again we start we deducing the type of `flatalgcoalg` from the context, which gives us:

```
flatalgcoalg :: TREE elem [elem] → LIST elem (Either [elem] (TREE elem [elem]))
```

Note that this coalgebra produces the next layer of the result list using a value of type `TREE elem [elem]` as a seed. The type more or less directs what the body of `flatalgcoalg` should be, namely:

```
flatalgcoalg :: TREE elem [elem] → LIST elem (Either [elem] (TREE elem [elem]))
flatalgcoalg (NODE [] e r)     = CONS e (Left r)
flatalgcoalg (NODE (x:xs) e r) = CONS x (Right (NODE xs e r))
```

However, the final goal is to specify `flatalgcoalg` in terms of `flattenCore`:

```
flatalgcoalg :: TREE elem [elem] → LIST elem (Either [elem] (TREE elem [elem]))
flatalgcoalg = flattenCore ∘ fmap (splitRight out)
```

First, verify that the type of `flatalgcoalg` is indeed correct. Are you able to define `flattenCore`? If not, try to repeat the (dual of the) above procedure for:

```
flatten2 :: Tree elem → [elem]
flatten2  =  unfold  flatcoalg
```

This should result in the (dual) algebra

```
flatcoalgalg :: TREE a (Tree a, LIST a (Tree a)) → LIST a (Tree a)
```

By requiring that

```
flatcoalgalg = fmap (joinRight inn) ∘ flattenCore
```

and combining the direct definition of `flatcoalgalg` with that of `flatalgcoalg` you should be able to obtain `flattenCore`.

2. Combine the phases to form sorting algorithms—there are four combinations altogether. Do you recognize any of the algorithms? One of them implements a version of Quick Sort, where the search tree structure is made explicit.

Exercise 13.4 (Pencil and paper: uniqueness and fusion). In the lectures we have introduced a *generic* definition of fold that works for arbitrary recursive datatypes.

```
fold :: (Base f) ⇒ (f a → a) → (Rec f → a)
fold a  =  a ∘ fmap (fold a) ∘ out
```

The generic definitions enjoys generic properties! First of all, folds enjoy the following *uniqueness property*:

$$\texttt{f} = \texttt{fold a} \iff \texttt{f} \circ \texttt{inn} = \texttt{a} \circ \texttt{fmap f} \tag{22}$$

The law states that `fold` is the unique solution of its defining equation.

1. Show that the uniqueness property implies the *computation law*:

$$\texttt{fold a} \circ \texttt{inn} = \texttt{a} \circ \texttt{fmap (fold a)} \tag{23}$$

2. Show that the uniqueness property implies the *reflection law*:

$$\texttt{id} = \texttt{fold inn} \tag{24}$$

   Note that `inn :: f (Ref f)` $\longrightarrow$ `Rec f` is an algebra.

3. Finally, show that the uniqueness property implies the *fusion law*:

$$\texttt{h} \circ \texttt{a} = \texttt{b} \circ \texttt{fmap h} \Longrightarrow \texttt{h} \circ \texttt{fold a} = \texttt{fold b} \tag{25}$$

   The fusion law states a condition for fusing a function with a fold to form another fold.

4. Use the properties above to show that `inn` and `fold (fmap inn)` are inverses of each other:

$$\texttt{inn} \circ \texttt{fold (fmap inn)} = \texttt{id} \tag{26a}$$
$$\texttt{fold (fmap inn)} \circ \texttt{inn} = \texttt{id} \tag{26b}$$

   In other words, `fold (fmap inn) = out`. *Hint:* first show (26a), then use (26a) to establish (26b).

5. *Optional:* can you dualize the laws to unfolds?