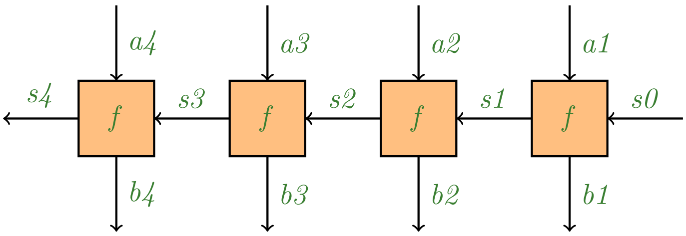
Prof. Dr. Ralf Hinze
M.Sc. Sebastian Schweizer

TU Kaiserslautern
Fachbereich Informatik
AG Programmiersprachen

Functional Programming: Exercise 5

Sheet published: Wednesday, May 15th
Submission deadline: Wednesday, May22nd, 12:00 noon

Exercise 5.1 (Skeleton: `Hardware.hs`)**.** Complex circuits are often assembled from simpler components using some regular "wiring pattern". A simple example is afforded by the ripple carry adder[1], which implements the school algorithm for addition in hardware. It consists of a series of full adders:



Assume we want to add two four bit numbers x and y. For each bit, a full adder (f) is required. It takes two summand bits (top input, i.e. each ai is a pair where one bit stems from x and the other from y) and a carry (right input). It produces a sum bit (bottom output) and a carry (left output).

a) Capture the wiring scheme as a higher-order function:

$$mapr :: ((a, state) \to (b, state)) \to (([a], state) \to ([b], state))$$

The second component of each pair can be seen as a state. In a sense, $mapr$ is a stateful version of map. Contrary to imperative programming, the state is explicit, not implicit. The diagram above shows that the state is threaded through the gates from right to left (hence the r in $mapr$).

b) We now consider some basic electronics. To represent a single bit in Haskell, we use the type **data** $Bit = O \mid I$. In our local electronics store, we found AND, OR, and XOR gates, they are represented in Haskell as follows:

$$
\begin{aligned}
&and, or, xor :: Bit \to Bit \to Bit \\
&and\ O\ \_\ = O \\
&and\ I\ \ b\ = b \\
&or\ O\ b = b \\
&or\ I\ \ \_ = I \\
&xor\ \ O\ O = O \\
&xor\ \ O\ I\ \ = I \\
&xor\ \ I\ \ O = I \\
&xor\ \ I\ \ I\ \ = O
\end{aligned}
$$

[1] `https://en.wikipedia.org/wiki/Adder_(electronics)#Ripple-carry_adder`

Using only these gates, build a half adder. It takes two bits as input and calculates the sum and a carry:

$$\textbf{type}\ Carry = Bit$$
$$halfAdder :: (Bit, Bit) \to (Bit, Carry)$$

Using only the gates and half adders, build a full adder.

$$fullAdder :: ((Bit, Bit), Carry) \to (Bit, Carry)$$

c) Implement a ripple carry adder using *mapr* with *fullAdder* (and any other Haskell function you need to use to prepare the inputs).

$$rippleAdder :: ([Bit], [Bit], Carry) \to ([Bit], Carry)$$

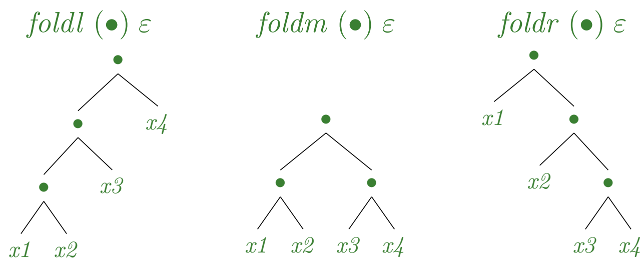
You can assume that both input lists have the same length.

Exercise 5.2 (Skeleton: `BoolMonoids.hs`)**.**

a) How many ways are there to turn the type *Bool* into a monoid? (There are sixteen *candidates* as there are sixteen functions of type $Bool \to Bool \to Bool$.) Define all of them using **newtype** definitions.

b) For each of the Boolean monoids, what is the meaning of *reduce*? Are any of these functions predefined (under a different name)?

Exercise 5.3 (Skeleton: `MapReduce.hs`)**.** In the lectures we have implemented *reduce* in terms of a higher-order function: $reduce = foldr\ (\bullet)\ \varepsilon$. Of course, this is a rather arbitrary choice: $reduce = foldl\ (\bullet)\ \varepsilon$ works equally well. Since the operation is associative, it does not matter how nested applications of '$\bullet$' are parenthesized. The overall result is bound to be the same. However, there is possibly a big difference in running time. For many applications, a balanced "expression tree" is actually preferable, for example in the mergesort algorithm. In particular, a balanced tree can in principle be evaluated in parallel!



The goal is to define a function

$$foldm :: (a \to a \to a) \to a \to [a] \to a$$

that constructs and evaluates a balanced expression tree.

a) The types of *foldl*, *foldm*, and *foldr* are quite different. Why?

b) Implement *foldm* using a *top-down* approach: Split the input list into two halves, evaluate each half separately, and finally combine the results using the monoid operation (*divide and conquer*).

c) Implement *foldm* using a *bottom-up* approach: Traverse the input list combining two adjacent elements e.g. $[x1, x2, x3, x4]$ becomes $[x1 \bullet x2, x3 \bullet x4]$. Repeat the transformation until the list is a singleton list.