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

TU Kaiserslautern
Fachbereich Informatik
AG Programmiersprachen

Functional Programming: Exercise 8

Sheet published: Friday, June 14th
Submission deadline: Wednesday, June 26th, 12:00 noon

Note regarding the public holiday On June 20th, there is a public holiday. We won't have an exercise session on that day. This sheet therefore is for two weeks, the submission deadline is on Wednesday, June 26th, 12:00 noon and it will be discussed the day after.

Exercise 8.1 (Skeleton: `WordCount.hs`). Implement a simplified version of the UNIX command `wc`, see Figure 1. For example, applied to two of the provided files, the program displays:

```
sebastian@laptop ~/Teaching/FP19/ex8 $ wc Echo.hs WordCount.hs
 10   24 154 Echo.hs
 10   25 160 WordCount.hs
 20   49 314 total
```

For accessing the program's command line arguments, the standard library *System.Environment* provides the command *getArgs* :: *IO* [*String*]. As an example, the program

```
module Main where
import System.Environment
main :: IO ()
main = do args ← getArgs
          putStrLn (unwords args)
```

echoes the command line arguments to the standard output. Notice that a stand-alone Haskell program must contain a *Main* module that contains a definition of *main* :: *IO* (). To compile the program, use GHC's built-in make facility, `ghc --make`, e.g.

```
sebastian@laptop $ ghc --make Echo.hs
[1 of 1] Compiling Main             ( Echo.hs, Echo.o )
Linking Echo ...
sebastian@laptop $ ./Echo hello world
hello world
```

```
WC(1)                    User Commands                   WC(1)

NAME
       wc - print newline, word, and byte counts for each file

SYNOPSIS
       wc [FILE]...

DESCRIPTION
       Print newline, word, and byte counts for each FILE, and
       a total line if more than one FILE is specified. A word
       is a non-zero-length sequence of  characters  delimited
       by white space.
```

Figure 1: An excerpt of the man page for `wc` (slightly simplified).

```haskell
main :: IO ()
main = do
  filePaths <- getArgs
  filesWithContent <- mapM (\f -> do c <- readFile f; return (f, c)) filePaths
  putStrLn $ format $ wc filesWithContent
format :: [(Int, Int, Int, String)] -> String
format ((t1, t2, t3, t4) : files) = (unlines $ map f files) ++ totals where
  totals = show t1 ++ " " ++ show t2 ++ " " ++ show t3 ++ " " ++ t4
  f (c1, c2, c3, p) = pad c1 l1 ++ " " ++ pad c2 l2 ++ " " ++ pad c3 l3 ++ " " ++ p
  l1 = length (show t1)
  l2 = length (show t2)
  l3 = length (show t3)
  pad n l = let sn = show n in (replicate (l - length sn) ' ') ++ sn
wc :: [(FilePath, String)] -> [(Int, Int, Int, String)]
wc [] = [(0, 0, 0, "total")]
wc ((path, content) : files) = updatedTotals : newEntry : rest where
  (oldTotals : rest) = wc files
  (c1, c2, c3) = count content
  newEntry = (c1, c2, c3, path)
  (t1, t2, t3, t4) = oldTotals
  updatedTotals = (t1 + c1, t2 + c2, t3 + c3, t4)
count :: String -> (Int, Int, Int)
count x = (length (lines x), length (words x), length x)
```

Exercise 8.2 (Skeleton: `Evaluator.hs`). Recall the expression datatype shown in the lectures. Extend the type and its evaluator by non-deterministic choice.

```
data Expr
  = Lit Integer        -- a literal
  | Expr :+: Expr      -- addition
  | Expr :*: Expr      -- multiplication
  | Div Expr Expr      -- integer division
  | Expr :?: Expr      -- non-deterministic choice
```

The expression *e1* :?: *e2* either evaluates to *e1* or to *e2*, non-deterministically. For example,

```
toss :: Expr
toss = Lit 0 :?: Lit 1
```

evaluates either to 0 or to 1. To illustrate, here are some example evaluations:

```
>>> evalN toss
[0,1]
>>> evalN (toss :+: Lit 2 :*: toss)
[0,2,1,3]
>>> evalN (toss :+: Lit 2 :*: (toss :+: Lit 2 :*: (toss :+: Lit 2 :*: toss)))
[0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15]
```

As you can see, the evaluator returns a list of all possible results. Implement this function

```
evalN :: Expr -> [Integer]
```

Haskell's list datatype is already an instance of *Functor*, *Applicative*, and *Monad*. So, all you have to do is to extend the interpreter in applicative style by adding one equation for :?: (you can copy the remaining equations from *evalA*).

```
evalN :: Expr -> [Integer]
evalN (Lit i)       = pure i
evalN (e1 :+: e2)   = pure (+) <*> evalN e1 <*> evalN e2
evalN (e1 :*: e2)   = pure (*) <*> evalN e1 <*> evalN e2
evalN (Div e1 e2)   = pure div <*> evalN e1 <*> evalN e2
evalN (e1 :?: e2)   = evalN e1 ++ evalN e2
```

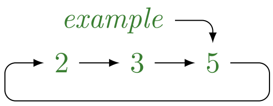
Exercise 8.3 (Skeleton: `Necklace.hs`). We introduce the following data type *Necklace* for non-empty cyclic linked lists:

```haskell
type NecklaceRef elem = IORef (Necklace elem)
data Necklace elem = Cons elem (NecklaceRef elem)
```

It is similar to the linked list introduced in the lectures, but there is no case for the empty list. Values of this data type are meant to be cyclic. For example, we can build:

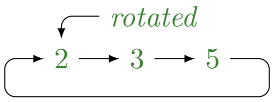
```haskell
example :: IO (NecklaceRef Integer)
example = mdo
  x <- newIORef (Cons 2 y)
  y <- newIORef (Cons 3 z)
  z <- newIORef (Cons 5 x)
  return z
```

We used the *mdo*-syntax[1] since we need mutual recursive assignments. With **do** instead of *mdo*, the first assignment would fail since y is not in scope. The result is the following cyclic construct:



The idea is to represent a list of elements by giving a reference to the *last* element. The example therefore represents the list $[2, 3, 5]$. This representation allows us to easily implement certain operations.

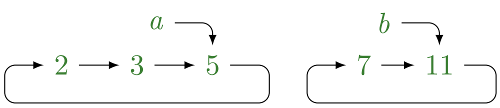
a) Implement a function $rotate :: NecklaceRef\ elem \to IO\ (NecklaceRef\ elem)$ that rotates the represented list. To do so, none of the references need to be changed, the result just points to another element in the necklace. The function therefore runs in constant time. With $rotated = example \gg= rotate$, we get:

   It represents the list $[3, 5, 2]$. The next rotation represents $[5, 2, 3]$ and the third one again the initial list $[2, 3, 5]$.

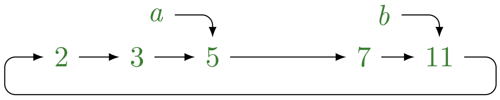
   ```haskell
   rotate :: NecklaceRef elem -> IO (NecklaceRef elem)
   rotate xs = do
     (Cons _ ys) <- readIORef xs
     return ys
   ```

b) Appending two necklaces is possible in constant time, too! Consider a and b as shown:

   To append a and b, we only need to update two references:

[1] https://wiki.haskell.org/MonadFix



Afterwards, b represents $[2,3,5] +\!\!+ [7,11]$ and a represents $[7,11] +\!\!+ [2,3,5]$. So we do not need to return a result, instead we change the provided arguments. Implement the function $append :: NecklaceRef\ elem \to NecklaceRef\ elem \to IO\ ()$.

```
swap :: IORef a -> IORef a -> IO ()
swap x y = do
  a <- readIORef x
  b <- readIORef y
  writeIORef x b
  writeIORef y a

append :: NecklaceRef elem -> NecklaceRef elem -> IO ()
append a b = do
  Cons _ anext <- readIORef a
  Cons _ bnext <- readIORef b
  swap anext bnext
```

Exercise 8.4. Obtain the free theorems for the following function types:

a) $a \to (a, a)$

$$
\begin{array}{ll}
& (f, f) \in a \to (a, a) \\
\Longleftrightarrow & \{ \text{ type as relation } \} \\
& \forall x\ y\ .\ (x, y) \in a \Rightarrow (f\ x, f\ y) \in (a, a) \\
\Longleftrightarrow & \{ \text{ relation of pairs } \} \\
& \forall x\ y\ .\ (x, y) \in a \Rightarrow (\mathit{fst}\ (f\ x), \mathit{fst}\ (f\ y)) \in a \wedge (\mathit{snd}\ (f\ x), \mathit{snd}\ (f\ y)) \in a \\
\Longrightarrow & \{ \text{ instantiate } a \text{ to a function } h \} \\
& \forall x\ y\ .\ h\ x = y \Rightarrow h\ (\mathit{fst}\ (f\ x)) = \mathit{fst}\ (f\ y) \wedge h\ (\mathit{snd}\ (f\ x)) = \mathit{snd}\ (f\ y) \\
\Longrightarrow & \{ \text{ choose } h = \mathit{const}\ y \} \\
& \forall x\ y\ .\ \mathit{const}\ y\ x = y \Rightarrow \mathit{const}\ y\ (\mathit{fst}\ (f\ x)) = \mathit{fst}\ (f\ y) \wedge \mathit{const}\ y\ (\mathit{snd}\ (f\ x)) = \mathit{snd}\ (f\ y) \\
\Longleftrightarrow & \{ \mathit{const}\ y\ \_ = y \} \\
& \forall y\ .\ y = y \Rightarrow y = \mathit{fst}\ (f\ y) \wedge y = \mathit{snd}\ (f\ y) \\
\Longleftrightarrow & \\
& \forall y\ .\ f\ y = (y, y)
\end{array}
$$

b) $(a, b) \to (b, a)$

$$
\begin{array}{ll}
& (f, f) \in (a, b) \to (b, a) \\
\Longleftrightarrow & \{ \text{ type as relation } \} \\
& \forall x\ y\ .\ (x, y) \in (a, b) \Rightarrow (f\ x, f\ y) \in (b, a) \\
\Longrightarrow & \{ \text{ restrict to pairs } \} \\
& \forall x1\ x2\ y1\ y2\ .\ ((x1, x2), (y1, y2)) \in (a, b) \Rightarrow (f\ (x1, x2), f\ (y1, y2)) \in (b, a) \\
\Longleftrightarrow & \{ \text{ relation of pairs } \} \\
& \forall x1\ x2\ y1\ y2\ .\ (x1, y1) \in a \wedge (x2, y2) \in b \Rightarrow \\
& \quad (\mathit{fst}\ (f\ (x1, x2)), \mathit{fst}\ (f\ (y1, y2))) \in b \wedge (\mathit{snd}\ (f\ (x1, x2)), \mathit{snd}\ (f\ (y1, y2))) \in a \\
\Longrightarrow & \{ \text{ instantiate } a \text{ to a function } h1 \text{ and } b \text{ to a function } h2 \} \\
& \forall x1\ x2\ y1\ y2\ .\ h1\ x1 = y1 \wedge h2\ x2 = y2 \Rightarrow \\
& \quad h2\ (\mathit{fst}\ (f\ (x1, x2))) = \mathit{fst}\ (f\ (y1, y2)) \wedge h1\ (\mathit{snd}\ (f\ (x1, x2))) = \mathit{snd}\ (f\ (y1, y2)) \\
\Longrightarrow & \{ \text{ choose } h1 = \mathit{const}\ y1 \text{ and } h2 = \mathit{const}\ y2 \} \\
& \forall x1\ x2\ y1\ y2\ .\ \mathit{const}\ y1\ x1 = y1 \wedge \mathit{const}\ y2\ x2 = y2 \Rightarrow \\
& \quad \mathit{const}\ y2\ (\mathit{fst}\ (f\ (x1, x2))) = \mathit{fst}\ (f\ (y1, y2)) \wedge \mathit{const}\ y1\ (\mathit{snd}\ (f\ (x1, x2))) = \mathit{snd}\ (f\ (y1, y2)) \\
\Longleftrightarrow & \{ \mathit{const}\ z\ \_ = z \} \\
& \forall y1\ y2\ .\ y1 = y1 \wedge y2 = y2 \Rightarrow y2 = \mathit{fst}\ (f\ (y1, y2)) \wedge y1 = \mathit{snd}\ (f\ (y1, y2)) \\
\Longleftrightarrow & \\
& \forall y1\ y2\ .\ f\ (y1, y2) = (y2, y1)
\end{array}
$$

c) $a \to a$ (can you show that this must be the identity function?)

$$
\begin{aligned}
& (f, f) \in a \to a \\
\iff & \quad \{ \text{ type as relation } \} \\
& \forall x\ y\ .\ (x, y) \in a \Rightarrow (f\ x, f\ y) \in a \\
\Longrightarrow & \quad \{ \text{ instantiate } a \text{ to a function } h \} \\
& \forall x\ y\ .\ h\ x = y \Rightarrow h\ (f\ x) = f\ y \\
\Longrightarrow & \quad \{ \text{ choose } h = \mathit{const}\ y \} \\
& \forall x\ y\ .\ \mathit{const}\ y\ x = y \Rightarrow \mathit{const}\ y\ (f\ x) = f\ y \\
\iff & \quad \{ \mathit{const}\ y\ \_ = y \} \\
& \forall y\ .\ y = y \Rightarrow y = f\ y \\
\iff & \\
& \forall y\ .\ y = f\ y \\
\iff & \quad \{ \text{ definition of } \mathit{id} \} \\
& f = \mathit{id}
\end{aligned}
$$

Note: The first few lines (those beginning with $\iff$) are fully automatic, they are the actual *free* part of the free theorem. The lines beginning with $\Longrightarrow$ describe a concious choice we made to derive a useful theorem.

Procedure:

1. Start with $(f, f) \in$ *type of your function*.
2. Use a suitable rule to interpret the type as a relation (slide 404).
3. Now you can choose what the type variables should be, often it is useful to interpret them as function (slide 407).
4. The theorem holds for *all* such functions. Either you can derive something useful where the function is left unspecified (often using map and list types), or you can make a choice what those functions should be (like we did here).
5. For the implication that is derived in the free theorem, either pick the function such that the left side trivially is true (as we did here), or assume that the left hand side is true and use that equation to transform something on the right hand side (as on slide 408).

Exercise 8.5 (Lecture Evaluation - Not graded)**.** Please contribute to the lecture evaluation. Your feedback helps us to improve this course. If you have a `@cs.uni-kl.de` email address, then you already have received an email with access details. If you do not have such an email account, please go to `https://vlu.informatik.uni-kl.de/teilnahme/` and enter your RHRK email address (`username@rhrk.uni-kl.de`). You then will receive an email with access details for the lecture evaluation system.

Access the system using the received URL and then select your lectures. Our lecture is *INF-36-51-V-6: "Funktionale Programmierung"*. Afterwards, you can start answering the questions and give us comments. The evaluation is of course anonymous.

Thanks for taking your time!