Functional Programming (NWI-IBC040)

Exam — Monday, 18 January 2021 – 8:30 - 11:30

Please read carefully before answering the questions:

- Check that the exercise set is complete: the exam consists of 5 questions.
- The 'cheat sheets' containing some prelude functions is available as a separate document.
- Read the questions carefully.
- Be concise and precise.
- This exam is closed book. You are not allowed to use lecture notes, personal notes, books, a notebook, or any other electronic devices, except for the computer needed to take the exam. It is not permitted to open GHCi or to search for solutions on the Internet.

1 Warm-up exercise (points: $3 \times 3 = 9$)

In this assignment you may only use basic/library functions and/or list comprehensions, but **not** recursion. However, you may use multiple equations if necessary. This applies to all three parts.

The function `intersperse :: a → [a] → [a]` takes an element and a list and 'intersperses' that element between (i.e put that element between) the elements of the list.

For example:

```
intersperse ',' "abcdef" = "a,b,c,d,e,f"
intersperse 0 [1..5] = [1,0,2,0,3,0,4,0,5]
intersperse "+" ["foo", "bar"] = ["foo","+","bar"]
intersperse 10 [1] = [1]
```

1.a) Give a definition of `intersperse`. It is of course also not allowed to directly call the function `intersperse` from the `List` library.

1.b) Define a function `allEqual :: Eq a ⇒ [a] → Bool` that checks if all elements of a list are equal.

1.c) Define the function `interspersed :: [a] → Bool` that checks whether the elements of a list are correctly separated by one and the same symbol. So, `interspersed xs == True` if and only if `xs = intersperse sep ys` for some `sep` and `ys`.

For example:

```
interspersed "a,b,c,d,e,f"      = True
interspersed [1,2,3]            = True
interspersed "a,b,c,d,ef"       = False
interspersed [1,0,2,0,3,0,4,0] = False
```

2 Functions and types (points: $4 \times 2 = 8$)

Define a *total* function for each of the following types, that is, provide a binding for each of the type signatures. You may assume that functions passed as arguments are total.

2.a) `f1 :: ((a → a) → a) → a`

2.b) `f2 :: Either a b → (a → b) → b`

2.c) `f3 :: Monad m ⇒ m a → m b → m (a,b)`

2.d) `f4 :: (a → c, b → c → d) → (a, b) → d`

3 Types (points: $4 \times 2 = 8$)

Give the *most general type* for the following functions.

3.a)
```
f5 f x y z = f (f x y) z
```

3.b)
```
f6 xs = reverse [(y,x) | (x,y) ← xs]
```

3.c)
```
f7 = foldMap (\x → [x + 10, x + 20])
```

3.d)
```
f8 f = do
  x ← f 0
  case x of
    Nothing → f 1
    y       → return y
```

4 Hash codes (points: 2 + 6 + 3 + 3 + 2 = 16)

A hash is a way to reduce a complex value to a single integer, with the property that two distinct values are very unlikely to have the same hash code. This can be useful to do quick comparisons and lookups.

Suppose that we have the following declarations for working with hash codes:

```
type Hash = ...
hashInt :: Int → Hash
hashCombine :: Hash → Hash → Hash
```

4.a) Define a class Hashable with a function hash, for types for which a hash code can be computed

4.b) Give instances of Hashable for Int, Either and pairs

4.c) Give a definition of the function

```
hashContainer :: (Foldable t, Hashable a) ⇒ t a → Hash
```

that computes the hash code of any foldable container

4.d) Pre-computed hash codes can be used to speed up comparisons.

Define a datatype `WithHash` that stores any value together with its hash code. Define a function `withHash` that converts a value into this datatype, also give a type signature. Give or derive an instance of `Eq` for `WithHash` that only looks at the values if the hash codes are equal.

5 Trees and Monads (points: 5 × 4 = 20)

A DNA sequence is composed of so-called base elements, which we can represent by the following datatype

```
data Base = A | C | G | T
```

Consider the following datatype of trees, containing values of type Base in the leaves.

```
data Tree  =  Leaf Base | Fork Tree Tree
```

Moreover, assume that the following datatype of binary digits is given.

```
data Bit = O | I
```

A tree can be represented as a sequence of bits. This is done by preorder traversal. Each internal node is rendered as a I and each leaf as a O. In the case of an internal node, the subsequent bits are representations of the two subtrees. For a leaf, the following two bits indicate the base value. Take, for example, the following trees.

```
t1 = Leaf A
t2 = Fork (Leaf C) (Leaf T)
t3 = Fork (Fork (Leaf C) (Leaf T))
          (Fork (Leaf A) (Fork (Fork (Leaf G) (Leaf A)) (Leaf G)))
```

These trees are converted into

```
[O,O,O]                                          -- t1
[I,O,O,I,O,I,I]                                  -- t2
[I,I,O,O,I,O,I,I,I,O,O,O,I,I,O,I,O,O,O,O,O,I,O]  -- t3
```

5.a) Write a function compressTree that encodes a given tree as a bit string.

For converting a bit string back to the tree we introduce a function inflate. The idea is to use a technique for this conversion that is similar to the way we construct parsers. The type of an 'inflator' is given by the following newtype declaration:

```
newtype Inflater a = IF { inflate :: [Bit] → Maybe (a, [Bit]) }
```

This declaration states that an inflater of type a is a function that takes a binary input string and produces either Nothing (in case of failure) or a pair comprising a result value of type a and the remainder of the input. In this assignment you may choose whether you use Monads or Applicatives.

5.b) Give appropriate instances of Inflater either for class Applicatives or for class Monad.

5.c) Also give an instance of class Alternative.

5.d) Introduce two primitive inflaters bit :: Bit → Inflater () and base :: Inflater Base. bit checks whether the input starts with the given bit, and base tries to recognize one of the base values.

5.e) Using these combinators and primitives, define the inflator inflateTree :: Inflater Tree that converts a sequence of bits into a tree.

6 Correctness (points: 4 + 4 + 10 = 18)

Foldr fusion

In this exercise we assume that the following definitions of `(++)`, `concat` and `reverse` in terms of `foldr` is given:

```
(++)    = flip (foldr (:))
concat  = foldr (++) []
reverse = foldr (\x r → r ++ [x]) []
```

The function `foldr` is defined as usual:

```
foldr op e []     = e
foldr op e (x:xs) = x ‘op‘ (foldr op e xs)
```

Recall the fusion law from the lectures:

$$\texttt{f} \circ \texttt{foldr g e} = \texttt{foldr h (f e)} \Longleftarrow \texttt{f (g x y)} = \texttt{h x (f y)}$$

6.a) Use the fusion law to show

$$\texttt{reverse} \circ \texttt{concat} = \texttt{foldr (flip (++)} \circ \texttt{reverse) []}.$$

If necessary you may use the following property of `reverse`

```
reverse (xs ++ ys) = reverse ys ++ reverse xs
```

Hint: It is crucial to determine carefully what f, g and h are from the FF-law.

6.b) Use the fusion law to show

$$\texttt{foldr k e (xs ++ ys)} = \texttt{foldr k (foldr k e ys) xs}$$

Hint: Use the definition of `(++)` to get the formula in the correct form. Determine what `f`, `g` and `h` are.

Induction

6.c) The function `foldl` is defined as:

```
foldl op e []     = e
foldl op e (x:xs) = foldl op (e ‘op‘ x) xs
```

Prove by induction that

$$\texttt{foldr f e (reverse xs)} = \texttt{foldl (flip f) e xs}.$$

Hint: Use the property proven in the previous part.