Functional Programming (NWI-IBC040)

Exam — Thursday, 20 January 2022 – 18:00 - 21:00

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.
- This exam has 6 questions for 52 points. The points for each question are shown in the margin.

Question 1: Warm-up .... (8 points)

We call a pair of numbers (a, b) a d-couple if a and b differ exactly by d. For instance, $(1, 4)$ is a 3-couple, and so is $(7, 4)$. The function `couples :: Int → [Int] → [(Int,Int)]` takes a (natural) number d and a list l of integers as arguments and returns a list of all d-couples (a, b) where a and b are consecutive elements of l. Examples

```
couples 2 [1,4,3,1,2,5,7,4] = [(3,1),(5,7)]
couples 3 [1,4,3,1,2,5,7,4] = [(1,4),(2,5),(7,4)]
couples 1 [1..5]            = [(1,2),(2,3),(3,4),(4,5)]
couples 7 [1..1000]         = []
```

(3) (a) Define `couples` using basic/library functions and/or list comprehensions, but **not** recursion.

Solution:

```
couples d xs = [(a,b) | (a,b) ← zip xs (tail xs), abs (a-b) == d]

couples₂ d = filter ((d ==) . abs . uncurry (-)) . pairs
  where pairs xs = zip xs (tail xs)
```

(3) (b) Define `couples` but this time using recursion, and not list comprehension or library functions. You may use multiple equations if necessary, but no helper functions (e.g. it is not allowed to implement the library functions used in the previous part yourself, and use the same solution).

Solution:

```
couples' d (a:l@(b:_))
  | a-b == d || b-a == d = (a,b) : couples' d l
  | otherwise            = couples' d l
couples' _ _             = []
```

(2) (c) Define a function `couples1to10 :: [Int] → [(Int,Int)]` that returns a list containing all d-couples for every $1 \le d \le 10$.

Solution:

```
couples1to10 ns = concatMap (flip couples ns) [1..10]
```

Question 2: Functions and types .... (8 points)

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 :: [Bool] → [Int]`

the output of `f₁` must depend on its input.

Solution: for example

```
f₁ = map (\x → if x then 1 else 0)

f₁' xs = [length xs]
```

(2) (b) `f₂ :: ((a → b) → c) → (a → d) → (d → b) → c`

Solution:

```
f₂ f g h = f (h . g)
```

(2) (c) `f₃ :: (Int,a) → Maybe (a → b) → Maybe b`

Your function must do something interesting, and not always return `Nothing`.

Solution:

```
f₃ (_,x) = fmap (\f → f x)

f₃' _     Nothing  = Nothing
f₃' (_,x) (Just f) = Just (f x)
```

(2) (d) `f₄ :: Monad f ⇒ f a → f b → f (a,b)`

Solution:

```
f₄ = liftM2 (,)

f₄' xs ys = (,) <$> xs <*> ys

f₄'' xs ys = do { x ← xs; y ← ys; return (x,y) }
```

Question 3: Type inference . . . . (8 points)

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

(2) (a)
```
g₁ f g x = f (g (f x))
```

Solution:

```
g₁ :: (a → b) → (b → a) → a → b
```

(2) (b)
```
g₂ (Just x) (Just y) = Left (x + y)
g₂ Nothing  v        = Right v
g₂ u        Nothing  = Right u
```

Solution:

```
g₂ :: Num a ⇒ Maybe a → Maybe a → Either a (Maybe a)
```

(2) (c)
```
g₃ xs = xs <*> [1..length xs]
```

Solution:

```
g₃ :: [Int → b] → [b]
```

(2) (d)
```
g₄ [] = do
      putStrLn "done"
g₄ (x:xs) = do
      print (fst x, snd x)
      putStrLn ":"
      g₄ xs
```

Solution:

```
g₄ :: (Show a, Show b) ⇒ [(a,b)] → IO ()
```

Question 4: Parsers, Foldables, and Traversables.....................(12 points)

In this question we work with representations of (Haskell-like) types. We use the following data structure for this:

```
data Type a = IntT | VarT a | ListT (Type a) | TupleT [Type a]
```

This should be interpreted as:

- The constant `IntT` represents the type of integers.
- The value `VarT vn` represents a type variable with name `vn`. Observe that `Type` is parametric in the type for those variable names.
- The value `ListT t` represents a list of values of type `t`,
- And the value `TupleT ts` represents a tuple of which each component has the corresponding type from `ts`. Hence, the length of `ts` also indicates the arity (i.e. number of arguments) of the tuple.

Some examples (on the left the representation, on the right the types as they are written in Haskell):

Value of `Type a`	Haskell equivalent
`IntT`	`Int`
`VarT "a"`	`a`
`ListT (VarT "a")`	`[a]`
`TupleT [ VarT "b", IntT, ListT IntT ]`	`(b,Int,[Int])`

First of all, we are going to write a parser that converts concrete syntax into the above representation. This concrete syntax is identical to the Haskell syntax and is given by the following grammar:

```
type = "Int"
     | identifier
     | '[' type ']'
     | '(' type {',' type}+ ')'
```

In this (EBNF-like) notation, `{ smth }+` means `smth` one or more times.

(5) (a) Turn this grammar into a parser using the *parser combinators*. Recall that the type of a parser is given by the following newtype declaration:

```
newtype Parser a = P { parse :: String → Maybe (a, String) }
```

The parser combinators are provided by the following instances (function bodies are omitted):

```
instance Functor Parser
instance Applicative Parser
instance Monad Parser
instance Alternative Parser
```

Besides these combinators you can use the following primitive parsers:

```
symbol     :: String → Parser String
identifier :: Parser String
```

The symbol parser can be used to recognize the given input string (with the string itself returned), whereas identifier is a parser that returns an identifier represented as a string.

Solution:

```
parseType  =  symbol "Int"  *> return IntT
          <|> VarT  <$> identifier
          <|> ListT <$> (symbol "[" *> parseType <* symbol "]")
          <|> TupleT <$> ((:) <$>
               (symbol "(" *> parseType) <*> some (symbol "," *> parseType) <*
symbol ")")
```

(3) (b) Define an instance of Foldable for type Type. Hint: it is easier to define foldMap than foldr.

Solution:

```
instance Foldable Type where
  foldMap f IntT         = mempty
  foldMap f (VarT v)     = f v
  foldMap f (ListT t)    = foldMap f t
  foldMap f (TupleT ts)  = foldMap (foldMap f) ts
```

(2) (c) Use foldr (or, if you prefer, foldMap) from class Foldable to define a function

```
vars :: (Eq a) ⇒ Type a → [a]
```

that returns the names of all variables appearing in the given type. The result list should not contain the same name more than once.

Solution:

```
vars = foldr (\v vs → [v] ‘union‘ vs) []
```

Question 5: Type classes .... (8 points)

(2) (a) Define a type class Enumerable a, that has one member, called values, that enumerates all values of type a. If necessary, you can assume that this class is only instantiated with types that are finite.

Solution:

```
class Enumerable a where
  values :: [a]
```

(2) (b) Give a suitable instance of Enumerable for type Bool.

Solution:

```
instance Enumerable Bool where
  values = [True, False]
```

(2) (c) Give a suitable instance of Enumerable for pairs.

Solution:

```
instance (Enumerable a, Enumerable b) ⇒ Enumerable (a,b) where
  values = [ (x,y) | x ← values, y ← values ]
```

(2) (d) Give a suitable instance of Enumerable for the type Either.

Solution:

```
instance (Enumerable a, Enumerable b) ⇒ Enumerable (Either a b) where
  values = map Left values ++ map Right values
```

Question 6: Correctness . . . . (8 points)

Consider the following functions, the first two of which are from Haskell's standard library:

```
iterate :: (a → a) → a → [a]
iterate f x = x : iterate f (f x)

(!!) :: [a] → Int → a
(x:_)  !! 0 = x
(_:xs) !! n = xs !! (n-1)

iterateN :: Int → (a → a) → a → a
iterateN 0 f x = x
iterateN n f x = iterateN (n - 1) f (f x)
```

(8) (a) Prove that for all `n ≥ 0`, `f` and `x`: `iterate f x !! n = iterateN n f x`

Solution: Proof by induction on n.

```
case n = 0
     iterate f x !! 0
   = { def iterate }
     (x : iterate f (f x)) !! 0
   = { def !! }
     x
   = { def iterateN }
     iterateN 0 f x

case n = m+1
   IH: for all f and x: iterate f x !! m = iterateN m f x
   To prove: for all f and x: iterate f x !! (m+1) = iterateN (m+1) f x
Proof:
     iterate f x !! (m+1)
   = { def iterate }
     (x : iterate f (f x)) !! (m+1)
   = { def !! }
     iterate f (f x) !! m
   = { IH }
     iterateN m f (f x)
   = { def iterateN }
     iterateN (m+1) f x
```