Functional Programming – Final exam – Thursday 9/11/2017

Name: MODEL SOLUTION
Student number:

Q:	1	2	3	4	5	Total
P:	20	30	20	15	15	100
S:						

Before you begin:

- Do not forget to write down your name and student number above.
- If necessary, explain your answers *in English*.
- Use *only* the empty boxes under the questions to write your answer and explanations in.
- At the end of the exam, only hand in the filled-in exam paper. Use the blank paper provided with this exam only as scratch paper (kladpapier).
- Answers will not only be judged for correctness, but also for clarity and conciseness.

In any of the answers below you may (but do not have to) use the following well-known Haskell functions and operators, unless stated otherwise: `id`, `(.)`, `const`, `flip`, `head`, `tail`, `(++)`, `concat`, `foldr` (and its variants), `map`, `filter`, `sum`, `all`, `any`, `not`, `(&&)`, `(||)`, `zip`, `reverse`, and all the members of the type classes `Show`, `Eq`, `Ord`, `Enum`, `Num`, `Functor`, `Applicative`, and `Monad`.

1. The function `zip` generates a list of pairs from a pair of lists. Note that if one list is shorter than the one, the final elements of the longest are thrown away.

```
zip []     _      = []
zip _      []     = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

We want to abstract this notion into a type class `Zippable`,

```
class Zippable f where
  zip :: f a -> f b -> f (a, b)
```

(a) (8 points) Consider the following data type for binary trees,
`data Tree a = Leaf | Node a (Tree a) (Tree a)`
Write its `Zippable` instance. As in the case of lists, you might need to throw away information from some subtrees to implement `zip`.

```
instance Zippable Tree where
  zip Leaf _ = Leaf
  zip _ Leaf = Leaf
  zip (Node x1 l1 r1) (Node x2 l2 r2)
    = Node (x1, x2) (zip l1 l2) (zip r1 r2)
```

(b) (7 points) Write the following generic function `zipWith`. This function applies a mapping over the elements of two `Zippable` containers, whenever the containers are also `Functors`.

```
zipWith :: (Functor f, Zippable f) => (a -> b -> c) -> f a -> f b -> f c
```

Hint: use the function `uncurry :: (a -> b -> c) -> (a, b) -> c`.

```
zipWith f xs ys = fmap (uncurry f) (zip xs ys)
--or
...             = uncurry f <$> zip xs ys
```

(c) (5 points) There is another way to implement a `zip` function for a list, namely,

```
instance Zippable [] where
  zip xs ys = [(x, y) | x <- xs, y <- ys]
```

Does this definition coincide with one given at the beginning of the exercise? If not, give an example in which each definition gives a different outcome.

No, the second def. does a cartesian product

Counterexample: `zip ['a'] [1,2]`

- 1st def: `[('a', 1)]`
- 2nd def: `[('a',1), ('a',2)]`

2. Remember that an operation `(<>) :: m -> m -> m` and an element `e :: m` form a commutative monoid over type `m` if the following laws are satisfied,

```
(1) x <> (y <> z) = (x <> y) <> z
(2) x <> e = x
(3) e <> x = x
(4) x <> y = y <> x
```

Given the following definitions for `foldr`, `reverse`, `(.)`, and `filter`,

```
(a) foldr (<>) e []     = e
(b) foldr (<>) e (x:xs) = x <> foldr (<>) e xs

(c) reverse []     = []
(d) reverse (x:xs) = reverse xs ++ [x]

(e) (f . g) x = f (g x)

(f) filter p []     = []
(g) filter p (x:xs) | p x       = x : filter p xs
(h)                 | otherwise = filter p xs
```

(a) (5 points) Using equational reasoning, prove that the following holds if `(<>)` and `e` form a commutative monoid.

`foldr (<>) e [x] = x`

```
  foldr (<>) e [x]
= (syntax)
  foldr (<>) e (x : [])
= (b)
  x <> foldr (<>) e []
= (a)
  x <> e
= (4)
  x
```

(b) (15 points) Prove by induction that the following holds if `(<>)` and `e` form a commutative monoid,

`foldr (<>) e = foldr (<>) e . reverse`

You are allowed to use the following lemma in your proof:

`foldr (<>) e (xs ++ ys) = (foldr (<>) e xs) <> (foldr (<>) e ys)`

Clearly state the cases you consider, the induction hypothesis, and justify each equality.

By extensionality, we need to prove

foldr (<>) e xs = foldr (<>) e (reverse xs)

By induction over xs:

Case []

```
foldr (<>) e []
```

```
foldr (<>) e (reverse [])
= (c)
  foldr (<>) e []
```

Case (x:xs)

IH: foldr (<>) e xs = foldr (<>) e (reverse xs)

```
foldr (<>) e (x : xs)
= (b)
  x <> foldr (<>) e xs
= (4)
  foldr (<>) e xs <> x
```

```
foldr (<>) e (reverse (x : xs))
= (d)
  foldr (<>) e
    (reverse xs ++ [x])
```

Z.O.Z.

= (lemma with reverse xs, [x])
  foldr (<>) e (reverse xs)
    <> foldr (<>) e [x]
= (prev. exercise)
  foldr (<>) e (reverse xs) <> x
= (IH)
  foldr (<>) e xs <> x

(c) (10 points) Prove that the following holds for any predicate `p :: a -> Bool`,
`length (filter p xs) + length (filter (not . p) xs) = length xs`
Use induction. State and prove here the [] case.

Case []

We need to prove
length (filter p []) + length (filter (not. p) []) = length []

We know that filter f [] = []
length [] = 0

So the equality can be rewritten as 0 + 0 = 0

State the induction hypothesis and prove here the `(x:xs)` case. You need to distinguish two cases, depending on whether the predicate p holds for the element x or not.

Case (x:xs)

IH: length (filter p (x:xs)) + length (filter (not. p) (x:xs))
= length (x:xs)
= 1 + length xs

For the left-hand side we distinguish two cases:

* (p x) holds ⇔ (not.p) x does not hold

length (filter p (x:xs)) + length (filter (not.p) (x:xs))

= (def of length)

length (x : filter p xs) + length (filter (not.p) xs)

= 1 + length (filter p xs) + length (filter (not.p) xs)

* (p x) does not hold ⇔ (not.p) x holds

length (filter p (x:xs)) + length (filter (not.p) (x:xs))

= (def of length)

length (filter p xs) + length (x: filter (not.p) xs))

= length (filter p xs) + 1 + length (filter (not.p) xs)

In both cases, we apply IH and get

1 + length xs

= length (x:xs)

3. Consider the following data type of simple arithmetic expressions,

```
data Expr = Literal Integer | Add Expr Expr | Mult Expr Expr
```

which comes with two operations to evaluate and optimize an expression,

```
eval :: Expr -> Integer
opt  :: Expr -> Expr
```

(a) (6 points) Give definitions for the following QuickCheck properties:

- Adding zero to an expression evaluates to the same result.
- Multiplication is commutative.

```
addZero e = eval e == eval (Add e (Literal 0))

comm x y = eval (Mult x y) == eval (Mult y x)
```

(b) (4 points) Write a QuickCheck property `checkOptimizer` which checks that the evaluation of an expression `e` gives the same result after optimization.

```
checkOptimizer e
  = eval e == eval (opt e)
```

(c) (5 points) Write the `Arbitrary` instance for `Expr`.

```
class Arbitrary a where
  arbitrary :: Gen a
```

Hint: below you can find some of the primitives of QuickCheck random generation.

```
choose    :: Random a => (a, a) -> Gen a
frequency :: [(Int, Gen a)] -> Gen a
elements  :: [a] -> Gen a
```

```
instance Arbitrary Expr where
  arbitrary = do n <- arbitrary
                 e1 <- arbitrary
                 e2 <- arbitrary
                 elements [ Literal n,
                          , Add e1 e2, Mult e1 e2]

  -- other option
  arbitrary = frequency
                [(3, Literal <$> arbitrary)
                , (1, Add <$> arbitrary <*> arbitrary)
                , (1, Mult <$> arbitrary <*> arbitrary)]
```

(d) (5 points) In order to check that our optimizer works correctly, we want to write a property which are only executed if the random expression given by QuickCheck contains a subexpression of the form `Mult (Literal 0) e` or `Mult e (Literal 0)`. We write the following code:

```
checkZeroMult e = hasZeroSubexpression e ==> checkOptimizer e
```

What is the problem with this definition? What could you do to solve the problem?

Problem: low probability of randomly generating an expression which satisfies *hasZeroSubexpressions*

Solution: write a custom generator

4. The `Result` data type is a close relative of `Maybe`. A value of type `Result e r` may describe a succesful computation, or a failure along with a description of the problem.

```
data Result e r = Fail e | Ok r
```

This type is a functor, as witnessed by the following declaration,

```
instance Functor (Result e) where
  fmap f (Fail e) = Fail e
  fmap f (Ok   r) = Ok   (f r)
```

Note that in the case of a failure we do not apply the function to the inner result. Instead, we keep the description of the problem untouched.

(a) (10 points) The type `Result e` is also a monad. Complete the corresponding instance declaration.

```
instance Monad (Result e) where
  return :: a -> Result e a
  return = ...
  (>>=) :: Result e a -> (a -> Result e b) -> Result e b
  x >>= f = ...
```

return = Ok

Ok x >>= f = f x

Fail e >>= _ = Fail e

(b) (5 points) Sinterklaas has to visit every town in the Netherlands to give presents. Unfortunately, both travelling from town to town and giving the presents in each town might go wrong, which we model by a couple of functions,

```
travel :: Town -> Town -> Result String ()
give   :: Town -> Result String ()
```

Using those functions, we can write a function which, given a list of the towns in the order in which they have to be visited, executes the whole Sinterklaas tour.

```
tour :: [Town] -> Result String ()
tour []       = Fail "No towns to visit!"
tour [x]      = give x
tour (x:y:zs) = case give x of
                  Fail e1 -> Fail e1
                  Ok   _  -> case travel x y of
                               Fail e2 -> Fail e2
                               Ok   _  -> tour (y:zs)
```

Rewrite the function `tour` using `do` notation.

```
tour [] = fail "..."
tour [x] = give x
tour (x: y: zs) = do give x
                     travel x y
                     tour (y: zs)
```

5. *Multiple choice questions.* Choose *one* answer.

(a) (5 points) Which is the result of `0 'seq' (\x -> undefined)`?

A. This expression is not well-typed.
B. `0`.
C. `undefined`.
(D.) `\x -> undefined`.

(b) (5 points) Given the following expressions:

1. `\f -> (f True, f 'a')`
2. `\f -> f (True, 'a')`

A. None of them is well-typed.
B. (1) is well-typed and (2) is not well-typed.
(C.) (1) is not well-typed and (2) is well-typed. §
D. Both of them are well-typed.

(c) (5 points) Which of the following is true?

A. You can always replace `return (return x)` by `return x`.
(B.) You can write an expression of type `IO (IO Int)`. *
C. Every functor is also a monad.
D. Evaluation in Haskell occurs eagerly.

* return (return 3)

§ In (1), the type of f would need to be Bool → α and Char → β at the same time

The type of (2) is ((Bool, Char) → a) → a