

EXAM FUNCTIONAL PROGRAMMING

Tuesday the 1st of October 2016, 08.30 h. - 10.30 h.

Name:

Student number:

Before you begin: Do not forget to write down your name and student number above. If necessary, explain your answers (in English or Dutch). For multiple choice questions, clearly circle what you think is the (one and only) best answer. Use the empty boxes under the other questions to write your answer and explanations in. Use the blank paper provided with this exam only as scratch paper (kladpapier). At the end of the exam, only hand in the filled-in exam paper. Answers will not only be judged for correctness, but also for clarity and conciseness. A total of 100 points can be obtained. Good luck!

In any of your answers below you may (but do not have to) use the following well-known Haskell functions/operators, unless stated otherwise: *replicate*, *id*, *concat*, *foldr* (and variants), *map*, *filter*, *const*, *all*, *any*, *flip*, *fst*, *snd*, *not*, *(.)*, *elem*, *take*, *drop*, *takeWhile*, *dropWhile*, *head*, *tail*, *zip*, *reverse*, *(++)*, *lookup*, *max*, *min* and all members of the type classes *Eq*, *Num*, *Ord*, *Show* and *Read*.

1. LISTS

- (i) Define a function $voww :: [a] \rightarrow [(a, a)]$ that takes a list xs , and pairs the first with the last, the second with the one-but-last, and so on. If the list has odd length, the middle element is paired with a copy of itself. So $voww [1, 2, 3, 4, 5] = [(1, 5), (2, 4), (3, 3)]$ and $voww [1, 2, 3, 4, 5, 6] = [(1, 6), (2, 5), (3, 4)]$. Implement this function without using recursion yourself, and without list comprehensions.

.../8

- (ii) Define a function ulh that takes a list xs , and pairs each element with all other elements in xs . $ulh [1, 2, 3, 2] = [(1, 2), (1, 3), (1, 2), (2, 1), (2, 3), (2, 2), (3, 1), (3, 2), (3, 2), (2, 1), (2, 2), (2, 3)]$, in that order. Do this using recursion and without list comprehensions. Moreover, of the above listed functions you may only use $(++)$. You may define your own local definitions.

.../9

- (iii) Would (ii) be easy to do with a list comprehension? If so, explain how; if not, explain what makes it hard.

.../4

2. DATATYPES Consider the following simplified datatype for representing boolean expressions (propositions), where variable names consist of a single character:

```
data Prop = Cons Bool
          | Vari Char
          | Not Prop
          | Prop :/\: Prop
```

Here, the constructor `:/\:` represents conjunction \wedge , and the constructor `Not` represents the negation symbol \neg .

- (i) Give the value of type `Prop` that represents the proposition $(\neg v \wedge w) \wedge \mathbf{tt}$ where v and w represent variables, and `tt` represents the value true.

.../6

- (ii) Write an evaluator `eval :: (Char -> Bool) -> Prop -> Bool` that takes a function that maps variables to booleans, and a proposition, and returns the boolean value of that proposition.

.../10

- (iii) For all propositions p , $\neg(\neg p) = p$. Write a simplifier $simpl :: Prop \rightarrow Prop$ that uses this (and only this) equality as much as it can to simplify boolean propositions. For example, $simpl\ prop = Cons\ True\ :\backslash:\ Not\ (Vari\ 'v')$ where $prop = Not\ (Not\ (Not\ (Not\ (Cons\ True))\ :\backslash:\ Not\ (Vari\ 'v')))$.

.../8

- (iv) We also have some equalities for conjunction (\wedge): $p \wedge \mathbf{ff} = \mathbf{ff} = \mathbf{ff} \wedge p$ and $p \wedge \mathbf{tt} = p = true \wedge p$. Now, $simpl\ prop = Not\ (Vari\ 'v')$ for the $prop$ given in (iii). Extend $simpl$ to apply also these optimisations.

.../6

3. SUBLISTS

In this question we deal with a function $subs :: [a] \rightarrow [[a]]$ which returns all the sublists of the argument list (i.e., all the lists that result by deleting elements from the argument list in any possible way).

- (i) How many sublists does $[1,2,3,4]$ have?

| |
|-------|
| .../4 |
|-------|

- (ii) Explain how you can compute $subs (x:xs)$ from $subs xs$ (for example by using concrete values for x and xs).

| |
|-------|
| .../4 |
|-------|

- (iii) Now, write the function $subs :: [a] \rightarrow [[a]]$ exploiting sharing where you can.

| |
|-------|
| .../6 |
|-------|

4. MULTIPLE CHOICE .../20

The following multiple choice questions are each worth 5 points.

- (i) I For a left-associative operator \oplus , the expression $a \oplus b \oplus c$ should be interpreted as $a \oplus (b \oplus c)$.
 II When we say `infix :<>` in Haskell, we mean that the operator `:<>` is associative, so that we can write expressions like $a :<> b :<> c$.
- Both I and II are true
 - Only I is true
 - Only II is true
 - Both I and II are false
- (ii) I `[const '2', (flip const) "2"]`, where `flip` flips the first two arguments of a function, is well-typed.
 II `\ x -> [(flip . flip) x, id]` is well-typed.
- Both I and II are true
 - Only I is true
 - Only II is true
 - Both I and II are false
- (iii) What is the type of `map ('}') . map reverse . map ('{')`?
- `[[a]] -> [[a]]`
 - `[[String]] -> [[String]]`
 - `[[Char]] -> [[Char]]`
 - The expression is type incorrect.
- (iv) The function `intersperse :: a -> [a] -> [a]` puts its first argument between all the elements of a non-empty list. Thus `intersperse 'a' "xyz"` results in `"xayaz"`. Which definition is correct, assuming the argument `as` is not empty?
- `intersperse a as = foldr (\ e r -> (e : a : r)) [] as`
 - `intersperse a as = foldl (\ r e -> (a : e : r)) [] as`
 - `intersperse a as = (tail . concat . map (\ x -> [a, x])) as`
 - `intersperse a as = tail [(a : e) | e <- as]`

5. TYPE INFERENCE

Determine the type of *foldr concatMap*, where $\text{concatMap} :: (a \rightarrow [b]) \rightarrow [a] \rightarrow [b]$. You should not just write down the type below, but also explain how you arrived at that type (for example, in the way that this is done in the lecture notes of this course).

.../15