Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik
Datenbanksysteme · Prof. Dr. Grust

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Functional Programming

SS 2017
Benjamin Dietrich

Assignment #2

Submission Deadline: Thu, 11.5.2017

Exercise 1: Patterns (2 Points)

Simplify the following patterns:

1. `f x | x == 42 = ...`
2. `g xs | not $ null xs = ...`
3. `h (_, _) = ...`
4. `i (a,b,c,xs) | a == b && c == 42 && length xs == 2 && head xs == 'a' = ...`

Exercise 2: List Processing (12 Points)

This exercise is concerned with list processing in Haskell, both via explicit recursion and using functions from the prelude and `Data.List`.

You can import the module `Data.List` via

```
import Data.List
```

at the top of your Haskell source file or in GHCi. It contains a large number of functions on lists. You may freely use functions from this module unless we explicitly state differently.

API documentation for the prelude, `Data.List` and other modules included in the Haskell Platform can be found online at http://hackage.haskell.org/package/base-4.9.1.0/docs/Data-List.html, Hoogle (https://www.haskell.org/hoogle/) or in your local Haskell Platform documentation.

1. Function `map :: (a -> b) -> [a] -> [b]` that applies a function to every element of a list should be well-known. Please write a similar function `mapEveryOther` that applies it's functional argument only to every second element of the input list and leaves the other elements as-is.

   Example: `mapEveryOther (+ 42) [1,2,3,4]` ≡ `[43,2,45,4]`

   Before you write down the definition, write down the function's polymorphic type. Compare the types of `map` and `mapEveryOther` and explain the difference.

2. The following function performs some computation on lists.

```
questionable :: (a -> Bool) -> (a -> b) -> [a] -> [b]
questionable p f as =
    if null as
    then []
    else if p (head as)
          then questionable p f (tail as)
          else dubious f as

  where
    dubious g xs =
        if null xs
        then []
        else g (head xs) : dubious g (tail xs)
```

Although this is is legal Haskell code, it's clearly hard to read and understand.

(a) Write an equivalent function `acceptable` that still uses explicit recursion but utilizes multiple equations with pattern matching and guards instead of conditional expressions and the list deconstructors `head` and `tail`.

(b) Write an equivalent function `reasonable` that does not use explicit recursion and is defined exclusively using list-processing functions from the prelude and `Data.List`.

Hint: Use the following search string on Hoogle to narrow down the number of list functions that might be used to implement one aspect of `questionable`.

```
+Data.List (a -> Bool) -> c
```

3. Write a function that moves a window of a given size n over a list. A list element's window contains the element itself as well as the $n-1$ preceding elements. See the examples below for clarification. Every element of the input list is mapped to the value obtained by applying a function to the element's window. Elements for which the window size is smaller than specified are mapped to a default value.

The function has the following type:

```
type Window a = [a]

slidingWindow :: Int -> (Window a -> b) -> b -> [a] -> [b]
```

Consider the following examples:

```
slidingWindow 3 id [] [1..5] ≡ [[],[],[1,2,3],[2,3,4],[3,4,5]]
slidingWindow 3 sum 0 [1,2,3,4,5,6,7] ≡ [0,0,6,9,12,15,18]
slidingWindow 4 (!! 2) 'x' "abcdef" ≡ "xxxcde"
```

Exercise 3: Fold **(6 Points)**

Formulate the following functions without using explicit recursion. Instead, make use of the prelude function

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```
[1]

Applying the function (`foldr f z l`), we can right-associatively fold a list (`l :: [a]`) using a binary operator (`f :: (a -> b -> b)`). This is, to reduce the list, starting from a initial value `z` of type `b` (typically the operator's identity), applying the operator element by element from right to left:

```
foldr f z [x1, x2, ..., xn] ≡ x1 `f` (x2 `f` ... (xn `f` z)...)
```

Take for example the `sum'` function presented in the lectures. It can be rewritten using a fold:

```
sum' :: [Integer] -> Integer
sum' = foldr (+) 0
```

Applied to a list `[4,2,6]` the sum it is evaluated as follows:

```
sum' [4,2,6] ≡ foldr (+) 0 [4,2,6] ≡ 4 + (2 + (6 + 0)) ≡ 12
```

Note: Obviously you must not use specialized prelude or module functions (e.g. `length`, `intercalate`, etc.) to solve the following problems.

1. `length' :: [a] -> Integer` – to count the size of a list.

   Hint: You may use function `const :: a -> b -> a` which returns its first argument and ignores a second one.

2. `commaSep :: [String] -> String` – to concatenate a list of strings, separated by commas (`','`).

   Example: `commaSep ["Hello","World"] ≡ "Hello,World"`

3. Rewrite the following function, using `foldr` instead of explicit recursion. The function removes duplicate elements from a sorted list:

```
removeDups :: [Integer] -> [Integer]
removeDups []                    = []
removeDups [x]                   = [x]
removeDups  (x:y:ys) | x == y    = removeDups (y:ys)
                     | otherwise = x:removeDups (y:ys)
```

[1]Actually the prelude function `foldr` has a more general type. However, applied to lists its type can be described in this concrete version.