Assignment 2

Advanced Functional Programming, 2019
(Avancerad funktionell programmering, 2019)

due 9 December 2019, 23:59

1 Stack Permutations (perm.hs & perm.pdf, 3 + 1 points)

Solve the stack permutations problem from Assignment 1, but this time in Haskell.

Task

Write a Haskell module (`perm.hs`) that returns `True` if the output list is a stack permutation of the input list. Your module should export one function, called `perm`:

```
perm :: [Int] -> [Int] -> Bool
```

Much like in Erlang, to get the maximum points for this part (**3 points**) your solution needs to be efficient: it should return an answer in one second or less for inputs with up to 1 million elements.

Your solution should also include QuickCheck-based tests, for appropriately chosen parts of the implementation. Include a report (`perm.pdf`) detailing your algorithm and the properties you have tested for. If your algorithm is the same as in the Erlang part, it is fine to use the same text, but you should also property-test for at least as many properties that you tested your Erlang program. (**1 point**)

Input / Output

The function takes two lists filled with numbers of type `Int` (not `Integer`). The lists follow the same specifications as in Assignment 1. The function should return a Haskell boolean (type `Bool`) value indicating if the output queue is a stack permutation of the input queue or not. We show two sample calls for the function:

```
Prelude> :load Perm
[1 of 1] Compiling Perm             ( Perm.hs, interpreted )
Ok, one module loaded.
*Perm> :t perm
perm :: [Int] -> [Int] -> Bool
*Perm> perm [1, 2, 3] [3, 2, 1]
True
*Perm> perm [1, 2, 3] [2, 3, 1]
False
```

2 Graph Coloring (`kcolor.hs` & `kcolor.pdf`, 4 + 2 points)

Implement a functional graph data structure in Haskell and use it to **lazily** solve the graph k-coloring problem **using the brute force algorithm**.

Task

In Assignment 1, we asked you to solve the graph k-coloring algorithm in Erlang. For the Haskell version, we want you to utilize two of Haskell's defining features: Monads and laziness. Specifically:

- **Create a Graph data type in Haskell** and implement appropriate functions. See Assignment 1 for the details. **(1 point)**
- **Create appropriate QuickCheck-based tests** for your Graph implementation. Explain in the report (`kcolor.pdf`) what properties you tested and how. Note that you will probably need to find some way of generating random Graph instances. **(1 point)**
- **Implement the brute-force k-coloring algorithm** on your graph **(1 point)** utilizing laziness **(1 point)** and monads **(1 points)**. Specifically, you should find a way of creating all possible combinations of k-colors lazily and then find the fist correct combination taking advantage of the monadic properties of `Maybe`. You can read more about the `Maybe` monad here: http://learnyouahaskell.com/a-fistful-of-monads. Explain in the report how you used laziness and monads in this assignment. **(3 points)**
- **Test your k-coloring algorithm** using QuickCheck. Write in the report how you tested and it and why you did it this way. **(1 point)**

Note that, much like in the Erlang assignment, the report is necessary for getting any points for the Graph implementation, the testing, and the laziness/monads part.

Input / Output

The input will have the same format as in the Erlang assignment, but this time using Haskell's syntax for tuples. A list of nodes and their adjacent nodes will be given, followed by the required number of colors. The output should also be the same as the Erlang assignment, returning `Just` a list of Haskell tuples if a coloring has been found, or `Nothing` otherwise.

```
*Kcolor> kcolor [(1, [2, 3]), (2, [1, 3]), (3, [1, 2])] 3
Just [(1, 'a'), (2, 'b'), (3, 'c')]
*Kcolor> kcolor [(1, [2, 3]), (2, [1, 3]), (3, [1, 2])] 2
Nothing
```

3 Fringes and Fibonacci Trees (`fringe.hs` & `fringe.pdf`, 4 points)

In the lectures, we mostly saw how to use GHCi, the interactive environment of the GHC distribution. However, GHC also comes with a compiler that produces executable files. One just needs to define a `main` function with type `IO ()`, which will be the entry point of the program as shown below.

```
$ cat hello.hs
main :: IO ()
main = putStrLn "Hello, World!"
$ ghc -O hello.hs
[1 of 1] Compiling Main             ( hello.hs, hello.o ) [Optimisation flags changed]
Linking hello ...
$ ./hello
Hello, World!
```

For this exercise, you need to hand in a file `fringe.hs` that contains the Haskell code for questions **(a)**–**(j)** below that ask for such code. Your report should contain text-based answers to the remaining questions. Your file should be compilable with the command `ghc -o fringe fringe.hs` and should run according to what is asked in question **(i)** below.

Consider a data structure `Tree a` that represents binary trees whose interior nodes contain information of type `a`.

```
data Tree a = E | T a (Tree a) (Tree a)
  deriving (Show, Read)
```

For example, the tree of Figure 1 is represented as:

```
T a (T b (T d E E) (T e (T i E E) (T j E E)))
    (T c (T f (T k E E) (T l E E)) (T g E E))
```

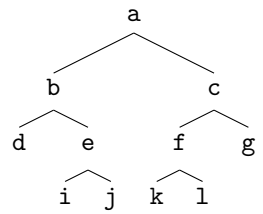

Figure 1: Example tree.

(a) Write a function `mirror :: Tree a -> Tree a` that constructs the *mirror* of such a tree (i.e., a tree whose left children have all been transposed with the right ones).

In such a tree, *leaves* are all nodes of the form `T x E E`. The *fringe* of a tree is defined as the list of all leaves of a tree as produced by an infix traversal. For example, the fringe of the tree of Figure 1 is the list `[d, i, j, k, l, g]`.

(b) Write the function `fringe_naive :: Tree a -> [a]` that finds the fringe of a tree with the "most obvious" recursive algorithm you can think of. What is the complexity of your function (measured in terms of n, the number of leaves of a tree)?

(c) Write a better function `fringe :: Tree a -> [a]` that finds the fringe of a tree in a better way. What is the complexity of your function?

(d) Write a function `same_fringe :: Eq a => Tree a -> Tree a -> Bool` that returns `True` if the fringes of two trees are equal. What is the complexity of `same_fringe`? How does it behave if the two fringes differ?

(e) If Haskell were an *eager* language instead, how would your answer to the previous question change? How easy is to write an equally efficient `same_fringe` function in e.g., Erlang or in some other eager functional language? Write one or sketch its implementation in sufficient detail.

For every natural number n, we define the Fibonacci tree T_n as follows. T_n has as root the n-th Fibonacci number F_n, as defined in e.g., http://en.wikipedia.org/wiki/Fibonacci_number. Trees T_0 and T_1 are leaves. For every n, the tree T_{n+2} has as its left child the tree T_{n+1} and as its right child the tree T_n.

For example, the Fibonacci tree T_5 is shown in Figure 2.

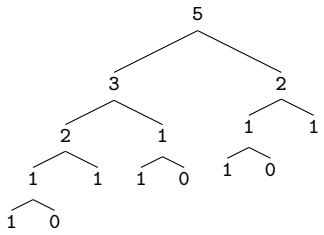

Figure 2: The tree T_5.

(f) Write a function `fibtree_naive :: Int -> Tree Int` that constructs T_n in the "most obvious" recursive way. How much space does tree T_n occupy in memory?

(g) Write a better function `fibtree :: Int -> Tree Int` that constructs T_n in a way such that its memory consumption is $O(n)$.

(h) How much memory space is occupied by the tree that gets produced by the evaluation of expression `mirror (fibtree n)`?

(i) Submit a program that combines all the above. Your program should ask the user for a natural number n and print two lines. The first one should be "`yes`" if the fringe of T_n is equal to that of its mirror; otherwise, it should be "`no`". The second line should be "`yes`" if the fringe of T_n is equal to the fringe of T_{n+1}; otherwise "`no`".

(j) Try your program for successively increasing values of n. Why, as n increases, the first line is shown immediately, but the second line is delayed more and more? Is this in accordance with the answers you gave in the previous questions?

You may find the following skeleton of `fringe.hs` handy:

```
import System.IO

... -- Your code here

main = do  hSetBuffering stdin  NoBuffering
           hSetBuffering stdout NoBuffering
           putStr "Give n: "
           s       <-  getLine
           let n   =   read s
           let t1  =   fibtree n
           let t2  =   mirror t1
           putStr (if same_fringe t1 t2 then "yes\n" else "no\n")
           let t3  =   fibtree (n+1)
           putStr (if same_fringe t1 t3 then "yes\n" else "no\n")
```

4 Type classes (`showme.hs`, 2 points)

The Glorious Haskell Compiler, GHC, can employ the C++ preprocessor if given the command-line argument `-cpp`. This allows the use of preprocessor commands like `#ifdef` and `#include`, as shown below:

vector.hs

```
module Vector where

type Vector   = [Integer]
data Expr     = V Vector
              | VO VectorOp Expr Expr
              | SO ScalarOp IntExpr Expr
data IntExpr  = I Integer
              | NO NormOp Expr
data VectorOp = Add | Sub | Dot
data ScalarOp = Mul | Div
data NormOp   = NormOne | NormInf

#include "showme.hs"
```

Task

Define the contents of `showme.hs` so that given the above `vector.hs` you can have the following interaction with the interpreter:

Shell

```
$ ghci -cpp
GHCi, version 8.0.1: http://www.haskell.org/ghc/  :? for help
Prelude> :load vector.hs
[1 of 1] Compiling Vector           ( vector.hs, interpreted )
Ok, modules loaded: Vector.
*Vector> VO Dot (VO Add (V [1,2]) (V [3,4])) (SO Mul (NO NormOne (V [2])) (V [5,6]))
{'dot', {'add', [1,2], [3,4]}, {'mul', {'norm_one', [2]}, [5,6]}}
```

In general, given any part of an expression that is using the above constructors the interpreter should print the equivalent "pretty" version, as it appeared in the specification of the `vector_server` in the first assignment.

Hint

The type class `Show` may have something to do with this question...

5 Fulltext indexing (`indexing.hs`, 4 points)

Your task is to speed up, as much as possible, a program that performs fulltext indexing and searching of documents by using the power of multicores. The sequential program and its inputs can be found in course's page; you will need to download and unpack the set of sample documents:

```
$ tar xvf docs.tar.bz2
```

You can try out the program like this:

```
$ ghc -O indexing.hs && ./indexing docs/*
search (^D to end): <type your search terms here>
```

For example, you could enter `parallel concurrent` and the program would list the filenames of all the documents that contain both those words. To benchmark the program, run it like this:

```
$ echo "parallel concurrent" | ./indexing docs/* +RTS -s
```

The program works by creating a mapping from words to the set of documents that contains that word.

Sample

```
$ # original indexing.hs
$ ulimit -n 4096 && ghc -O indexing.hs  && echo "keyword" | ./indexing docs/* +RTS -s
...
 Total   time    4.034s  (  4.020s elapsed)
...
$ # make indexing.hs faster
$ ghc -O indexing.hs -rtsopts -threaded
$ ulimit -n 4096 && echo "parallel concurrent" | ./indexing docs/* +RTS -N2 -s
...
  Total   time    3.703s  (  1.972s elapsed)
...
$ ulimit -n 4096 && echo "parallel concurrent" | ./indexing docs/* +RTS -N4 -s
...
  Total   time    4.384s  (  1.253s elapsed)
...
$ ulimit -n 4096 && echo "parallel concurrent" | ./indexing docs/* +RTS -N8 -s
...
  Total   time    5.211s  (  0.835s elapsed)
...
$ ulimit -n 4096 && echo "parallel concurrent" | ./indexing docs/* +RTS -N16 -s
...
  Total   time    9.992s  (  0.827s elapsed)
...
```

Grading

Your final points for this exercise is the sum of the speedup divided by the number of schedulers in each case from $[2, 4, 8, 16]$ cores measured on a machine with $>= 16$ cores. In the sample case, the total points for this exercise would be $4.02 * (1/1.972/2 + 1/1.253/4 + 1/0.835/8 + 1/0.827/16) = 2.73$.

Submission instructions

- All questions regarding the assignment and its grading should go to Christos, the course's assistant (`christos.sakalis@it.uu.se`). It is of course OK to cc: the teacher in the mails, but it is actually the assistant that will do the grading and can probably give you more accurate answers about what it is actually required to get a full grade in the assignment.
- Each student must send her/his own individual submission.
- For this assignment, you must submit a single `afp_assignment2.zip` file at the relevant section in Studentportalen. `afp_assignment2.zip` should contain eight files (without any directory structure):
  - The seven files requested (`perm.hs`, `perm.pdf`, `kcolor.hs`, `kcolor.pdf`, `fringe.hs`, `fringe.pdf`, `showme.hs`, and `indexing.hs`) which should conform to the specified interfaces regarding exported functions, handling of input and format of output.
  - A text file named `README.txt` whose first line should be your name. You can include any other comments about your solutions in this file.

Have fun!