CHALMERS TEKNISKA HÖGSKOLA
Datavetenskap
Dennis Björklund

October 25th, 2003
Functional programming
INN040/TDA450

Exam
Functional Programming

Saturday, October 25th, 2003, 8.45 - 12.45.
Examiner: Dennis Björklund, telephone office: 772 5402, cell: 073 636 51 79.

Permitted aids:
English-Swedish or English-other language dictionary.

- Begin each question on a new sheet. Write your personal number on every sheet.
- You may lose marks for unnecessarily long, complicated, or unstructured solutions.
- Full marks are awarded for solutions which are elegant, efficient, and correct.
- You are free to use any Haskell standard functions, including those whose definitions are attached, unless the question specifically forbids you to do so.
- You may use the solution of an earlier part of a question to help solve a later part, even if you did not succeed in solving the earlier part.
- The exam consists of 5 questions, worth 9, 4, 16, 12, and 19 points. For a Chalmers students the grade limits are: 3: 24p, 4: 36p, 5: 48p. For a Gothenburg University student they are: G: 28p, VG: 48p.

1. Give the most general type for each of the following functions:

   (a) `fa (a,b) = a` (1 p)

   (b) given the definition

   ```
   data Either a b = Left a | Right b
   ```

   give the type for

   ```
   fb (Left a)  = a
   fb (Right b) = b
   ```

   (2 p)

   (c) `fc [] c = []`
   `fc (x:xs) c = (c,x) : fc xs c` (2 p)

   (d) `fd [] g z = z`
   `fd (x:xs) g z = g x (fd xs g z)` (2 p)

   (e) `fe x [] = False`
   `fe x (y:ys) = x == y || fe x ys` (2 p)

2. Evaluate the following expression by hand. You only need to show the final answer

   (a) Given:

   ```
   f []  = 0
   f [x] = 1
   f (x:y:xs) = y + f xs
   ```

   Evaluate:

   ```
   f [1,2,3,4,5]
   ```

   (2 p)

   (b) Given:

   ```
   g []         rs = rs
   g ((a,b):xs) rs = g xs (a : b : rs)
   ```

   Evaluate:

   ```
   g [ (1,2), (3,4) ] []
   ```

   (2 p)

3. Given the definition:

```
data Expr
  = Const Int
  | Plus Expr Expr
  | Mult Expr Expr
```

that is used to represent arithmetic expressions in haskell.

(a) Write the expression

$$(1 + 2) * (3 + 4)$$

as a value in the datatype `Expr` (2 p)

(b) Write a function with the following name and type

```
eval :: Expr -> Int
```

that computes an integer value by evaluating the expression. For the example above we would get 21 as a result. (3 p)

(c) We want to add variables to the expressions. Our variables are described by the type:

```
type VarId = (String,Int)
```

That is, a variable consists of a name and a number. For example the variable x_2 would be represented as `("x",2)`. Rewrite the type `Expr` to include variables. (1 p)

(d) An environment is a data structure that associates values with variables. It's used to lookup a variable and get the associated value.

We model environments using the following type:

```
type Env = VarId -> Maybe Int
```

The datatype `Maybe` is defined as:

```
data Maybe a = Just a | Nothing
```

We use `Nothing` to handle the case where you try to lookup a variable that is not in the environment.

Define

```
emptyEnv :: Env
```

that is an environment where no variables at all are associated with values. Next, define a function to extend a given environment with a new association of a variable and a value:

```
extendEnv :: Env -> VarId -> Int -> Env
```

This function works so that:

```
extendEnv env var i
```

returns an environment that is the same as `env` except that it also associates the integer `i` with the variable `var`. (4 p)

(e) Using the functions in part 3d write an expression that is an environment where x_1 is associated with the value 13 and x_3 is associated with the value 42.

You should see the enviroments as an abstract datatype and only use the functions defined for it. (1 p)

(f) Define a function

```
lookupEnv :: Env -> VarId -> Maybe Int
```

that given an environment and a variable produces a value associated with the variable (if the variable is in the environment), We use the `Maybe` type to be able to either return an integer value or an error value. (1 p)

(g) Using the new datatype in part 3c and the environments above, define a new evaluation function:

```
eval :: Env -> Expr -> Maybe Int
```

that makes use of the new expression type in part 3c and the enviroments we defined above.

Notice that we use the `Maybe` type here too to handle errors, in this case when you have a formula that contains variables that are not in the environment.

As before, you should see the enviroment as an abstract datatype and only use the functions defined for it. (4 p)

4. We say that a string `xs` is a *subword* of another string `ys` if one can obtain `xs` by removing a number of characters from `ys`. So for example `"so"` is a subword of `"subword"` and `"apa"` is a subword of `"apparat"`. We do not restrict ourselves to strings containing letters, so for example `"j u"` is a subword of `"hej du"`.

   (a) Define a function
   `subWords :: String -> [String]`
   which gives as its result the list of all subwords of its argument. For example:

   ```
   ? subWords "apa"
   ["", "a", "p", "pa", "a", "aa", "ap", "apa"]
   ```

   You need not worry about producing the subwords in the same order as they are given here. (6 p)

   (b) Define a function
   `isSubWord :: String -> String -> Bool`
   such that `isSubWord xs ys` tests whether `xs` is a subword of `ys`. This can be done using the definition

   ```
   isSubWord xs ys = elem xs (subWords ys)
   ```

   but this is much too inefficient – the number of subwords of a string of length n is 2^n. Give a more efficient definition of `isSubWord`. (6 p)

5. A scieintfic study made at a unsrievity in England has shown that if the first two and the last two letters in every word are plcaed corrcetly, then it does not matter in what order the other letetrs in the word come. The text is rebdaale even if the rest is in radnom order. This is beacuse people don't read each single letter but one word at a time.

   If you can't read the paragraph above the full text is in this footnote[1].

   We will write functions to randomize text like that.

   (a) The first part is to write a function that works on a single word. At first one might think that we should write a function of type `String -> String`, but we can not. Functions in haskell are pure, which means that when called several times with the same input they produce the same output. For that reason, we can not write a function with that type that randomizes the middle part of the word.

   Instead we will write a function that also takes a list of random numbers as input, and as output gives the result string and the list of random numbers that is did not consume:

   Define the function

   ```
   randomizeWord :: [Int] -> String -> ([Int], String)
   ```

   Hint! To reorder the characters in a string one can pair each character c with a random number r as (r,c) and then simply sort the result (it works since tuples are sorted in a lexicographic order). (8 p)

   (b) Using the function above we shall now write a function that works on a full sentence. Our function has the following type:

   ```
   randomizeSentence :: [Int] -> String -> ([Int],String)
   ```

   This function works just as the function `randomizeWord` above, except that the strings now are complete sentences. The integer lists are just as above the lists of random numbers.

   The input sentence will consist of only simple letters and spaces, you do not need to worry about other characters, like question marks or similar.

   For example, given that we have an infinite list of random numbers, `randList` we can call the function as follows:

   ```
   snd (randomizeSentence randList
     "A scientific study made at a university in England")
   ```

   and as result get

   ```
   "A scieintfic study made at a unsrievity in England"
   ```

   (7 p)

[1] A scientific study made at a university in England has shown that if the first two and the last two letters in every word are placed correctly, then it does not matter in what order the other letters in the word come. The text is readable even if the rest is in random order. This is because people don't read each single letter, but one word at a time.

(c) Write a small main program that works like this when compiled and run:

```
bash> ./randomize
Enter the string to randomize: sweden europe
swdeen euorpe
Enter the string to randomize: A scientific study
A scetifniic study
```

You can use the following definition to generate an infinite list of random numbers

```
gen :: IO [Int]
```

Make sure you use `gen` only once during an execution of your program. Once you have the infinite list of random numbers, you use that list to pick from, over and over again. (4 p)