COMP313-08A Programming Languages

Second Haskell coursework

Solutions

1. Write a function

`halve :: [a] -> ([a],[a])`

that splits an even-lengthed list it two equal-lengthed halves, so:

`halve [1,2,3,4,5,6] = ([1,2,3],[4,5,6])`

Using the Prelude-defined functions

```
take :: [a] -> Integer -> [a]
```

and

```
drop :: [a] -> Integer -> [a]
```

we have

```
halve xs = (take k xs, drop k xs)
            where k = length xs `div` 2
```

2. Define a function

`mix :: [Int] -> [Int] -> [Int]`

which mixes two sorted lists of integers together, to get a sorted list which contains all the elements from the two original lists, so:

`mix [3,6,9] [1,8,10] = [1,3,6,8,9,10]`

The definition should use explicit recursion and no other pre-defined sorting functions.

```
mix [] ys = ys
mix xs [] = xs
mix (x:xs) (y:ys)
   |  x <= y = x : mix xs (y:ys)
   |  x >= y = y : mix (x:xs) ys
```

3. Use the function `mix` *from question two to define a function*

`sortints :: [Int] -> [Int]`

which sorts a list of integers. The empty list and a singleton list are already sorted, and any other list should be sorted by mixing together the two lists that result from sorting the two halves of the list separately.

As the question says, the strategy here is to mix together the two, already sorted, halves of the original list. So, we halve the original list, sort it and then mix:

```
sortints [] = []
sortints [x] = [x]
sortints xs = mix (sortints xs1) (sortints xs2)
              where (xs1, xs2) = halve xs
```

Note that, in fact, the halving as defined in question one above in fact works on odd- or even-lengthed lists, so this definition is OK.

5. Do exercise 7.5 from the text book.

To answer this we just need to add an *environment* which records the bindings of variables to values, make sure this is carried around by the evaluation function, and give ways of adding to and looking-up (finding) in it.

```
data Expr = C Float | Expr :+ Expr | Expr :-Expr| Expr :* Expr |
            Expr :/ Expr | V String| Let String Expr Expr

evaluate :: Expr -> Float
evaluate exp = evaluate' exp [] ---set-up an   empty environment

evaluate' (C x)         env = x

evaluate' (e1 :+ e2)    env = evaluate' e1 env + evaluate' e2 env

evaluate' (e1 :- e2)    env = evaluate' e1 env - evaluate' e2 env

evaluate' (e1 :* e2)    env = evaluate' e1 env * evaluate' e2 env

evaluate' (e1 :/ e2)    env = evaluate' e1 env / evaluate' e2 env

evaluate' (V v)         env = find v env

evaluate' (Let v e1 e2) env = evaluate' e2 (extend v e1 env)

find v  []          = error ("Unbound variable: " ++ v)

find v1 ((v2,e):es) = if v1 == v2 then e else find v1 es

---We extend with variables that may already appear in
---the environment so as to have a sensible block
---structure, so, for example,
---evaluate (Let "x" (C 5) (Let "x" (C 4) (V "x"))
---gives 4.0 and not 5.0

extend v e env = (v, evaluate' e env) : env
```

6. Do exercise 8.1 from the text book.

We get a never-ending list of unit-radius circles by taking the list `manyCircles` and forming a single region out of them:

```
foldr Union Empty manyCircles
```

and then intersect this with a five-circles-long rectangle (i.e. it will be 10 units long—five two-unit-diameter circles, and two units high) which is first translated so its centre is at the centre of the third circle, i.e. four units to the right):

```
Translate (4,0) (Shape (Rectangle 10 2))
```

which gives us the final region given by the function:

```
fiveUnitCircles = (foldr Union Empty manyCircles) `Intersect`
                       (Translate (4,0) (Shape (Rectangle 10 2)))
```

The main problem with this solution is that, once we decide to display it, and because the low-level graphics functions are, of course, not lazy (physical systems tend not to be), the very long list of circles never gets completely represented on the pixel grid that the low-level graphics compiles, and so we never get around to doing the intersection, i.e. nothing gets displayed, though a lot of computation takes place. However, as an expression this is conceptually simpler—we don't have to say how many circles should be displayed, just that the rectangle obscures all but five of them.

7. Do exercise 8.3 from the text book.

We can do this by taking the complement of a circle whose radius is given by the first argument (this gives us all of the plane with a "hole" in it given by the smaller circle) and then intersecting this with a circle whose radius is the second argument (so all of the plane beyond the larger circle drops away) leaving just the required ring:

```
annulus :: Radius -> Radius -> Region

annulus innner outer =
                      (Complement (Ellipse inner inner))
                             `intersect` (Ellipse outer outer)
```

8, Do exercise 9.2 from the text book.

We have, by definition,

```
flip f x y = f y x
```

so, for arbitrary `x` and `y`,

```
flip (flip f) x y
-> flip f y x
-> f x y
```

So

```
flip (flip f) = f
```

9. Do exercise 9.9 from the text book.

To find the principal type of `fix` we reason as follows: Assume `f` has type `a` and `fix` has type `b`. From the LHS, `b` must be `a -> c` for some `c` since `fix` is clearly a function that takes something like `f` of type `a`.

From the RHS, `a` must be `c -> c`, since `f` is clearly a function which takes things like `(fix f)` which has type `c` to things like `(fix f)`(i.e. the LHS) which has type `c`. Putting all this together we have

`b =a -> c`

`a =c -> c` which means

`b= (c -> c) ->c`

and remember `b`

was the type of `fix`, so

```
fix :: (c -> c) -> c
```

First abstract on the arguments, to get:

```
remainder = \a -> \b -> if a<b then a else remainder (a-b) b
```

then define a new function, which I'll call `absrem`, by:

```
absrem = \r -> \a -> \b -> if a<b then a else r (a-b) b
```

Note that *this* definition is not recursive.

Also note that

```
absrem remainder
->
(\r -> \a -> \b -> if a<b then a else r (a-b) b) remainder
->
\a -> \b -> if a<b then a else remainder (a-b) b
->
remainder
```

i.e.

```
remainder = absrem remainder                    (1)
```

Now consider the equation defining `fix`:

```
fix f = f (fix f)
```

and the evaluation sequence

```
fix absrem
```

`->` (unfolding `fix`)

```
absrem (fix absrem)
```

Clearly, if we make the (non-recursive) definition:

```
remainder = fix absrem                          (2)
```

then we get

```
remainder

->

fix absrem

->

absrem (fix absrem)

->

absrem remainder
```

as required by (1). So, (2) is the required non-recursive definition for `remainder`.

This process can be used to make any recursive definition into a non-recursive one, except for `fix`. Using `fix` to remove itself from a recursive definition is clearly not going to work.

If we step outside of the strong typing (every expression has exactly one principal type) of languages like Haskell into a language with no types then we can use this process to make any function non-recursive. In such a language (like the λ-calculus) then the expression

$\lambda f.\ (\lambda x.\ f\ (x\ \ x))(\ \lambda x.f\ (x\ \ x))$

behaves like `fix` above, but does not use itself in its own definition. This expression cannot be well-formed in a typed language since we have "x x", i.e. x being applied to itself, and no typed expression can be given that works in that way (there is always an extra -> needed in the type for the left-hand x compared with the right-hand x, so x must be given two types, which means the expression cannot be well-typed and so is not well-formed).

10. Do exercise 9.10 from the text book.

```
 (y+1)/2
```
is
```
(/2) ((+1) y)
```
which is
```
(/2).(+1) y
```
which is
```
(\x -> (x+1)/2) y
```

So,
```
\x -> (x+1)/2 = (/2) . (+1)
```

So,
```
map (\x -> (x+1)/2) xs
```
is
```
map (/2).(+1) xs
```
and this is
```
map (\x -> (x+1)/2) = map (/2).(+1)
```

11. Do exercise 9.11 from the text book.

```
map f (map g xs)
```
->
```
(map f . map g) xs
```
->
```
map (f . g) xs
```

So, the earlier
```
map (/2).(+1) xs
= map (/2) (map (+1) xs)
```

12. Do exercise 11.1 from the text book.

Prove `putCharList cs = map putChar cs` for all finite lists `cs`. Use induction on `cs`.

Base case: Prove `putCharList [] = map putChar []`

`putCharList []`

-> (unfold `putCharList`)

[]

-> (fold map)

`map putChar []`

Assume: For arbitrary `cs`, `putCharList cs = map putChar cs`
To prove: For arbitrary `c`, `putCharList (c:cs) = map putChar (c:cs)`

```
putCharList (c:cs)
-> (unfold putCharList)
putChar c : putCharList cs
-> (by assumption)
putChar c : map putChar cs
-> (fold map)
map putChar (c:cs)
Q.E.D.
```

Prove `listProd xs = fold (*) 1 xs` for all finite lists `xs`. Use induction on `xs`.

Base case: Prove `listProd [] = fold (*) 1`

```
listProd []
-> (unfold listProd)1
-> (fold fold)
fold (*) 1 []
```

Assume: For arbitrary `xs`, `listProd xs = fold (*) 1 xs`
To prove: For arbitrary `x`, `listProd (x:xs) = fold (*) 1 (x:xs)`

```
listProd (x:xs)
-> (unfold listProd)
x * listProd xs
-> (by assumption)
x * fold (*) 1 xs
-> (fold fold)
fold (*) 1 (x:xs)
Q.E.D.
```

Your answers for all the questions above are due at 1000 on Wednesday 16th April 2008.
*You must submit your answers as a plain text file via Moodle. This **MUST** be a plain text file (**not** a PDF, **not** a MS-format file or any other sort of particular format) since we will want to load your solutions and try them out in hugs. Please also put your name and number at the top of the file.*