Weekly Assignments 5

To be submitted: Monday, 12 February 2018

Note that some tasks may deliberately ask you to look at concepts or libraries that we have not yet discussed in detail. But if you are in doubt about the scope of a task, by all means ask.

Please try to write high-quality code at all times! This means in particular that you should add comments to all parts that are not immediately obvious. Please also pay attention to stylistic issues. The goal is always to submit code that does not just correctly do what was asked for, but also could be committed without further changes to an imaginary company codebase.

W5.1 Packaging

Prepare a Cabal package to contain all your solutions so that it can easily be built using cabal-install or stack.

Note that one package can contain one library (with arbitrarily many modules) and possibly several executables and test suites.

Please include a `README` file in the end explaining clearly where within the package the solutions to the individual subtasks are located.

Please do *NOT* try to upload your package to Hackage.

W5.2 Delayed computations

The type constructor `Delayed` can be used to describe possibly non-terminating computations in such a way that they remain "productive", i.e., that they produce some amount of progress information after a finite amount of time.

```
data Delayed a = Now a | Later (Delayed a)
```

We can now describe a productive infinite loop as follows:

```
loop :: Delayed a
loop = Later loop
```

This is productive in the sense that we can always inspect more of the result, and get more and more invocations of `Later`.

We can also use `Later` in other computations as a measure of cost or effort. For example, here is a version of the factorial function in the `Delayed` type:

```
factorial :: Int -> Delayed Int
factorial = go 1
```

```
  where
    go !acc n
      | n <= 0    = Now acc
      | otherwise = Later (go (n * acc) (n - 1))
```

We can extract a result from a `Delayed` computation by traversing it all the way down until we hit a `Now`, at the risk of looping if there never is one:

```
unsafeRunDelayed :: Delayed a -> a
unsafeRunDelayed (Now x)   = x
unsafeRunDelayed (Later d) = unsafeRunDelayed d
```

Subtask 5.2.1

Define a function

```
runDelayed :: Int -> Delayed a -> Maybe a
```

that extracts a result from a delayed computation if it is guarded by at most the given number of `Later` constructors, and `Nothing` otherwise.

Subtask 5.2.2

The type `Delayed` forms a monad, where `return` is `Now`, and `>>=` combines the number of `Later` constructors that the left and the right argument are guarded by.

Define the `Functor`, `Applicative`, and `Monad` instances for `Delayed`.

Subtask 5.2.3

Assume we have

```
tick :: Delayed ()
tick = Later (Now ())

psum :: [Int] -> Delayed Int
psum xs = sum <$> mapM (\ x -> tick >> return x) xs
```

Describe what `psum` does.

Subtask 5.2.4

The type `Delayed` is actually a free monad. Define the functor `DelayedF` such that `Free DelayedF` is isomorphic to `Delayed`, and provide the witnesses of the isomorphism:

```
fromDelayed :: Delayed a -> Free DelayedF a
toDelayed   :: Free DelayedF a -> Delayed a
```

Subtask 5.2.5

We can also provide an instance of `Alternative`:

```
instance Alternative Delayed where
  empty = loop
  (<|>) = merge

merge :: Delayed a -> Delayed a -> Delayed a
merge (Now x) _           = Now x
merge _ (Now x)           = Now x
merge (Later p) (Later q) = Later (merge p q)
```

Define a function

```
firstSum :: [[Int]] -> Delayed Int
```

that performs `psum` on every of the integer lists and returns the result that can be obtained with as few delays as possible.

Example:

```
runDelayed 100 $
    firstSum [repeat 1, [1,2,3], [4,5], [6,7,8], cycle [5,6]]
```

should return `Just 9`.

Subtask 5.2.6

Unfortunately, `firstSum` will not work on infinite (outer) lists and

```
runDelayed 200 $
    firstSum $
        cycle [repeat 1, [1,2,3], [4,5], [6,7,8], cycle [5,6]]
```

will loop.

The problem is that `merge` schedules each of the alternatives in a fair way. When using `merge` on an infinite list, all computations are evaluated one step before the first `Later` is produced. Define

```
biasedMerge :: Delayed a -> Delayed a -> Delayed a
```

that works on infinite outer lists by running earlier lists slightly sooner than later lists. Write

```
biasedFirstSum :: [[Int]] -> Delayed Int
```

which is `firstSum` in terms of `biasedMerge`. Note that `biasedFirstSum` will not necessarily always find the shortest computation due to its biased nature, but it should work on the infinite outer list example above and also in

```
runDelayed 200 $
    biasedFirstSum $
        replicate 100 (repeat 1) ++ [[1]] ++ repeat (repeat 1)
```

to return `Just 1`.

W5.3 Type checking

Assume the environment

```
Gamma = epsilon
      , map :: forall a b . (a -> b) -> [a] -> [b]
      , singleton :: forall a . a -> [a]
```

Using the DHM type rules given on the slides, provide valid type judgements (full trees) that explain that

```
Gamma |- map map :: [x -> y] -> [[x] -> [y]]
```

and

```
Gamma |- map (\ x -> singleton x) :: [x] -> [[x]]
```

W5.4 Difference lists

Consider (following the `dlist` package):

```
data DList a = DL { unDL :: [a] -> [a] }

fromList :: [a] -> DList a
fromList = DL . (++)

toList :: DList a -> [a]
toList = ($ []) . unDL

empty :: DList a
empty = DL id

singleton :: a -> DList a
singleton = DL . (:)

append :: DList a -> DList a -> DList a
append xs ys = DL (unDL xs . unDL ys)
```

```
foldr :: (a -> b -> b) -> b -> DList a -> b
foldr f b = Data.List.foldr f b . toList
```

Convince yourself that the "naive" way of writing `reverse` for lists

```
reverseList :: [a] -> [a]
reverseList = Data.List.foldr (\ x r -> r ++ [x]) []
```

is much less efficient than going via a `DList`:

```
reverseDList :: DList a -> DList a
reverseDList = foldr (\ x r -> append r (singleton x)) empty

reverseListViaDList :: [a] -> [a]
reverseListViaDList = toList . reverseDList . fromList
```

Can you explain (or even prove?) why?