Monads in functional programming - a homework

Patryk Kajdas

Please send solutions to `patry93k@gmail.com`. The deadline is 29.03.2015. You can send solutions to both 5.1 and 5.2 exercises, but you will obtain only $max(c1, c2)$ points, where ci is the number of points obtained in exercise $5.i$.

Ex 1 (1 point). Write a tail-recursive function equivalent to `map`, i.e. to

```
map f [] = []
map f x:xs = f x:map f xs
```

Ex 2 (1 point). Consider the following definition of `cfold`:

```
cfold f z [] = z
cfold f z (x:xs) = f x z (\y -> cfold f y xs)
```

Define `foldl` and `foldr` in terms of `cfold`

Ex 3 (2 points). We define the `MaybeT` type by

```
newtype MaybeT m a = MaybeT (m (Maybe a))
```

What if we had defined it by

```
newtype MaybeT m a = MaybeT (Maybe (m a))
```

instead? What are the semantics of such type then? Could we still have defined a monad transformer based on it? If so, define it.

Ex 4 (2 points). Why is it that the `lift` function has to be defined seperately for each monad, whereas `liftM` can be defined in a universal way?

Ex 5.1 (3 points). Define a `Diagnostics` monad that, like the `Writer` monad (wasn't covered during the lecture - become acquainted with it!), allow the programmer to log debug messages, but in addition provides an `abort` operation that stops the program when invoked. Make sure that the log is not erased when `abort` is called!

Ex 5.2 (3 points). When implementing program transformations or other symbolic processing, it is often necessary to generate fresh identifiers. Define a `NameSupply` monad that offers an operation `gensym`. Every time `gensym` is used, it returns the next name form the sequence `a, ..., z, a1, ..., z1, a2, ...`

Ex 6 (4 points). Devise a continuation monad transformer – give its (informal) semantics and implement it. Then explain the characteristics of a monad obtained from combining:

- your transformer with a list monad
- the list monad transformer with a continuation monad

Ex 7 (6 points). Define a function `callCCexp` (exp from “explicit”) that acts just as `callCC` but with the difference that it never returns to the current continuation **unless** a value is specifically passed to that continuation (so, recalling the example from the lecture, if we call `exit ‘‘foo’’`, then eveything works as before – but if we never call `exit` and reach the end of the inner do-block, we shouldn’t get back to the outer block (where we invoked `callCCexp`)). Can you define `callCC` and `callCCexp` in terms of each other? If so, do it.