Weekly Assignments 6

To be submitted: Tuesday, 20 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.

W6.1 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.

W6.2 foldr-build fusion

One optimisation technique that GHC employs for list functions is so-called foldr-build fusion.

We know what `foldr` is; it is a way to systematically consume a list. Let us consider `build`:

```
build :: (forall r . r -> (a -> r -> r) -> r) -> [a]
build builder = builder [] (:)
```

The function `build` takes a builder, which is a polymorphic function parameterized by two arguments more or less matching the types of the list constructors.

Here is an example of `replicate` written using `build`:

```
replicate :: Int -> a -> [a]
replicate n x =
  build (\ nil cons ->
    let
      go n =
```

```
        if n <= 0
          then nil
          else cons x (go (n - 1))
    in
      go n)
```

Convince yourself that `replicate` really behaves like `replicate` should.

Subtask 2.1

Define `fromTo :: Int -> Int -> [Int]` where

```
fromTo 1 5 = [1,2,3,4,5]
fromTo 4 4 = [4]
fromTo 5 4 = []
```

with the help of the `build` function.

Subtask 2.2

The foldr-build fusion law says that

```
foldr op e (build builder) = builder e op
```

In particular, on the right hand side, no list has to ever be built.

Again, convince yourself that given an instance of `foldr` and one of the builder examples above, the left and right hand side really evaluate to the same results.

There are, however, some tricky side conditions with respect to laziness and undefined that are required for this law to hold. Find a counterexample.

Subtask 2.3

Define `filter` as an instance of both `build` and `foldr`, i.e.,

```
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = build (\ nil cons -> foldr ...)
```

W6.3 Explicit dictionaries

Internally, GHC represents class declarations as declarations of record types, and instance declarations as the definitions of constants or functions resulting in values of these record types.

These records are often called “dictionaries”, because they allow the compiler to look up the implementation of class methods in a given context.

For example, consider the dictionary type for `Functor`:

```
data FunctorDict f = FunctorDict { fmap_ :: forall a b . (a -> b) -> f a -> f b }
```

The instance for lists corresponds to:

```
listFunctor :: FunctorDict []
listFunctor = FunctorDict map
```

Consider the sum and composition of functors:

```
data Sum f g a = Inl (f a) | Inr (g a)
newtype Compose f g a = Compose { getCompose :: f (g a) }
```

Functors are closed under sums and composition, corresponding to instances with the following headers

```
instance (Functor f, Functor g) => Functor (Sum f g)
instance (Functor f, Functor g) => Functor (Compose f g)
```

These instances correspond to dictionary-transforming functions of the following types:

```
sumFunctor :: FunctorDict f -> FunctorDict g -> FunctorDict (Sum f g)
composeFunctor :: FunctorDict f -> FunctorDict g -> FunctorDict (Compose f g)
```

Define these!

W6.4 Performance of free monads

Recall the type `GP` from W1:

```
data GP a =
    End a
  | Get (Int -> GP a)
  | Put Int (GP a)
```

This is an instance of a free monad, and the monad instance is as follows:

```
instance Monad GP where
  return = End
  End x >>= f = f x
  Get k >>= f = Get (\ x -> k x >>= f)
  Put x k >>= f = Put x (k >>= f)
```

(with the obvious instances for `Functor` and `Applicative`)

We also have wrappers:

```
get :: GP Int
get = Get End
```

```
put :: Int -> GP ()
put x = Put x (End ())
```

And a function to simulate a program given a list of inputs:

```
simulate :: GP a -> [Int] -> a
simulate (End x)   _        = x
simulate (Put _ k) is       = simulate k is
simulate (Get k)   (i : is) = simulate (k i) is
```

(Our original simulation function was also producing the list of outputs, but we don't need that in this task.)

Subtask 4.1

Consider:

```
askMany :: Int -> GP Int
askMany 0 = return 0
askMany n = askMany (n - 1) >>= \ n -> get >>= \ x -> return (n + x)
```

What happens if you execute `simulate (askMany 100000) (repeat 1)`? Explain why.

Subtask 4.2

We're going to apply the same trick here that we applied in W5 for difference lists. One way to view the step going from ordinary lists to difference lists is to remove the empty list and replace it with a continuation, a pointer to a list to append to the current one.

In our situation, we're going to remove the `End` constructor and replace it with a continuation, a pointer to another `GP` computation to execute next.

Unlike list append, monadic bind can make use of the result of the previous computation and change the type of the result. So if we're considering a computation of type `GP a`, the continuation has to be of type `a -> GP b` for some `b`.

This motivates the following definition:

```
newtype GP' a = GP' { unGP' :: forall b . (a -> GP b) -> GP b }
```

Here, `GP` is the old `GP` type. Since we don't know the ultimate result type `b` of the computation, we keep it polymorphic.

Define functions

```
fromGP :: GP a -> GP' a
toGP   :: GP' a -> GP a
```

that map back and forth between a `GP a` and `GP' a`. In fact, the two types are isomorphic. (Hint: both functions are one-liners).

Subtask 4.3

Use the conversion functions to define

```
get'      :: GP' Int
put'      :: Int -> GP' ()
simulate' :: GP' a -> [Int] -> a
```

that work in the new type.

Subtask 4.4

Define a `Monad` instance for `GP'`. This is perhaps a bit tricky, but if you follow the guidance of the types, there is not much else you can do but the right thing. Note that the `Monad` instance for `GP'` is *not* reusing the monad instance for `GP`.

The `Functor` and `Applicative` instances are as usual.

Subtask 4.5

Now reimplement `askMany` using `GP'` and convince yourself that the problem is fixed.

W6.5

The package `monad-par` defines a computation type `Par` of parallel computations. It is quite similar to `async`, but where async can be used to concurrently and asynchronously run side-effecting computations, the computations in the `Par` type are supposed to be pure and deterministic.

There's a function

```haskell
spawn :: NFData a => Par a -> Par (IVar a)
```

that runs a `Par` computation in parallel to the current one, and delivers the result in an `IVar`.

There's also a function

```haskell
runPar :: Par a -> a
```

that allows to run a parallel computation, just getting a plain `a` out, because everything is supposed to be deterministic.

Look at the sources of the package and find the definitions of `Par` and `Trace`. Copy them and explain in what ways they are similar or different from the definition of `GP`.

Can you rewrite the definition so that it actually makes use of `Free`?