4 Monads and applicative functors

1. Recall the interface for monads given in the lecture:

$$\begin{array}{l}\textbf{class}\ \mathit{Monad}\ m\ \textbf{where}\\ \quad \mathit{return} :: a \to m\ a\\ \quad (\gg\!=)\ \ :: m\ a \to (a \to m\ b) \to m\ b\end{array}$$

These operators should also satisfy three laws:

$$\begin{array}{lll} \mathit{return}\ x \gg\!= k &= k\ x & \text{-- left unit}\\ m \gg\!= \mathit{return} &= m & \text{-- right unit}\\ (m \gg\!= k) \gg\!= l &= m \gg\!= (\lambda x \to k\ x \gg\!= l) & \text{-- associativity}\end{array}$$

which can be seen as similar to the unit and associativity laws of monoids. There is an alternative presentation of monads:

$$\begin{array}{l}\textbf{class}\ \mathit{Functor}\ m \Rightarrow \mathit{Monad'}\ m\ \textbf{where}\\ \quad \mathit{return} :: a \to m\ a\\ \quad \mathit{join}\ \ \ :: m\ (m\ a) \to m\ a\end{array}$$

subject to three different laws:

$$\begin{array}{lll} \mathit{join} \circ \mathit{fmap}\ \mathit{return} &= \mathit{id} & \text{-- left unit}\\ \mathit{join} \circ \mathit{return} &= \mathit{id} & \text{-- right unit}\\ \mathit{join} \circ \mathit{fmap}\ \mathit{join} &= \mathit{join} \circ \mathit{join} & \text{-- associativity}\end{array}$$

in which the similarity to monoids is perhaps easier to spot. Here, *Functor* is another type class:

$$\begin{array}{l}\textbf{class}\ \mathit{Functor}\ m\ \textbf{where}\\ \quad \mathit{fmap} :: (a \to b) \to m\ a \to m\ b\end{array}$$

subject to two laws:

$$\begin{array}{lll} \mathit{fmap}\ \mathit{id} &= \mathit{id} & \text{-- identity}\\ \mathit{fmap}\ (g \circ f) &= \mathit{fmap}\ g \circ \mathit{fmap}\ f & \text{-- composition}\end{array}$$

It turns out that the presentation of monads in terms of *join* is mathematically more natural, but the presentation in terms of $\gg\!=$ is more convenient for programming. Given that M has a well behaved instance for *Monad*, show that it can also be given a well behaved instance for *Functor* and *Monad'*; and conversely.

2. Instead of $\gg=$, one could equivalently use *Kleisli composition*:

   $$(>\!\!=\!\!>) :: \mathit{Monad}\ m \Rightarrow (a \rightarrow m\ b) \rightarrow (b \rightarrow m\ c) \rightarrow a \rightarrow m\ c$$

   Give a definition of $>\!\!=\!\!>$ in terms of $\gg=$, and vice versa.

3. In the lecture, the exception-raising monadic evaluator *evalE* uses the *Maybe* monad: either it succeeds or fails, but if it fails it gives no error message. Here is a more informative datatype for return values:

   $$\textbf{data}\ \mathit{Exc}\ a = \mathit{Error}\ \mathit{String} \mid \mathit{Success}\ a$$

   Redo the evaluator to return an error message when the expression involves division by zero. What is the *Monad* instance for *Exc*?

4. Extend the type of expressions to specify an alternative term to evaluate in case of an exception, and correspondingly modify the monadic evaluator with exceptions *evalE* to handle this extension. More specifically, extend the type *Expr* to include a new case *Try Expr Expr*. To evaluate *Try d e*, first evaluate *d*; if it succeeds, return its value, but if evaluation raises an exception, then evaluate *e*. Here's a transcript of how the program should behave:

   Main⟩ $\mathit{eval}\ (\mathit{Try}\ (\mathit{Div}\ (\mathit{Lit}\ 1)\ (\mathit{Lit}\ 0))\ (\mathit{Lit}\ 42))$
   42

   Define a corresponding *try* operation on *Maybe* values to facilitate the modification.

5. In the monadic evaluator *evalC* that counts the number of divisions, the use of the state monad (called *Counter* in the lecture) is somewhat heavy-handed. Instead of keeping track of a current state, each computation could simply return a value paired with the number of operations required to compute it:

   $$\textbf{data}\ \mathit{Counter}\ a = C\ (a, \mathit{State})$$

   (As hinted in the lecture, you can't actually define *Monad* instances for **type** synonyms, but you can for **data** types; but this requires you to introduce a constructor like *C* above.) Modify the evaluator to use this new computation type.

6. Conversely, modify the monadic *evalC* evaluator with state to be able to access the current state in a computation (so the shortcut of the previous exercise won't work). More specifically, extend the type *Expr* to include a new term *Count*. The value of *Count* is the number of operations performed so far, a quantity that is retrieved by accessing the state. Here's a transcript of how the program should behave:

   Main⟩ *eval* (*Div* (*Div* (*Div* (*Lit* 6) (*Lit* 3)) *Count*) *Count*)
   1

   The answer here is 1 because the first instance of *Count* evaluates to 1 (since the division of 6 by 3 is performed earlier) and the second instance returns 2 (since the divisions of 6 by 3 and 2 by 1 are performed earlier). Define a corresponding *count* operation on the type *Counter* to facilitate the modification. (Again, you'll need to use **data** instead of **type**:

   data *Counter a* = *C* (*State* → (*a*, *State*))

   in order to define a *Monad* instance.)

7. Modify the monadic tracing evaluator *evalT* so that it only traces selected parts of the computation. More specifically, extend the type *Expr* with two extra terms, *Trace Expr* and *Untrace Expr*. Tracing should be turned on for all subterms surrounded by *Trace*, and turned off for all subterms surrounded by *Untrace*. To support this change, a computation should be represented by a function, the argument of which is a boolean that indicates whether tracing is on. To facilitate the modification, define suitable operations on computations to set and access the tracing status.

8. Recall the interface for applicative functors given in the lecture:

   class *Functor m* ⇒ *Applicative m* **where**
     pure :: *a* → *m a*
     (⊛) :: *m* (*a* → *b*) → *m a* → *m b*

   Instances should satisfy the following four laws:

   $$\begin{array}{lll} pure\ id \circledast v & = v & \text{-- identity} \\ pure\ (\circ) \circledast u \circledast v \circledast w & = u \circledast (v \circledast w) & \text{-- composition} \\ pure\ f \circledast pure\ x & = pure\ (f\ x) & \text{-- homomorphism} \\ u \circledast pure\ y & = pure\ (\lambda f \to f\ y) \circledast u & \text{-- interchange} \end{array}$$

These four laws are rather hard to remember, but they again give something analogous to monoidal properties. Here is an alternative presentation:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Functor}\ m \Rightarrow \mathit{Applicative'}\ m\ \textbf{where} \\
\quad \mathit{unit} :: m\ () \\
\quad (\otimes)\ :: m\ a \rightarrow m\ b \rightarrow m\ (a, b)
\end{array}
$$

subject to the following (much easier to remember) three laws:

$$
\begin{array}{lll}
\mathit{unit} \otimes m & = \mathit{fmap}\ (\lambda x \rightarrow ((), x))\ m & \text{-- left unit} \\
m \otimes \mathit{unit} & = \mathit{fmap}\ (\lambda x \rightarrow (x, ()))\ m & \text{-- right unit} \\
(m \otimes n) \otimes p & = \mathit{fmap}\ \mathit{assoc}\ (m \otimes (n \otimes p)) & \text{-- associativity} \\
 & \quad \textbf{where}\ \mathit{assoc}\ (x, (y, z)) = ((x, y), z) &
\end{array}
$$

in which the unit and associativity properties are much clearer. Show that any well behaved instance of *Applicative* can also be made a well behaved instance of *Applicative'*, and vice versa.

9. In the lecture, we saw that *Const A* is an applicative functor when *A* is a monoid (such as integers with addition). What goes wrong if you try to make *Const A* a monad?

10. Another functor that is applicative but not a monad is the Haskell datatype of lists, which includes both the finite and infinite lists. In the alternative presentation, the definition is:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Applicative'}\ [\,]\ \textbf{where} \\
\quad \mathit{unit} = \mathit{repeat}\ () \\
\quad (\otimes)\ = \mathit{zipWith}\ (,)
\end{array}
$$

That is, the unit is an infinite list of unit values, and pairing is a zip (yielding a result the same length as the shorter argument). It looks like this should correspond to a monad, but it does not... what goes wrong?

11. One of the things that is nicer about applicative functors than monads is that they compose well. Show that if *M* and *N* are well behaved applicative functors, then so is their composition *Compose M N*, where

$$
\textbf{data}\ \mathit{Compose}\ m\ n\ a = C\ (m\ (n\ a))
$$

Why can't you do the same thing for monads?

12. As we saw in the lecture, applicative functors support traversal perfectly well: you don't need the full power of monads for this. In Haskell, this is captured by the type class

$$
\begin{array}{l}
\textbf{class}\ \mathit{Traversable}\ t\ \textbf{where} \\
\quad \mathit{traverse} :: \mathit{Applicative}\ m \Rightarrow (a \to m\ b) \to t\ a \to m\ (t\ b)
\end{array}
$$

(again, subject to some laws, which we will ignore). I gave the obvious left-to-right definition of traversal over lists:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Traversable}\ [\,]\ \textbf{where} \\
\quad \mathit{traverse}\ f\ [\,] \qquad\quad = \mathit{pure}\ [\,] \\
\quad \mathit{traverse}\ f\ (x : xs) = \mathit{pure}\ (:) \circledast f\ x \circledast \mathit{traverse}\ f\ xs
\end{array}
$$

For a class M of effects, and an effectful body $f :: A \to M\ B$, applying *traverse* f to a list of type $[A]$ will perform the effects of f on each A in turn, from left to right, then return a list of type $[B]$. But other traversal orders can be specified: right-to-left, odd-positioned before even-positioned...

What does *traverse* do when the applicative functor is *Const* applied to the monoid of integers under addition?