Homework 5

Declaring Monad instances

Due: Wednesday 2023-10-11, 10:00AM

Read the blog post *Functors, Applicatives, and Monads — in pictures.*
https://www.adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html

Assignment

Provide a `Monad` instance for each of the following type constructors

1. "Safe" computations that either return a value of type `a` or fail with an informative error message:

   ```
   data Safe a = Value a | Error String
   ```

2. The type of *finitely branching* trees. In these trees, all the data occur on the leaves, but the nodes can have any number of children.

   ```
   data FTree a = Leaf a | Node [FTree a]
   ```

3. The type of logical propositions (boolean expressions) with variables of type a:

   ```
   data Prop a = PVar a | Const Bool | And (Prop a) (Prop a)
               | Or  (Prop a) (Prop a) | Not (Prop a) | Iff (Prop a) (Prop a)
   ```

4. The type of lambda terms using the "nested type" encoding.

   ```
   data Lam a = Var a | App (Lam a) (Lam a) | Abs (Lam (Maybe a))
   ```

   You should think of this datatype as representing binary trees with data only at the leaves, but which have one more special constructor for *unary* nodes.

   Every time a unary node is encountered, the number of possible data at the leaves below that node *increases by one.*

   ```
   lam :: Lam Bool
   lam = App (Var True) (Abs (App (Var (Just False)) (Var Nothing)))
   ```

Hint. First implement the following function:

```
lift :: (a -> Lam b) -> Maybe a -> Lam (Maybe b)
```

This function will help you fix a type mismatch that will arise in the `Abs` case when you are writing the `bindLam` function.

5. The type of polynomials with variables of type a.

   A polynomial is either a monomial — which is just a floating-point coefficient together with a list of variables to be all multiplied together — or else it is a sum of polynomials.

```
data Poly a = Mono Double [a] | Sum (Poly a) (Poly a)

poly1 = Mono 1.0 []                                       -- 1
poly2 = Mono 2.0 ["X", "X"]                               -- 2x^2
poly3 = Sum (Mono 3.0 ["Y", "Y"]) (Mono (-0.5) ["X","Y"])  -- 3y^2-xy/2
```

   Hint. This is by far the most difficult problem, and will require you to relate the elements of the datatype to the polynomials they actually represent.

   - The unit of this polynomial type constructor should send a variable `x :: a` to an element of type `Poly a` representing the identity polynomial $p(x) = x$.

     Note that this polynomial is a monomial, with coefficient 1, and a single variable being multiplied.

   - The bind of this polynomial type constructor should substitute the variables $x_1, x_2, x_3, \ldots$ occurring inside some polynomial $p(x_1, x_2, x_3, \ldots)$ with new polynomials $q_1(\vec{y}), q_2(\vec{y}), q_3(\vec{y}), \ldots$.

     For example, if $p(x_1, x_2) = 3x_1 + x_2$, $q_1(y, z) = 2y$, $q_2(y, z) = 1 + 3z^2$, then

```
p      = Sum (Mono 3.0 ["x1"]) (Mono 1.0 ["x2"])
q "x1" = Mono 2.0 ["y"]
q "x2" = Sum (Mono 1.0 []) (Mono 3 ["z","z"])

bindPoly :: (a -> Poly b) -> Poly a -> Poly b
bindPoly q p = Sum (Mono 6.0 ["y"]) (Sum (Mono 1.0 []) (Mono 3.0 ["z","z"]))
```

   In solving this problem, you will need to write a function that multiplies out a list of polynomials. This can be easily achieved via fold, if you first write a function that multiplies out TWO polynomials.

   If you are unsure about how to start, write out a few small examples to understand how the function should behave.

   (For example, $(2x^2+1)*(3y^2 - \frac{xy}{2}) = 6x^2y^2 - x^3y + 3y^2 - \frac{xy}{2}$. Your function should implement this operation using the representation above.)

Example Problem

Provide a `Monad` instance for the following type of *bi-infinite sequences* of type a.

```
data Seq a = Seq (Integer -> a)

seq1 :: Seq Integer -- (...9,4,1,0,1,4,9,...)
seq1 = Seq (\n -> n*n)
seq2 :: Seq String  -- (..."","","","","a","aa","aaa",...)
seq2 = Seq (\n -> take (fromIntegral n) (repeat 'a'))
```

Solution

Here is one possible route to a successful `Monad` instance.

```
instance Functor Seq where
  fmap f (Seq s) = Seq g
    where g n = f (s n)

unitSeq :: a -> Seq a
unitSeq x = Seq (const x)

bindSeq :: (a -> Seq b) -> Seq a -> Seq b
bindSeq f (Seq g) = Seq h where  --       h :: Integer -> b
  h n = case f (g n) of          -- f (g n) :: Seq b
          Seq s -> s n           --       s :: Integer -> b

instance Applicative Seq where
  pure = unitSeq
  fs <*> as = bindSeq (<$> as) fs

instance Monad Seq where
  return = unitSeq
  as >>= f = bindSeq f as
```

Additional Hints

Let `X` $\in$ {`Safe`, `FTree`, `Prop`, `Lam`, `Poly`}.
The `Monad` instance for `X` can be obtained via the following recipe.

1. Implement the unit and the bind for the type constructor `X`:

   ```
   unitX :: a -> X a
   bindX :: (a -> X b) -> X a -> X b
   ```

2. Using the above two functions, declare `Functor`, `Applicative`, and `Monad` instances for `X`, using the following code:

   ```
   instance Functor X where
     fmap f  = bindX (unitX . f)

   instance Applicative X where
     pure    = unitX
     f <*> t = bindX (<$> t) f

   instance Monad X where
     return  = unitX
     t >>= f = bindX f t
   ```

The above recipe reduces the problem of declaring `Monad` instance to implementing `unitX` and `bindX`.

For most examples in this homework, `unitX` is quite trivial, and it usually corresponds to a constructor for datatype `X` that takes a pure value of type `a` as input.

The definition of `bindX` becomes progressively more difficult, however.

- For the `Lam` datatype, you should first implement the `lift` helper function as suggested in the question. That function also has a wrinkle: you will need to find a way to get `Lam (Maybe b)` from something of type `Lam b`.

  Conceptually, what we need to do here is to map the `Just` function (`b -> Maybe b`) over the `Lam` type. But this requires `Lam` to implement the `fmap` AKA `<$>` operator — that is, to be a `Functor` already!

  It is therefore easiest to first define the `Functor` instance for `Lam` directly, and then use that to complete the definition of `lift` and `bindLam`.

- For the `Poly` datatype, the `Mono` case of `bindPoly` will require you to multiply out a list of `Poly b`. You can do this by using a fold, supplied with a binary function to multiply two polynomials. To multiply two polynomials together, you will also need a way to multiply a polynomial by a monomial.

  I therefore suggest you to implement the following functions before `bindPoly`:

  ```
  multMono :: Double -> [a] -> Poly a -> Poly a
  multPoly :: Poly a -> Poly a -> Poly a
  ```