Lab 10 - Monad-based language interpreters apud Philip Wadler

This tutorial explores the use of monads to structure functional programs.

Monads increase the ease with which programs may be modified. They can mimic the effect of impure features such as exceptions, state, and continuations; and also provide effects not easily achieved with such features. The types of a program reflect which effects occur.

As part of the tutorial, a simple interpreter is modified to support various extra features: error messages, state, output, and non-deterministic choice.

Introduction: Shall I be pure or impure?

Pure functional languages, such as Haskell, offer the power of lazy evaluation and the simplicity of equational reasoning. Impure functional languages, such as OCaml or Scheme, offer a tempting spread of features such as state, exception handling, or continuations.

One factor that should influence my choice is the ease with which a program can be modified. Pure languages ease change by making manifest the data upon which each operation depends. But, sometimes, a seemingly small change may require a program in a pure language to be extensively restructured, when judicious use of an impure feature may obtain the same effect by altering a mere handful of lines.

Say I write an interpreter in a pure functional language. To add error handling to it, I need to modify the result type to include error values, and at each recursive call to check for and handle errors appropriately. Had I used an impure language with exceptions, no such restructuring would be needed.

To add an execution count to it, I need to modify the the result type to include such a count, and modify each recursive call to pass around such counts appropriately. Had I used an impure language with a global variable that could be incremented, no such restructuring would be needed.

To add an output instruction to it, I need to modify the result type to include an output list, and to modify each recursive call to pass around this list appropriately. Had I used an impure language that performed output as a side effect, no such restructuring would be needed.

Or I could use a monad.

This tutorial walks you through using monads to structure an interpreter so that the changes mentioned above are simple to make. In each case, all that is required is to redefine the monad and to make a few local changes. This programming style regains some of the flexibility provided by various features of impure languages. It also may apply when there is no corresponding impure feature.

Though this tutorial concentrates on the use of monads in a program tens of lines long, it also sketches our experience using them in a program three orders of magnitude larger.

Basic (monadic) interpreter

Let us start with a simple interpreter for lambda calculus.

Abstract syntax

A term is either a variable, a constant, a sum, a lambda expression, or an application.

```
type Name = String

data Term = Var Name
          | Con Integer
          | Term :+: Term
          | Lam Name Term
          | App Term Term
  deriving (Show)
```

The interpreter has been kept small for ease of illustration. It can easily been extended to deal with additional values, such as booleans, pairs, and lists; and additional term forms, such as conditional and fixpoint.

The following will serve as test data.

```
term0 = (App (Lam "x" (Var "x" :+: Var "x")) (Con 10 :+: Con 11))
```

In more conventional notation this would be written `((\ x -> x + x) (10 + 11))`. The value corresponding to the evaluation of `term0` is "42".

Values

A value is either `Wrong`, a number, or a function. The value `Wrong` indicates an error, such as an unbound variable, an attempt to add non-numbers, or an attempt to apply a non-function.

```
  data Value = Num Integer
             | Fun (Value -> M Value)
             | Wrong

  instance Show Value where
    show (Num x) = show x
    show (Fun _) = ""
    show Wrong   = ""
```

What is a monad?

For our purposes, a monad is a triple `(M,return,>>=)` consisting of a type constructor `M` and a pair of polymorphic functions.

```
return :: a -> M a
(>>=) :: M a -> (a -> M b) -> M b
```

The basic idea in converting a program to monadic form is this: a function of type `a -> b` is converted to one of type `a -> M b`. Thus, in the definition of `Value`, functions have type `Value -> M Value` rather than `Value -> Value`, and the interpreter function `interp` has type `Term -> Environment -> M Value` rather than type `Term -> Environment -> Value`.

Similarly for the auxiliary functions `lookupEnv`, `add`, and `apply`.

```
  type Environment = [(Name, Value)]

  interp :: Term -> Environment -> M Value
  interp (Var x) env = lookupM x env
  interp (Con i) _ = return (Num i)
  interp (t1 :+: t2) env = do
    v1 <- interp t1 env
    v2 <- interp t2 env
    add v1 v2
  interp (Lam x e) env = return $ Fun $ \ v -> interp e ((x,v):env)
  interp (App t1 t2) env = do
```

```
    f <- interp t1 env
    v <- interp t2 env
    apply f v

lookupM :: Name -> Environment -> M Value
lookupM x env = case lookup x env of
    Just v  -> return v
    Nothing -> return Wrong

add :: Value -> Value -> M Value
add (Num i) (Num j) = return (Num (i + j))
add _ _             = return Wrong

apply :: Value -> Value -> M Value
apply (Fun k) v = k v
apply _ _       = return Wrong

test :: Term -> String
test t = showM $ interp t []
```

The identity function has type `a -> a`. The corresponding function in monadic form is `return`, which has type `a -> M a`. It takes a value into its corresponding representation in the monad.

Consider the case for constants.

```
interp (Con i) _ = return (Num i)
```

The expression `(Num i)` has type `Value`, so applying `return` to it yields the corresponding `M Value`, as required to match the type of `interp`.

For the more interesting cases we use the `do` notation. Consider the case for sums.

```
  interp (t1 :+: t2) env = do
    v1 <- interp t1 env
    v2 <- interp t2 env
    add v1 v2
```

This can be read as follows: evaluate `t1`; bind `v1` to the result; evaluate `t2`; bind `v2` to the result; add `v1` to `v2`.

Application is handled similarly; in particular, both the function and its argument are evaluated, so this interpreter is using a call-by-value strategy.

Just as the type Value represents a value, the type M Value can be thought of as representing a computation. The purpose of `return` is to coerce a value into a computation; the purpose of `>>=` is to evaluate a computation, yielding a value.

Informally, `return` gets us into a monad, and `>>=` gets us around the monad. How do we get out of the monad? In general, such operations require a more ad hoc design. For our purposes, it will suffice to provide the following.

```
showM :: M Value -> String
```

This is used in `test`.

By changing the definitions of `M`, `return`, `>>=`, and `showM`, and making other small changes, the interpreter can be made to exhibit a wide variety of behaviours, as will now be demonstrated.

Variation zero: Standard interpreter

To begin, we will use the trivial (Control.Monad.Identity) monad.

```
type M a = Identity a
showM ma = show (runIdentity ma)
```

`Identity` encapsulates the identity function on types, `return` is the identity function, `>>=` is postfix application, and `showM` extracts the value and shows it. Simplifying in the above definition we obtain the standard meta-circular interpreter for lambda calculus:

```
interp :: Term -> Environment -> Value
interp (Var x) e = lookup x e
interp (Con i) e = Num i
interp (u :+: v) e = add (interp u e) (interp v e)
interp (Lam x v) e = Fun (\ a -> interp v ((x,a):e))
interp (App t u) e = apply (interp t e) (interp u e)
```

The other functions in the interpreter simplify similarly.

For this variant of the interpreter, evaluating `test term0` returns the string `"42"`, as we would expect.

Exercises: Monadic Variations

For each of the following exercises (variations), copy `var0.hs` as `varn.hs` and update it as the exercise requires.

Variation one: Partial evaluation

Instead of using the `Wrong` value to record failed evaluations, define `M` as the `Maybe` monad and use `Nothing` to record failed evaluations.

You need to remove `Wrong` and its aparitions and need to redefine `showM`, and change the definitions of `lookupM`, `add`, and `apply`.

Variation two: Error messages

To improve the error messages, use the `Either` monad.

The `showM` function must display either the successful result introduced by "Success: ", or the error message introduced by "Error: ".

To modify the interpreter, substitute monad E for monad M, and replace each occurrence of `return Wrong` by a suitable `Left` expression. The only occurrences are in `lookupM`, `add`, and `apply`. No other changes are required. The error messages should be:

- unbound variable: < name >
- should be numbers: < v1 >, < v2 >
- should be function: < v1 >

Evaluating `test term0` should return "Success: 42"; and evaluating

```
test (App (Con 7) (Con 2))
```

should return "Error: should be function: 7".

In an impure language, this modification could be made using exceptions or continuations to signal an error.

Variation three: State

To illustrate the manipulation of state, the interpreter is must be modified to keep count of the number of reductions that occur in computing the answer. The same technique could be used to deal with other state-dependent constructs, such as extending the interpreted language with reference values and operations that side-effect a heap.

The monad of state transformers is the State monad.

A state transformer takes an initial state and returns a value paired with the new state. The `return` function returns the given value and propagates the state \ unchanged. The `>>=` function takes a state transformer `ma :: M a` and a function `k :: a -> M b`. It passes the initial state to the transformer `ma`; this yields a value paired with an intermediate state; function `k` is applied to the value, yielding a state transformer `(k a :: S b)`, which is passed the intermediate state; this yields the result paired with the final state.

To model execution counts, take the state to be an integer.

```
type M a = State Integer a
```

The `showM` should use the function `runState` with the initial state 0 and should prints the final value and counter.

Evaluating `test term0` should return "Value: 42; Count: 3".

To achieve this, define the computation to increase the counter:

```
tickS :: M ()
```

and modify functions `add` or `apply` using `tickS` to increase the counter for each call to them.

You can either use the `state :: (s -> (a,s)) -> M a` function to embed a simple state action into the monad, or the `modify :: (s -> s) -> M ()` to map an old state to a new state inside the state monad.

A further modification extends the language to allow access to the current execution count.

First, define a function to return the current count.

```
fetchS :: S s
```

You can either use the `state :: (s -> (a,s)) -> M a` function or the `get :: M s` which returns the state from the internals of the monad.

Second, extend the term data type, and add a new term `Count`.

Third, define `interp` for `Count` to fetches the number of execution steps performed so far, and returns it as the (`Num`) value of the term. For example, applying `test` to

```
(Con 1 :+: Con 2) :+: Count
```

should return "Value: 4; Count: 2", since only one addition occurs before `Count` is evaluated.

In an impure language, these modifications could be made using state to contain the count.

Variation four: Output

Next we modify the interpreter to perform output. The state monad seems a natural choice, but it's a poor one: accumulating the output into the final state means no output will be printed until the computation finishes.

Instead, we will use the Writer monad.

```
type M a = Writer String a
```

The `Writer` monad behaves as follows. Each value is paired with the output produced while computing that value. The `return` function returns the given value and produces no output. The `>>=` function performs an application and concatenates the output produced by the argument to the output produced by the application.

Modify `showM` function to print the output followed by the value. You can use the `runWriter :: M a -> (a, w)` to extract the value and the output from a computation.

Extend the language an output operation, by adding a new term `Out Term`.

Evaluating `Out u` causes the value of `u` to be sent to the output, and also returned as the value of the term.

For example, applying `test` to

```
Out (Con 41) :+: Out (Con 1)
```

should return "Output: 41; 1; Value: 42".

In an impure language, this modification could be made using output as a side effect.

Variation five: Non-deterministic choice

We now modify the interpreter to deal with a non-deterministic language that returns a list of possible answers.

To do so, we will use the monad of lists.

```
type M a = [a]
```

Extend the interpreted language with two new constructs: `Fail` and `Amb Term Term`.

Evaluating `Fail` should return no value, while evaluating `Amb u v` should return all values returned

by either `u` or `v`. Extend `interp` to achieve this semantics.

For example, applying `test` to

```
App (Lam "x" (Var "x" :+: Var "x")) (Amb (Con 1) (Con 2))
```

should return "[2,4]".

It is more difficult to see how to make this change in an impure language. Perhaps one might create some form of coroutine facility.

Source: The essence of functional programming