1 Monad Review

In category theory a monad is a triplet, (C, η, μ), containing an endofunctor over categories along with two natural transformations. The endofunctor over categories, C, takes an input of a value in some category (for example, integers in the category of sets), and gives an output that is also in that category, therefore, it has type $C : T \to T$. The two transformations it must define are a unit function that maps a value into the monad (known as η, with type $I_C \to T$), and a multiplication function, known as μ, which goes from a repeated application of the endofunctor to a single, represented as having type $T^2 \to T$ or $T \circ T \to T$. These transformations must be defined in such a way that they follow certain coherence properties.

This concept has been adopted by functional programming language designers as a way of decorating certain data types. These decorations may be things like the `Maybe` monad, which extends any type to be a category with the option of being `Nothing`. In some functional languages, most notably Haskell, they are also used to augment series execution of functions, to indicating if these functions are allowed to have "impure" side effects, such as I/O. This allows Haskell programmers to separate functions whose output will always be the same as for a given input (referentially transparent) from ones whose output may vary (such as reading from an input buffer). Eugenio Moggi is the Italian researcher who put down the framework for the use of Monads in a functional language, as was discussed at the Cornell Programming Languages Discussion Group last year.

1.1 Monads in Haskell

The papers that we are going to be discussing today are both based on the idea of augmenting Haskell with concurrency monads. Therefore, it will be helpful to review how monads are implemented and how they are used in Haskell.

In Haskell a monad is a kind of typeclass, a structure not entirely different from a Java Interface, where a type specification is given for several functions that anyone that would like to be members of this typeclass must satisfy. Additionally, a Haskell typeclass may define default implementations of the functions, such as in the case of the `Eq` typeclass, where not equals is simply defined as the boolean negation of equals..

The type specification for a Haskell monad is:

```
class Monad m where
  (>>=) :: m a -> ( a -> m b) -> m b
  return :: a -> m a
```

The first operator, `(>>=)` is known as the "bind" operator. It takes an argument of type `a` wrapped in the monad `m`, a function which maps from values of type `a` to values of type `b` wrapped in monad `m`, and returns a value of type `b` wrapped in the monad `m`. For example, in the set monad, this would let you take a set (like, $\{1, 5\}$) take a function such as `f x = {x, 2x}`, and would generate an output of $\{1, 2, 5, 10\}$.

This is the operator that is used for composing multiple monadic actions in series. It is analogous to the μ mapping from category theory monads.

The second operator, `return`, takes a value of type `a` and wraps it in monad `m`. For sets of integers, it would take a single integer and return a set (e.g. `return 1` would result in $\{1\}$) It is analogous to the η operator.

1.2 Examples

An example used in the first paper discussed below is the `writer` monad. It decorates the result of some computation with a string, which could be used for logging actions. In the paper it is first defined as a monadic typeclass that defines a function `write`, which moves a `String` into the monad.

```
class Monad m => Writer m where
  write :: String -> m ()
```

To use the `writer` monad, we need to create a type which is an instance of this typeclass. This instance must also be an instance of the `monad` typeclass. In the paper, this instance was called `W`.

```
type W a = (a, String)

instance Monad W where
  (a, s) >>= k = let (b, s') = k a in (b, s++s')
  return x     = (x, "")

instance Writer W where
  write s = ((), s)
```

What the above code does is to declare a new type named `W` that takes an argument of some arbitrary type `a`, and creates a type `(a, String)`. It then makes this type an instance of the `monad` typeclass, where the bind operator when given a pair of some value `(a, s)` and a function `k` applies `k` to the value `a`, extracts the result as `(b, s')`, and then returns `(b, s++s')`, where `++` is the Haskell concatenation operator.

The return operator is much simpler, simply taking in some value `x`, and returning a pair with that value and the empty string. This type `W` is also made into an instance of the `Writer` typeclass, where the `write` function returns a pair with the unit type, `()`, and the input string.

To use this monad we need one additional construct: a *run* function. For the writer monad, this consists of extracting the string and tossing away the value that was computed. In Haskell, this is not atypical, as we're often more interested in a monad's side effects than the actual value that it contains. Here, that function is called `output` and is defined as:

```
output :: W a -> String
output (a, s) = s
```

This monad can then be used to compose several functions together, and make notes about their intermediate state. This can be done in two separate ways. Most directly, the bind operator can be used to string several functions together. However, more readably, you can use Haskell *do* syntax, where statements can contain bindings to variables which will be passed to subsequent statements.

For example, the following two code segments represent equivalent functionality, but the one written in *do* notation looks more natural, and does not require every line to be made into an anonymous function (done here with the backslash operator, a Haskell operator chosen since it looks somewhat similar to λ, but can be typed by a standard keyboard).

```
expr1 >>= \x ->          do x <- expr1
expr2 >>= \_ ->             ; expr2
expr3 >>= \y ->             ; y <- expr3
return expr4                ; return expr4
```

2 Functional Pearls: A Poor Man's Concurrency Monad

This paper describes the implementation of a concurrency monad entirely within the language of Haskell without the addition of any language features. In the paper, the bind operator, `>>=` is represented by $\star$, however in these notes, I will be representing it as `>>=` for the sake of consistency with the way Haskell is typically input.

The "Functional Pearls" in the title of this paper refers to a class of paper. These papers are designed to be accessible and well-written documents discussing an interesting technique or methodology. They are not required to be ground-breaking when they are published, but due to the cleverness of the ideas that they discuss, and the level of accessibility they are designed for, they are almost always worth reading.

To build the concurrency monad, this paper creates a Monad transformer. A monad transformer takes in a monad of one type, and then creates a monad of another type. For instance, if you wanted to take a value you got from an I/O operation, and perform an analysis on it which might fail, you could do this by creating a monad that has the properties of both the `IO` monad and the `Maybe` monad. In this paper, we'll be discussing a monad transformer called `C`, which allows any monadic action lifted into it to be considered as atomic in concurrent operations.

As the author of this paper decided he would not add any primitives to the language, the model of concurrency created does not actually exhibit actual concurrency, but instead interleaves the execution of several monadic operators. To do this, it needs to have some way to suspend the operation. This is being done with continuations, where each computation is given a place to go once its computation is done, and can postpone going to that location. We can think of the continuation as the "future" of the computation. To think of that in the typesystem, if we have some computation with the type of `Action m` (where `m` is some monad), a function that uses a continuation with result type `a` has the type:

```
type C m a = ( a -> Action m ) -> Action m
```

Given that, the monad transformer `C` is defined as:

```
instance Monad m => Monad (C m) where
  f >>= k    = \c -> f ( \a -> k a c)
  return x   = \c -> c x
```

To understand this, we can look at the types of the different operators and think about the kinds of types we would like to get out.

The simplest one is first, the return operator. We would like to give it a value of some type `a`, and have it return a type of `C m a`. To do this, all we have to do is to take a continuation and call it.

The bind operator is much more complicated. We'd like to take a value of `C m a` (or `(a -> Action m) -> Action m`)), a function of type `a -> C m B` (or `a -> ( B -> Action m) -> Action M`), and return a value of `C m B`, (or `(B -> Action m) -> Action m`)). To do this, we'll be using some anonymous functions to take in continuations and apply them to actions in order to expose their values. Below is the bind operator with type annontations:

```
f:((a -> Action m)->Action m) >>= k:(a->(B->Action m)->Action m)) =
    \c : ( B -> Action m ) -> f ( \a : (a) . k a c )
```

The paper then defines the specifics of the `Action` operator type, which is one of an Atomic action inside the monad (`Atom`), a fork operation, which forks off two different actions (`Fork`) and a stop operation, which ends the current thread of execution (`Stop`). These three different kinds of actions provide all of the primitives needed for a basic implementation of concurrency.

Because the concurrency design in this paper isn't actual parallel concurrency made with separate threads, it requires a scheduler in the language to take `Actions` and execute them one at a time. The example given in the paper is a round-robin scheduler, where a stack of executable processes is maintained, with a single element being popped off of it at a time. If an Atomic action is popped off, it is allowed to execute one step, and then its continuation is put back on the bottom of the stack. If a fork action is popped off, it is split into two actions, which are placed on the bottom. When a stop action is popped off, it is simply allowed to die.

In the paper, the round robin scheduler is used in conjunction with the writer monad, to provide for interleaved output. Two examples are shown, one where doing the entire printing action is lifted as an atomic action (and hence, the interleaving happens at the string level), and another where each character print is lifted as an atomic action, causing the interleaving to happen at that level.

3 Combining Events and Threads for Scalable Network Services

This paper is an extension of the work from the previous paper, however, its implementation of a concurrency monad uses language features to allow for actual parallel execution. It continues to use a cooperative execution model, and use lazy execution of continuations to allow for interruption.

This paper claims that the model of concurrency it uses elegantly bridges the gap between thread-based and event-based parallelism. This is the same goal that the Actor abstraction discussed on the 20th of October had. Thread-based execution is typically considered to interface with shared memory, and have no control over when preemption may occur. Event-based parallelism must set up waits for each action that it would like to respond to, and wait for that event to occur before they run, which may make reasoning about control flow difficult. This paper proposes a hybrid model that allows programmers to write in the somewhat more natural threaded style, but with the thread scheduler allowing for event handlers.

The reason the paper cites for choosing the hybrid model is actually the similarity of the two models. In 1978 Lauer and Needham described a mapping between Threads and Events, which argued that fundamentally, they are the same, though they may be interfaced with and implemented differently. During class discussion, there were some disagreements with the exact duality, but some of the points still stood. In the paper, they chose to hybridize by using thread abstractions to represent control flow, and event abstractions to provide scheduling control. In particular, they used continuation-passing-style (CPS) event handlers to blend control and scheduling flow. This model was implemented in Haskell using monadic operators in a similar way to the Claessen paper above. The main difference is the scheduler allows for true parallelism, instead of just interleaving of operations. Exceptions are handled through the continuations.