Lecture 19: Monads

Brianna Marshall

CS4400/5400 Fall 2024

1 Introduction

We saw mutable state in lecture 16 and exceptions in lecture 18. Both features are a form of effect. In both cases, the implementation involved adding a new expression that, as part of being evaluated, implicitly does something to the machine's state, like changing memory or redirecting control flow. This is in addition to, or instead of, the value returned by the expression.

But that isn't the only way. It's possible to implement mutable state and exceptions using only purely functional language features that we're already familiar with. Doing it this way has a number of advantages. In particular, we avoid all of the problems described in the section "Some Consequences of Mutation" in the Lecture 16 Notes. It also leads directly to a powerful abstraction with broader consequences for the design of programming languages in general: monads.

A clue for how to pull this off lies in the implementation of our interpreters for MutLang and ExnLang (see the Lecture 16 and Lecture 18 Notes). The interpreters are actually pure functions, even though they are interpreting impure languages. We can apply similar techniques inside of the languages themselves.

This is going to be a very practical approach to monads. We will go through a series of progressively more complicated examples written in Haskell, culminating in the definition of a monad and a demonstration of what you can do with code written to be generic over any monad.

2 Haskell crash course

Haskell is a typed functional programming language similar to OCaml, but it has some language features that are especially useful for monads, which we will use later. Let's briefly go over some of the basics.

2.1 Functions

A top-level function is declared like this:

```
foo x y z = x + y * z
```

The name of the function is `foo` and its parameters are `x`, `y`, and `z`. The body is the expression `x + y * z` which follows the equals sign.

Top-level functions usually include a type signature, which is written on its own line before the definition:

```
add1 :: Int -> Int
add1 x = x + 1
```

The double colon `::` indicates a type annotation and is read "`add1` has type `Int -> Int`."

Anonymous functions (lambda expressions) start with a backslash `\`, which looks like a λ with a missing leg. This is another way to write the same `add1` function:

```
add1 = \x -> x + 1
```

2.2 Types

Algebraic data types are declared with the `data` keyword. Each data type has zero or more constructors separated by a vertical bar `|`; each constructor has zero or more fields with types separated by spaces. For example, this is a sum type that contains either a pair of integers or a boolean:

```
data IntPairOrBool = Left Int Int | Right Bool
```

Types can depend on other types. For example, a generic version of `IntPairOrBool` could look like:

```
data Either a b = Left a | Right b
```

where `a` and `b` are type variables. The type can then be *instantiated* with type arguments:

```
leftOrRight :: Either (Int, Int) Bool
leftOrRight = Left (1, 2)
```

2.3 Pattern matching

Sum types can be pattern matched against using the `case` expression:

```
add1OrNot :: IntPairOrBool -> IntPairOrBool
add1OrNot v = case v of
  Left x y -> Left (x + 1) (x + 1)
  Right b -> Right (not b)
```

2.4 Comments

Comments start with two dashes `--` and go until the end of the line.

```
-- Increment the given integer by one.
add1 :: Int -> Int
add1 x = x + 1
```

3 Functions that fail

Let's start with something familiar: an interpreter for a simple language with booleans, numbers, if-then-else, and let-bindings. The AST for the language is defined as follows:

```haskell
data Expr
  = EVar String
  | EBool Bool
  | ENum Int
  | ELet String Expr Expr
  | EIf Expr Expr Expr
  | EAdd Expr Expr

data Value = VBool Bool | VNum Int
```

3.1 Error-returning style

We will implement an environment-based interpreter, so the `interp` function needs to take an environment and the expression to interpret and return a value. As usual, we need to pattern match on the expression. This will look something like:

```haskell
interp :: Map String Value -> Expr -> Value
interp env expr = case expr of
  EVar x -> _
  EBool b -> _
  ENum n -> _
  ELet x e1 e2 -> _
  EIf e1 e2 e3 -> _
  EAdd e1 e2 -> _
```

Right away, we run into an issue when trying to implement the `EVar` case:

```haskell
  EVar x -> Map.lookup x env
```

The compiler tells us: "Couldn't match expected type 'Value' with actual type 'Maybe Value'."

`Map.lookup` is like Racket's `hash-ref`. However, while `hash-ref` raises an exception when the key isn't found, `Map.lookup` returns a special value instead. The type `Maybe a` is defined:

```haskell
data Maybe a = Nothing | Just a
```

When `Map.lookup` returns `Maybe Value`, it means there could be a value, but there might not be. We need to use pattern matching to find out. While Racket lets us ignore failures, letting them implicitly bubble up as exceptions, Haskell forces us to think about how to handle each potential failure case.

We can anticipate that there are several ways for an interpreter of this language to fail:

1. An unknown variable is used.
2. One of the subexpressions of `EAdd` evaluates to a boolean.
3. The first subexpression of `EIf` evaluates to a number.

Let's define a sum type to indicate whether the interpreter succeeded or failed, and another sum type to indicate which kind of failure happened. The return type of `interp` also needs to change accordingly. We will call this *error-returning style* (in contrast to *exception style*) because the error is explicitly returned as a value.

```
data Result e a = Err e | Ok a
data EvalError = UnknownVar | ExpectedNum | ExpectedBool

interp :: Map String Value -> Expr -> Result EvalError Value
```

The `EVar` case now involves pattern matching on the return value of `Map.lookup` and returning the corresponding constructor for `Result`:

```
  EVar x -> case Map.lookup x env of
    Nothing -> Err UnknownVar
    Just v -> Ok v
```

The `EBool` and `ENum` cases are straightforward: they can never fail.

```
  EBool b -> Ok (VBool b)
  ENum n -> Ok (VNum n)
```

Our first recursive case is `ELet`. We need to recursively call `interp` on `e1`, but this could fail. If that happens, the only thing to do is propagate the error up, since there won't be a value to use. This is expressed by the `Err e -> Err e` case below, and it mimicks the behavior of exceptions.

```
  ELet x e1 e2 -> case interp env e1 of
    Err e -> Err e
    Ok v1 ->
      let env' = Map.insert x v1 env
       in interp env' e2
```

The `EIf` case follows similarly, but there's an additional wrinkle that `e1` needs to evaluate to a boolean. If calling `interp` on `e1` succeeds, we case on the returned value, and if it's not a `VBool`, we return an `ExpectedBool` error.

```
  EIf e1 e2 e3 -> case interp env e1 of
    Err e -> Err e
    Ok v1 -> case v1 of
      VNum _ -> Err ExpectedBool
      VBool b -> if b then interp env e2 else interp env e3
```

Finally, `EAdd` requires both of its subexpressions to evaluate to numbers. We use another, larger sequence of cases to handle failures one-by-one.

```
  EAdd e1 e2 -> case interp env e1 of
    Err e -> Err e
    Ok v1 -> case v1 of
      VBool _ -> Err ExpectedNum
      VNum n1 -> case interp env e2 of
        Err e -> Err e
        Ok v2 -> case v2 of
          VBool _ -> Err ExpectedNum
          VNum n2 -> Ok (VNum (n1 + n2))
```

3.2 Bubbling up errors

Did you notice that `Err e -> Err e` showed up every time we made a recursive call? As mentioned, this mimicks the behavior of exceptions, which "bubble up" the call stack until someone handles the exception. We can factor out this programming pattern into a separate function that automatically returns `Err e` if the `Result` is `Err e`, but runs a function on the `Ok` value otherwise.

```
andThen :: Result e a -> (a -> Result e b) -> Result e b
andThen r f = case r of
  Err e -> Err e
  Ok x -> f x
```

It can then be used like:

```
andThen (foo x) (\y -> Ok (y + 1))
```

where `foo x` is a function call that returns a `Result`, and `\y -> Ok (y + 1)` is a lambda expression that is called on a successful value and returns another `Result`. We can rewrite this to look a bit more English-like by using another Haskell language feature: infix function calls. Surrounding the function name in backticks ` means that we can place it *between* its two arguments instead of before:

```
foo x `andThen` \y -> Ok (y + 1)
```

3.3 Booleans and numbers

Another repeated operation is extracting a boolean or number from a value, and failing if the value isn't the right type. In our Racket interpreters, we sometimes defined `to-bool` and `to-num` helper functions that raised an exception if they couldn't convert the value to the expected type. We can write something similar in Haskell, but returning an `Err` instead of raising an exception:

```
toBool :: Value -> Result EvalError Bool
toBool v = case v of
  VNum _ -> Err ExpectedBool
  VBool b -> Ok b

toNum :: Value -> Result EvalError Int
toNum v = case v of
  VBool _ -> Err ExpectedNum
  VNum n -> Ok n
```

3.4 Refactoring the interpreter

After refactoring our interpreter to use the three helper functions defined above, it now looks like this:

```
interp :: Map String Value -> Expr -> Result EvalError Value
interp env expr = case expr of
  EVar x -> case Map.lookup x env of
    Nothing -> Err UnknownVar
```

```
    Just v -> Ok v
  EBool b -> Ok (VBool b)
  ENum n -> Ok (VNum n)
  ELet x e1 e2 ->
    interp env e1 `andThen` \v1 ->
      let env' = Map.insert x v1 env
       in interp env' e2
  EIf e1 e2 e3 ->
    interp env e1 `andThen` toBool `andThen` \b ->
      if b then interp env e2 else interp env e3
  EAdd e1 e2 ->
    interp env e1 `andThen` toNum `andThen` \n1 ->
      interp env e2 `andThen` toNum `andThen` \n2 ->
        Ok (VNum (n1 + n2))
```

4 Purifying state

In Lecture 14, we implemented a compiler from TinyCalc to MicroASM. We used a function called `fresh` to get a new memory address where we could store the result of an expression:

```
(define counter (box 0))
(define (fresh)
  (define cur (unbox counter))
  (set-box! counter (+ 1 cur))
  cur)
```

Notice how the implementation uses a mutable box to store the current address. Let's look at how this can be done in Haskell without using mutation. The expression and instruction languages we will be using are defined like this:

```
data Expr
  = ENum Int
  | EAdd Expr Expr

type Addr = Int
type Reg = Int
type Value = Int

data Inst
  = ISet Reg Value
  | IStore Reg Addr
  | ILoad Reg Addr
  | IAdd
  | IHalt
```

(`type` declares a type alias; `Addr`, `Reg`, and `Value` are just other names for the `Int` type.)

4.1 State-passing style

Let's break down everything that `fresh` is doing:

1. Get the current value of `counter`.
2. Set the next value of `counter`.
3. Return the fresh address.

Without a mutable variable that we can perform side effects on, our only choice to describe these actions is through the input and output of the function. If we can't get the current value of a mutable variable, let's ask for it instead by adding a function parameter. Similarly, if we can't set the value of a mutable variable, let's tell the caller what we would have done by returning an extra value. The first component of the pair is the new value of the variable and the second component of the pair is the requested fresh address.

```
fresh :: Addr -> (Addr, Addr)
fresh counter = (counter + 1, counter)
```

This is called *state-passing style*. In our compiler, we need to carefully thread through the latest value of the counter, which changes after every recursive call or call to `fresh`:

```
compile :: Expr -> Addr -> (Addr, ([Inst], Addr))
compile expr counter = case expr of
  ENum n ->
    let (counter', addr) = fresh counter
        insts = [ISet 0 n, IStore 0 addr]
     in (counter', (insts, addr))
  EAdd e1 e2 ->
    let (counter', (insts1, addr1)) = compile e1 counter
        (counter'', (insts2, addr2)) = compile e2 counter'
        (counter''', addr) = fresh counter''
        insts = [ILoad 1 addr1, ILoad 2 addr2, IAdd, IStore 0 addr]
     in (counter''', (insts1 ++ insts2 ++ insts, addr))
```

4.2 The `State` type

Let's define how this "mutable" variable works a bit more formally. Everything that uses the variable needs to follow the pattern of taking in the current value and returning a pair of the new value and the result. We can define the type `State s a` as a function that threads the state of the variable through itself, where `s` is the type of the state and `a` is the type of the result.

```
newtype State s a = State {runState :: s -> (s, a)}
```

(`newtype` is basically the same as `data`; `runState` is the name of a field that has type `s -> (s, a)`.)

There are two operations you can do with a mutable variable: get the current value and set the current value. We can define those in terms of `State`:

```
get :: State s s
get = State (\c -> (c, c))

put :: s -> State s ()
put c = State (\_ -> (c, ()))
```

`get` leaves the value of the variable unchanged (the first `c` in `(c, c)`) and also returns it as the result (the second `c` in `(c, c)`. `put` *ignores* the current value of the variable and instead replaces it with the value given to it; it returns nothing interesting (`()` is the unit type and value).

As a technicality, we also need a way to create a `State` computation that simply returns a normal value without getting or setting the variable:

```
yield :: a -> State s a
yield x = State (\c -> (c, x))
```

We would also like to avoid having to manually thread the latest value of the variable through our computation; we just care about the result values and would prefer the current state to be implicitly updated behind the scenes, like a real mutable variable. We can define a helper function that takes in a `State` and a function to call on the result value, which returns another `State`:

```
andThen :: State s a -> (a -> State s b) -> State s b
andThen s f =
  State
    ( \c ->
        let (c', x) = runState s c
         in runState (f x) c'
    )
```

(`runState s c` is equivalent to `(runState s) c`; first it accesses the `runState` field of `s`, which is a function that takes in the current value of the variable, then it applies that function to `c`.)

Now we can rewrite `fresh` to use these new operators:

```
fresh :: State Addr Addr
fresh =
  get `andThen` \counter ->
    put (counter + 1) `andThen` \() ->
      yield counter
```

Then we can rewrite `compile`:

```
compile :: Expr -> State Addr ([Inst], Addr)
compile expr = case expr of
  ENum n ->
    fresh `andThen` \addr ->
      let insts = [ISet 0 n, IStore 0 addr]
       in yield (insts, addr)
  EAdd e1 e2 ->
    compile e1 `andThen` \(insts1, addr1) ->
      compile e2 `andThen` \(insts2, addr2) ->
```

```
        fresh `andThen` \addr ->
          let insts = [ILoad 1 addr1, ILoad 2 addr2, IAdd, IStore 0 addr]
           in yield (insts1 ++ insts2 ++ insts, addr)
```

Finally, we can use `runState` to kick off a stateful computation starting from an initial value of the variable (`0` in this case), then discard the final value when we're done with it.

```
compileHalt :: Expr -> [Inst]
compileHalt expr =
  let (_, (insts, addr)) = runState (compile expr) 0
   in (insts ++ [ILoad 0 addr, IHalt])
```

5 The monad

Take another look at the two `andThen` functions we defined earlier:

```
andThen :: Result e a -> (a -> Result e b) -> Result e b
andThen r f = case r of
  Err e -> Err e
  Ok x -> f x

andThen :: State s a -> (a -> State s b) -> State s b
andThen s f =
  State
    ( \c ->
        let (c', x) = runState s c
         in runState (f x) c'
    )
```

Although their implementations are completely different, their types are almost identical. Where the first one uses `Result e`, the second one uses `State s`.

The other thing to notice is that both `Result` and `State` have a way of taking any value of type `a` and wrapping it in something of type `Result e a` or `State s a`. For `State`, that was the function `yield`; for `Result`, that was the constructor `Ok`.

It turns out that many types share a structure that allows functions like `andThen` and `yield` to be implemented for them. These types are monads.

5.1 The `Monad` type class

Haskell comes with an abstraction for monads. This is the `Monad` *type class*. A type class is kind of like an interface from object-oriented languages—it's a collection of *methods* for a type—although there are some differences that we won't get into here.

There are two methods that we need to define for a monad `m`:

```
(>>=) :: m a -> (a -> m b) -> m b
pure  :: a -> m a
```

`>>=`, pronounced "bind," is the monad sequencing operator and it corresponds to the `andThen` functions we defined. `pure` converts a "pure" value to a "monadic" value and it corresponds to `yield` and `Ok` from earlier.

A well-behaved monad instance also needs to follow three laws:

- `pure x >>= f` should be the same as `f x`. Intuitively, this is because bind runs `f` on a successful result of the previous monadic computation; if the previous computation simply converted a pure value to a monadic value, we should be able to run `f` on that value directly and skip the conversion.
- `m >>= pure` should be the same as `m`. Intuitively, this is because taking the result value of `m` and converting it back into a monadic value shouldn't change anything.
- `m >>= (\x -> f x >>= g)` should be the same as `(m >>= f) >>= g`. This is associativity: intuitively, it says that it shouldn't matter whether monadic binds are built up left-to-right or right-to-left.

It's possible to verify that the monad implementations for `Result` and `State` satisfy these laws.

5.2 A monad for `Result` and `State`

For technical reasons, `pure` is defined in a separate type class called `Applicative`—we won't go into what `Applicative` is, but it is another type class related to monads. In short, every `Monad` is also an `Applicative`.

The implementations (defined with the `instance` keyword) for our `Result` type are, with some boilerplate omitted:

```
instance Monad (Result e) where
  (>>=) = andThen

instance Applicative (Result e) where
  pure = Ok
```

And the implementations for our `State` type are:

```
instance Monad (State s) where
  (>>=) = andThen

instance Applicative (State s) where
  pure = yield
```

5.3 `do` notation

Now that we have our `Monad` instances, we can do something cool.

Did you notice that the programs written with `andThen` became increasingly indented as a new lambda expression was used each time? This is known as *rightward drift* and it's common for monadic code, because each new step takes place "inside" the previous step, similar to continuation-passing style.

Haskell provides a language feature to avoid rightward drift when using monads: `do` notation. `do` notation is syntactic sugar for successive calls to a monad's bind operator. When you would write:

```
foo `andThen` \x ->
  bar `andThen` \y ->
    pure (x + y)
```

Or equivalently, but more generally:

```
foo >>= \x ->
  bar >>= \y ->
    pure (x + y)
```

You can instead use a `do` expression:

```
do
  x <- foo
  y <- bar
  pure (x + y)
```

The `do` expression is equivalent to the version with `>>=` above, but it looks more like an imperative program with statements.

We can rewrite our interpreter to use `do` notation:

```
interp :: Map String Value -> Expr -> Result EvalError Value
interp env expr = case expr of
  EVar x -> case Map.lookup x env of
    Nothing -> Err UnknownVar
    Just v -> Ok v
  EBool b -> Ok (VBool b)
  ENum n -> Ok (VNum n)
  EAdd e1 e2 -> do
    v1 <- interp env e1
    n1 <- toNum v1
    v2 <- interp env e2
    n2 <- toNum v2
    Ok (VNum (n1 + n2))
  EIf e1 e2 e3 -> do
    v1 <- interp env e1
    b <- toBool v1
    if b then interp env e2 else interp env e3
  ELet x e1 e2 -> do
    v1 <- interp5 env e1
    let env' = Map.insert x v1 env
    interp env' e2
```

And we can rewrite our compiler:

```
fresh :: State Addr Addr
fresh = do
```

```haskell
  next <- get
  put (next + 1)
  yield next

compile :: Expr -> State Addr ([Inst], Addr)
compile expr = case expr of
  ENum n -> do
    addr <- fresh
    let insts = [ISet 0 n, IStore 0 addr]
    yield (insts, addr)
  EAdd e1 e2 -> do
    (insts1, addr1) <- compile e1
    (insts2, addr2) <- compile e2
    addr <- fresh
    let insts = [ILoad 1 addr1, ILoad 2 addr2, IAdd, IStore 0 addr]
    yield (insts1 ++ insts2 ++ insts, addr)
```

6 Same program, different monad

So far, we've only seen programs that are written for a specific monad (`Result` or `State`). But the true power of the monad comes from being able to write code that is generic over *multiple* monads.

Consider the function `genstr`, which generates a string with a given length and alphabet:

```haskell
genstr :: (Monad m, MonadChoice m) => Int -> String -> String -> m String
genstr len alphabet suffix =
  if len <= 0
    then pure suffix
    else do
      c <- choose alphabet
      genstr (len - 1) alphabet (c : suffix)
```

The key here is the use of a new type class that we defined, `MonadChoice`. It contains one method, `choose`, which takes a list of elements and returns a single element (in the monad).

```haskell
class MonadChoice m where
  choose :: [a] -> m a
```

Since `genstr` is generic over the monad it uses, anyone who calls it can choose which monad they would like, as long as it has a `choose` method. The behavior of each monad could potentially be completely different. Basically, the choice of monad can reinterpret the *same* program with *different* semantics!

6.1 Nondeterminism

Let's try giving `genstr` nondeterministic semantics: we want it to collect every possible string that it could generate into one big list. It turns out that a monad with these semantics already exists in Haskell: it's the list type! We only need to define the `choose` method as the identity function:

```haskell
instance MonadChoice [] where
  choose = id
```

Now all we need to do is tell Haskell that we would like `genstr` to give us a list of strings. We can use `::` to indicate that the return type should be `[String]`. In the REPL, this generates all strings of length 3 using the characters `a` or `b`:

```
ghci> genstr 3 "ab" "" :: [String]
["aaa","baa","aba","bba","aab","bab","abb","bbb"]
```

6.2 Sampling

For long strings with many different characters, it would take too long to generate every possible string. There are too many combinations. However, what if we only need need a small number of strings? A monad that randomly *samples* the search space instead of exhaustively enumerating it could generate one string at a time much more quickly.

We'll define the type `Sample` as a function that takes in a pseudorandom number generator (PRNG), then returns the new (possibly changed) state of the PRNG and a result. This type has both a `Monad` and `MonadChoice` instance; the implementation for `choose` uses the PRNG to pick one element of the list.

```haskell
data Sample a = Sample {runSample :: StdGen -> (StdGen, a)}

instance Monad Sample where
  m >>= f =
    Sample
      ( \g ->
          let (g', x) = runSample m g
           in runSample (f x) g'
      )

instance MonadChoice Sample where
  choose xs =
    Sample
      ( \g ->
          let (i, g') = uniformR (0, length xs - 1) g
           in (g', xs !! i)
      )

instance Applicative Sample where
  pure x = Sample (\g -> (g, x))
```

For convenience, we'll also define a `sample` function that initializes the PRNG to a random seed and runs the sample function with it:

```haskell
sample :: Sample a -> IO a
sample s = do
```

```
  g <- initStdGen
  pure (snd (runSample s g))
```

Now we can generate very long strings with many different characters quickly. This would've taken forever to run under the nondeterministic semantics, but under the sampling semantics, it's virtually instant:

```
ghci> sample (genstr 72 ['a' .. 'z'] "")
"yttdqpabyeqixgfixbtvywhyuldqbygbmkjadcmczonqgcptpwmgezemgmzvurzddvnkfogj"
```