CS 115
Functional Programming

Lecture 20: May 18, 2016

State Monads
(part 2)

Previously

- State monads
- The `State` datatype
- `(State s)` as a monad
- The `runState` function

Today

- More on state monads
- The `MonadState` type class
  - the `get` and `put` methods to retrieve/change values in the state being passed around
- Examples using state monads

Recall

- State monads are a way to encapsulate a "state-passing" mode of computation
- Can use state monads to simulate either local or global variables in an imperative-style computation
- Unlike the `IO` monad, can easily exit from a state monad computation using the `runState` function

Recall

- State-passing computations can be written like this (for a given state type `s`):

```
(a, s) -> (b, s)
```

- Or with the first argument curried:

```
a -> s -> (b, s)
```

- The `State` datatype wraps the `s -> (b, s)` part into a constructor:

```
data State s a = State (s -> (a, s))
```

Recall

- Using the **`State`** datatype, state-passing computations are represented as:

```
a -> State s b
```

- For a given state type **`s`**, **`State s`** is a monad

Recall

- The **`Monad`** instance definition for **`(State s)`** is as follows:

```
instance Monad (State s) where
  return x = State (\st -> (x, st))
  mv >>= g =
    State (\st ->
      let (State ff) = f x
          (y, st') = ff st
          (State gg) = g y
      in gg st')
```

Recall

- Even though the `Monad` instance for (`State s`) definition looks complex
  - the definition of `return` is equivalent to the state-passing identity function `id_state (x, st) = (x, st)` translated to use the `State` datatype
  - the definition of `>>=` is equivalent to composing two state-passing functions (translated to use the `State` datatype) to create a third
- So the `Monad` instance is completely natural and can be derived without even using the monad laws

Example

- Big deal... what does this buy us in practice?
- We'll use state monads to write a function which is structurally equivalent to imperative C code, but without using the `IO` monad
- Previously we saw how to write the `gcd` function using `IORef`s
- Now we'll see how to do the same thing (more simply!) using a state monad

Example

- Recall the C version of **`gcd`**:

```
int gcd(int i, int j) {
  while (i != j) {
    if (i > j) {
      i = i - j;
    } else {
      j = j - i;
    }
  }
  return i;
}
```

Example

- We will translate the C `gcd` function into the equivalent state monad version in Haskell
- To do this, we will need a way to put values into the state and withdraw them from the state
- We will show how to do this in general, then introduce the `MonadState` type class, which will do it for us

Retrieving the state

- In a state monad, we are combining state transformers which have the type `s -> (a, s)` for some state type `s` (wrapped in a `State` constructor)
- Each state transformer specifies the way it changes the state as well as a "return value"
- If the value we want to return is the state itself, what will the type of the state transformer be?
- Answer: `s -> (s, s)`

Retrieving the state

- Let's write such a state transformer and call it **getState**:

```
getState :: State s s
getState = State (\st -> (st, st))
```

- In a state monad computation, we would use it like this:

```
do ...
   st <- getState
   ... -- (computations involving st)
```

Changing the state

- Changing the state requires that we have a state transformer that can change the existing state
- Written in the purely-functional style, this would look like this:

```
putState' :: (s, s) -> ((), s)
putState' (st', st) = ((), st')
```

- Writing this with the `State` datatype, it becomes:

```
putState :: s -> State s ()
putState st' = State (\st -> ((), st'))
-- substitute new state st' for old state st
```

Changing the state (2)

- `getState` and `putState` are the only essential functions needed to interact directly with the state in a state monad
- However, another useful function is called `modifyState`
- It takes a function and applies it to the state, yielding a new state

```
modifyState :: (s -> s) -> State s ()
modifyState f = State (\st -> ((), f st))
```

`gcdState` (version 1)

- With these functions, we can write our first version of the GCD function using state monads
- We will define a GCD state transformer called `gcdState`
- The GCD computation requires what state variables?
  - two integers, which we'll call `i` and `j`
- Therefore, the type signature of `gcdState` is:

```
gcdState :: State (Integer, Integer) Integer
```

`gcdState` (version 1)

```
gcdState :: State (Integer, Integer) Integer
```

- Note that the "return" type of the state transformer is `Integer`
- Meaning: when the computation is done, the result will be an integer
- Before we discuss how to write this state transformer, let's see how to use it!
- We will use the `runState` function to define the `gcd` function itself

`gcdState` (version 1)

- `runState` takes the initial state and the state transformer and returns the final state, plus the return value
- We assume the return value will be the GCD itself
- Therefore, we have:

```
gcd :: Integer -> Integer -> Integer
gcd i j = fst $ runState gcdState (i, j)
```

- `runState gcdState (i, j)` returns a (value, state) pair
- Take the value part, which is the GCD

`evalState` and `execState`

- Most state monad computations either want just the return value or the final state (not both)
- Therefore, in the `Control.Monad.State` module (where all the state monad stuff is defined) are two useful helper functions:

```
evalState :: State s a -> s -> a
evalState trans init_st = fst $ runState trans init_st

execState :: State s a -> s -> s
execState trans init_st = snd $ runState trans init_st
```

evalState and execState

- We can simplify our gcd function a tad using `evalState`:

```
gcd :: Integer -> Integer -> Integer
gcd i j = evalState gcdState (i, j)
```

- Now we merely have to define `gcdState`!

`gcdState` (version 1)

- Here is our first version of **`gcdState`**:

new i new j

```
gcdState :: State (Integer, Integer) Integer
gcdState = do
  (i, j) <- getState
  if i > j
     then do putState (i - j, j)
             gcdState
     else if i < j
             then do putState (i, j - i)
                     gcdState
             else  -- i == j
               return i
```

`gcdState` (version 2)

- The nested `if` statements are gross, so let's use the `compare` function instead
- `compare` outputs a value of the `Ordering` type:

```
data Ordering = LT | EQ | GT
```

- This will allow us to clean up the code without changing anything significant

gcdState (version 2)

```
gcdState :: State (Integer, Integer) Integer
gcdState = do
  (i, j) <- getState
  case compare i j of
    GT -> do putState (i - j, j)
             gcdState
    LT -> do putState (i, j - i)
             gcdState
    EQ -> return i
```

`gcdState` (version 2)

- Notice this code:

```
do putState (i - j, j)
   gcdState
```

- This combines two state transformers to get one bigger state transformer
- The new state transformer changes the state, then runs the `gcdState` transformer again

MonadState

- We will improve this code still more, but first I want to introduce a very useful type class called `MonadState`
- `MonadState` has this definition:

```
class Monad m => MonadState s m | m -> s where
  get :: m s
  put :: s -> m ()
  state :: (s -> (a, s)) -> m a
```

MonadState

- What **MonadState** does is to generalize the **getState** and **putState** functions to arbitrary state-like monads
- The definition for the **(State s)** monad is as follows:

```
instance MonadState s (State s) where
  get = State (\st -> (st, st))
  put st' = State (\st -> ((), st'))
  state = State  -- won't be using this
```

- Look familiar?

MonadState

- Why does `MonadState` need to be a type class?
- Why not define `get` and `put` to just be `getState` and `putState`?
- Answer: It's more general!
- Monads other than `(State s)` can have `get` and `put` methods
- One class of these are aggregate monads built on top of `(State s)` using what are called *monad transformers* (advanced topic! coming soon!)

MonadState

- Also notice the line:

```
class Monad m => MonadState s m | m -> s where
```

- Note the functional dependency `| m -> s`
- This means: any given monad `m` which is an instance of `MonadState` uniquely determines the state type `s` for the instance
- In our case, `m` is `(State s)` and `s` is `s`, so the connection is pretty obvious

MonadState

- We can use **MonadState**'s methods to shrink our **getState** code a tiny bit:

```
gcdState :: State (Integer, Integer) Integer
gcdState = do
  (i, j) <- get
  case compare i j of
    GT -> do put (i - j, j)
             gcdState
    LT -> do put (i, j - i)
             gcdState
    EQ -> return i
```

MonadState

- We can also define a function called `modify` which generalizes `modifyState`:

```
modify :: (s -> s) -> State s ()
modify f = do s <- get
              put (f s)
```

- We could also define it as:

```
modify :: (s -> s) -> State s ()
modify f = State (\s -> (), f s)
```

`runState` again

- We have seen how to use `runState` to "run" a computation in a state monad and get a result
- A different use for `runState` is to extract the state transformer function out of a `State` value
- Recall the definition of `runState`:

```
runState :: State s a -> s -> (a, s)
runState (State f) init_st = f init_st
```

- Leaving off the final `init_st`, we have:

```
runState (State f) = f   -- extract f
```

whileState

- We will use the **MonadState** methods, **modify** and **runState** to define **whileState**, a while loop that operates in a state monad
- Here is the definition:

```
whileState :: (s -> Bool) -> State s () -> State s ()
whileState test body = do
  s0 <- get
  when (test s0)
       (do modify (snd . runState body)
           whileState test body)
```

- Let's walk through this step-by-step

whileState

- Look at the type declaration:

```
whileState :: (s -> Bool) -> State s () -> State s ()
whileState test body = ...
```

- The first argument is a test function
  - It takes the existing state, examines it to see if the computation has completed, and returns a boolean value
- The second argument is the body of the while loop
  - It is a state transformer that changes the state and returns nothing
- The return value is also a state transformer that returns nothing

whileState

```
whileState :: (s -> Bool) -> State s () -> State s ()
whileState test body = do
  s0 <- get
  when (test s0)
       (do modify (snd . runState body)
           whileState test body)
```

- First, we get the state and bind it to s0

whileState

```
whileState :: (s -> Bool) -> State s () -> State s ()
whileState test body = do
  s0 <- get
  when (test s0)
       (do modify (snd . runState body)
           whileState test body)
```

- **`when`** is a monadic **`if`** statement with no **`else`**
- Definition:

```
when :: (Monad m) => Bool -> m () -> m ()
when b s = if b then s else return ()
```

whileState

```
whileState :: (s -> Bool) -> State s () -> State s ()
whileState test body = do
  s0 <- get
  when (test s0)
       (do modify (snd . runState body)
           whileState test body)
```

- When the test **`(test s0)`** returns **`True`**,
  - execute the line **`modify (snd . runState body)`**
  - then run **`whileState`** all over again
- Otherwise, you're done!

`whileState`

```
modify (snd . runState body)
```

- `body` is the body of the while loop, as a state transformer of type `State s ()`
- `runState body` is the state transforming function, of type `s -> ((), s)`
- `(snd . runState body)` is a function of type `(s -> s)`; it takes in a state `s`, runs `runState body` on it to get a value of type `((), s)`, and takes the second part of the tuple (of type `s`)

whileState

`modify (snd . runState body)`

- `(snd . runState body)` is thus a function that uses the state transformer `body` to modify the state
- `modify` takes the function `(snd . runState body)` and actually modifies the state
- then `whileState` is repeated again, until it's done
- Now we can use `whileState` to simplify our `gcdState` state transformer

whileState

- One small tweak...
- `(snd . runState body)` is equivalent to `(execState body)`
- Rewriting, we have:

```
whileState :: (s -> Bool) -> State s () -> State s ()
whileState test body = do
  s0 <- get
  when (test s0)
       (do modify (execState body)
           whileState test body)
```

gcdState

- New version of `gcdState`:

```
gcdState :: State (Integer, Integer) Integer
gcdState = do
  whileState (\(i, j) -> i /= j)
    (do (i, j) <- get
        if i > j
           then put (i - j, j)
           else put (i, j - i))
  (i, _) <- get
  return i
```

gcdState

- Compare with C version:

```haskell
gcdState = do
  whileState (\(i, j) -> i /= j)
    (do (i, j) <- get
        if i > j
           then put (i - j, j)
           else put (i, j - i))
  (i, _) <- get
  return i
```

```c
int gcd(int i, int j) {
  while (i != j) {
    if (i > j)
         { i = i - j; }
    else { j = j - i;}
  }
  return i; }
```

- Very similar!
- But we can do even better...

gcdState

```
gcdState :: State (Integer, Integer) Integer
gcdState = do
  whileState (\(i, j) -> i /= j)
    (do (i, j) <- get
        if i > j
           then put (i - j, j)
           else put (i, j - i))
  (i, _) <- get
  return i
```

icky

gcdState

```
...
              then put (i - j, j)
              else put (i, j - i)
...
```

- The `put` lines are icky because you only need to modify either `i` or `j`, not both
- Let's define two functions `putI` and `putJ` that will only modify `i` or `j`

gcdState

- Here you go:

```
putI :: Integer -> State (Integer, Integer) Integer
putI i = modify (\(_, j) -> (i, j))

putJ :: Integer -> State (Integer, Integer) Integer
putJ j = modify (\(i, _) -> (i, j))
```

gcdState

- Now `gcdState` becomes:

```
gcdState :: State (Integer, Integer) Integer
gcdState = do
  whileState (\(i, j) -> i /= j)
    (do (i, j) <- get
        if i > j
          then putI (i - j)
          else putJ (j - i))
  (i, _) <- get
  return i
```

gcdState

- Compare with C version now:

```
gcdState = do
  whileState (\(i, j) -> i /= j)
    (do (i, j) <- get
        if i > j
           then putI (i - j)
           else putJ (j - i))
  (i, _) <- get
  return i
```

```
int gcd(int i, int j) {
  while (i != j) {
    if (i > j)
         { i = i - j; }
    else { j = j - i;}
  }
  return i; }
```

- Almost identical!
- Downside: had to define trivial helper functions **`putI`** and **`putJ`**

Caveats

- Some of the presentation here has been simplified
- Actual `ghc` library code can be quite complex (hyper-general versions of everything)
- Modules also often import other modules and re-export the same functions
- Sometimes modules restrict access to constructors
  - Example: the `State` constructor is not exported; need to use the `state` function which does the exact same thing
- When in doubt, consult Hoogle and read the source code!

Caveats

- Module name `Control.Monad.State` may be found in more than one library ("package") depending on your Haskell setup
- GHC will not let you import a module if there are multiple candidates for importing (just get a warning message)
- If this happens, specify `Control.Monad.Trans.State` instead and everything should work

Conclusion

- State monads allow us to simulate imperative code that uses local or global variables
- Advantages:
  - can easily get into/out of monad
  - purely functional, often simpler than using `IORef`s
- Disadvantages:
  - can't do I/O (unless you use a monad transformer)
  - usually slower than using `IO` monad
- State monads are a useful tool in many kinds of programming

Next time

- Parsing with parser combinators