CS 115
Functional Programming

Lecture 19:

State Monads
(part 1)

Previously

- Error-handling monads
- Functional dependencies
- Existential types

Today

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

Motivation

- We have seen that we can use the `IO` monad to write imperative computations in Haskell
  - `IO` is not just for input and output!
- We can use `IORef`s where mutable variables are used in imperative languages
- We can use `IOArray`s where mutable arrays are used in imperative languages
- All this power comes with a major price tag, though
  - What is it?

Motivation

- Using the `IO` monad for imperative programming is a one-way trip!
- Once your code enters the `IO` monad, it never exits it!
- It would be good if there was a way of doing imperative (or imperative-like) computations without being forced to stay in the `IO` monad

Motivation

- We have four options to do imperative-like computations without having to stay in the `IO` monad:
- Option 1: Do them in the `IO` monad, but use `unsafePerformIO`
  - almost never a good idea!
  - need to be able to prove that this doesn't break referential transparency, or the code will not behave properly

Motivation

- Option 2: do computations in the `ST` monad
  - `ST` is an `IO`-like monad that allows you to exit back into normal functional code
  - allows you to use `STRef` and `STArray` (analogous to `IORef` and `IOArray`)
  - However, you can't do input and output in the `ST` monad
  - "Under the hood", `ST` is actually using the `IO` monad to run its computations and (safely) uses `unsafePerformIO`
- [Reference: "Lazy Functional State Threads" paper by John Launchbury and Simon Peyton-Jones]

Motivation

- Option 3: manually thread state variables in each function that needs them
  - Use helper functions with extra state variables and change the state variables when calling helper functions (often recursively)
  - A purely functional approach (no `IO` or `ST` monads)
  - This approach covered in CS 4 in detail
  - Perfectly OK to do this if the number of state variables is small (1 or 2)
  - With more state variables, this gets very cumbersome

Motivation

- Option 4: do computations in a *state monad*
  - State monads are a *purely functional* way to encapsulate state and pass it around in computations that use that state
  - State monads are completely unconnected from the `IO` or `ST` monads
  - State monads allow us to simulate state variables in a purely functional setting
  - It's easy to break out of a state monad whenever you want

Motivation

- State monads are particularly useful when you don't interact with the state much, but need to occasionally read state variables or write state variables
- As we will see, state monads are just a monadic form of option 3 (the standard passing state through extra variables trick we learned in CS 4)

State monads vs `ST` monad

- Why use state monads instead of the `ST` monad?
- `ST` monad is preferable when you have an unbounded number of stateful items you want to work with (*e.g.* large numbers of `STRef`s or `STArray`s) or where efficiency is essential
- When the number of stateful items is small, and many functions share the same state, state monads are a more natural fit
- You can combine state monads with the `IO` monad using *monad transformers* (coming up in a few lectures), but `ST` and `IO` cannot be combined

Stateful computations

- Recall, once again; monads are used to model different "notions of computation" in Haskell
- Here, the "notion" is: computations that interact with state (local or global)
- Conceptually, we can draw the type signature of the characteristic functions of this monad as:

```
a ---[access/modify state variables]--> b
```

- Such functions take in a value of type `a`, possibly interact with (access/modify) some state variables, and output a value of type `b`

Stateful computations

- We need to take this schematic diagram

```
a ---[access/modify state variables]--> b
```

- and convert it into something Haskell can use
- Let's assume that we have a datatype representing all the state variables in our computations (as a tuple or a record)
- If this datatype is called `s` we can rewrite the type signature as:

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

Stateful computations

- `(a, s) -> (b, s)`
- This is the type signature of a pure function which passes a state value of type `s` into the input and retrieves it from the output
- The input and output `s` values may be different
- Computations of this form are said to *thread the state* through the computation
- We can write all state-handling functions we want in this manner, but doing so will be very tedious
  - like using `Maybe` or `(Either e)` types without monads

Stateful computations

- Ultimately, we want to be able to write monadic functions with a type signature like this:

```
a -> m b
```

- We will have to do a few transformations to convert the "natural" type for state-passing functions:

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

- into the monadic form

Transformation #1

- We will start by noting that any function with the type

`(a, b) -> c`

- can be curried to give an equivalent function of type

`a -> b -> c`

- Applying this to functions of type

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

- gives functions of type

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

- These functions can represent exactly the same computations as ones of type `(a, s) -> (b, s)`

Transformation #2

- Since the `->` in type signatures associates to the right, it is legitimate to rewrite the type signature

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

- as:

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

- Objects that have the type `(s -> (b, s))` will be referred to henceforth as "state transformers"
- They are functions that take in a state value of type `s`, and return a value of type `b`, along with a (possibly different) state value of type `s`

Transformation #3

- We can create a new datatype to wrap around the return type of functions with this type:

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

- We'll call it `State`, and define it as:

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

- `State` is a binary type constructor like `Either` with the kind `* -> * -> *`
- We use the name `State` as the name of the (only) value constructor as well as the type constructor's name (this is legal in Haskell)

Transformation #3

- Using `State`, we can rewrite our state-passing functions so that they have the type signature:

```
a -> State s b   -- or: a -> (State s) b
```

- Comparing this to the characteristic type signature of a monadic function:

```
a -> m b
```

- We see that our monad is going to have to be `(State s)` which will be a unary type constructor (*i.e.* which will have the kind `* -> *`)
- We will refer to this as "the" `State` monad

State monads

- Notice that the monadic values of the `State` monad:

`State s a`

- are actually *functions* of type `(s -> (a, s))` modulo the `State` wrapper
- We have talked about monadic values in the `IO` monad as being "actions" or "undercover functions"
- Here is a monad where the monadic values actually *are* functions!
  - *Not* a coincidence (see why later)

State monads

- Our job now is to write the `Monad` instance definition for the `State s` monad
- We will fill in this code:

```
instance Monad (State s) where
  return x = {- to be filled in -}
  mv >>= f = {- to be filled in -}
```

- As before, we will define `>>=` based on what we want the monad to achieve

Deriving the >>= operator

- Let's start by assuming we have two functions in the `State s` monad with these type signatures:

```
f :: a -> State s b
g :: b -> State s c
```

- and we would like to compose them to give a function with the type signature:

```
h :: a -> State s c
```

- Let's rewrite these type signatures in a non-monadic form so we can better see what's going on

Deriving the >>= operator

- Non-monadic versions of `f`, `g`, and `h` might have the type signatures:

```
f' :: (a, s) -> (b, s)
g' :: (b, s) -> (c, s)
h' :: (a, s) -> (c, s)
```

- Now what composing `f'` and `g'` to give `h'` means is clear:
  - the state output of `f'` (type `s`) is the state input to `g'`
  - the value output of `f'` (type `b`) is the value input to `g'`
- The monad's job will be to handle the state-passing for us

Deriving the >>= operator

- We can easily define `h'` in terms of `f'` and `g'`:

```
h' :: (a, s) -> (c, s)
h' (x, st) =
  let (y, st')  = f' (x, st)
      (z, st'') = g' (y, st')
      -- initial state of g' = final state of f'
  in (z, st'')
```

- This could be simplified all the way down to:

```
h' = g' . f'
```

- but we'll stick to the expanded form for clarity

Deriving the >>= operator

- Going back to the original functions **f**, **g**, and **h**, we have

```
h = f >=> g
```

- which is equivalent to:

```
h x = f x >>= g
```

- which is equivalent to:

```
h x = f x >>= \y -> g y
```

- which is equivalent to:

```
h x = do y <- f x
         g y
```

Deriving the >>= operator

- The interpretation of:

```
h x = do y <- f x
         g y
```

- goes like this:
  1. We compute `f x` (possibly using/changing the state) to get the value `y`
  2. We compute `g y` (possibly using/changing the state) to get the final result
- The state is handled "under the surface" by the monad so we can just concentrate on the values `x` and `y` (the state is there whenever we need it)

Deriving the >>= operator

- Let's go back to `f'` and `g'`:

```
f' :: (a, s) -> (b, s)
g' :: (b, s) -> (c, s)
```

- and write curried versions of them with these type signatures:

```
f'' :: a -> s -> (b, s)
g'' :: b -> s -> (c, s)
```

- in terms of `f'` and `g'`

Deriving the >>= operator

- We have:

```
f'' :: a -> s -> (b, s)
f'' x st = f' (x, st)

g'' :: b -> s -> (c, s)
g'' y st = g' (y, st)
```

- Or, written slightly differently:

```
f'' x = \st -> f' (x, st)
g'' y = \st -> g' (y, st)
```

Deriving the >>= operator

- If we wrap the right-hand sides of `f''` and `g''` in a `State` constructor, we have the definitions of `f` and `g` in terms of `f'` and `g'`:

```
f :: a -> State s b
f x = State (\st -> f' (x, st))

g :: b -> State s c
g y = State (\st -> g' (y, st))
```

Deriving the >>= operator

- Similarly, we can define the monadic composition of `f` and `g` (`h`) in terms of the composition of `f'` and `g'` (`h'`) as follows:

```
h :: a -> State s c
h x = State (\st -> h' (x, st))
```

- Now we are ready to derive the `>>=` operator for the `(State s)` monad

Deriving the >>= operator

- Recall:

```
h = f >=> g
```

- which is equivalent to:

```
h x = f x >>= g
```

- Reversing this equation, we have:

```
f x >>= g = h x
```

- Expanding `h x`, we have:

```
f x >>= g = State (\st -> h' (x, st))
```

Deriving the >>= operator

- Let us calculate:

```
f x >>= g
  = State (\st -> h' (x, st))
  = State (\st ->  -- expand using definition of h'
      let (y, st')  = f' (x, st)
          (z, st'') = g' (y, st')
      in (z, st''))
  = State (\st ->
      let (y, st') = f' (x, st) in
        g' (y, st'))  -- (z, st'') stuff is redundant
```

Deriving the >>= operator

- Continuing:

```
f x >>= g
  = State (\st ->
        let (y, st') = f' (x, st) in
          g' (y, st'))
-- Recall:
-- f x = State (\st -> f' (x, st))
  = State (\st ->
        let (State ff) = f x  -- unpack (f x)
            -- ff = \st -> f' (x, st)
            (y, st') = ff st
        in g' (y, st'))
```

Deriving the >>= operator

- Notice that this definition is no longer dependent on `f'` but only on `f`:

```
f x >>= g
  = State (\st ->
      let (State ff) = f x
          (y, st') = ff st
      in g' (y, st'))
```

- Let's eliminate `g'` in favor of `g` the same way

Deriving the >>= operator

- Continuing:

```
f x >>= g
  = State (\st ->
      let (State ff) = f x
          (y, st') = ff st
          (State gg) = g y  -- unpack (g y)
          -- gg = \st' -> g' (y, st')
      in gg st')  -- gg st' = g' (y, st')
```

Deriving the >>= operator

- Substitute **`mv`** for **`f x`** to get:

```
mv >>= g
  = State (\st ->
      let (State ff) = mv
          (y, st') = ff st
          (State gg) = g y
      in gg st')
```

- This is the correct definition of **`>>=`** for the **`(State s)`** monad
- It may seem unintuitive, but it's just a translation of the way **`f'`** and **`g'`** compose to get **`h'`**

Deriving the >>= operator

- We can also write it like this:

```
mv >>= g
  = State (\st ->
      let (y, st') = runState mv st
      in runState (g y) st')
```

- where

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

Deriving the `return` method

- We still need to derive the `return` method
- Usually we do this using the monad laws
- Here, there is a much easier way!
- Recall: the `return` method for a particular monad is the monadic version of the identity function
- Monadic functions in the `(State s)` monad have type signatures of the form:

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

Deriving the `return` method

- The non-monadic state-passing functions have type signatures of the form:

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

- The identity function in this form would be:

```
id_state (x, st) = (x, st)
id_state' x st = (x, st)          -- curried
id_state' x = \st -> (x, st)     -- written differently
```

- Written as a function in the `(State s)` monad, this becomes:

```
id_state_monad :: a -> State s a
id_state_monad x = State (\st -> (x, st))
```

Deriving the `return` method

```
id_state_monad :: a -> State s a
id_state_monad x = State (\st -> (x, st))
```

- This is the identity function in the `(State s)` monad
- Therefore, it is also the `return` method:

```
return :: a -> State s a
return x = State (\st -> (x, st))
```

- What `return` does is to take a value and output a state transformer which takes a state, doesn't change it and returns the original value

The **Monad** instance

- Putting this all together, we get the **Monad** instance for the **(State s)** monad:

```
instance Monad (State s) where
  return x = State (\st -> (x, st))

  mv >>= g
    = State (\st ->
        let (State ff) = mv
            (y, st') = ff st
            (State gg) = g y
        in gg st')
```

Validating the `Monad` instance

- Once we have a putative `Monad` instance, we must use the monad laws to validate it
- Unfortunately, for state monads this is pretty grungy even for monad laws 1 and 2
  - and *really* ugly for monad law 3!
- I will refer you to a detailed derivation on my blog:

`http://mvanier.livejournal.com/5406.html`

- Upshot: this `Monad` instance does in fact obey the monad laws

Getting out of the monad

- We said that, unlike the `IO` monad, the `(State s)` monad allows us to break out of the monad at any time
- This is actually trivial!
- A monadic value in the `(State s)` monad has the type `State s a`, which is equivalent to a function of type `(\s -> (a, s))` wrapped up in a `State` constructor
- To extract the (value, state) pair, we must unpack the function from the `State` constructor and apply it to an initial state value

Getting out of the monad

- We saw a library function called `runState` which does this; we can rewrite it as:

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

  - (The actual definition may be different from this, but it will be equivalent)
- A computation in the `(State s)` monad can be "run" by passing it, and an initial state, to `runState`, which will return the final state and the final result value

Using state monads

- State monads are found in the Haskell module called `Control.Monad.State`
- This module also defines `runState` as well as the `MonadState` type class (subject of next lecture)

Next time

- 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