COMP3141

Software System Design and Implementation

Effects and IO Monad Practice

Curtis Millar
CSE, UNSW

2 July 2021

QuickCheck and Search Trees

1. `mysteryProp`
2. `mysterious`
3. `astonishing`

Sequential composition

A function of the type `a -> (a, b)` is an *operation* that will produce a value of type `b` by *mutating* a value of type `a`.

If I have another function of type `a -> (a, c)`, I can *sequentially compose* them together. I can mutate the value of type `a` using the first function, then the second, and return the result of the second.

```
compose :: (a -> (a, b)) -> (a -> (a, c)) -> (a -> (a, c))
compose first second input =
  let
    (afterFirst, \_) = first input
  in
    second afterFirst
```

We compose the functions together by appying the first and discarding the resulting value of type `b`.

Using the output of an operation

If we determine our second operation based on the *result* of the first operation, we can represent it with a function of the type `b -> (a -> (a, c))`.
It is a function that consumes the result of a previous operation and produces a new operation. We can then create a function to sequence these operations together.

```
bind :: (a -> (a, b)) -> (b -> (a -> (a, c))) -> (a -> (a, c))
bind first second input =
  let
    (afterFirst, firstResult) = first input
  in
    second firstResult afterFirst
```

This is similar to `compose`, but rather than throw away the value of type `b`, we apply the function that produces the next opreation to that value.

State Monads

What we have in fact constructed is the State monad.

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

The State monad encapsulates functions that mutate a state value and produce a result of that mutation.
The State monad has an instance of the Monad type class (we'll discuss further in week 7), so uses the >> operator rather than compose and the >>= operator rather than the bind function.

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

We also get a return function that creates an operation that does nothing and returns a constant value.

```
return :: Monad m => a -> m a
```

Do notation

As writing expressions using the >> and >>= operators can be tedious, Haskell provides do notation.

```
-- Using >> operator
qux = foo >> bar

-- Using do
qux = do
  { foo
  ; bar
  }
```

```
-- Using >>= operator
    qux = foo >>= (\result -> bar)

-- Using do (without {;})
qux = do
  result <- foo
  bar
```

Functions on State Monads

The State monad provides three functions to interact with the internal state, get, put, and modify.

```
-- Get the current state
get :: State s s
get = State $ (s -> (s, s))

-- Update the state
put :: s -> State s ()
put newState =
  State $ (_ -> (newState, ()))
```

```
-- Mutate the state
modify :: (s -> s) -> State s ()
modify f = do
  s <- get
  put $ f s

-- Execute from starting state
runState :: State s a -> s -> (s, a)
runState (State op) = op
```

Mini Processor

Example

Mini processor done in editor

The IO Type

A procedure that performs some side effects, returning a result of type a is written as IO a.

World interpretation

IO a is an abstract type. But we can think of it as a function:

```
RealWorld -> (RealWorld, a)
```

(that's how it's implemented in GHC)

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

getChar :: IO Char
readLine :: IO String
putStrLn :: String -> IO ()
readFile :: Strig -> IO String
```

QuickChecking Monads

QuickCheck lets us test IO (and ST) using this special property monad interface:

```
monadicIO :: PropertyM IO () -> Property
pre       :: Bool -> PropertyM IO ()
assert    :: Bool -> PropertyM IO ()
run       :: IO a -> PropertyM IO a
```

Testing a Tic-Tac-Toe A.I

Example

Testing A.Is for Tic-Tac-Toe

Done in editor

Homework

1. Next week is flexibility week
2. Last week's quiz is due on Today. Make sure you submit your answers.
3. The fourth programming exercise is due by the start if my next lecture (in 14 days).
4. This week's quiz is also up, it's due Friday week (in 14 days).