Monads

Lecture 12

Monads

- A language without side effects can't do I/O
  - Side effects are critical in some applications
    - For expressiveness and/or efficiency

- Haskell has a general mechanism for side effects and much more
  - based on *monads*

An Example: A Counter

- How do we keep track of the # of times a function f is called?

- In C: a global (or static) variable count
  - f(x) = { count++; … body of function … }

- In Haskell: an extra parameter to f

  f :: Int -> Foo -> (Int * Bar)

  f count x = (count+1, … body of function …)

An Example (Cont.)

- Expanding on this program:

```
g count y = …  (c,v) = h(count,a) ….

h count z = … (c,w) = f(count,b) …

f count x = (count+1, … body of function …)
```

A Problem

- The functional solution has a problem
- The counter must be maintained by f's caller
  - And possibly f's caller's caller
  - Or even the entire program
- Passing around the "state" explicitly is painful
  - And poor software engineering

An Idea

- Could we hide the state in a type?

- What does a state transformer do?
  - Maps a state to a new state
  - And (possibly) returns a value

```
type StateTrans s a = s -> (s,a)
```

Statements

- A *statement* is a state transformer
  - E.g., x = a*b in C

- A sequence of statements successively transforms the state

s1; s2

- Define a sequencing combinator:

```
(>>=) :: StateTrans s a -> (a -> StateTrans s b) -> StateTrans s b
(>>=) f g s0 = let (s1,v) = f s0 in g v s1
```

Sequencing

- Look at this combinator more closely

  ```
  (>>=) :: StateTrans s a -> (a -> StateTrans s b) -> StateTrans s b
  ```

- This combinator must evaluate the first state transformer and then the second

Sequencing (Cont.)

- Now expand the type:

  (>>=) :: StateTrans s a -> (a -> StateTrans s b) -> StateTrans s b

  (>>=) :: (s -> (s,a)) -> (a -> (s -> (s,b))) -> (s -> (s,b))

- Note two things:
  - The manipulation of state is hidden inside the type StateTrans
    - If this type is abstract, access to the state is restricted
  - The statements *have not been executed yet*
    - >>= is a "script builder'

Example Revisited

- Consider the counter example

  f count x = (count+1, … body of function …)

- The state is an integer:

  (>>=) :: StateTrans Int a -> (a -> StateTrans Int b) -> StateTrans Int b

- We need a state transformer that updates the state:

```
++ :: Int -> (Int, ())
++ i = (i+1, ())
```

The Example, Cont.

- Consider the counter example

  f count x = (count+1, … body of function …)

- Rewrite:

  f :: Foo -> StateTrans Int Bar

  f x = ++ >>= _…. body of function …

The Example, Cont.

- This isn't quite right:

```
f :: Foo -> StateTrans Int Bar
f x = ++ >>= \_.... body of function ...
```

- "Body of function" needs to be a state transformer, too!

- Use a new function:

```
unit :: a -> StateTrans s a
unit v s = (s,v)
```

The Original Example

g count y = … (c,v) = h(count,a) ….

h count z = … (c,w) = f(count,b) …

f count x = (count+1, … body of function …)

Rewritten as a State Transformer

```
g y = unit(…) >>= \_.h(a) >>= \v.unit(…)

h z = unit(…) >>= \_.f(b) >>= \w.unit(…)

f x = ++ >>= \_. unit (… body of function …)
```

Dicussion

- The "plumbing" of the state is hidden
  - State only referred to explicitly where needed
  - Just as in C
- The types help
- State still affects global program structure
  - But functional sub-parts are clearly delineated

Discussion

- What is state in a functional world?

- Answer:
  - A hidden "extra" parameter to every function
  - This extra parameter is restricted
    - Cannot be copied
    - Strict sequencing of operations must be observed

- This is what >>= does
  - Threads the state using higher-order functions

Another Example of a State Monad

```
data SM a = SM (Int -> (Int,a))
SM c1 >>= fc2 =
        SM (\s0 -> let (s1,r) = c1 s0 in fc2 r s1)
unit k  = SM \s -> (s, k)

a >> b      = a >>= \_ -> b
read        = SM \s -> (s, s)
inc         = SM \s -> (s+1,())
init        = \i.SM \s -> (I,())

(init 0) >> inc >> read >>= \x -> … compute with x …
```

Observation

- Consider the type StateTrans again:

  type StateTrans s a = s -> (s,a)

- Note that we really used

  type StateTrans Int a = Int -> (Int,a)

- The type of the state was fixed
- But the type of computations on state is parameterized by a

Monads

A Monad is a type M a with two operations:

```
unit :: a -> (M a)
bind :: M a -> (a -> M b) -> M b
```

And

- bind (or >>=)is associative

```
(a >>= \x -> b x) >>= \y -> c y     =
a >>= (\x -> (b x >>= \y -> c y))
```

- unit is an identity

A Critical Distinction

- Let M be a monad
- If a is a type of *values*
  - E.g., Int, Char, Int -> Int
- Then M a is a type of *computations*
  - E.g., State transformers on Ints

Monadic Programming

- Monads allow one to:
  - Hide “extra” arguments to functions
  - Add primitives to access those hidden values
  - Compositionally build computations

- None of this is specific to state

- There are many other applications

I/O

- IO monad provides input/output operations
  - The state is the external world
  - Primitive operations to read/write devices

IO :: World -> (World, a)

- Computations in the IO monad map the state of the world to a new world value
  - The world is the state

I/O Operations

```
getcIO :: IO Char
putcIO :: Char -> IO Char

bindIO :: IO a -> (a -> IO b) -> IO b
unitIO :: a -> IO a
```

More Useful IO Operations

- With IO, we often don't care about the return value, so we can define functions to ignore it

```
(>>) :: IO a -> IO b -> IO b
(>>) s1 s2 = s1 >>= \_.s2

doneIO :: () -> IO ()
doneIO () = unitIO ()
```

An Example

```
echo = getcIO >>= \a ->
            if (a == eof) then
                doneIO
            else
                putcIO a >> echo
```

IO in Haskell

- The IO Monad is *the* way to I/O in Haskell

- Programs must have the type IO a
  - A whole program is an I/O performing computation

- The World values are crucial
  - Explicit in the compiler, like all other values
  - Show the dependencies between computations
  - Ignored by the code generator

Single-Threadedness

- Critical to correctness is that the state (or world) is single threaded
- If a monad is abstract, this is guaranteed
  - Bind/unit are single-threaded
    - Take one reference, produce one reference
  - These are the only operations on the type
- Also used for, e.g., efficient array upate

Non Single-Threadedness

- Haskell does have some “dangerous” I/O primitives that are not single-threaded
  - E.g., to do asynchronous I/O

- These may lead to race conditions
  - But that is part of the desired functionality

Other Uses of Monads

- Continuations
  - Hide the continuation in the monad
- Exceptions
  - Special case of continuations
- Passing state backwards
  - How many calls of this function are left in the execution?
  - Just reverse passing of state in bind
  - Depends on lazy evaluation

Composable Interpreters

- Make each language feature a monad
- Build an interpreter with exactly the features you want by composing monads
  - Guy Steele's work
- Unfortunately, composing monads doesn't work in general
  - But this is the closest we've gotten to compositional language design

Conclusions

- Monads are a way of structuring programs
- Depend critically on higher-order functions
- A new idea