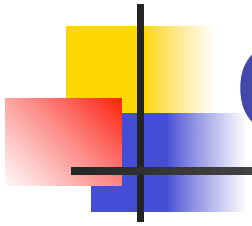
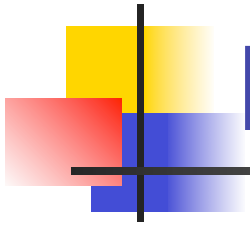


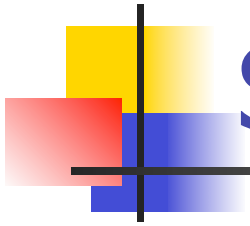
CS 11 Haskell track: lecture 5

- This week:
  - State monads



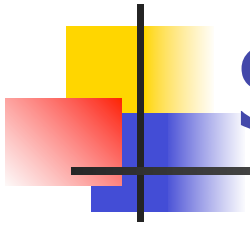
Reference

- "Monads for the Working Haskell Programmer"
- http://www.engr.mun.ca/~theo/Misc/haskell_and_monads.htm
- Good explanation of state monads
- Today's lecture shamelessly ripped off from this



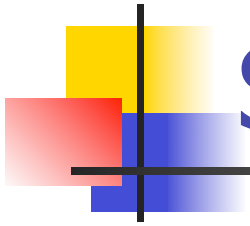
Stateful computations (1)

- Most programming languages use state all over the place
- Functions can receive inputs, return outputs, and also modify the global state
- Internally, functions often work by modifying local state of function on a line-by-line basis



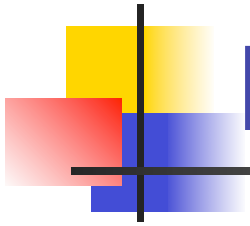
Stateful computations (2)

- Haskell is a purely functional programming language
  - can’t modify state locally or globally
- Can always turn a stateful computation into a stateless computation – how?



Stateful computations (3)

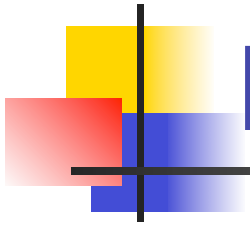
- Can "thread the state" through functions by adding state as extra argument
  - though functions become more cumbersome
- E.g. `f(x)` → `f(state, x)`
- Managing threaded state becomes inconvenient
- How can we retain advantages of functional programming while still threading state?



Modeling state in Haskell (1)

- Recall that monads provide a way of structuring computations that are function-like but not necessarily strictly functional
- We can create a monadic interface to functions that manipulate local state
- Conceptually, our "functions" will look like:

```
          local state
input -------------> output
```

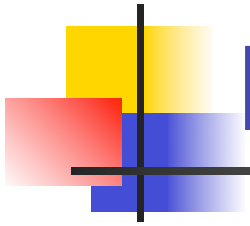


Modeling state in Haskell (2)

- To make this functional, we have to put the local state in the inputs and outputs as an additional argument in each:
- Now our functions look like this:

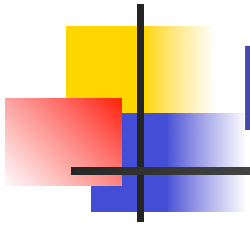
```
(input, state) -> (state, output)
```

- The function takes in an input value, plus the initial value of the local state, and returns the output value, plus the final value of the local state



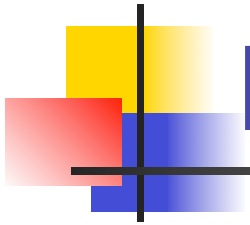
Modeling state in Haskell (3)

- We can curry the input argument to get:
  `input -> state -> (state, output)`
- This will be the characteristic shape of the monadic functions we'll be working with
- The monadic values will represent functions of the form `state -> (state, output)`



Modeling state in Haskell (4)

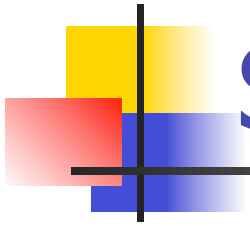
- Monadic functions:
  `input -> state -> (state, output)`
- Corresponds to `a -> m b` where `a` is input and `m b` is `(state -> (state, output))`
- Monadic values: `(state -> (state, output))` or `m b` for the appropriate monad `m`
- Real state monads are a thin wrapper around this notion



Running example

- Imperative algorithm to compute greatest common divisor (GCD) of two positive integers:

```
int gcd(int x, int y) {
  while (x != y) {
    if (x < y)
        y = y - x;
    else
        x = x - y;
  }
  return x;
}
```

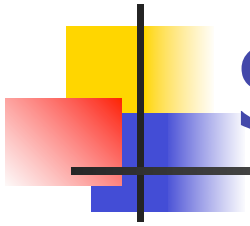


Stateful data types (1)

- First, want to encapsulate notion of threading state into our data types:

```
newtype StateTrans s a = ST (s -> (s, a))
```

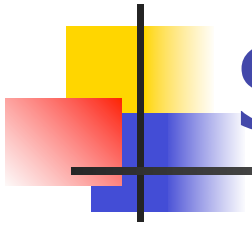
- `newtype` declaration is like a `data` declaration with only one option
- Now a `StateTrans` object encapsulates some kind of state (`s`) and some kind of value (`a`)



Stateful data types (2)

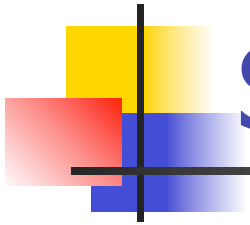
```
newtype StateTrans s a = ST (s -> (s, a))
```

- Notice that this type defines a whole family of state-passing types
- For any given computation, must assign a particular kind of state and a particular kind of value
- Can specify how to combine different instances of this type



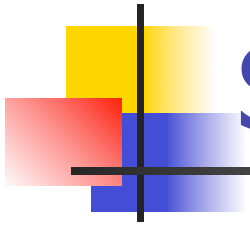
Stateful data types (3)

- Can probably assume that state type stays constant throughout computation
  - represents all possible aspects of state in the computation e.g. as a tuple
- Value types may change for every step of the computation



State monads (1)

- Can think of stateful computation as a composition of several smaller stateful computations
- To manage different "notions of computation", we use monads
  - `IO` – computations that perform I/O
  - `Maybe` – computations that may fail
  - `List` – computations that may return multiple results
  - `StateTrans` – computations that transform state

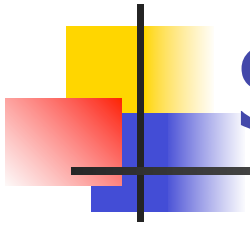


State monads (2)

- Let's build up the `instance` declaration:

```
instance Monad (StateTrans s)
  where
    -- return :: a -> StateTrans s a
    return x = ST (\s0 -> (s0, x))
```

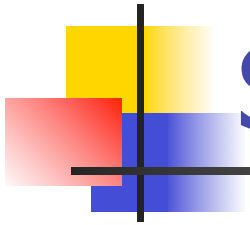
- `return` just returns a value, leaving the state unchanged



State monads (3)

- Still need the bind operator:

```
-- (>>=) :: StateTrans s a ->
--          (a -> StateTrans s b) ->
--          StateTrans s b
(ST p) >>= k  =
  ST (\s0 ->
             let (s1, x) = p s0
                 (ST q)  = k x
             in q s1)
```

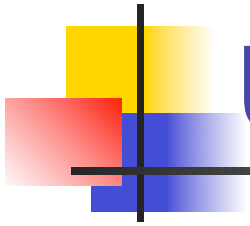


State monads (4)

- Meaning of the bind operator:

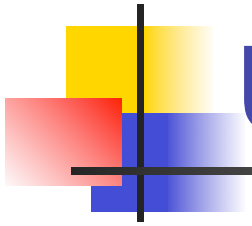
```
(ST p) >>= k  =
  ST (\s0 -> let (s1, x) = p s0
                 (ST q)  = k x
             in q s1)
```

- Given state transformer `p`, return new state transformer that
  - takes a state `s0`, applies `p` to it to get `(s1, x)`
  - applies `k` to `x` to get new state transformer `ST q`
  - applies `q` to new state `s1` to get final state/value pair



Useful auxiliary functions (1)

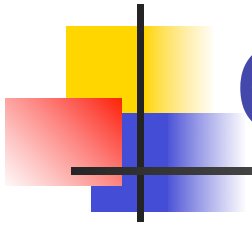
```
-- Extract the state from the monad.
readST :: StateTrans s s
readST =  ST (\s0 -> (s0, s0))


-- Update the state of the monad.
updateST :: (s -> s) -> StateTrans s ()

updateST f =  ST (\s0 -> (f s0, ()))
```



Useful auxiliary functions (2)

```
-- Evaluate a stateful computation.
runST :: StateTrans s a -> s -> (s, a)
runST (ST p) s0 = p s0
```

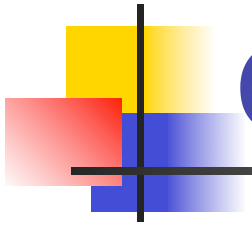
- This starts off the entire computation
  - by passing a state to a particular transformer
  - result is the final state/value pair



GCD example (1)

- The state represents?
  - the current `x` and `y` values.

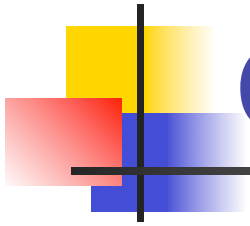
```
type GCDState = (Int, Int)
```



GCD example (2)

- Getting values from the state:

```
getX :: StateTrans GCDState Int
-- getX = ST (\s0 -> (s0, fst s0))
getX = do s0 <- readST
          return (fst s0)

getY :: StateTrans GCDState Int
-- getY = ST (\s0 -> (s0, snd s0))
getY = do s0 <- readST
          return (snd s0)
```



GCD example (3)

- Evaluation of `getX`

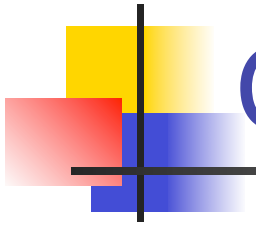
```
getX = do s0 <- readST
          return (fst s0)
```

- Desugar `do`, equivalent to:

```
getX = readST >>= \s0 -> return (fst s0)
```

- Evaluate `readST`:

```
getX = ST (\s0 -> (s0, s0)) >>=
         \s0 -> return (fst s0)
```



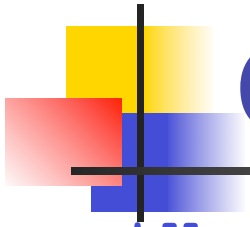
GCD example (4)

- Evaluation of `getX`

```
getX = readST >>= \s0 -> return (fst s0)
     = ST (\s0 -> (s0, s0)) >>=
          \s0 -> return (fst s0)
```

- Unpack `>>=` operator for state monad
- Recall:

```
(ST p) >>= k  =
  ST (\s0 -> let (s1, x) = p s0
                 (ST q)  = k x
             in q s1)
```

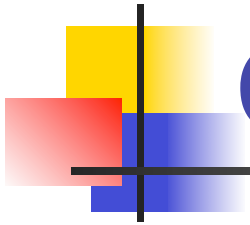


GCD example (5)

```
getX = ST (\s0 -> (s0, s0)) >>=
          \s0 -> return (fst s0)
(ST p) >>= k  =
  ST (\s0 -> let (s1, x) = p s0
                 (ST q)  = k x
             in q s1)
```

- Here, `p s0 = (s0, s0)`
- `k = \s0 -> return (fst s0)`

```
getX = ST (\s0 -> let (s1, x) = (s0, s0)
                      (ST q) = return (fst s0)
             in q s1)
```



GCD example (6)

```
getX = ST (\s0 -> let (s1, x) = (s0, s0)
                      (ST q) = return (fst s0)
                  in q s1)
```

- Recall:

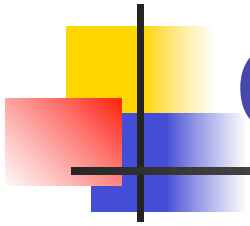
```
return x = ST (\s0 -> (s0, x))
```

- Therefore:

```
ST q = ST (\s0 -> (s0, fst s0))
```

- Continuing...

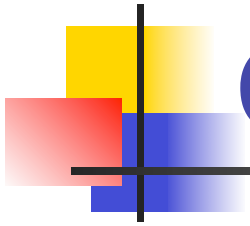
```
getX = ST (\s0 -> q s1)
     = ST (\s0 -> q s0) -- s1 == s0 here
     = ST (\s0 -> (s0, fst s0))  -- QED
```



GCD example (7)

- Putting values into the state:

```
putX :: Int -> StateTrans GCDState ()
-- putX x' = ST (\(x, y) -> ((x', y), ()))
putX x' = updateST (\s0 -> (x', snd s0))

putY :: Int -> StateTrans GCDState ()
-- putY y' = ST (\(x, y) -> ((x, y'), ()))
putY y' = updateST (\s0 -> (fst s0, y'))
```



GCD example (8)

- Compute the GCD:

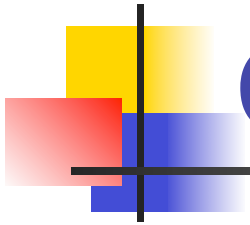
```
gcdST :: StateTrans GCDState Int
gcdST = do x <- getX
           y <- getY
           (if x == y
                then return x
            else if x < y
                 then do putY (y - x)
                         gcdST
                 else do putX (x - y)
                         gcdST)
```

GCD example (9)

- Compute the GCD:

```
gcdST :: StateTrans GCDState Int
gcdST = do x <- getX
           y <- getY
           (if x == y
                then return x
            else if x < y
                 then do putY (y - x)
                         gcdST
                 else do putX (x - y)
                         gcdST)
```

looks like recursive function call, but isn't really



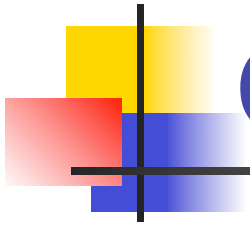
GCD example (10)

```
do putY (y - x)
   gcdST
```

- Equivalent to:

```
putY (y - x) >> gcdST
```

- Combines two state transformers to get a new state transformer
- Recursive data definition
  - not recursive function call
  - like `ones = 1 : ones`

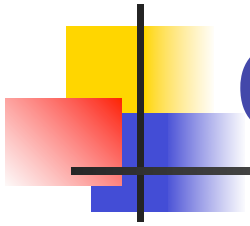


GCD example (11)

- Running the GCD:

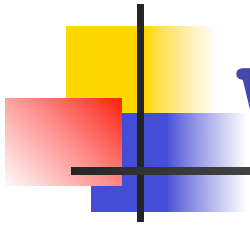
```
mygcd :: Int -> Int -> Int
mygcd x y = snd (runST gcdST (x, y))
```

- Initialize GCD state transformer with `(x, y)`
- Run it until it returns a final (state, value) pair
- Return the second element of the pair (the result value)



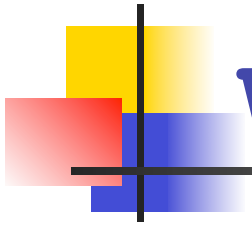
GCD example (12)

- Could write more helper functions
  - e.g. `whileST`
- to more accurately imitate the imperative algorithm
- Common Haskell practice to write higher-order monad combinators



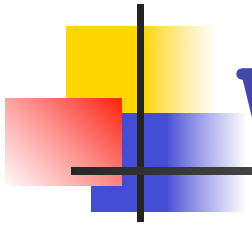
`whileST` (1)

- Let's try to write `whileST`
- State monad version of an imperative "while" loop
- Inputs?
  - a "test" (to see if we continue the loop)
  - a "body" (the contents of the loop)
- Output?
  - a state transformer implementing the while loop



`whileST` (2)

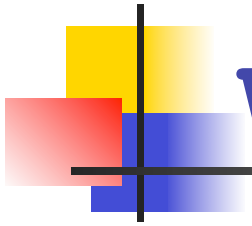
- Type of the inputs?
- test
  - a function mapping ... ?
  - the state to a boolean (`s -> Bool`)
- body
  - a state transformer returning ... ?
  - nothing! (unit type `()`)
  - `StateTrans s ()`



`whileST` (3)

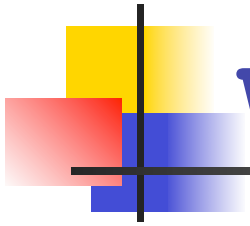
- Type of the output?
  - a state transition returning ... ?
  - nothing! (unit type `()`)
  - `StateTrans s ()`
- The function thus has type

```
(s -> Bool) -> StateTrans s ()
-> StateTrans s ()
```



whileST (4)

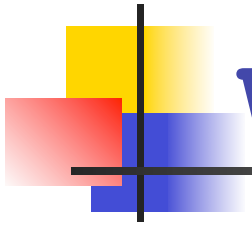
```
whileST :: (s -> Bool) -> StateTrans s () ->
  StateTrans s ()
whileST test body =
  do s0 <- readST
     if (test s0)
        then do updateST (fst . b)
                whileST test body
        else return ()
  where ST b = body
```



whileST (4)

```
whileST :: (s -> Bool) -> StateTrans s () ->
  StateTrans s ()
whileST test body =
  do s0 <- readST
     if (test s0)
        then do updateST (fst . b)
                whileST test body
        else return ()
  where ST b = body
```

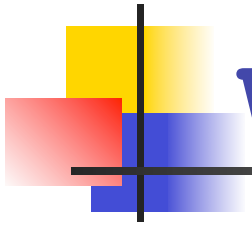
read the current state



whileST (4)

```
whileST :: (s -> Bool) -> StateTrans s () ->
  StateTrans s ()
whileST test body =
  do s0 <- readST
     if (test s0)
        then do updateST (fst . b)
                whileST test body
        else return ()
  where ST b = body
```

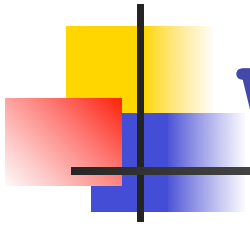
if the test is true



whileST (4)

```
whileST :: (s -> Bool) -> StateTrans s () ->
  StateTrans s ()
whileST test body =
  do s0 <- readST
     if (test s0)
        then do updateST (fst . b)
                whileST test body
        else return ()
  where ST b = body
```

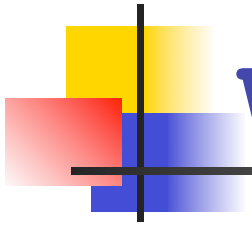
change the state using the body of the loop



whileST (4)

```
whileST :: (s -> Bool) -> StateTrans s () ->
  StateTrans s ()
whileST test body =
  do s0 <- readST
     if (test s0)
        then do updateST (fst . b)
                whileST test body
        else return ()
  where ST b = body
```

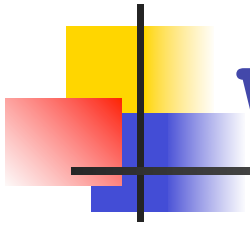
repeat the loop



whileST (4)

```
whileST :: (s -> Bool) -> StateTrans s () ->
  StateTrans s ()
whileST test body =
  do s0 <- readST
     if (test s0)
        then do updateST (fst . b)
                whileST test body
        else return ()
  where ST b = body
```

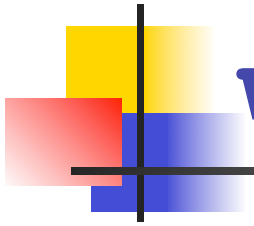
otherwise, we're done



whileST (5)

- GCD function using **whileST**:

```
gcdST :: StateTrans GCDState Int
gcdST = do whileST (\(x, y) -> x /= y)
                (do x <- getX
                    y <- getY
                    if x < y
                      then putY (y - x)
                      else putX (x - y))
           getX
```

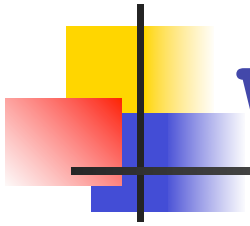


whileST (5)

- GCD function using **whileST**:

```
gcdST :: StateTrans GCDState Int
gcdST = do whileST (\(x, y) -> x /= y)
                (do x <- getX
                    y <- getY
                    if x < y
                      then putY (y - x)
                      else putX (x - y))
           getX
```

test

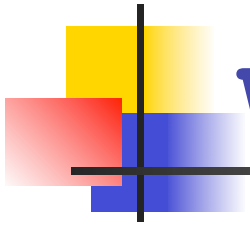


whileST (5)

- GCD function using **whileST**:

```
gcdST :: StateTrans GCDState Int
gcdST = do whileST (\(x, y) -> x /= y)
                   (do x <- getX
                       y <- getY
                       if x < y
                          then putY (y - x)
                          else putX (x - y))
                   getX
```

body

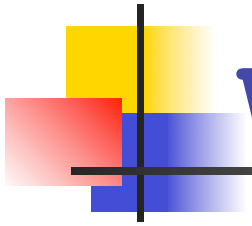


whileST (5)

- GCD function using **whileST**:

```
gcdST :: StateTrans GCDState Int
gcdST = do whileST (\(x, y) -> x /= y)
                (do x <- getX
                    y <- getY
                    if x < y
                      then putY (y - x)
                      else putX (x - y))
```

getX result is x



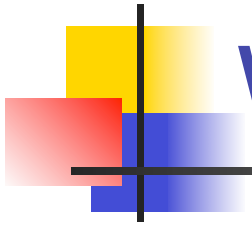
whileST (6)

- Haskell

```
do whileST (\(x, y) -> x /= y)
     (do x <- getX
         y <- getY
         if x < y
           then putY (y - x)
           else putX (x - y))
   getX
```

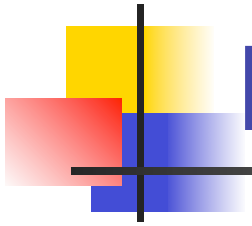
- C

```
while (x != y) {
  if (x < y) {
      y = y - x;
  } else {
      x = x - y;
  }
}
return x;
```



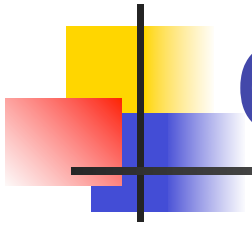
What have we accomplished?

- We can now write any function in Haskell that would have used "internal state" in another language in essentially the same way
- Could have done this before if we were willing to convert imperative function into a functional form
  - now we don't have to



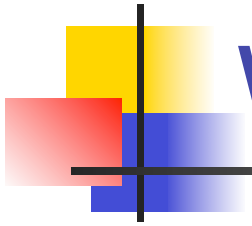
Bottom line

- State monads can be used to implement imperative computations in a functional setting
- Requires a change of perspective:
  - functions don't just map values to values
  - functions map state transformers to state transformers
  - monads make this convenient



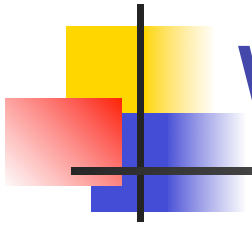
Quote

- "Haskell is the world's best imperative language"



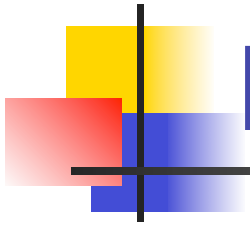
Warning! (1)

- Just because we can express stateful computations in Haskell, doesn't mean they run faster
- Sometimes, would like to write code in imperative style just so it runs faster (like raw C code)
- Haskell provides different tools to do this



Warning! (2)

- To represent the notion of a mutable value, can use
  - `IORef a` -- mutable value of type `a`
  - `STRef a` -- ditto
- `IORef a` runs in `IO` monad, `STRef a` runs in `ST` monad (which we haven't discussed)
- If you do this, code will run very fast



Next time

- Wrap up the lectures
  - Module system
  - Arrays
  - "Maybe" some more monads