

Introduction to stateful monads and computation expressions in F#

2012 James Litsios

<http://equational.blogspot.com/>

Introduction

Monads (aka computation expressions) are good!

- They bring “even more” type safety
- They simplify your code
- They bring quality and productivity

Goal

- Show where they come from
- How they can use them

Using state and maybe monad as stepping stones

Some stateful code

```
type State1 =  
{  
    keyValues:Map<string,float>;  
    counter: int;  
}
```

```
let test1 state =  
    let a = get1 state "a"  
    let b = get1 state "b"  
    let state1 = set1 state "c" (a+b)  
    let (counter, state2) =  
        increment1 state1  
(counter, state2)
```

Some example functions

```
let get1 state key =
  let value = Map.find key state.keyValues
  value

let set1 state key value =
  let newState = { state with
    keyValues = Map.add key value state.keyValues; }
  newState

let increment1 state =
  let newCounter = state.counter+1
  let newState = { state with counter = newCounter; }
  (newCounter, newState)
```

Running test1

```
let testState1 = {  
    keyValues=Map.ofList ["a",1.0;"b",2.0];  
    counter=100;  
}
```

```
let test1 state =  
    let a = get1 state "a"  
    let b = get1 state "b"  
    let state1 = set1 state "c" (a+b) →  
    let (counter, state2) = increment1 state1 →  
        (counter, state2)
```

test1 testState1

```
(101, {  
    keyValues = map [  
        ("a", 1.0);  
        ("b", 2.0);  
        ("c", 3.0)];  
    counter = 101;  
}) )
```

Signature of the functions

get1: State1 -> string -> float

```
let get1 state key =
    let value = Map.find key state.keyValues
    value
```

set1: State1 -> string -> float -> State1

```
let set1 state key value =
    let newState = { state with
        keyValues = Map.add key value state.keyValues; }
    newState
```

increment1: State1 -> int * State1

```
let increment1 state =
    let newCounter = state.counter+1
    let newState = { state with counter = newCounter; }
    (newCounter, newState)
```

Refactor return to be the same as increment1: State1 -> int * State1

```
let get1 state key =  
  let value = Map.find key state.keyValues  
  value  
    float
```

```
let set1 state key value =  
  let newState = { state with  
    keyValues =  
      Map.add key value state.keyValues;  
  }  
  newState  
    State1
```

```
let get2 state key =  
  let value = Map.find key  
    state.keyValues  
  (value, state)  
    float*State1
```

```
let set2 state key value =  
  let newState = { state with  
    keyValues =  
      Map.add key value state.keyValues;  
  }  
  ((), newState)  
    unit*State1
```

test2 with refactored return

```
let testState1 = {  
    keyValues=Map.ofList  
    ["a",1.0;"b",2.0];  
    counter=100;  
}  
  
let test2 state =  
    let (a,state1) = get2 state "a"  
    let (b,state2) = get2 state1 "b"  
    let (_,state3) =  
        set2 state2 "c" (a+b)  
    let (counter, state4) =  
        increment1 state3  
(counter, state4)
```

```
test2 testState1  
(101, {  
    keyValues = map [  
        ("a", 1.0);  
        ("b", 2.0);  
        ("c", 3.0)];  
    counter = 101;  
} )
```

Refactor state as last of arguments

now state is always last!

```
let get2 state key =  
  let value = Map.find key state.keyValues  
  (value, state)
```

```
let set2 state key value =  
  let newState = { state with keyValues =  
    Map.add key value state.keyValues; }  
  ((), newState)
```

```
let increment1 state =  
  let newCounter = state.counter+1  
  let newState = { state with counter =  
    newCounter; }  
  (newCounter, newState)
```

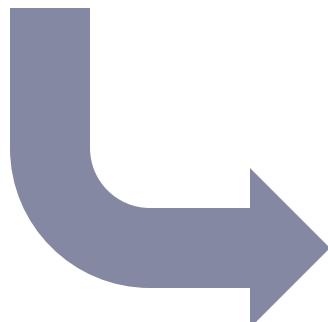
```
let get3 key state =  
  let value = Map.find key state.keyValues  
  (value, state)
```

```
let set3 key value state =  
  let newState = { state with  
    keyValues = Map.add key value state.keyValues;  
  }  
  ((), newState)
```

```
let increment3 state =  
  let newCounter = state.counter+1  
  let newState = { state with counter = newCounter; }  
  (newCounter, newState)
```

test3 with refactored state as last argument

```
let test2 state =  
  let (a,state1) = get2 state "a"  
  let (b,state2) = get2 state1 "b"  
  let (_,state3) = set2 state2 "c" (a+b)  
  let (counter, state4) = increment1 state3  
  (counter, state4)
```



```
let test3 state =  
  let (a,state1) = get3 "a" state  
  let (b,state2) = get3 "b" state1  
  let (_,state3) = set3 "c" (a+b) state2  
  let (counter, state4) = increment3 state3  
  (counter, state4)
```

Refactor as two argument functions

```
let get3 key state =
  let value = Map.find key state.keyValues
  (value, state)

let set3 key value state =
  let newState = { state with
    keyValues =
      Map.add key value state.keyValues;
  }
  ((), newState)

let increment3 state =
  let newCounter = state.counter+1
  let newState = { state with
    counter = newCounter; }
  (newCounter, newState)
```

```
let get4 key state =
  let value = Map.find key state.keyValues
  (value, state)

let set4 (key, value) state =
  let newState = { state with
    keyValues =
      Map.add key value state.keyValues;
  }
  ((), newState)

let increment4 _ state =
  let newCounter = state.counter+1
  let newState = { state with
    counter = newCounter; }
  (newCounter, newState)
```

Refactor state to be an explicit function

```
let get4 key state =
  let value = Map.find key state.keyValues
  (value, state)
```

```
let set4 (key, value) state =
  let newState = { state with
    keyValues =
      Map.add key value state.keyValues;
  }
  ((), newState)
```

```
let increment4 state =
  let newCounter = state.counter+1
  let newState = { state with
    counter = newCounter; }
  (newCounter, newState)
```

```
let get4 key =
  fun state ->
    let value = Map.find key state.keyValues
    (value, state)
```

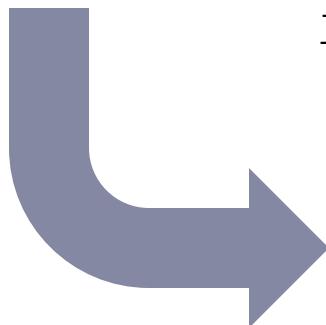
```
let set4 (key, value) =
  fun state ->
    let newState = { state with
      keyValues =
        Map.add key value state.keyValues;
    }
    ((), newState)
```

```
let increment4 _ =
  fun state ->
    let newCounter = state.counter+1
    let newState = { state with
      counter = newCounter; }
    (newCounter, newState)
```

Not systematically used in this presentation to save space !!!

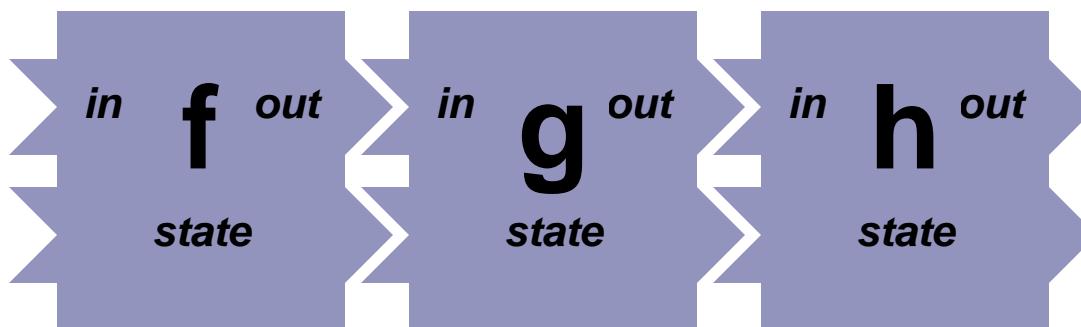
test4: refactored with two argument

```
let test3 state =  
  let (a,state1) = get3 "a" state  
  let (b,state2) = get3 "b" state1  
  let (_,state3) = set3 "c" (a+b) state2  
  let (counter, state4) = increment3 state3  
  (counter, state4)
```



```
let test4 state =  
  let (a,state1) = get4 "a" state  
  let (b,state2) = get4 "b" state1  
  let (_,state3) = set4 ("c", (a+b)) state2  
  let (counter, state4) = increment4 () state3  
  (counter, state4)
```

Idea: assembling functions by matching arguments and returned values



Feeding output into inputs

Regrouping inputs

```
let bind2 f g arg state =  
  let (rf, state2) = f arg state  
  let (rg, state3) = g rf state2  
  (rg, state3)
```

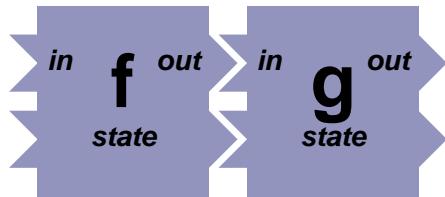
```
let bind3 f g h arg state =  
  let (rf, state2) = f arg state  
  let (rg, state3) = g rf state2  
  let (rh, state4) = h rf state3  
  (rh, state4)
```

```
let bind mf g state =  
  let (rf, state2) = mf state  
  let (rg, state3) = g rf state2  
  (rg, state3)
```

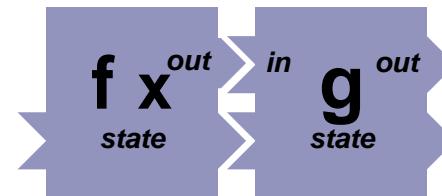
```
let bind31 mf g h =  
  bind (bind mf g) h
```

What did we create?

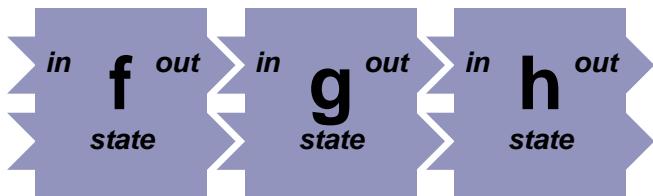
bind2 f g



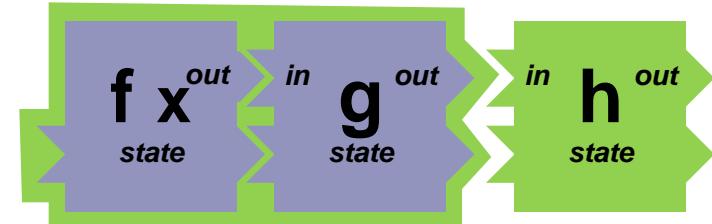
bind (f x) g



bind3 f g h

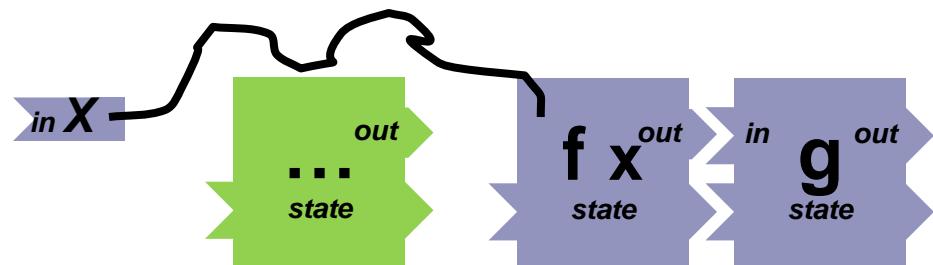


bind31 (f x) g h



Some observations

- *bind* seems to be limited because first parameter is “already applied”
 - Not an issue, first parameter can be introduced at a call point
 - E.g. `fun x -> bind (f x) h`
- Even better
 - First parameter can be brought in “from anywhere you want”!
- E.g. `fun x -> ... bind (f x) g ...`



A handy operator

```
let (>>=) mf g =  
  bind mf g
```

```
let tbind32 mf g h =  
  mf >>= g >>= h
```

```
let bind31 mf g h =  
  bind (bind mf g) h
```

```
let tbind32 mf g h =  
  (mf >>= g) >>= h
```

Nice!

“Binding” get followed by get

bind (f x) g or (f x) >>= g

Looks like:



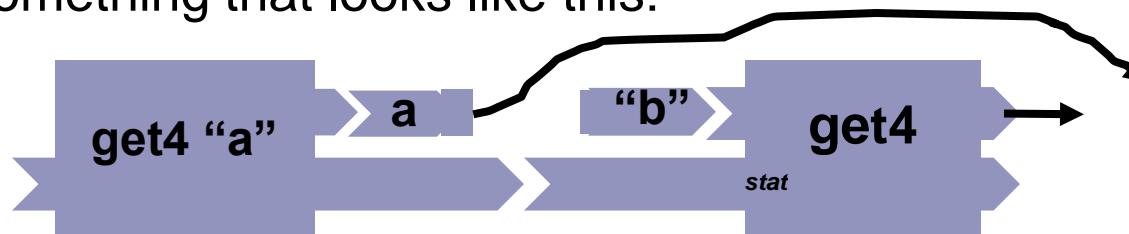
So... (get4 “a”) >>= get4

Looks like this:



Will take the value returned by *get4 “a”* and give it as input to *get4...*
And that is not good!

We want something that looks like this:

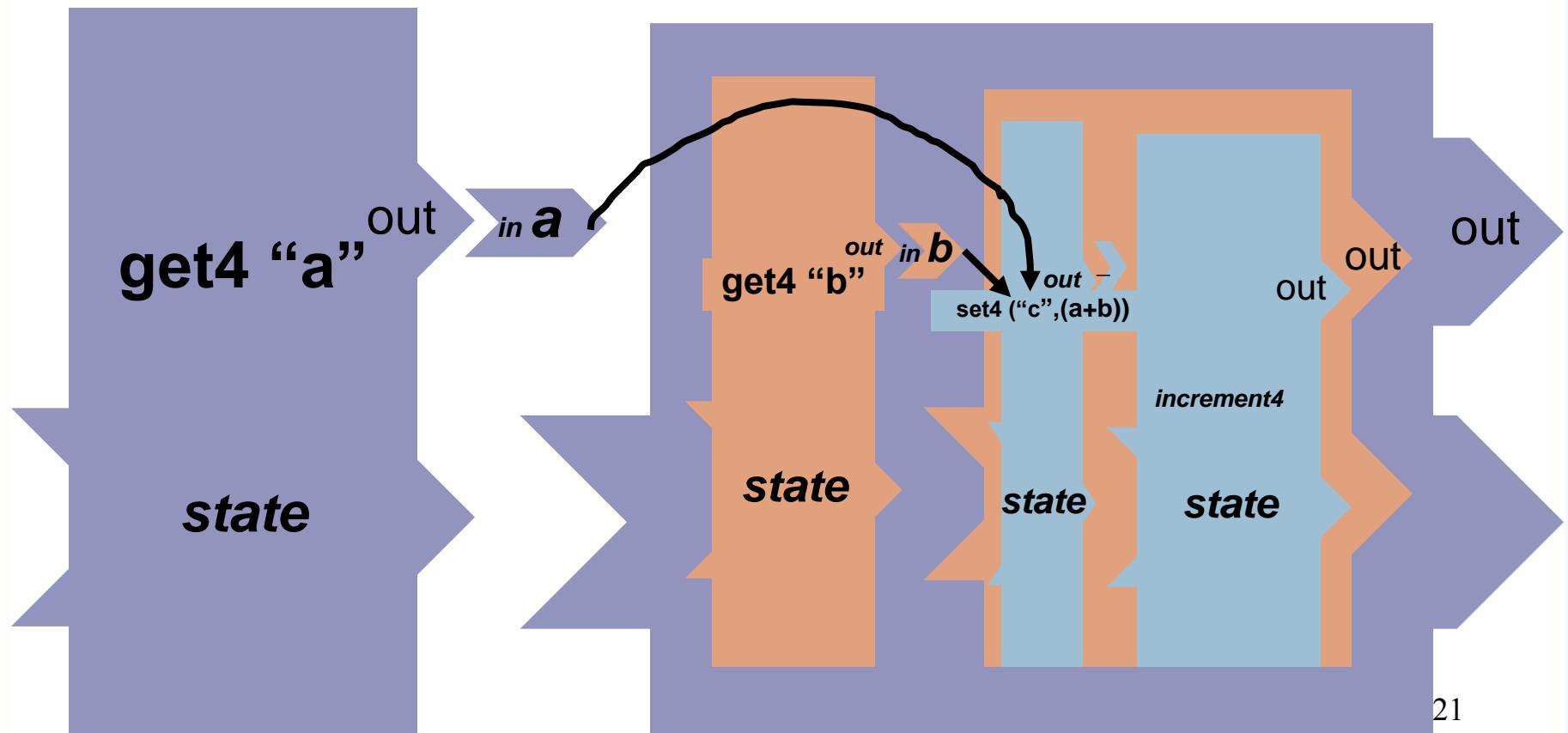


Which can be written as: *(get4 “a”) >>= (fun a -> get4 “b”)*

Really nice!

```
let test6 =
    get4 "a"
  >>= fun a ->
    get4 "b"
  >>= fun b ->
    set4 ("c", (a+b))
  >>= increment4
                                          test6 testState1
                                          (101, {
                                                keyValues = map [
                                                    ("a", 1.0);
                                                    ("b", 2.0);
                                                    ("c", 3.0)];
                                                counter = 101;
                                              } )
```

A graphical illustration



Cleaning up functions with no args

```
let (>>/) mf mg =  
  mf >>= (fun _ -> mg)
```

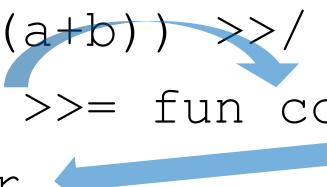
```
let increment5 state =  
  let newCounter = state.counter+1  
  let newState = { state with  
    counter = newCounter;  
  }  
(newCounter, newState)
```

```
let test7 =  
  get4 "a" >>= fun a ->  
  get4 "b" >>= fun b ->  
  set4 ("c", (a+b)) >>/  
  increment5
```

A first example of “monadic programming”

```
let ret x = fun state -> (x, state)
```

```
let test8 =  
  get4 "a"  >>= fun a ->  
  get4 "b"  >>= fun b ->  
  set4 ("c", (a+b)) >>/  
  increment5 >>= fun counter ->  
ret counter
```



```
test8 testState1  
(101, {  
  keyValues = map [  
    ("a", 1.0);  
    ("b", 2.0);  
    ("c", 3.0)];  
  counter = 101;  
} )
```

The “do” notation: computation expressions

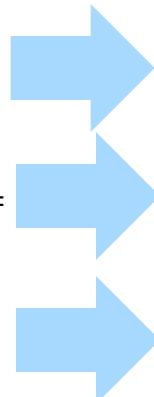
```
let test8 =  
  get4 "a" >>= fun a => >>= becomes let!  
  get4 "b" >>= fun b =>  
  set4 ("c", (a+b)) >>/ >>/ becomes do!  
  increment5 >>= fun counter =>  
ret counter
```

```
let test9 =  
  state1 {  
    let! a = get4 "a"  
    let! b = get4 "b"  
    do! set4 ("c", (a+b))  
    let! counter = increment4  
    return counter  
  }
```

Setting up the “do” notation

```
type State1() =  
    member b.Bind (m, f) =  
        m >>= f  
    member b.Combine (m1, m2) =  
        m1 >>/ m2  
    member b.Return x =  
        ret x
```

```
let state1 = State1()
```



```
let test9 =  
    state1 {  
        let! a = get4 "a"  
        let! b = get4 "b"  
        do! set4 ("c", (a+b))  
        let! counter = increment4  
        return counter  
    }
```

“Do” notation is just a notation!

```
let test10 =
  get4 "a" >>= fun a ->
  get4 "b" >>= fun b ->
state1 {
  do! set4 ("c", (a+b))
  let! counter = increment5
  return counter
}

test10 testState1
(101, {
  keyValues = map [
    ("a", 1.0);
    ("b", 2.0);
    ("c", 3.0)];
  counter = 101;
})
```

Applying the do notation

```
let get4 key state =
  let value = Map.find key state.keyValues
  (value, state)
```



```
let getState f =
  (fun s -> (f s, s))
let mapState f =
  (fun s -> ((), f s))
let getKeyValues state =
  state.keyValues
let getCounter state =
  state.counter
```

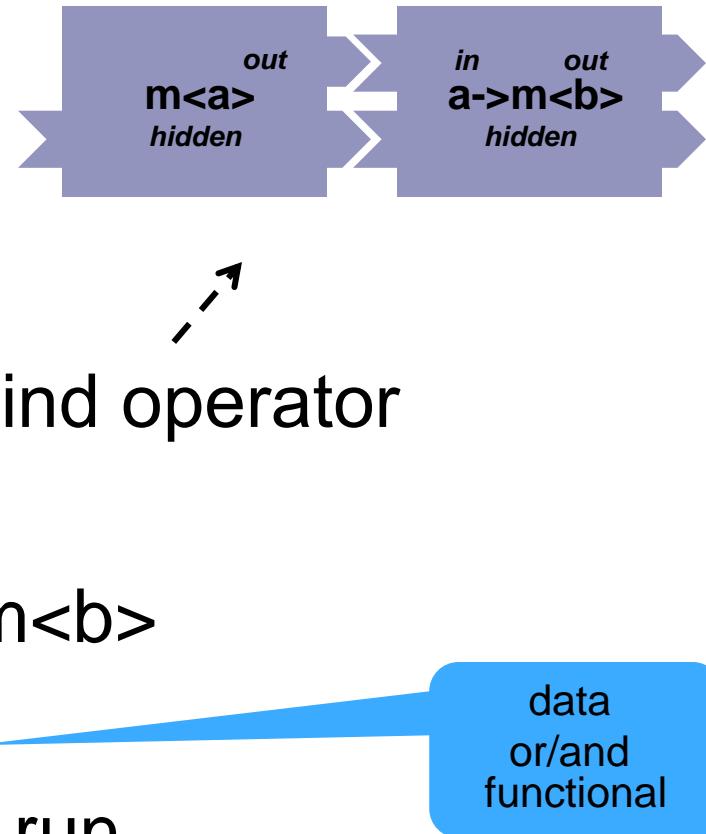
```
let get5 key =
  state1 {
    let! kv = getState getKeyValues
    return Map.find key kv
  }
```

```
let set6 (key, value) = ...
let increment6 = ...
```

Welcome to the monad!

A monad is

- polymorphic
- composable
- flexibly “linkable” with the bind operator
 - “all ready”: $m< a >$
 - “waiting for an arg”: $a \rightarrow m< b >$
- carries something hidden
- still needs to be executed / run



Making the execution more explicit

run1 takes a state monad and “runs it” with a state

```
let run1 m state = m state
```

test12 testState1

```
let test12 state = run1 test11  
state
```

(**101**, {
keyValues = map [

```
let test11 =  
state1 {  
let! a = get5 "a"  
let! b = get5 "b"  
do! set5 ("c", (a+b))
```

("a", 1.0);
("b", 2.0);
("c", 3.0)];
counter = **101**;

```
let! counter = increment6  
return counter
```

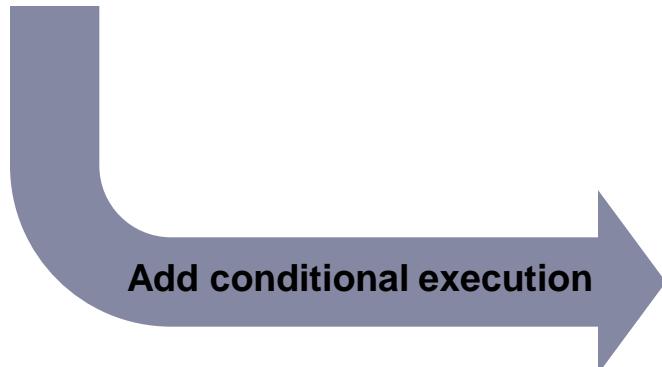
```
}
```

Giving the monad its own type

```
type SM's, 'r> =  
  SM of ('s -> 'r*'s)  
  
let getState f =  
  SM (fun s -> (f s, s))  
  
let mapState f =  
  SM (fun s -> ((), f s))  
  
let ret x =  
  SM (fun s -> (x, s))  
  
let run m state =  
  match m with  
  | SM f -> f state  
  
let get6 key =  
  SM(fun state->  
    ...)  
  
let bind mf g =  
  SM(fun state->  
    let (rf, state2) =  
      run mf state  
    let (rg, state3) =  
      run (g rf) state2  
    (rg, state3))
```

The stateful maybe/option monad

```
let bind m g state =  
  let (rf, state2) =  
    run m state  
  let (rg, state3) =  
    run (g rf) state2  
(rg, state3)
```



```
let bind m g state =  
  let (rf, state2) =  
    run m state  
  match rf with  
  | Some rfv ->  
    let (rg, state3) =  
      run (g rfv) state2  
    match rg with  
    | Some _ ->  
      (rg, state3)  
    | None ->  
      (None, state)  
  | None ->  
    (None, state)
```

Refactor using stateful maybe monad

```
let get5 key =
  SM(fun state->
    let value =
      Map.find key state.keyValues
    (value, state))
```

```
let set5 (key, value)=
  SM(fun state->
    let newState = { state with
      keyValues =
        Map.add key value
      state.keyValues;
    }
    (((), newState)))
```

```
let increment6 =
  SM(fun state->
    let newCounter = state.counter+1
    let newState = { state with
      counter = newCounter; }
    (newCounter, newState))
```

```
let get6 key =
  SM(fun state->
    let optValue =
      Map.tryFind key state.keyValues
    (optValue, state))
```

```
let set6 (key, value) =
  SM(fun state->
    let newState = { state with
      keyValues =
        Map.add key value state.keyValues;
    }
    (Some (), newState))
```

```
let increment7 =
  SM(fun state->
    let newCounter = state.counter+1
    let newState = { state with
      counter = newCounter; }
    (Some newCounter, newState))
```

Refactor using stateful maybe monad (do)

```
type OptState() =
    member b.Bind (m, f) = ...
    member b.Combine (m1, m2) = ...
    member b.Return x = ...

let get5 key =
    state1 {
        let! kv = getState getKeyValues
        return Map.find key kv
    }

let set5 (key, value) =
    state1 {
        do! mapState
        (mapKeyValues (Map.add key value))
    }

let increment6 =
    state1 {
        ...
    }

let getState' f =
    (fun s -> (Some (f s), s))

let mapState' f =
    (fun s -> (Some (), f s))

let someValue optX =
    (fun s-> (optX, s))

let get7 key =
    optState {
        let! kv = getState' getKeyValues
        let! value =
            someValue (Map.tryFind key kv)
        return value
    }

let set7 (key, value) =
    optState {
        do! mapState'
        (mapKeyValues (Map.add key value))
    }

let increment8 =
    optState {
        ...
    }
```

Stateful and conditional execution: ok

```
let test14 =
  optState {
    let! a = get6 "a"
    let! b = get6 "b"
    do! set6 ("c", (a+b))
    let! counter = increment7
    return counter
  }
  run test14 testState1
  (Some 101, {
    keyValues = map [
      ("a", 1.0);
      ("b", 2.0);
      ("c", 3.0)];
    counter = 101;
  } )
```

Stateful and conditional execution: fail

```
let test15 =  
  optState {  
    let! a = get6 "a"  
    let! b = get6 "x"  
    do! set6 ("c", (a+b))  
    let! counter = increment7  
    return counter  
  }  
  
run test15 testState1  
  
(None, {  
  keyValues = map [  
    ("a", 1.0);  
    ("b", 2.0)];  
  counter = 100;  
})
```

Working with hierarchical states

```
type State2 =
    {
        kstate1:Map<string,State1>;
        zstate:State1;
    }

let testState2 = {
    kstate1=Map.ofList
        ["x",testState1;
         "y",testState1];
    zstate1=testState1;
}
```

```
let zState1InState2 f state2 =
    let (r, state1) = f state2.zstate1
    (r, { state2 with zstate=state1 })

let adaptM fold m =
    SM (fun state ->
        fold (run m) state)

E.g.
adaptM zState1InState2 (...)
```

Example with inner state: apply to zstate1 field

```
let test16 =  
  adaptM zState1InState2  
    (state1 {  
      let! a = get4 "a"  
      let! b = get4 "b"  
      do! set4 ("c", (a+b))  
      let! counter = increment5  
      return counter  
    })
```

```
(101,  
{ kstate1 =  
  map  
    [ ("x", {keyValues = map [ ("a", 1.0); ("b", 2.0)];  
           counter = 100;});  
     ("y", {keyValues = map [ ("a", 1.0); ("b", 2.0)];  
           counter = 100;})];  
  zstate1 = {keyValues = map [ ("a", 1.0); ("b", 2.0); ("c", 3.0)];  
            counter = 101;}; })
```

Working with “keyed” hierarchical states

```
type State2 =
  {
    kstate1:Map<string,State1>;
    zstate1:State1;
  }

let kState1InState2 k f state2 =
  let (r, state1) =
    f (Map.find k state2.kstate1)
  let state2' = { state2 with
    kstate1=
      (Map.add k state1 state2.kstate1) }
  (r, state2'')
```

```
let adaptM fold m =
  SM (fun state -> fold (run m) state)
```

E.g.

```
adaptM (kState1InState2 "x") { ... }
```

```

let test17 =
adaptM (kState1InState2 "x")
(state1 {
    let! a = get5 "a"
    let! b = get5 "b"
    do! set5 ("c", (a+b))
    let! counter = increment6
    return counter
})

```

```

(101,
{kstate1 =
map
[ ("x", {keyValues = map [ ("a", 1.0); ("b", 2.0); ("c", 3.0)];
            counter = 101; });
  ("y", {keyValues = map [ ("a", 1.0); ("b", 2.0) ];
            counter = 100; })];
zstate1 = {keyValues = map [ ("a", 1.0); ("b", 2.0) ];
            counter = 100; }; })

```

Example with inner state: apply to kstate1 element

Working with monadic functions on hierarchical states

```
let get6 key = SM(...)  
let get7 key = state1{...}  
let set7 (key, value) = SM(...)  
let set7 (key, value) = optState { ... }  
...  
let getInZ1 key = adaptM zState1InState2 (get6 key)  
...  
let adaptM' fold m' =  
    fun x -> adaptM fold run (m' x)  
  
let getInZ2 = adaptM' zState1InState2 get6  
let setInZ2 = adaptM' zState1InState2 set6  
...
```

```

let test18 =
state1 {
  let Z m = adaptM zState1InState2 m
  let KVX m = adaptM (kState1InState2 "x") m
  let Z' m' = adaptM' zState1InState2 m'
  let KVX' m' = adaptM' (kState1InState2 "x") m'
  let! a = Z (get5 "a")
  let! b = (Z' get5) "b"
  do! (KVX' set5) ("c", (a+b))
  let! counter1 = KVX increment6
  let! _ = KVX increment6
  let! counter2 = Z increment6
  return counter2
}

```

(101,

```

{kstate1 =
map
[("x", {keyValues = map [("a", 1.0); ("b", 2.0); ("c", 3.0)];
          counter = 102; });
 ("y", {keyValues = map [("a", 1.0); ("b", 2.0)];
          counter = 100; })];
zstate1 = {keyValues = map [("a", 1.0); ("b", 2.0)];
           counter = 101; }; })

```

Example with inner state: apply at multiple levels

```

let test19 =
  optState {
    let Z m = adaptM zState1InState2 m
    let KVX m = adaptM (kState1InState2 "x") m
    let Z' m' = adaptM' zState1InState2 m'
    let KVX' m' = adaptM' (kState1InState2 "x") m'
    let! a = Z (get6 "a")
    let! b = (Z' get6) "b"
    do! (KVX' set6) ("c", (a+b))
    let! counter1 = KVX increment7
    let! _ = KVX increment7
    let! counter2 = Z increment7
    return counter2
  }

```

```

(Some 101,
{kstate1 =
map
[("x", {keyValues = map [("a", 1.0); ("b", 2.0); ("c", 3.0)];
counter = 102;});
("y", {keyValues = map [("a", 1.0); ("b", 2.0)];
counter = 100;})];
zstate1 = {keyValues = map [("a", 1.0); ("b", 2.0)];
counter = 101;};})

```

**Example with
inner state:
apply at
multiple levels
in maybe monad
(ok)**

```

let test20 =
  optState {
    let Z m = adaptM zState1InState2 m
    let KVX m = adaptM (kState1InState2 "x") m
    let Z' m' = adaptM' zState1InState2 m'
    let KVX' m' = adaptM' (kState1InState2 "x") m'
    let! a = Z (get6 "a")
    let! b = (Z' get6) "b"
    do! (KVX' set6) ("c", (a+b))
    let! counter1 = KVX increment7
    let! _ = KVX increment7
    let! counter2 = Z increment7
    let! err = (Z' get6) "x"
    return counter2
  }

  (null,
   {kstate1 =
    map
      [ ("x", {keyValues = map [ ("a", 1.0); ("b", 2.0) ];
                 counter = 100; });
       ("y", {keyValues = map [ ("a", 1.0); ("b", 2.0) ];
                 counter = 100; })];
    zstate1 = {keyValues = map [ ("a", 1.0); ("b", 2.0) ];
               counter = 100; }; })

```

**Example with
inner state:
apply at
multiple levels
in maybe monad
(fail)**

Wrapping it up: Summary

Refactored

- return
- state as last arg
- Just two args
- Match arg and ret;
and states

bind function

bind operator ($>>=$)

bind op no return ($>>/$)

“do” notation

- computation expressions

“do” notation class def

Presented the monad

- Wrapped in type
and run function

Run in bind

Maybe monad

Hierarchical states

- No args (adaptM)
- With args (adaptM')
- Over many expr
- For single expr

Leave a comment!

<http://equational.blogspot.com/2012/12/introduction-to-stateful-monads.html>