Monads and State machines

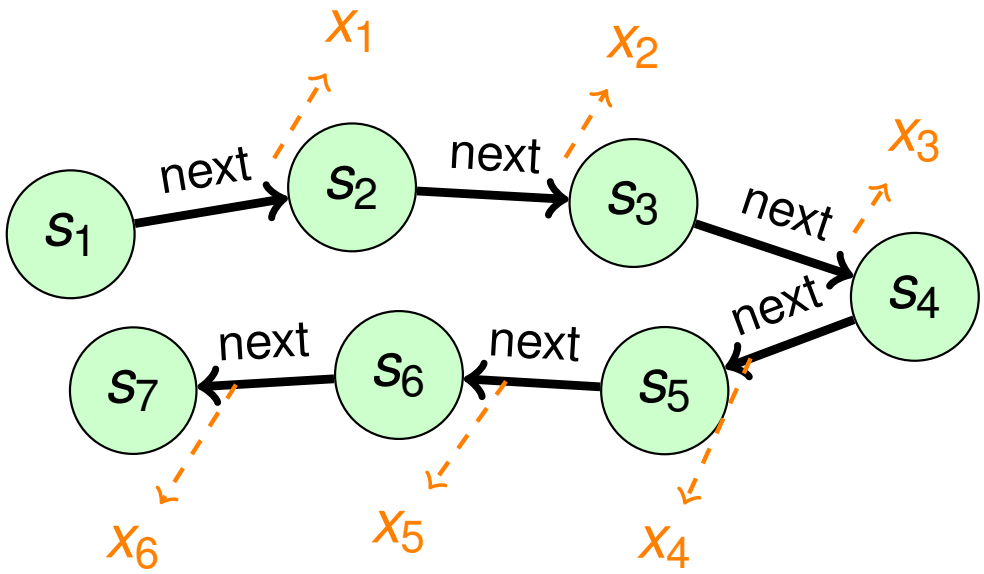
Functional Programming and Intelligent Algorithms

Prof Hans Georg Schaathun

Høgskolen i Ålesund

14th February 2017

The state machine



- `next :: State -> (State,Int)`
- Lehmer:
```
next s = (s',s')
    where s' = (a + x*s) `mod` m
```
- Cipher: `next s = (s + 1 `mod` m, encrypt k s)`

State machines in functional programming

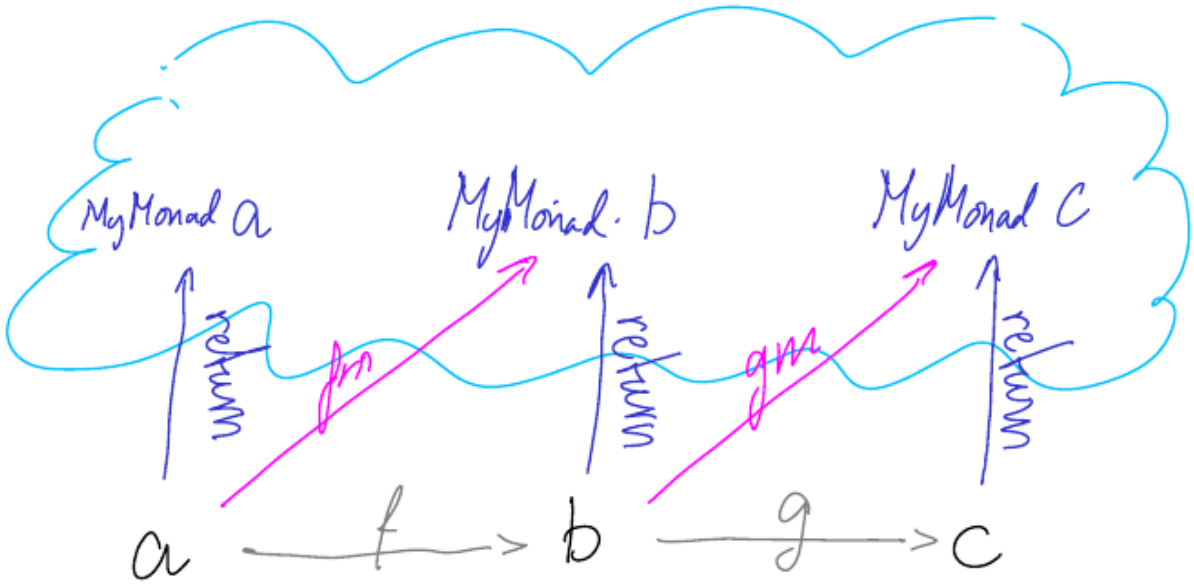
What is special about state in functional programming?

State machines in functional programming

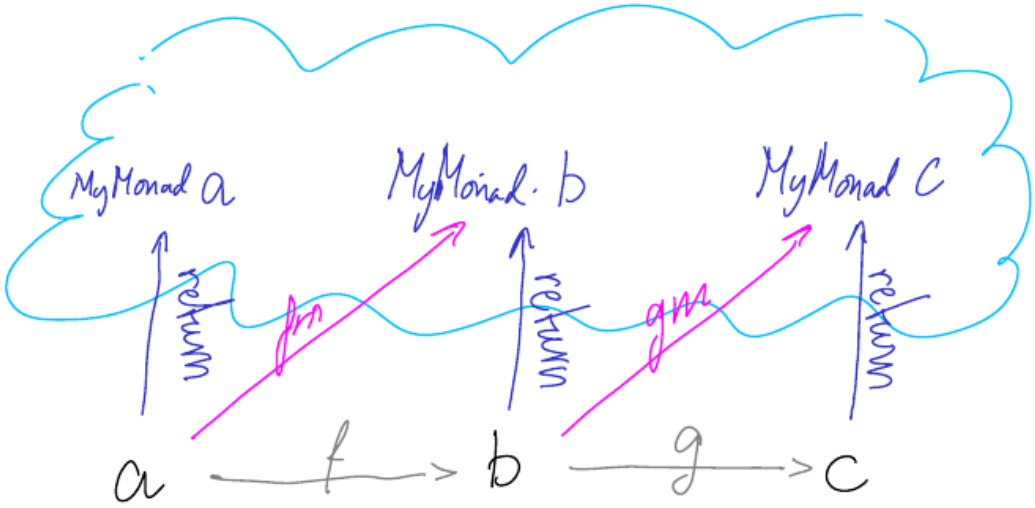
What is special about state in functional programming?

Haskell uses monads

Hiding in the Clouds



Monadic and non-monadic functions



Pure functions	`f :: a -> b`
	`g :: b -> c`
Monadic functions	`fm :: a -> MyMonad b`
	`gm :: b -> MyMonad c`
Return	`return :: x -> MyMonad x`

Function composition

Pure functions	
1. $h = f \circ g$ 2. $h(x) = f(g(x))$	
Or in Haskell	
1. `h = f . g` 2. `h x = f $ g x`	

Function composition

Pure functions	Monads (Binding operations)
1. $h = f \circ g$ 2. $h(x) = f(g(x))$	1. `fm :: a -> MyMonad b` 2. `gm :: b -> MyMonad c` 3. `hm = fm >>= gm` 4. `hm :: a -> MyMonad c`
Or in Haskell	
1. `h = f . g` 2. `h x = f $ g x`	

Function composition

Pure functions	Monads (Binding operations)
1. $h = f \circ g$ 2. $h(x) = f(g(x))$	1. `fm :: a -> MyMonad b` 2. `gm :: b -> MyMonad c` 3. `hm = fm >>= gm` 4. `hm :: a -> MyMonad c`
Or in Haskell	**Equivalently**
1. `h = f . g` 2. `h x = f $ g x`	1. `hm x = do` 1.1 `y <- fm x` 1.2 `gm y`

Mixing pure and monadic functions

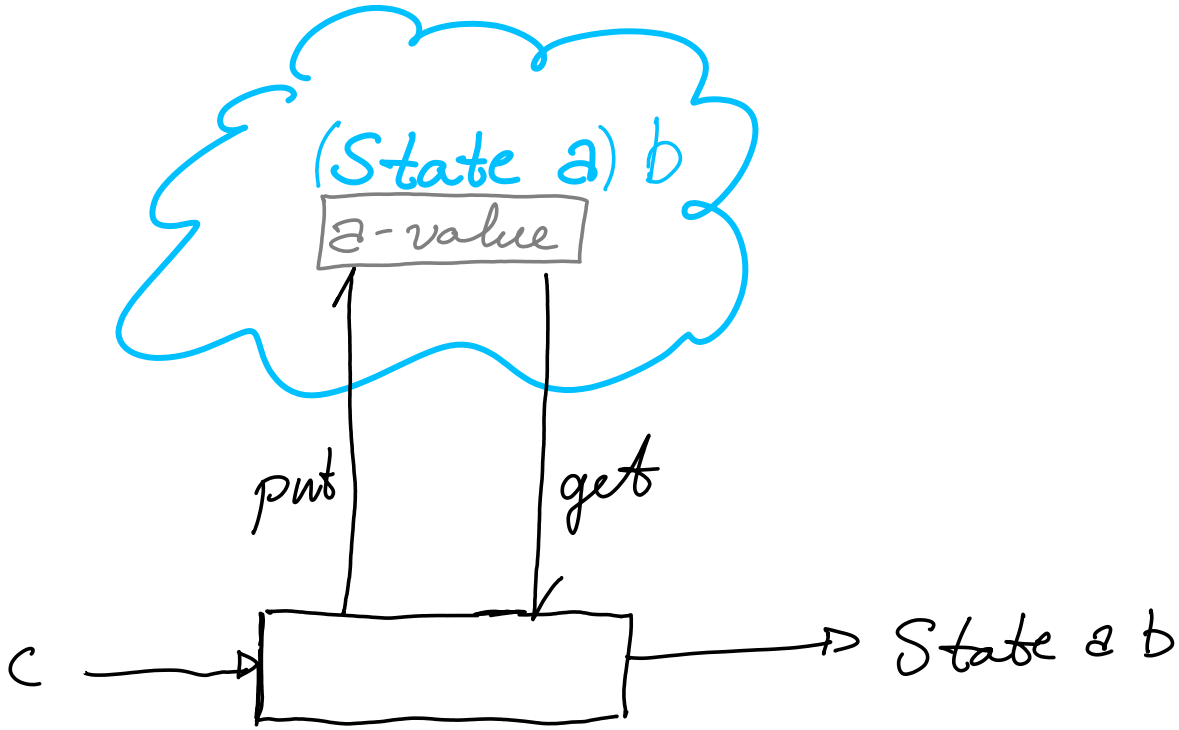
```
hm x = do
     y <- fm x
     let z = g y
     return z
```

Mixing pure and monadic functions

```
hm x = do
     y <- fm x
     let z = g y
     return z
```

`fx :: MyMonad a -> b` *is impossible*

The State Monad



A State Machine for Random Numbers

```
1. import Data.Word32
2. getRandom ::  State TFGen Word32
3. getRandom = do
   3.1 s <- get
   3.2 let (r,s') = next s
   3.3 put s'
   3.4 return r
```

Running the state machine

1. `f :: IO State TFGen a`
2. `g :: TFGen`
3. `runState f g :: (a,TFGen)`

Summary

— The State monad enables a PRNG state
  - without explicitly passing the state in and out of every function

— To use it, functions must be monadic
  - just like IO

— Compose stateful actions using `do`
  - or, if you prefer, `>>=` and `>>`