Monads and State machines

Functional Programming and Intelligent Algorithms

Prof Hans Georg Schaathun

Høgskolen i Ålesund

10th February 2015

Outline

The PRNG is a state machine

1. The state decides what the PRNG will output
   - *Lehmer's algorithm* x_{i-1} is the state
   - *Counter mode* i is the state
2. State transition
   - *Lehmer's algorithm* $x \mapsto a + cx \mod m$
   - *Counter mode* $i \mapsto i + 1$
3. Output function
   - *Lehmer's algorithm* $x \mapsto a + cx \mod m$
   - *Counter mode* $i \mapsto e_k(i + 1)$

State machines in functional programming

How do we handle state machines in functional programming?

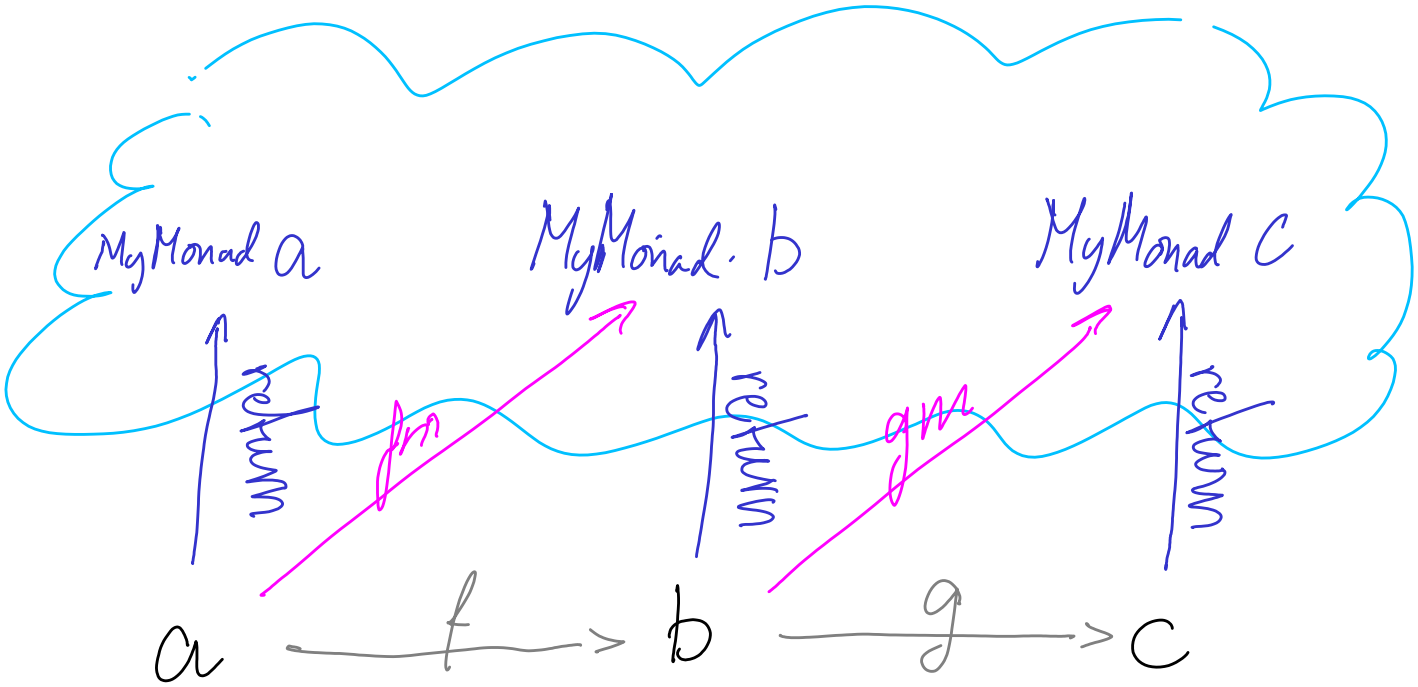
1. What is special about functional programming?
2. What is difficult?
3. How can we do it?

Outline

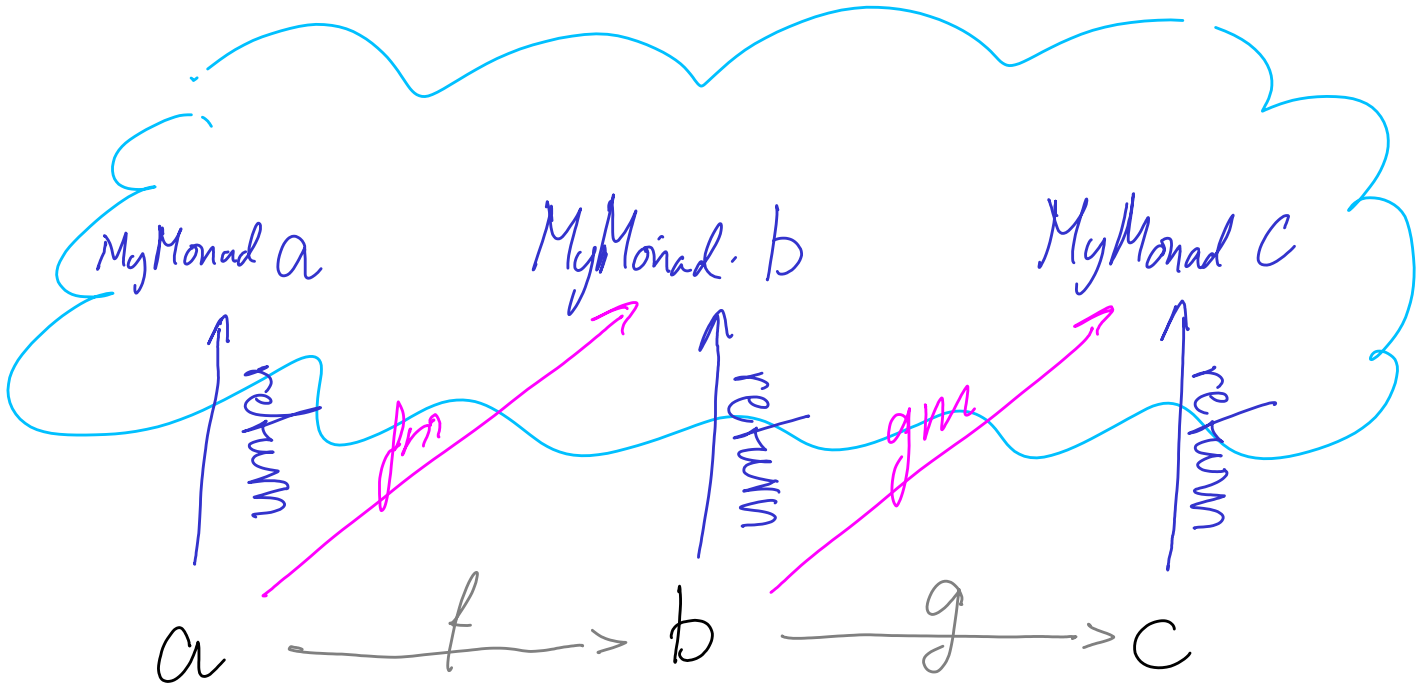
Monads

1. Category theory
2. Operations on types

Hiding in the Clouds



Hiding in the Clouds



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

Function composition

Combining 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`

Binding operations

Combining monadic functions

1. `fm :: a -> MyMonad b`
2. `gm :: b -> MyMonad c`
3. `hm :: a -> MyMonad c`
4. `hm = fm >>= gm`

Equivalently

1. `hm x = do`
    1. `y <- fm x`
    2. `gm y`

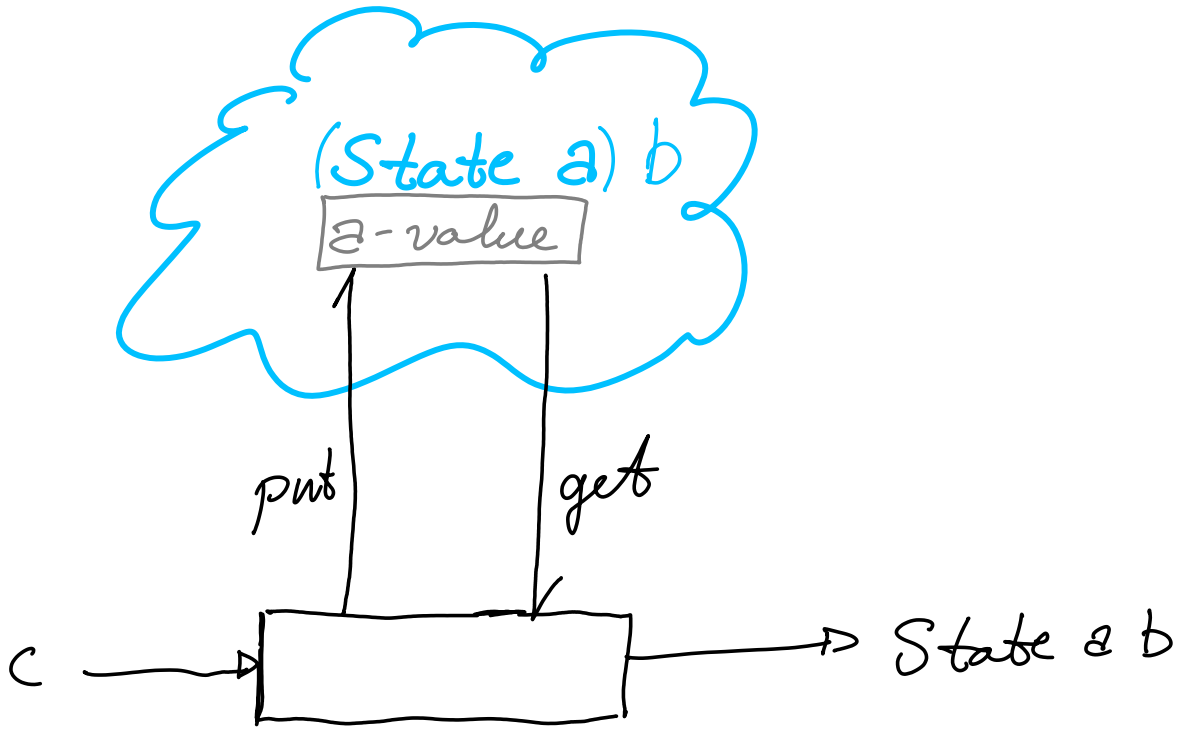
Mixing pure and monadic functions

1. `hm x = do`
   1. `y <- fm x`
   2. `let z = g y`
   3. `return z`

- If you use a monad, you have to return a monad
- `fx :: MyMonad a -> b` is impossible

Outline

The State Monad



A State Machine for Random Numbers

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

Composing actions

1. The State monad allows stateful actions
   - in the same way as IO actions
2. Compose them using `do`
   1. Access to `put` and `get`
   2. Use custom stateful actions like `getRandom`

Running the state machine

If you have a composite State action `f`, you can run the state machine.

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

The output is the contents of the action `f` and the final state.

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 `>>`