

@AndrzejWasowski@scholar.social

Andrzej Wąsowski
Florian Biermann

Advanced Programming

State Monad

IT UNIVERSITY OF COPENHAGEN

SOFTWARE
QUALITY
RESEARCH

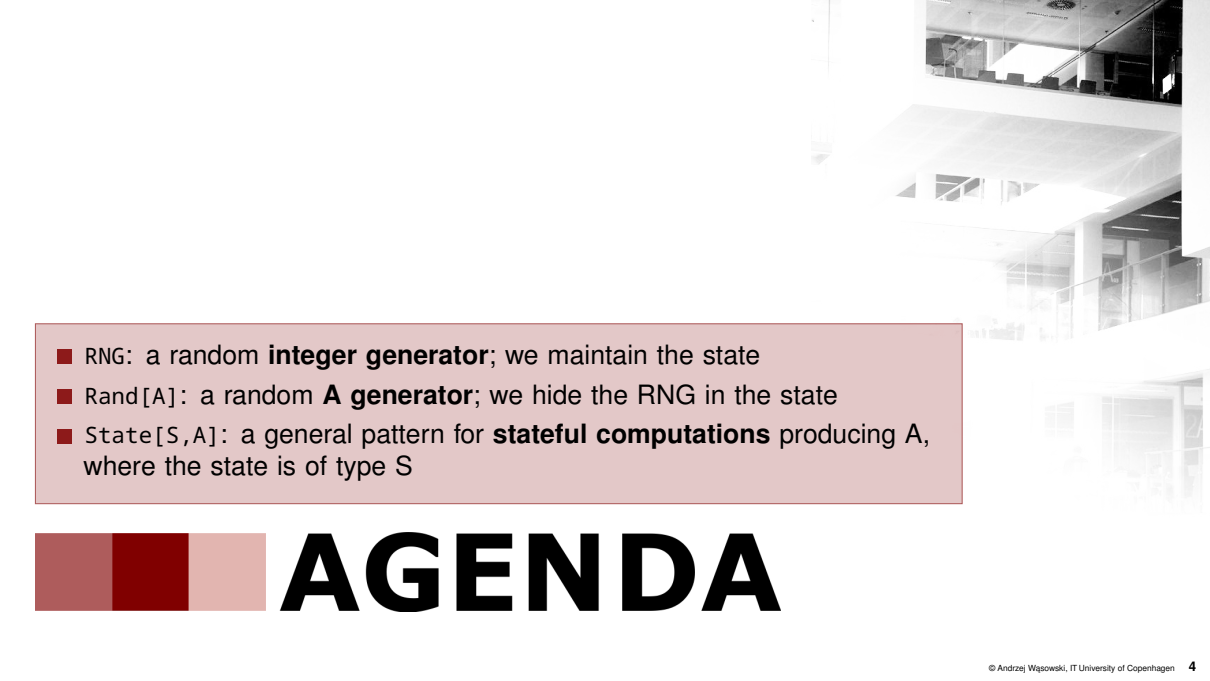
Does FP Eliminate State?

Picture: You after four weeks of ADPRO

- **Variables** and fields are state
- Program **stack&heap** are state
- **Databases** are state
- **Program counter** is state!
- Variable **assignments** set state
- Variable **accesses** read state
- **Loops** must change state
- **Exceptions** modify program counter and stack
- We have **disallowed** almost all of these?

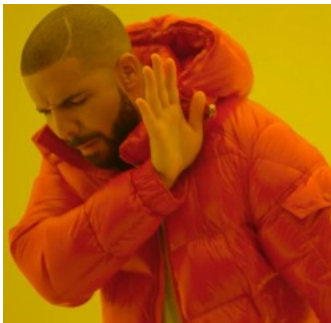
- In FP we often make **state explicit**:
 - Converted loops to **recursive functions**
 - A function is a **state transform**
 - **Arguments** are the state explicitly
 - **No other implicit**, hidden state that can be **changed** by others
- Today, a pure **pattern** for state transforms:
 - A more **recipe-like** way to encode state
 - Allows to hide the state, make it **implicit** like in imperative programming
 - Still the state is **encapsulated** in a well defined value
 - Still **no other** encapsulated state that can be **changed** by others



- 
- RNG: a random **integer generator**; we maintain the state
 - Rand[A]: a random **A generator**; we hide the RNG in the state
 - State[S,A]: a general pattern for **stateful computations** producing A, where the state is of type S



AGENDA



var



State[S,A]

A Typical Stateful Imperative API

```
var rng = new scala.Util.Random  
  
// returns a random number form 0 to 5  
def rollDie: Int = rng.nextInt(6)
```

- We call `rollDie` and observe a value 5 the first time, and 0 the second time
- Mentimeter 5427 6049: What is the result of `rollDie + rollDie`?
- What does it tell us about referential transparency of `rollDie`?
- `rng` is an external, implicit state, that can be changed by others
- To make `rollDie` referentially transparent, make the state explicit

Converting RNG to explicit state

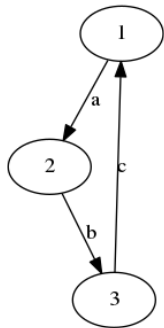
- We had: `RNG.nextInt: () => Int`
- Lets **return new state explicitly**, instead of modifying old (RT)

```
trait RNG:  
  def nextInt: (Int, RNG)  
  
object RNG:  
  def nextInt(rng: RNG): (Int, RNG) =  
    rng.nextInt
```

- **In general** a function: `State => (Output, State)`
- **Wrap** this as `case class State[S, +A] (run: S => (A, S))`
- So RNG becomes `State[RNG, Int]` (`run = RNG.nextInt`)
- **Intuition 1:** Automaton or Transition would be better names than State
- **Intuition 2:** step would be a better name than run

Consider a Simple Automaton

Stateful by definition



```
var state = 1
while true do
  state match
    case 1 => print("a"); state = 2
    case 2 => print("b"); state = 3
    case 3 => print("c"); state = 1
```

```
var x = 0
while true do
  print(x)
  x+=1
```

```
def step(state: Int): (String, Int) =
  state match
    case 1 => ("a", 2)
    case 2 => ("b", 3)
    case 3 => ("c", 1)
```

```
def loop(state: Int = 1) =
  val (output, state1) = step(state)
  print(output)
  loop(state1)
```

- This automaton as an **instance of State**: `State[Int,String](step)`
- More precisely the instance is: `State[Int,String](run = step)`
- Convert the second while loop to a similar instance of State (5min)

States vs Streams

- We can **unroll** (unfold) a state machine from an initial state, producing a word of actions (a lazy list)
- **Discuss:** What is the lazy list from our first automaton?
- **Discuss:** What is the lazy list from our second automaton?
- **Mentimeter 5427 6049:** another stream
- An **exercise** implementing this mapping as a function
- **Laziness** of streams is useful here, why?

Anything stateful maps to the state pattern

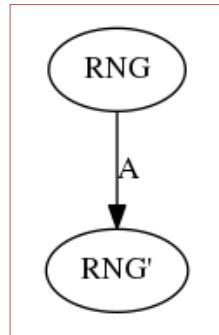
Recap

- Random number generators (state: RNG seed)
- Websites with modality (session state)
- Database backed applications (DB state)
- Communication protocols (protocol state)
- etc.

Random Number Generator as an Instance of State

```
type Rand[A] = State[RNG,A]
```

- RNG is the state of the random generator (usually some large number encapsulated)
- The textbook gives a simple implementation of RNG based on multiplication with large primes module 64 bits
- Rand[A] is a **computation** that we can run, then it will produce a random A and a new state RNG
- Another useful intuition: Rand[A] is a **generator of random A's**
- Or even just a **"random A"**



How do I use this generator of random numbers?

```
type Rand[A] = State[RNG, A]
val r: Rand[Int] = ...
val (i, r1) = r.run(SimpleRNG(42))
```

- SimpleRNG is the book's concrete implementation of the RNG trait
- 42 is the initial seed (state)
- (r1,i) is a new state and a random number
- **Question:** How do I get the next random number?
- **Question:** What happens if I call `r.run` again?

What can we do with Automata/State?

State is a monad, similar key operations as for List, Option, and Stream

```
def map[S, A, B](s: State[S, A])(f: A => B): State[S, B]
```

Can use this to generate even numbers:

```
val even: Rand[Int] = map[Int](r) { n => n * 2 }
```

Automata can be composed [1/2]

flatMap can be used to compose generators:

```
def flatMap[S, A, B](s: State[S, A])(f: A => State[S, B]): State[S, B]
```

Function `f` takes values produced by `s` and uses them to construct a new automaton.
In the context of our `rand`:

```
def flatMap[A, B](r: Rand[A])(f: A => Rand[B]): Rand[B] =  
  flatMap[A, B](r: State[RNG, A])(f: State[RNG, B]): State[RNG, B]
```

flatMap can compose generators (compute a random size list of random even integers):

```
val int: Rand[Int] = ... (assume you have it)  
def ints(n: Int): Rand[List[Int]] = ... (assume creates a random list of given length)  
val ns: Rand[List[Int]] = int.flatMap { n => ints(n) }
```

The state RNG passed **implicitly**; size generated with different state than each number

Automata can be composed [2/2]

The `map2` function computes a zipping of two automata over the same state space:

```
map2[S, A, B, C](sa: State[S, A])(sb: State[S, B])(f: (A, B) => C): State[S, C]
```

Could be used to create a product automaton synchronizing two computations, then `C` is `(A, B)`.

More fun in exercises :)

Next Week

- Next week, we will use the generators of random numbers to implement a modern testing framework
- So: keep reading the chapters and solve the exercises!