INFO0054-1
Programmation Fonctionnelle

Chapter 06: Purely functional state

Christophe Debruyne

(c.debruyne@uliege.be)

References

- Chapter 6: Purely functional state.
  Paul Chiusano and Runar Bjarnason. Functional Programming in Scala, Manning Publications, 2015.

Overview

- Purely functional state
  - Problems of (APIs with) side effects
  - A functional random number generator
  - Generalization step 1: Pure APIs for state transitions
  - Generalization step 2: A state transition data type
  - Simulating finite-state machines: a coin-operated turnstile

The thought exercise of identifying patterns and creating abstractions is important here!

Part 01:
Purely functional state

Purely functional state

- Manipulating state is convenient, and we have seen how we can manipulate state with side effects in other courses. Remember, side-effects make things more difficult to test, manage, etc.

- In this part of the lesson, we will ask ourselves how we can model "state" in functional programming. It turns out that this is simple:
  - We will use objects that represent states and state transitions.
  - We will write functions that accept a state as an argument and returns a new state along with its result.

Random numbers with side-effects

- Scala's library for generating pseudo-random numbers relies on side-effects.

```
scala> val r = new scala.util.Random
val r: scala.util.Random = scala.util.Random@74fda9ed

scala> r.nextInt(10)
val res0: Int = 2

scala> r.nextInt(10)
val res1: Int = 3
```

Generating random integers between [0,10[.

- After each invocation, the internal state of `r` is changed. This may have a negative impact on the "testability" of our code. For example…

A virtual D20

```scala
def rolld20: Int = {
    val r = new scala.util.Random
    r.nextInt(20)
}
```

The specification of rolld20 is simple: an invocation returns a random integer in the interval [1,20].

This implementation contains a bug but satisfies the specification most of the time.

Indeed, it returns numbers between [0,19] and thus fails, statistically, 1 in 20 times. 20 is never returned either.

If we were to test this function, we may end up in a situation where this test fails sometimes, but usually succeeds.

By the way:

- A specification prescribes that a function does and usually involves information about preconditions, postconditions, inputs, outputs, etc.
- A specification gives us information on what a function does, and not how it is done. In fact, multiple implementations may satisfy a specification and the caller is unaware how it is implemented.

A virtual D20

```
~~def rolld20: Int = {~~
~~    val r = new scala.util.Random~~
~~    r.nextInt(20)~~
~~}~~
```

What about providing the random number generator as an argument?

```
import scala.util.Random
def rolld20(r: Random): Int = r.nextInt(20)
```

When a test fails, we now have the random generator that caused the test to fail. But does this solve our problem? No! The internal state of that object has changed, so there's little chance that it will fail again. And recreating a new random generator object with the same state is difficult.

How can we solve this? The answer lies in avoiding side-effects and recovering referential transparency by making state changes (or state updates) explicit.

A functional random number generator

```scala
trait RNG:
  def nextInt: (Int, RNG)

object RNG:
    case class Simple(seed: Long) extends RNG:
        def nextInt: (Int, RNG) =
            // `&` is bitwise AND. We use the current seed to generate a new seed.
            val newSeed = (seed * 0x5DEECE66DL + 0xBL) & 0xFFFFFFFFFFFFL
            // The next state, which is an `RNG` instance created from the new seed.
            val nextRNG = Simple(newSeed)
            // `>>>` is right binary shift with zero fill.
            // The value `n` is our new pseudo-random integer.
            val n = (newSeed >>> 16).toInt
            // The return value is a tuple containing both a pseudo-random
            // integer and the next `RNG` state.
            (n, nextRNG)
```

Random Number Generator

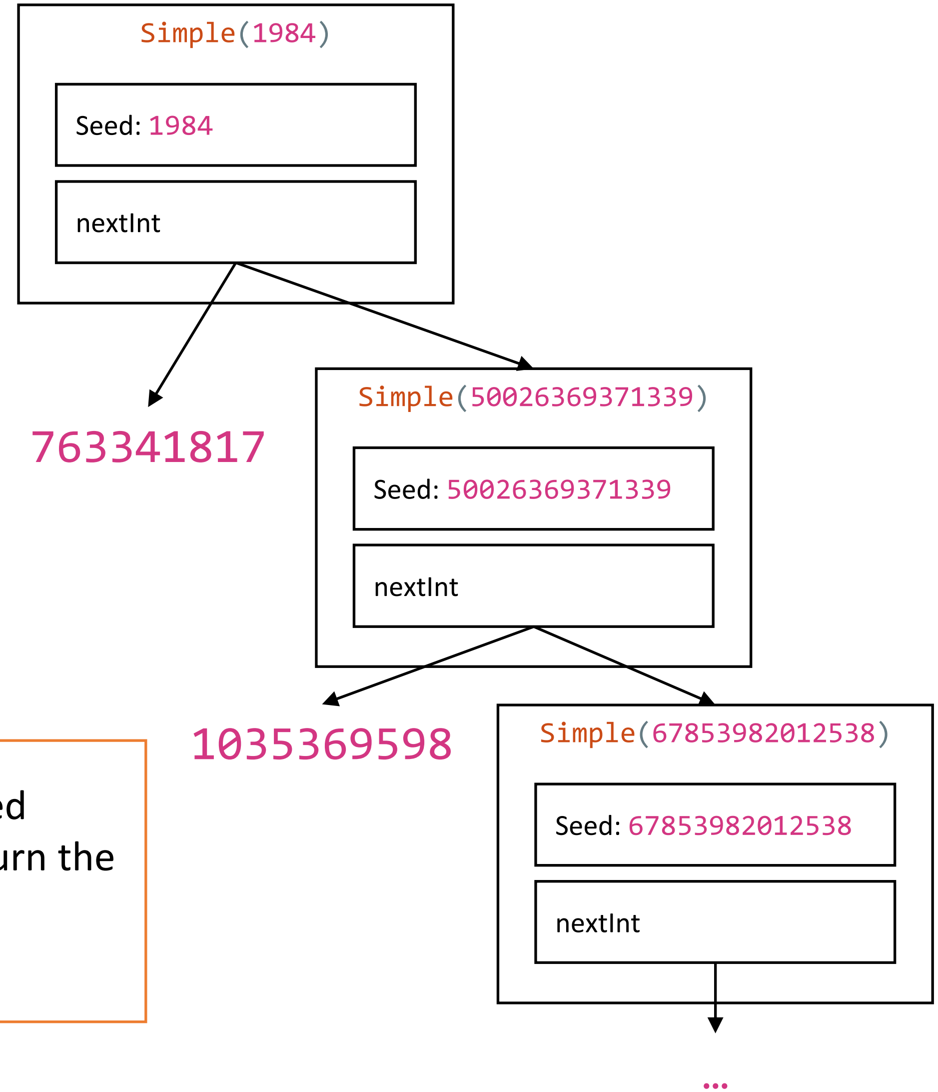
We create an interface for RNGs. Notice that it returns tuples containing an integer and an RNG.

Implémentation pas forcément pertinente

By the way: the code for generating the next integer is not important.

A functional random number generator

```
scala> val r1 = Simple(1984)
val r1: RNG.Simple = Simple(1984)

scala> val (i1, r2) = r1.nextInt
val i1: Int = 763341817
val r2: RNG = Simple(50026369371339)

scala> val (i2, r3) = r2.nextInt
val i2: Int = 1035369598
val r3: RNG = Simple(67853982012538)
```



Rather than returning only the randomly generated number and updating some internal state, we return the randomly generated number and the new state.
<u>We leave the old state unmodified</u>.

A functional random number generator

```
scala> val r1 = Simple(1984)
val r1: RNG.Simple = Simple(1984)

scala> val (i1, r2) = r1.nextInt
val i1: Int = 763341817
val r2: RNG = Simple(50026369371339)

scala> val (i2, r3) = r2.nextInt
val i2: Int = 1035369598
val r3: RNG = Simple(67853982012538)

scala> r1.nextInt
val res1: (Int, RNG) = (763341817,Simple(50026369371339))
```

By the way…

- We can always make impure APIs pure with this approach.

- By doing so, we make our solutions seemingly less efficient. Why? Rather than modifying data in place (which requires one to allocate memory once), we now need more memory. <u>But…</u>

  - Depending on the type of problem, some functional data structures may mitigate this problem.
  - There are cases where data can be mutated in place without loss of referential transparency.

- But let's take it one baby step at a time. These topics will be covered later in this course.

https://www.pexels.com/nl-nl/foto/man-liefde-spelen-gelukkig-4933789/

Pure APIs

Let's create some functions to generate random positive integers, pairs of random integers, list of random integers, etc.

```
def randomPositiveInt(r: RNG): (Int, RNG) =
    val (int1, r1) = r.nextInt
    (if int1 < 0 then -(int1 + 1) else int1, r1)
```

Int.MinValue (-2147483648) does not have a positive counterpart.

```
def randomPair(r: RNG): ((Int, Int), RNG) =
    val (int1, r1) = r.nextInt
    val (int2, r2) = r1.nextInt
    ((int1, int2), r2)

def randomList(n: Int)(r: RNG): (List[Int], RNG) =
    if(n == 0) (List(), r)
    else
        val (int1, r1) = r.nextInt
        val (rest, r2) = randomList(n - 1)(r1)
        (int1 :: rest, r2)
```

Pure APIs for state transitions

- We can see a pattern emerging:
  `RNG => (A, RNG)`
- These are called state transitions because they transform RNG states from one to the next.
- These patterns indicate we have some repetitive code. Is there a way to avoid these repetitions? Yes, with combinators.
- Combinators are higher-order functions that take care of state transitions.

```scala
def randomPositiveInt(r: RNG): (Int, RNG) =
    val (int1, r1) = r.nextInt
    (if int1 < 0 then -(int1 + 1) else int1, r1)
```

```scala
def randomPair(r: RNG): ((Int, Int), RNG) =
    val (int1, r1) = r.nextInt
    val (int2, r2) = r1.nextInt
    ((int1, int2), r2)
```

```scala
def randomList(n: Int)(r: RNG): (List[Int], RNG) =
    if(n == 0) (List(), r)
    else
        val (int1, r1) = r.nextInt
        val (rest, r2) = randomList(n - 1)(r1)
        (int1 :: rest, r2)
```

Pure APIs for state transitions

- We observed a pattern: `RNG => (A, RNG)`. Let's make an abstraction using a type alias. A type alias allows us to create a "synonym" for types. This is handy for giving simple names to complex types.

```
type Rand[+A] = RNG => (A, RNG)
```

- The type `Rand[+A]` represents a function that:
  - depends on some RNG as input,
  - uses that RNG to generate of type A,
  - transitions the RNG into a new state, and
  - returns the generated value and new state.

Pure APIs for state transitions

```scala
type Rand[+A] = RNG => (A, RNG)
```

- We can now represent methods of RNG as values of that new type.

```scala
val int: Rand[Int] = _.nextInt
```

- Here's an example of how int functions.

```scala
scala> val x = Simple(1)
val x: RNG.Simple = Simple(1)
scala> int(x)
val res4: (Int, RNG) = (384748,Simple(25214903928))
```

Pure APIs for state transitions

- Here are some state transitions that we can define: `unit` and `map`.
  - The `unit` state transition passes the state without using it and always returns a constant value.

```
def unit[A](a: A): Rand[A] = state => (a, state)
```

  - The `map` state transition transforms the output of a state transition without modifying the state.

```
def map[A,B](s: Rand[A])(f: A => B): Rand[B] =
    state1 => {
        val (a, state2) = s(state1)
        (f(a), state2)
    }
```

Pure APIs for state transitions

The whole point of the Rand[+A] type alias is to write combinators combining Rand values <u>without relying on explicitly passing the RNG state</u>. For example:

```
def randomPositiveInt(r: RNG): (Int, RNG) =
    val (int1, r1) = r.nextInt
    (if int1 < 0 then -(int1 + 1) else int1, r1)
```

We're explicitly passing RNG states.

```
def randomPositiveEvenInt: Rand[Int] =
    map(randomPositiveInt)(i => i - (i % 2))
```

We're not explicitly passing RNG states. But remember Rand[+A] is an alias for RNG => (A, RNG).

```
----------
```

- scala> val x = Simple(2)
- val x: RNG.Simple = Simple(2)

- scala> randomPositiveInt(x)
- val res0: (Int, RNG) = (769497,Simple(50429807845))

- scala> randomPositiveEvenInt(x)
- val res1: (Int, RNG) = (769496,Simple(50429807845))

Pure APIs for state transitions

- We have seen that `map` allows us to write functions without explicitly relying on passing state. But does that mean that we can write all functions using this implementation of `map`?
- The answer is no, as there are functions that require us to manage states being passed.
- One such function is `randomPositiveIntLessThan` for generating random numbers between 0 and n.

Pure APIs for state transitions

- Since the largest integer (`Int.MaxValue`) value is not necessarily devisable by n, some values will have a "higher" probability to appear. E.g., for `randomPositiveIntLessThan(20)`, we have:

Random pos int:	2147483637	2147483638	2147483639	2147483640	2147483641	2147483642	2147483643	2147483644	2147483645	2147483646
Modulo 20	17	18	19	0	1	2	3	4	5	6

```
def randomPositiveIntLessThan(n: Int): Rand[Int] =
    state1 => {
        val(int1, state2) = randomPositiveInt(state1)
        val mod = int1 % n
        if (int1 + (n - 1) - mod >= 0) (mod, state2)
        else randomPositiveIntLessThan(n)(state2)
    }
```

- If the randomly generated integer is larger than the largest multiple of n that fits in an integer, then `int1 + (n - 1) - mod` will result in a negative integer. In that case, try again.

Pure APIs for state transitions

```
def randomPositiveIntLessThan(n: Int): Rand[Int] =
    state1 => {
        val(int1, state2) = randomPositiveInt(state1)
        val mod = int1 % n
        if (int1 + (n - 1) - mod >= 0) (mod, state2)
        else randomPositiveIntLessThan(n)(state2)
    }
```

- Notice how we manually pass the state.
- It would be better to have a combinator that passes the state for us.
- This is where we introduce such a combinator, `flatMap`.

Pure APIs for state transitions

The function flatMap generates a random A with Rand[A], and then takes that A to choose a Rand[B] based on its value. flatMap allows us to <u>pass states along state transitions</u>. In other words, flatMap allows us to nest state transitions.

```
def flatMap[A, B](s: Rand[A])(f: A => Rand[B]): Rand[B] =
    state1 => {
        val (a, state2) = s(state1)
        f(a)(state2)
    }
```

We can now rewrite our function:

```
def randomPositiveIntLessThan(n: Int): Rand[Int] =
    flatMap(randomPositiveInt){
        i =>    val mod = i % n
                if (i + (n - 1) - mod >= 0) unit(mod)
                else randomPositiveIntLessThan(n)
    }
```

Pure APIs for state transitions

```
def flatMap[A, B](s: Rand[A])(f: A => Rand[B]): Rand[B] =
    state1 => {
        val (a, state2) = s(state1)
        f(a)(state2)
    }

def randomPositiveIntLessThan(n: Int): Rand[Int] =
    flatMap(randomPositiveInt){
        i =>    val mod = i % n
                if (i + (n - 1) - mod >= 0) unit(mod)
                else randomPositiveIntLessThan(n)
    }
----------
scala> randomPositiveIntLessThan(20)
val res2: Rand[Int] = RNG$$$Lambda$1763/0x000000080117a790@16f2d883

scala> randomPositiveIntLessThan(20)(Simple(1))
val res1: (Int, RNG) = (8,Simple(25214903928))
```

This returns a state transition.

Given a state, the state transition returns a new state.

Pure APIs for state transitions

We now implement our d20 with `randomPositiveIntLessThan`.

```
def rolld20: Rand[Int] = randomPositiveIntLessThan(20)
```

Even though it still contains a mistake, we can soon find states that return zeroes and recreate those exceptions in a reliable manner.

```
scala> List.range(0,5).map((x: Int) => rolld20(Simple(x)))
val res6: List[(Int, RNG)] = List((0,Simple(11)),
(8,Simple(25214903928)), (17,Simple(50429807845)),
(6,Simple(75644711762)), (15,Simple(100859615679)))
```

How do we fix this? We can use `map`.

```
def rolld20: Rand[Int] = map(randomPositiveIntLessThan(20))(_ + 1)
```

```
scala> List.range(0,5).map((x: Int) => rolld20(Simple(x)))
val res0: List[(Int, RNG)] = List((1,Simple(11)),
(9,Simple(25214903928)), (18,Simple(50429807845)),
(7,Simple(75644711762)), (16,Simple(100859615679)))
```

Part 02:
Further generalizing purely functional state

Generalizing our approach with a state transition data type

- You may have noticed that, unlike the book, I try to write functions using variables that start with "`state`" instead of "`rng`".

- But currently, all our functions are written in terms of Rand. State transitions that rely on an RNG, or random number generator.

- Are functions such as `unit`, `map`, `flatMap`, etc. specific to random number generators? No! They are function that can work with state transitions, not only RNG state transitions.

- Can we change our code to generalize these aspects? The answer is yes.

Generalizing our approach with a state transition data type

- Can we change our code to generalize these aspects? The answer is yes.

```
// def map[A, B](s: Rand[A])(f: A => B): Rand[B] =
//     state1 =>
//         val (a, state2) = s(state1)
//         (f(a), state2)

def map[S, A, B](s: S => (A, S))(f: A => B): S => (B, S) =
    state1 =>
        val (a, state2) = s(state1)
        (f(a), state2)

// def flatMap[A, B](s: Rand[A])(f: A => Rand[B]): Rand[B] =
//     state1 =>
//         val (a, state2) = s(state1)
//         f(a)(state2)

def flatMap[S, A, B](s: S => (A, S))(f: A => S => (B, S)): S => (B, S) =
    state1 =>
        val (a, state2) = s(state1)
        f(a)(state2)
```

Remember: `type Rand[+A] = RNG => (A, RNG)`

But look how easily we can change the signature to make it more general <u>without changing the code</u>!

All these functions are declared as part of the RNG companion object. Let's move them.

Generalizing our approach with a state transition data type

The functions `map`, `flatMap`, `unit`, … are still declared in the RNG companion object. Let's now create a class for state transitions ST. If possible, we would like to call these functions on objects of ST.

Generalizing our approach with a state transition data type

Let's create a new type to <u>represent state transitions</u>:

```
//type Rand[+A] = RNG => (A, RNG)
type ST[S, +A] = S => (A, S)
type Rand[+A] = ST[RNG, A]
```

And use these new types to simplify our functions' signatures:

```
// def map[S, A, B](s: S => (A, S))(f: A => B): S => (B, S) = …
def map[S, A, B](s: ST[S, A])(f: A => B): ST[S, B] =
    state1 =>
        val (a, state2) = s(state1)
        (f(a), state2)

// def flatMap[S, A, B](s: S => (A, S))(f: A => S => (B, S)): S => (B, S) = …
def flatMap[S, A, B](s: ST[S, A])(f: A => ST[S, B]): ST[S, B] =
    state1 =>
        val (a, state2) = s(state1)
        f(a)(state2)
```

These functions are still declared in the RNG companion object. Let's now create one for state transitions ST.

Code so far…

```
type ST[S, +A] = S => (A, S)
type Rand[+A] = ST[RNG, A]

object ST:
    extension [S, A](run: ST[S, A])
      def map[B](f: A => B): ST[S, B] =
          state1 => {
              val (a, state2) = run(state1)
              (f(a), state2)
          }

      def flatMap[B](f: A => ST[S, B]): ST[S, B] =
          state1 => {
              val (a, state2) = run(state1)
              f(a)(state2)
          }

      def map2[B, C](sb: ST[S, B])(f: (A, B) => C): ST[S, C] =
          state1 => {
              val (a, state2) = run(state1)
              val (b, state3) = sb(state2)
              (f(a, b), state3)
          }

    def unit[S, A](a: A): ST[S, A] = state => (a, state)
```

We created a singleton, that is a class with only one instance that both are referred to with the same name.

As the class and object have already been declared, we can only extend it. The `extension` keyword allows us to extend the object and class with additional fields such as functions.

For this course, this is a technical detail. People familiar with OOP might more easily understand the "problem."

The function map2

We need a couple of utility functions that are covered in more detail in the book. In the book, the reader is expected to implement some functions in RNG and then refactor them into other objects. We have already done this for map and flatMap.

The function map2, which belongs to an ST, takes a state transition and a function f. The state transition of the ST and the one that is passed are used to compute a A and a B. Those two are then applied to f and returned with the newest state.

```
def map2[B, C](sb: ST[S, B])(f: (A, B) => C): ST[S, C] =
    state1 =>   val (a, state2) = run(state1)
                val (b, state3) = sb(state2)
                (f(a, b), state3)
```

We provide a state transition object another state transition object and a function. The two RNG objects generate two random integers. They are used to create a list and the state of the last randomly generated integer is returned as the new state.

```
----------

scala> int.map2(int)(_ :: _ :: Nil)(x)

val res0: (List[Int], RNG) = (List(384748, -1151252339),Simple(206026503483683))
```

The function traverse I

traverse allows us to traverse a list of values and transform these into a list of corresponding state transitions and then sequence them together. As we go back to front, we end up with a list of state transitions with the first state transition corresponding with the first element of our list.

```
def traverse[S, A, B](l: List[A])(f: A => ST[S, B]): ST[S, List[B]] =
  l.foldRight(unit[S, List[B]](Nil))((a, acc) => f(a).map2(acc)(_ :: _))
```

Passes the state and returns an empty list of type B.

From right to left, take the next value and the previous result (or initial value) in the accumulator.

The function f takes the value and returns a state transition. We then call map2 and apply it acc, which is also a state transition, to cons the state transitions into a list.

Remember: the foldRight function of foldRight takes an ***associative*** binary operator function as input and will use it to collapse elements from the collection from right to left. The foldRight function allows you to also specify an initial value.

Simulating finite-state machines

Our state combinators we currently have defined are very powerful. It turns out that we can define three simple combinators that simulate *getting* a state, *setting* a state and *modifying* a state in a functional manner.

The `get` state transition allows us to pass the state and returns that same state as a value.

```
def get[S]: ST[S, S] = s => (s, s)
```

The `set` state transition "ignores" the incoming state and returns the new state and `Unit` as a value.

```
def set[S](s: S): ST[S, Unit] = _ => ((), s)
```

The `modify` state transition relies on `get` to get the current state, which is transformed with `f`, and then uses `set` to return a new state with an empty value.

```
def modify[S](f: S => S): ST[S, Unit] =
    get.flatMap(s => set(f(s)))
```

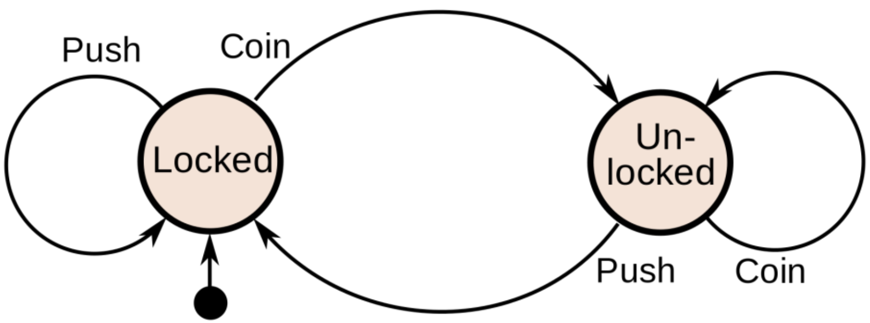
Using traverse

```
scala> val x = Simple(1)
val x: RNG.Simple = Simple(1)

scala> val list = List(1,2,3)
val list: List[Int] = List(1, 2, 3)

scala> val res = traverse(list)(randomList(_))
val res: ST[RNG, List[List[Int]]] =
ST$$$Lambda$1937/0x00000008011e0d38@29b2a94c

scala> res(x)
val res5: (List[List[Int]], RNG) = (
    List(
        List(384748),
        List(-1151252339, -549383847),
        List(1612966641, -883454042, 1563994289)),
    Simple(102497929776471))
```

Example of a finite-state machine

A coin-operated turnstile (src: Wikipedia)

There are two states: locked and unlocked. As a source of income, we also track the number of coins in a turnstile.



Source picture: https://www.pexels.com/photo/adult-black-nurse-passing-turnstile-in-metro-station-6097963/

Example of a finite-state machine

Diagram based on Wikipedia:

Current State	Input	Next State	Output
Locked	coin	Unlocked	Unlocks the turnstile so that the customer can push through.
	push	Locked	None • *I.e., you cannot pass.*
Unlocked	coin	Unlocked	None • We assume that the turnstile does not lock after a while. • We also assume that the turnstile, once open, will return any additional coins to the user.
	push	Locked	When the customer has pushed through, locks the turnstile.

Implementing the coin-based turnstile

```
sealed trait Input
case object Coin extends Input
case object Push extends Input

case class Turnstile(locked: Boolean, coins: Int)

object Turnstile:
  def simulate(inputs: List[Input]): ST[Turnstile, Int] =
    val states = ST.traverse(inputs)(i => ST.modify(nextState(i)))
    states.flatMap(_ => get.map(s => s.coins))

  def nextState(i: Input)(s: Turnstile) =
    (i, s) match
      case(Coin, Turnstile(true, coins)) => Turnstile(false, coins + 1)
      case(Push, Turnstile(true, _)) => s
      case(Coin, Turnstile(false, _)) => s
      case(Push, Turnstile(false, coins)) => Turnstile(true, coins)
```

Implementing the coin-based turnstile

```scala
sealed trait Input
case object Coin extends Input
case object Push extends Input

case class Turnstile(locked: Boolean, coins: Int)

object Turnstile:
  def simulate(inputs: List[Input]): ST[Turnstile, Int] =
    val states = ST.traverse(inputs)(i => ST.modify(nextState(i)))
    states.flatMap(_ => get.map(s => s.coins))

  def nextState(i: Input)(s: Turnstile) =
    (i, s) match
      case(Coin, Turnstile(true, coins)) =>
        { println("Thank you for paying!") ; Turnstile(false, coins + 1) }
      case(Push, Turnstile(true, _)) => { println("You need to pay!") ; s }
      case(Coin, Turnstile(false, _)) => { println("Ehm. You already paid...") ; s }
      case(Push, Turnstile(false, coins)) =>
        { println("Au revoir!") ; Turnstile(true, coins) }
```

Adding some `println` for demonstration purposes. Remember, `println` constitute a side effect that we (arguably) want to avoid in a program.

Using the coin-based turnstile

```
scala> val turnstile = Turnstile(true, 0)
val turnstile: Turnstile = Turnstile(true,0)

scala> val inputs = List(Push, Push, Push, Coin, Push, Coin, Coin, Push, Coin, Push)
val inputs: List[Input] = List(Push, Push, Push, Coin, Push, Coin, Coin, Push, Coin, Push)

scala> val simulation = Turnstile.simulate(inputs)
val simulation: ST[Turnstile, Int] = ST$$$Lambda$1940/0x00000008011deca8@6d7005e2

scala> val state = simulation(turnstile)
You need to pay!
You need to pay!
You need to pay!
Thank you for paying!
Au revoir!
Thank you for paying!
Ehm. You already paid...
Au revoir!
Thank you for paying!
Au revoir!
val state: (Int, Turnstile) = (3,Turnstile(true,3))
```

Summary

- We have seen how we can represent state in purely functional programs
- The trick is to use functions that accept states and return values with the next state. In other words, functions that represent state transitions.
- The approach we have covered in this part of the course may allow you to rewrite functions that require state into a pure manner.

Lexicon

- Finate-state machine – automate fini
- State – état