FP in Scala

Chapter 6:
Purely functional state
(recap)

Topics

- Random numbers
- Random numbers API in Scala
- Referential transparency
- Separation of concerns
- Meaning of Pure API
- Awkwardness in FP
- State actions or state transitions
- Function composition
- Type of State
- FP v/s imperative programming

Random numbers

- random numbers generations can be used **as an example** on how to write purely functional programs that manipulate state
- it is a **simple domain** to demonstrate this concept

Random numbers API in Scala

- `scala.util.Random` is a Scala class to generate random numbers
- the API has been implemented in the **imperative style**
- the API relies on **side effects**
- the implementation uses the *linear congruential generator** algorithm

an algorithm that yields a sequence of pseudo-randomized numbers calculated with a discontinuous piecewise linear equation. The method represents one of the oldest and best-known pseudorandom number generator algorithms. The theory behind them is relatively easy to understand, and they are easily implemented and fast, especially on computer hardware which can provide* ***modulo arithmetic by storage-bit truncation**.*

Random numbers API in Scala

A usage of the `scala.util.Random` class:

```
val rng = new scala.util.Random
```

- the object `rng` has **internal state**
- on invocation it **updates/mutates** internal state
- has **side effects**
- **not** referentially transparent*
- **not** testable, composable, modular or easily parallelized

*****_makes it easier to reason about the behavior of programs. An expression always evaluates to the same result in any context_

Random numbers API in Scala

Results of the usage:

```
rng.nextInt(6)
```

- has off-by-one **error**
  - returns a value between 0 and 5 instead of 1 and 6
- test method will satisfy the specification **only** 5 out of 6 times
- but test failure **isn't reproducible**
- bug is **obvious** and easy to reproduce

Note: above is an easier problem, but with a **complex problem** the bug would be **harder** to reproduce in a reliable way

Random numbers API in Scala

How to fix such an issue?

Hint: pass the random number generator **into the function** that caused the test to fail

- the generator must be created with the **same seed**
- the generator must be created in the **same state**

Referential transparency

- means it is easier to **reason** about the behavior of programs
- an expression **always evaluates** to the same result in any context

additional vocabulary:

referentially transparent

Referential transparency

Key to recovering *referential transparency**

- make state **updates explicit**
  - **return** the state
  - **return** the value we are generating
  - **do not** modify old state

**makes it easier to reason about the behavior of programs. An expression always evaluates to the same result in any context.*

An example of the implementation via an interface (`trait`):

```
trait RNG {
    def nextInt: (Int, RNG)
}
```

Separation of concerns

The old way:

- do not make state **updates explicit**
    - **not return** state
    - **return only the value** we are generating
    - **mutate** internal state

Referential transparency

The new way:

- make state **updates explicit**
  - **return** the state
  - **return** the value we are generating
  - **do not** modify old state

Separation of concerns

Advantage of the new way - *separation of concerns:*

- **computing** what the next state is
- **communicating** the new state to the rest of the program

Note: the state returned is still **encapsulated**, **without any leakage** of the underlying random number generation implementation

Meaning of Pure API

- always return **the same value** for the given input
  - (for the **same seed** in the case of the random number generator)
- also always return **the same value** using the *function** returned
  - (for the **same seed** in the case of the random number generator)

**generator function in the case of the random number generator*

Awkwardness in FP

- ***awkwardness*** *in functional programming (maintaining purity) is a sign of some* ***missing abstraction*** *waiting to be discovered*
- *look for* ***common patterns*** *that you can factor out, that others might have discovered*
- *with practice, experience, and more familiarity with the FP idioms, expressing a program* ***functionally*** *will become* ***effortless*** *and* ***natural***
- *programming using* ***pure*** *functions greatly* ***simplifies*** *the design space*

State actions or state transitions

```
RNG => (A, RNG)
```

- functions of such types are called **state actions** or **state transitions**
    - transforms `RNG` states ⇒ form one **state** to the next
- state actions can be **combined** using combinators (higher-order functions)
    - combinators: combines state actions
- type alias for `RNG` state action data type:

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

State actions or state transitions

`Rand[A]`

- **randomly** generated `A`
- it is a **state action**
- **depends** on some `RNG`
- uses it to **generate** an `A`
- transitions the `RNG` to a **new state** to be used by **another** action

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

State actions or state transitions

Simple `RNG` state transition is the **unit** action:

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

*`Rand[A]` is **type alias** for a function type `RNG => (A, RNG)`*

Function composition

Implementation of `map`:

```
def map[A, B](s: Rand[A])(f: A => B): Rand[B] =
  rng => {
    val (a, rng2) = s(rng)
    (f(a), rng2)
  }
```

demonstrating composition

Function composition

Example usage of `map`:

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

demonstrating composition

Type of state

`State` - short for computation that carries some ***state*** along or ***state action***, ***state transition*** or even ***statement***

Type of state

General **signature** for the `map` function:

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

General **type** for handling any type of `State`:

```
type State[S,+A] = S => (A,S)
```

`Rand` is a **type alias** for `State`:

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

`State` written as its **own class**:

```
case class State[S,+A](run: S => (A,S))
```

FP v/s imperative

- functional programming is programming with **no side effects**
    - on the other hand imperative programs **may have side effects**
- functional programming programs are ***referentially transparent****
    - **may not be the case** with imperative programs
- they can be **reasoned** about equationally
    - **may not be the case** with imperative programs

**makes it easier to reason about the behavior of programs. An expression always evaluates to the same result in any context*

Summary

- write purely functional programs that **have state**
    - helps write programs that are *referentially transparent*
    - they are easier to test, composable, modular and easily parallelized
- the idea behind it is to
    - use a pure function that **accepts state** as its argument
    - it returns the new state **alongside** its result