Open in app ↗

5

✦ Member-only story

Functional Reactive Programming in Scala from Scratch (Part 1)

Timo Stöttner · Follow
Published in ITNEXT
8 min read · Apr 18, 2019

Listen Share More

In this series of posts we want to develop a little framework for Functional Reactive Programming in Scala from scratch. This is the first part of the series. The remaining parts can be found here: Part 2, Part 3.

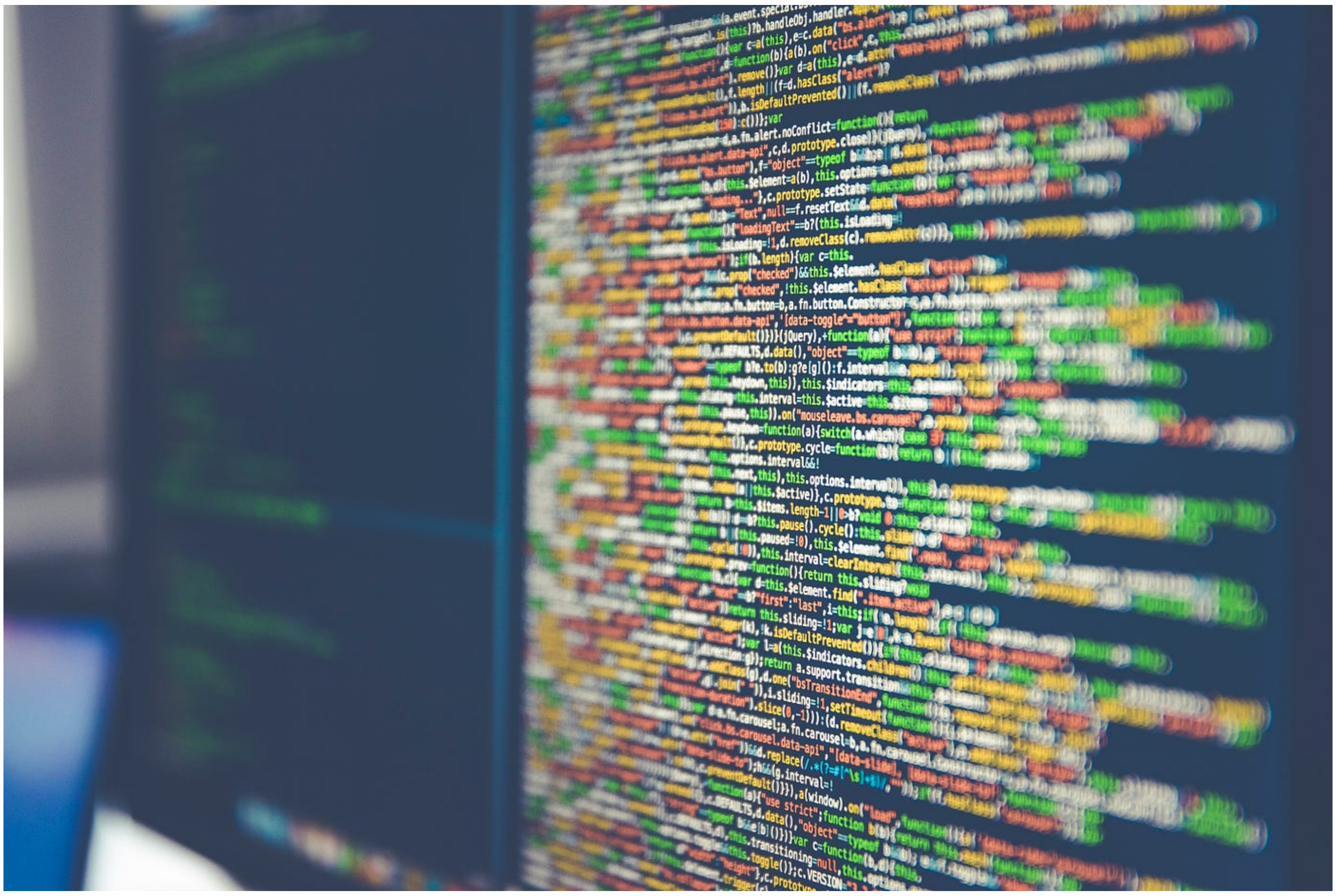


Photo by Markus Spiske on Unsplash

Since the reactive manifesto has been published, reactive programming has been experiencing quite a bit of a hype. A bunch of libraries have been developed for basically every major programming language which facilitate reactive programming. Among the most prominent of them are the ReactiveX Libraries that exist for Javascript, Python, Scala and a quite few others.

While these libraries are great, they are not to be confused with *functional* reactive programming. As is stated in the documentation of ReactiveX itself:

> *It is sometimes called "functional reactive programming" but this is a misnomer. ReactiveX may be functional, and it may be reactive, but "functional reactive programming" is a different animal.*

The main point of difference between reactive libraries such as ReactiveX and Functional Reactive Programming is that these libraries mostly just look at *events* and not at *behaviours*. Events are discrete values that are emitted over time, such as mouse clicks. Behaviours are continuous values that always have a current value, such as the mouse position.

(A mouse click itself does not have a value — it's just an event that gets fired everytime the user clicks somewhere. A mouse position, on the other hand, always has a current value — but it doesn't get "fired" at certain points in time.)

If you're confused about all of this, have a look at my article Demystifying Functional Reactive Programming discussing all of these buzz words I just dropped in much more detail.

In this series of articles we are going to develop a little framework for Functional Reactive Programming in Scala, step-by-step and entirely from scratch. The code we'll look at is a simplified version of Martin Odersky's and Ingo Maier's Scala.React that is also described in the paper Deprecating the Observer Pattern with Scala.React

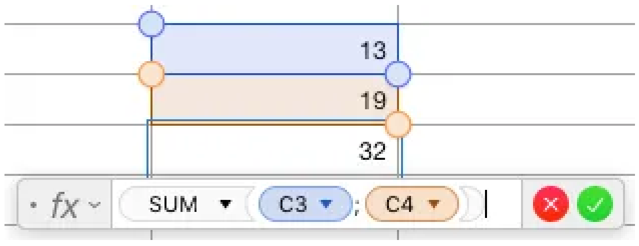
I got inspired to write this by watching a video of Martin Odersky's online course on Functional Programming, where he shortly explains parts of Scala.React. However, if you're not Martin Odersky, you might find the video pretty hard to follow. There's quite a lot going on in the code that doesn't immediately meet the eye. This series of posts will be an attempt to explain the concepts more gently. I believe the proposed programming framework results in very elegant code and even if you don't use it in your daily work, there's a lot to learn from it.

Notebooks containing all the code discussed here can be found on GitHub.

(Note: I'll assume some basic understanding of Scala. However, I'll try to explain the more advanced language features coming to play, so you should also be able to follow if you're not a Scala expert.)

Motivation

Reactive Programming is often explained with an analogy to a spreadsheet. Imagine a cell containing the sum of two other cells. If you change the value of one cell, the cell containing the sum will be immediately updated as well — it *reacts* to the other cell's update.



Spreadsheet: Sum of two cells

In the terms of classical Functional Reactive Programming, the cells' values are *behaviours*. They contain values that change over time. The sum is a *function* of time that depends on the other two cells' values.

Behaviours are sometimes also referred to as *signals*. In the following I will also call it *signal* because I find it to be a better word for it. (A signal doesn't just 'behave' the way it wants, but it reflects a value that might in turn depend on other signals.) Also, Scala.React calls it signal as well.

In this article we will look at a very similar but slightly simpler example, not to complicate things unnecessarily. We will look at an example of a BankAccount class the let's you deposit and withdraw money. A consolidator will then sum up the BankAccounts you tell it to depend on, so you know your total balance at all times. (The consolidator is the analog to the sum-cell in the spreadsheet example.)

So enough talking, let’s look at some code. What we’re after is to be able to write code that looks as follows:

```scala
class BankAccount {
  val balance = Var(0)

  def deposit(x: Int): Unit = {
    val curBalance = balance()
    balance() = curBalance + x
  }
  def withdraw(x: Int): Unit = {
    val curBalance = balance()
    balance() = curBalance - x
  }
}

def consolidated(accts: List[BankAccount]) =
  Signal(accts.map(_.balance()).sum)

val a = new BankAccount()
val b = new BankAccount()
val total = consolidated(List(a,b))
```

frp_bankaccount_motivation.scala hosted with ❤ by GitHub view raw

The `BankAccount` class simply lets you deposit and withdraw money and keeps track of the balance of your bank account. The `consolidated` function returns a `Signal` summing up the balances of the `BankAccount` s you pass to it.

Pretty elegant, isn’t it? In the end you’ll be able to simply call `total()` to get the current value of your two bank accounts whenever you feel like it. You won't have to keep track of observers and subjects as in the observer pattern. And you'll actually have a continuous value (as in FRP's 'behaviour') that depends on other values and gets updated automatically.

So how are we going to implement these mysterious `Signal` s and `Var` s? Let's get at it.

Understanding `Var`

First, to start off easy, let’s ignore the fact that we want to consolidate various bank accounts in the way shown above. Let’s only have a look at the class `BankAccount` itself. How could we implement `Var` to make the following code work?

```scala
class BankAccount {
  val balance = Var(0)

  def deposit(x: Int): Unit = {
    val curBalance = balance()
    balance() = curBalance + x
  }
  def withdraw(x: Int): Unit = {
    val curBalance = balance()
    balance() = curBalance - x
  }
}

val a = new BankAccount()
a deposit 20
print(a.balance()) // 20
```

frp_bankaccount_simple.scala hosted with ❤ by GitHub view raw

While it should be fairly straightforward to grasp what's going on in this piece of code, if you have a basic understanding of scala, there are quite a few instances of scala's syntactic sugar coming to play here. Without knowing them, you'll probably have a hard time coming up with an implementation of `Var` . So let's have a quick look at them.

a) `apply` in companion object

As you might know, if you define an `apply` method in the companion object of a class, you can create an instance of the class without the `new` keyword. A companion object is defined as an object with the exact same name as the class it is "accompanying". To make the statement `val balance = Var(0)` work, you need to do the following:

```scala
class Var(initVal:Int) {
    private var curVal = initVal
}
object Var {
    def apply(initVal:Int) = new Var(initVal)
}
```

frp_var_companionobject.scala hosted with ❤ by GitHub view raw

Here we just defined the class `Var` that takes an integer and a compantion object with an apply method that returns an instance of `Var` . Also, we added a variable `curVal` to `Var` in order to be able to change the class's value later.

b) `apply` **as a class method**

When adding a method `apply` to a class (not to its companion object), instances of the class can be called as if they were a method. The compiler converts this automatically into calls to the classes `apply` method. For example,

```
balance()
```

will be converted by the compiler into

```
balance.apply().
```

Equipped with this knowledge, it's quite straighforward to add the functionality that enables us to retrieve the current value of `balance` by calling `balance()`:

```scala
class Var(initVal:Int) {
    private var curVal = initVal
    def apply() = curVal
}
object Var {
    def apply(initVal:Int) = new Var(initVal)
}
```

frp_var_apply_class_method.scala hosted with ❤ by GitHub view raw

Now, when we define `balance` as an instance of `Var` and call `balance()`, we will retrieve the current value as stored in `curVal`.

(Actually, the `apply`-trick works in exactly the same way for the companion object as it does for the class method. When you call `Var()`, the compiler converts this to `Var.apply()`. Since you cannot call a class method without having an instance of it, this evokes the companion object's `apply` method. When calling `balance()`, this evokes the `apply` method of the class, that `balance` is an instance of.)

c) `update` **as a class method**

As we saw, the scala compiler converts `balance()` to `balance.apply()`. Similarly, the line

```
balance() = 5
```

gets converted to

`balance.update(5)` .

You might have been using this syntactic sugar unknowlingly quite a few times already. When having some array `arr` , the code `arr(1) = 5` gets converted to `arr.update(1,5)`

Knowing this, it should be quite straightforward to make the line `balance() = curBalance + x` work. The complete code to make our BankAccount class from above work is as follows:

```scala
class Var(initVal: Int) {
    private var curVal = initVal
    def apply(): Int = curVal
    def update(x: Int): Unit = curVal = x
}
object Var {
    def apply(initVal: Int) = new Var(initVal)
}
```

frp_var_update.scala hosted with ❤ by GitHub view raw

Understanding Signals

In the example so far I've glossed over the implementation of `Signal` . As you might've noticed, the function `consolidated` is supposed to return a `Signal` and we've only covered `Var` so far.

Actually, `Var` is supposed to be a subclass of `Signal` . What we're after is that we want to be able to define an immutable signal (one that always has the same value) with `Signal` and one whose value can change with `Var` .

So after defining `val s = Signal(3)` the value of `s` will remain to be 3, no matter what. However, as we've seen, we can update the `Var` balance like this, changing its value to 20:

```scala
val balance = Var(0)
balance() = 20
```

(Note that balance itself is an immutable value defined with `val` . Only the private variable of the instance of `Var` changes, while we cannot assign a new instance of

`Var` to the value `balance` itself.)

You might ask yourself: How is a `Signal` going to be of any use if it's immutable? Isn't `consolidated` supposed to update itself everytime a `BankAccount` s balance is updated?

Actually, `consolidated` 's value is a *function* of the `BankAccount` s values it depends on. The function isn't supposed to change. Only the value this function returns will change over time. If this isn't clear yet, it will hopefully become clearer later on.

So back to our definition of `Var` , how can we expand on the implementation from above to be able to update `balance` ? Let's have a look:

```scala
class Signal(initVal: Int) {
    private var curVal = initVal
    def apply(): Int = curVal
    protected def update(x: Int): Unit = curVal = x
}

class Var(initVal:Int) extends Signal(initVal: Int) {
    override def update(x: Int): Unit = super.update(x)
}


// Companion objects to enable instance creation without 'new' keyword
object Signal { def apply(initVal: Int) = new Signal(initVal) }
object Var { def apply(initVal: Int) = new Var(initVal) }
```

frp_simple_signal.scala hosted with ❤ by GitHub view raw

What we've done here is that we've basically just taken the implementation of `Var` from above and called it `Signal` . To reflect the fact that you should not be able to modify Signal from outside the class (or from subclasses) we added the `protected` keyword to the `update` method. Finally, `Var` extends `Signal` and the `update` method gets overridden without the `protected` keyword to be able to use the method from the outside world.

Looks good, right? Now we have all the building blocks we need. We've defined `Signal` s and `Var` s and the type checker is happy. So let's just give it a shot!

```scala
class BankAccount {
  val balance = Var(0)

  def deposit(x: Int): Unit = {
    val curBalance = balance()
    balance() = curBalance + x
  }
  def withdraw(x: Int): Unit = {
    val curBalance = balance()
    balance() = curBalance - x
  }
}

def consolidated(accts: List[BankAccount]) =
  Signal(accts.map(_.balance()).sum)

val a = new BankAccount()
val b = new BankAccount()
val total = consolidated(List(a,b))

a deposit 20
b deposit 30
print(s"Total balance: ${total()}") // prints "Total balance: 0"
```

frp_bankaccount_first_try.scala hosted with ❤ by GitHub view raw

Well, that didn't quite work out yet, unfortunately.

What happened is the following: The function `consolidated` retrieves the balances of each account, sums them up and returns a Signal with the result of that computation. If you look at the class definition of `Signal`, `initVal` is call-by-value. This means that its integer value gets computed on initialization and stays the same forever. The following little change in the code snippet illustrates this:

```scala
val a = new BankAccount()
a deposit 20                        // a.balance() == 20

val b = new BankAccount()
val total = consolidated(List(a,b)) // total() == 20

b deposit 30                        // b.balance() == 30
print(s"Total balance: ${total()}") // prints "Total balance: 20"
```

frp_first_try_explanation.scala hosted with ❤ by GitHub view raw

Here we added 20 to the balance of bank account `a` before defining `total`. Therefore `total()` equals 20, irrespective of any changes we might make to the accounts' balances afterwards.

In the next part of the series we'll have a look at how to get around this problem, so stay tuned! If you have any comments or suggestions so far, I'll appreciate the feedback.

Functional Programming Reactive Programming Software Development
Software Engineering Programming

Follow

Written by Timo Stöttner

161 Followers · Writer for ITNEXT

Data Scientist, Software Developer and Tech Enthusiast

More from Timo Stöttner and ITNEXT