Member-only story

Functional Reactive Programming in Scala from Scratch (Part 2)

Timo Stöttner · Follow
Published in ITNEXT
5 min read · Apr 26, 2019

Listen Share More

In this series of posts we want to develop a little framework for Functional Reactive Programming in Scala from scratch. If you haven't read the first part of the series yet, make sure to check it out here. Part 3 can be found here.

Photo by Émile Perron on Unsplash

Note: You can find the notebook with the entire code in this GitHub repository.

In the last article we started off with an implementation of a little framework for Functional Reactive Programming in Scala. Our goal was to write implementations for `Signal` and `Var` that enable us to do the following:

```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

Calling `total()` is then supposed to return the combined balance of both our `BankAccount` s at all times.

We achieved an implementation for `BankAccount` that worked in the way we expected in itself, but the consolidation didn't quite work out yet. The reason was that with our implementation of `Signal` and `Var` , `consolidated` got computed once on initialization and then stayed the same forever. To recap, this was our implementation of `Signal` and `Var` :

```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 made our implementation fail, is that we immediately evaluate `initVal` when passing it to the constructor of `Signal`. Later changes are irrelevant simply because `Signal`'s value doesn't get reevaluated.

So how do we get around this? Let's employ some tools from Functional Programming.

Making Use of Functional Programming

What we want to achieve is that `total` gets recomputed everytime one of the `BankAccount`s' balances changes. In other words, the `Signal` that's returned by `consolidated` is supposed to be a *function of the other Signals* it depends on. (If you skimmed over this part, you might want to read it again.)

Welcome to Functional Programming. So far we've only passed integer values around. Now we want to pass arbitrary functions around.

So how do we do that? First, we need to make sure that we can actually pass arbitrary expressions to our classes and we need to stop limiting ourselves to integers. We do that by replacing our Int-type declarations to generic types.

Second, we need to store our expression in a way that makes sure it doesn't get evaluated in a call-by-value manner. If we change our constructor parameter to be call-by-name, it can be reevaluated whenever something changes.

(If you're a little foggy on call-by-value and call-by-name, check out this brief and simple explanation).

Let's have a look at how this can be implemented:

```scala
class Signal[T](expr: => T) {
    private var curExpr: () => T = () => expr
    private var curVal: T = expr

    protected def update(expr: => T): Unit = {
        curExpr = () => expr
        curVal = expr
    }

    def apply() = curVal
}

class Var[T](expr: => T) extends Signal[T](expr) {
    override def update(expr: => T): Unit = super.update(expr)
}

// Companion objects to enable instance creation without 'new' keyword
object Signal { def apply[T](expr: => T) = new Signal(expr) }
object Var { def apply[T](expr: => T) = new Var(expr) }
```

frp_signal_call_by_name.scala hosted with ❤ by GitHub view raw

We made a few changes here:

1. We exchanged our integer types with generic types in all applicable places.
2. We renamed `initVal` to `expr` to illustrate that we are not just passing integer values around anymore but that we are working with arbitrary expressions. Also, we defined `expr` as call-by-name (`expr: => T`), meaning that it won't be evaluated immediately.
3. We added another variable `curExpr` that stores our expression without evaluating it and that can be updated to a new expression when needed. The syntax `var curExpr: () => T = () => expr` might need a few glances to wrap your head around. It defines a `var` of type `() => T` (an anonymous function) with the value `() => expr` . We can then call `curExpr()` to evaluate the expression.
4. Our `update` method now updates both `curExpr` and `curVal`

This gets us a step closer to our first working implementation. But as you might have guessed, it doesn't work yet. `expr` gets evaluated once we reach `private var curVal = expr` . So nothing really changes. (If you run the code from the end of the last article with these implementations of `Signal` and `Var` , you'll get the same results. I encourage you to try it out for yourself.)

As I wrote above, `total` needs to be recomputed everytime one of our `BankAccount` s balances changes. To ensure this, we need to keep track of the `Signal` s that depend on (i.e. "observe") our individual balances. If we don't know which `Signal` s depend on our balances, we don't know what we need to recompute once they change.

In the observer pattern we solve this problem by having the consolidator explicitly subscribe to every subject (i.e. `BankAccount`) it depends on. However, we want to write code that is more elegant than the observer pattern. The observer pattern requires quite a bit of boilerplate code that we want to avoid here.

So how can we solve this for our implementation of `Signal` and `Var` ? Let's have a look.

Keeping Track of Dependencies

One straightforward approach to keeping track of which `Signal` s need to be recomputed once a specific `Signal` changes, is to specifically pass the "subjects" it depends on to its constructor. When we initialize the `Signal` , we then need to tell it two things:

1. The expression it is supposed to compute
2. The other `Signal` s it is supposed to watch for changes, so it can recompute its value when they change

Let's have a look at how we could implement something like this:

```scala
class Signal[T](expr: => T, observed: List[Signal[_]] = Nil) {
  private var curExpr: () => T = () => expr
  private var curVal: T = expr

  private var observers: Set[Signal[_]] = Set()
  observed.foreach( obs => obs.observers += this )

  protected def computeValue(): Unit = {
    curVal = curExpr()
    observers.foreach(_.computeValue())
  }

  protected def update(expr: => T): Unit = {
    curExpr = () => expr
    computeValue()
  }

  def apply() = curVal

}

object Signal {
  def apply[T](expr: => T,  observed: List[Signal[_]] = Nil) = new Signal(expr, observe
}

// Leave the implementation of Var as it is for now
class Var[T](expr: => T) extends Signal[T](expr) {
  override def update(expr: => T): Unit = super.update(expr)
}
object Var { def apply[T](expr: => T) = new Var(expr) }
```

frp_signal_explicit_subjects.scala hosted with ❤ by GitHub view raw

We changed a few more things here:

1. There's now an optional constructor parameter `observed: List[Signal[_]] = Nil`. It can be used to pass a List of `Signal`s that the defined `Signal` depends on. As you can see, it defaults to `Nil`. So, if you don't pass anything, our newly defined `Signal` won't be updated when other `Signal`s change their values.

2. We added a `private var observers` that is initialized as an empty `Set`. When a new `Signal` is initialized, it iterates over `observed` and adds itself to `observers` of all its observed `Signal`s: `observed.foreach( obs => obs.observers += this )`

3. We added a method `computeValue` that updates the `Signal` 's current value by evaluating its current expression and has all its observers update their values as well.
4. `update` incorporates our new method `computeValue`

In order to make use of this implementation in our bank account example, we also need to make a little change to our function `consolidated` - we need to explicitly pass the `Signal` s our consolidator depends on:

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

frp_consolidated_explicit_subjects.scala hosted with ❤ by GitHub view raw

The good news is, this is our first working implementation of what we wanted to achieve! If you put the code together and run the following little test, you'll get the expected results:

```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: 50"
```

frp_second_try.scala hosted with ❤ by GitHub view raw

Congratulations on you're first working implementation of Functional Reactive Programming!

The bad news is, however, our code is quite repetitive and error-prone. When defining our function `consolidated` we need to pass both the function it computes (`accts.map(_.balance()).sum`) and the `Signal` s it depends on (`accts.map(_.balance)`) to the constructor of `Signal` . As you can see, this is almost the exact same code twice - at least in this simple case.

(Also, when updating our expression in a `Var` , we will currently not update the `observed` Signals. We could probably get around this by passing a new `observed` with calls to `update`, but this is deemed to get ugly pretty soon.)

Thank you for staying with me until the end of this pretty code-heavy article! If you've made it this far, it's only a little step to a much more elegant solution.

In the next article, we will have a look at how our `Signal`s can figure out for themselves what other `Signal`s they depend on. We'll come up with our final, much more elegant implementation.

Open in app ↗

5

Follow

Written by Timo Stöttner

161 Followers · Writer for ITNEXT

Data Scientist, Software Developer and Tech Enthusiast

More from Timo Stöttner and ITNEXT