✨ Member-only story

Functional Reactive Programming in Scala from Scratch (Part 3)

Timo Stöttner · Follow
Published in ITNEXT
7 min read · May 3, 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 two parts of the series yet, make sure to check them out here: Part 1, Part 2.

Photo by bert sz on Unsplash

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

In the last article we managed to write a working example of a little framework enabling us to do what we wanted: Keeping track of several bank accounts' balances with a consolidator. The consolidator made use of our little `Signal` framework handling all the complexity of updating the consolidated balances.

However, the API wasn't as elegant as we wanted yet. We needed to pass the observed signals to the consolidator explicitly, so it knew which other signals needed to be watched for changes. This was repetitive and error-prone. As a reminder, this is how our function `consolidated` looked like:

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

frp_consolidated_explicit_subjects.scala hosted with ❤ by GitHub view raw

The first argument to `Signal` constitutes the function to compute our `Signal`'s value. The second argument refers to the other `Signal`s our newly defined `Signal` depends on and therefore have to be watched for changes.

This is what we want it to look like:

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

frp_consolidated_final.scala hosted with ❤ by GitHub view raw

In words, we want to get rid of the second argument. Therefore, the new `Signal` has to figure out for itself, which other `Signal`s it depends on.

To solve this, we are going to make use of `DynamicVariable`s. I'll have to take a little bit of a detour to explain them, so bare with me. (If you know how to use `DynamicVariable`s or its Java counterpart `ThreadLocal`, feel free to skip this part.)

Using DynamicVariable to Keep Track of Dependencies

Scala's `DynamicVariable` is an implementation of the dynamic scoping pattern (thus also the name "dynamic" variable). Without going into more depth than necessary, a scope of a computer program refers to the region of that program where you can use an identifier (such as a variable name). There are two ways of scoping: *Lexical (or static) scoping* and *dynamic scoping*.

The standard is lexical scoping, which is what you most likely are used to. Here the scope depends on the location in the source code. For example, if you use a variable name in a class method, the compiler first looks for that name in the method itself and then on the class level.

Let's look at a little example to illustrate this:

```
class ScopingTest(){
    val value = "Class value"

    def print_class_value = println(s"Value: $value")

    def print_method_value = {
        val value = "Method value"
        println(s"Value: $value")
    }
}

val scope = new ScopingTest()

scope.print_class_value   // Prints "Value: Class value"
scope.print_method_value  // Prints "Value: Method value"
```

frp_static_scoping_example.scala hosted with ❤ by GitHub view raw

There are probably no big surprises here. In `print_class_value` there is no `value` defined, so the one defined on the class level is used. In `print_method_value` the `value` is defined within the method itself, which has a higher priority than the definition on the class level.

Dynamic scoping, on the other hand, depends on the *execution context*. Here, when you use a variable name in a class method, the compiler will first look for that name in the method itself and then *in the method that called that method* (and so on). There are a few programming languages implementing dynamic scoping, but the vast majority (such as scala) uses static scoping by default.

Dynamic scoping can make it pretty hard to understand where the value of a variable comes from. However, if you want to keep track of a value based on who called a method, dynamic scoping is the way to go.

If you think back to our bank account example, this is what we are after here. If we know that a `Signal` got called by another `Signal`, we can add it to a set of observers and update those observers whenever something changes. Luckily for us, `DynamicVariable`s implement such a pattern.

Let's first have a look at an easy example that illustrates how a `DynamicVariable` works before getting back to bank accounts:

```
1  import scala.util.DynamicVariable
2
3  val dynVar = new DynamicVariable[Int](1)
4
5  def dynVarTimesTwo = dynVar.value * 2
6
7  println(dynVarTimesTwo)     // Prints "2" (1*2)
8
9  dynVar.withValue(2){
10     println(dynVarTimesTwo) // Prints "4" (2*2)
11 }
12
13 println(dynVarTimesTwo)     // Prints "2" (1*2)
```

frp_dynvar_example.scala hosted with ❤️ by GitHub view raw

A few things to note here:

- When defining a DynamicVariable, we need to pass a default value. Here we set the default value to 1. This value is used unless we explicitly specify the value to be something else.
- We can access the DynamicVariable's value by calling `value`.
- DynamicVariable has a method `whithValue` that takes a new value and some arbitrary expression as parameters. If the expression accesses the current value of the `DynamicVariable`, it will receive the value that has been passed to `withValue`. *The variable's value therefore depends on the execution context* (as in dynamic scoping).

- If we access the variables value without a prior call to `withValue`, we will simply get the default value.

You can also nest calls to `withValue` like so:

```
println(dynVarTimesTwo)          // Prints "2"

dynVar.withValue(2){
    dynVar.withValue(3){
        println(dynVarTimesTwo)  // Prints "6"
    }
    println(dynVarTimesTwo)      // Prints "4"
}
```

frp_dynvar_nested_example.scala hosted with ❤️ by GitHub view raw

A few further notes on DynamicVariables:

- Those coming from Java might have noticed that `DynamicVariable`s are very similar to Java's `ThreadLocal`. When looking at the source code you can see that `DynamicVariable` actually uses Java's `InheritableThreadLocal`
- As the name of its Java counterpart suggests, `DynamicVariable`s are well suited to be used in threading. However, this is not relevant for us in this example
- Check out Wikipedia's page on scope if you want to learn more about static and dynamic scoping.

Back to our BankAccount Example

So how can we use DynamicVariables to keep track of the dependend `Signal`s in our `BankAccount` example? We are going to use them to keep track of who is calling a `Signal` by keeping track of the current caller with the `withValue` method. Whenever one `Signal` calls another `Signal`, we are going to add it to a set of observers. If another part of our program calls the `Signal`, our DynamicVariable will fall back to its default value with no effect.

Let's remind ourselves how our code looks like so far:

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

  private var observers: Set[Signal[_]] = Set()

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

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

  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_wo_explicit_dependencies.scala hosted with ❤️ by GitHub view raw

Here I just took the code from the end of the last article with a few minor changes: I got rid of the second parameter `observed` of `Signal`'s constructor and all references to it. We are not going to need it anymore once we introduce a `DynamicVariable` keeping track of the callers. Note that I kept the variable `observers`, though.

To incorporate the `DynamicVariable`, we are going to need a default value for it. Since our callers will be of type `Signal`, the default value has to be of type `Signal` as well:

```scala
object NoSignal extends Signal[Nothing](???) {
    override def computeValue() = ()
}
```

frp_nosignal.scala hosted with ❤️ by GitHub view raw

This is simply a dummy Signal of type `Nothing` with no implementation. We also override the method `computeValue` not to get stuck in infinite loops.

Next, we are going to define a dynamic variable in the companion object `Signal`, so the complete companion object looks like this:

```
object Signal {
  val caller = new DynamicVariable[Signal[_]](NoSignal)
  def apply[T](expr: => T) = new Signal(expr)
}
```

frp_signal_companion.scala hosted with ❤ by GitHub view raw

We defined `caller` as a `DynamicVariable` whose value is of type `Signal` and whose default value is `NoSignal`. (The type declaration `Signal[_]` refers to an existential type. Here this basically just means that we don't care about the type.)

Note here that we defined `caller` in the companion object and not in the class itself because we only want one instance of it to keep track of the currently calling `Signal`, not one instance per `Signal`.

We are going to use `caller` in two places. First, we will add the current caller to the set of observers everytime `apply` is called:

```
def apply() = {
    observers += caller.value
    curVal
}
```

frp_apply_w_caller.scala hosted with ❤ by GitHub view raw

Now everytime we access the `Signal`'s value, its caller gets added to the list of observers. If we call from somewhere outside the class without specifying `caller`'s value, the added observer will simpy be `NoSignal`, which has no effect. If we receive the `Signal`'s value from another `Signal`, we need to make sure that we specify the current caller. We will do that in `computeValue`:

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

frp_computevalue_w_dynvar.scala hosted with ❤️ by GitHub view raw

That's it! Putting it all together, our code looks as follows:

```scala
import scala.util.DynamicVariable

class Signal[T](expr: => T) {
  import Signal._ // Required for 'caller' defined in companion object
  private var curExpr: () => T = _
  private var curVal: T = _
  private var observers: Set[Signal[_]] = Set()

  update(expr)

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

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

  def apply() = {
    observers += caller.value
    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 {
  val caller = new DynamicVariable[Signal[_]](NoSignal)
  def apply[T](expr: => T) = new Signal(expr)
}
object Var { def apply[T](expr: => T) = new Var(expr) }

object NoSignal extends Signal[Nothing](???) { override def computeValue() = () }


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

frp_final_signal.scala hosted with ❤️ by GitHub view raw

I've just made one more little change: I didn't set values for `curVal` and `curExpr` directly and instead called `update` afterwards. This ensures two things: First, we don't have to repeat code in `computeValue` this way. Second, when initializing `NoSignal`, the constructor doesn't try to evaluate anything since it's method `computeValue` was overridden with an empty method.

Let's test it!

```
1  val a = new BankAccount()
2  val b = new BankAccount()
3  val c = consolidated(List(a,b))
4
5  a deposit 20
6  b deposit 30
7  print(c()) // Prints "50"
```

frp_final_test.scala hosted with ❤️ by GitHub view raw

Looks good! I hope you agree that this solution is much more elegant than the one we came up with in the last article. All thanks to `DynamicVariabe` s.

If you want to actually use this code, you'll need to take care of two more little things. Let's check them out.

A Few More Improvements

There are just two more little improvements I want to write about:

- Currently, observers will never get removed once they were added to the set of observers. When assigning a new expression to a `Signal`, it might stop depending on other `Signal` s it previously depended on. We should take this into account.
- It is currently possible to define `Signal` s that depend on each other. This cyclic dependency will lead to infinite loops if not handled properly. E.g. look at the following code where `b` is a function of `a` and `a` is a function of `b`:

```
1  val a = Var(0)
2  val b = Signal(a() + 1)
3  a() = b() + 1 // Will result in a java.lang.StackOverflowError
```

frp_cyclic_signal.scala hosted with ❤️ by GitHub view raw

Fortunately, both of these issues can be solved quite easily. In order to update the observers when necessary, we only need to reset the observers everytime `computeValue` is executed:

```
protected def computeValue(): Unit = {
    curVal = caller.withValue(this)(curExpr())
    val obs = observers
    observers = Set()
    obs.foreach(_.computeValue())
}
```

frp_reset_observers.scala hosted with ❤️ by GitHub view raw

Since all observers that still depend on the `Signal` will call the `Signal`'s apply method, they will be re-added to the set of observers shortly after being removed. All observers that no longer depend on the `Signal` will not.

Second, to prevent cyclic `Signal` definitions, we only need to add a little `assert` statement to the apply method:

```
def apply() = {
    observers += caller.value
    assert(!caller.value.observers.contains(this), "cyclic signal definition")
    curVal
}
```

frp_apply_w_assert.scala hosted with ❤️ by GitHub view raw

The assert statement will throw an error if the calling `Signal` observes the `Signal` whose value it retrieves.

That's it! Thank you for bearing with me during this rather lengthy explanation on how to implement a little framework for Functional Reactive Programming in Scala from scratch. I hope you learned something from it.

Where to Go From Here

- The actual implementation of Scala.React has quite a few more features that I haven't mentioned here. If you're interested to dig deeper check out the paper

Deprecating the Observer Pattern with Scala.React or the source code on GitHub.

- There are a few other frameworks for Functional Reactive Programming in Scala. I'm aware of scala.rx and reactify. (At least the first one of which is also heavily based on scala.react)
- A related, much more popular framework is RxScala. However, RxScala is more an implementation of Reactive Programming than of *Functional* Reactive Programming. (If you're confused about the differences, check out my article Demystifying Functional Reactive Programming). Depending on what you're after, though, RxScala might be a perfectly good and well-documented solution to your problem.

I'm planning to look into some of these libraries in future posts, so make sure to check back. Also, I'm planning to look into how the code from this series of posts could be translated into Python.

If you have any comments, please leave them!

Software Development | Software Engineering | Programming | Functional Programming | Reactive Programming