✨ Member-only story

Demystifying Functional Reactive Programming

Timo Stöttner · Follow
Published in ITNEXT
10 min read · Apr 6, 2019

Listen Share ••• More

Photo by Taras Shypka on Unsplash

Reactive programming has been gaining a lot of attention in the past couple of years. The reactive manifesto argues for a reactive software design and has been signed thousands of times. Reactive libraries like ReactiveX now exist for basically every major programming language.

With the advent of Big Data and Spark, functional programming is also experiencing quite of a hype. Leading to even more confusion, *functional reactive programming* gets thrown into the mix and frequently is used to describe reactive libraries such as ReactiveX.

In this article I want to get a hold of all the buzz words. So what is Functional Programming, Reactive Programming, and Functional Reactive Programming, how the heck do they relate to each other and why should you care?

(Note: I'm going to use some scala snippets here and there to explain the concepts but I'll keep them simple so it should also be easy to follow if you're not familiar with scala.)

Functional Programming

Functional programming is all about functions. (Yeah, right?) Well, actually it's all about *pure* functions. Pure functions are functions without any side effects. In other words, they don't write to a database, they don't change the value of anything outside their scope and they don't print anything.

This has an important consequence: Pure functions always must return something. A function not changing any state and not returning anything would be pretty much useless.

Imagine this:

```
1  def useless_function_times_two(x: Int): Unit = {
2      val useless_result = x * 2
3  }
```

frp_useless_func.scala hosted with ❤ by GitHub view raw

Obviously, this function doesn't do anything useful. (In case it's not obvious because you're not familiar with scala: The function doesn't return its result and doesn't change any external value.)

Referential Transparency

Also, pure functions only rely on the arguments passed to them as an argument. They don't access anything from the outside world.

This has one major advantage: *referential transparency*. When calling a pure function with the same arguments, it will *always* return the same result.

This makes it a lot easier to reason about code. If you've tested your individual functions and you're sure they work as they should, you can also be sure that the those functions interacting with each other give you the results you expect.

Imagine the following impure function:

```
1  var value = 2
2  def value_times_two(): Int = value * 2 // Impure function - bad idea
```

frp_impure_func.scala hosted with ❤️ by GitHub view raw

There is no way to predict what this function will return if you don't know a) that it depends on `value` and b) what the value of `value` is.

While this example might be a bit contrived, probably every seasoned programmer has already faced something similar and has been wondering where the heck their return value came from. Most functions are a bit more complicated than the one above and if they have external dependencies, changing these dependencies might leed to completely unforeseen results.

The following is generally a much better idea:

```
1  def times_two(x: Int): Int = x * 2
```

frp_pure_func.scala hosted with ❤️ by GitHub view raw

Functional programmers will tell you that this is the only right way to write functions and some functional programming languages will not allow you to write any functions that are impure. Whether you're a hardcore functional programmer or not, I think referential transparency is always a goal to shoot for.

Immutability

Concurrency leads to another requirement for pure functions: Variables passed as an input to functions must be immutable. Otherwise another thread could change the value of a variable after passing it to the function, which would break referential transparency.

Incidentially, since you want to be able to pass the output of a function as an input to another function, the return result of a pure function also has to be immutable.

Some functional programming languages take it as far as not to allow anything to be mutable. Want to double the values of a list? Pass it to a function and have it return a new list instead. Want to iterate over a list and need a variable to keep track of where you are in the list? Use recursion instead.

This has considerable advantages when dealing with concurrency and distributed systems that need to be able to recover from failure. If you cannot mutate anything, you don't need to worry about race conditions or deadlocks. Also, if you have a distributed system that does some heavy computations and one of your nodes fails, it is straightforward to recover if you still have the original immutable data and you know the computations to perform. (Apache Spark, a popular framework for processing large amounts of data, works exactly like this.)

There's a lot more to functional programming and many books have been written about it, but this is the basic gist of it.

Reactive Programming

Reactive programming is often explained with an analogy to a spreadsheet: Imagine a cell that calculates the input of two other cells. Once you change one of the inputs, the sum updates as well. The cell *reacts* to the changes and updates itself.

This is very similar to dataflow programming. Conceptually, the focus here lies on the flow of the data instead of on the flow of control.

One way to implement a reactive behaviour is with Futures (or Promises in other languages) that provide a callback. For instance, consider the following code:

```
1  import scala.concurrent.{Future}
2  import scala.util.{Failure, Success}
3  import scala.concurrent.ExecutionContext.Implicits.global
4
5  val expensive_computation = Future{
6      Thread.sleep(1000) // Simulate long-running computation
7      2 * 2
8  }
9
10 expensive_computation.onComplete{
11     case Success(value) => println(s"Result of expensive computation: $value")
12     case Failure(e) => e.printStackTrace
13 }
```

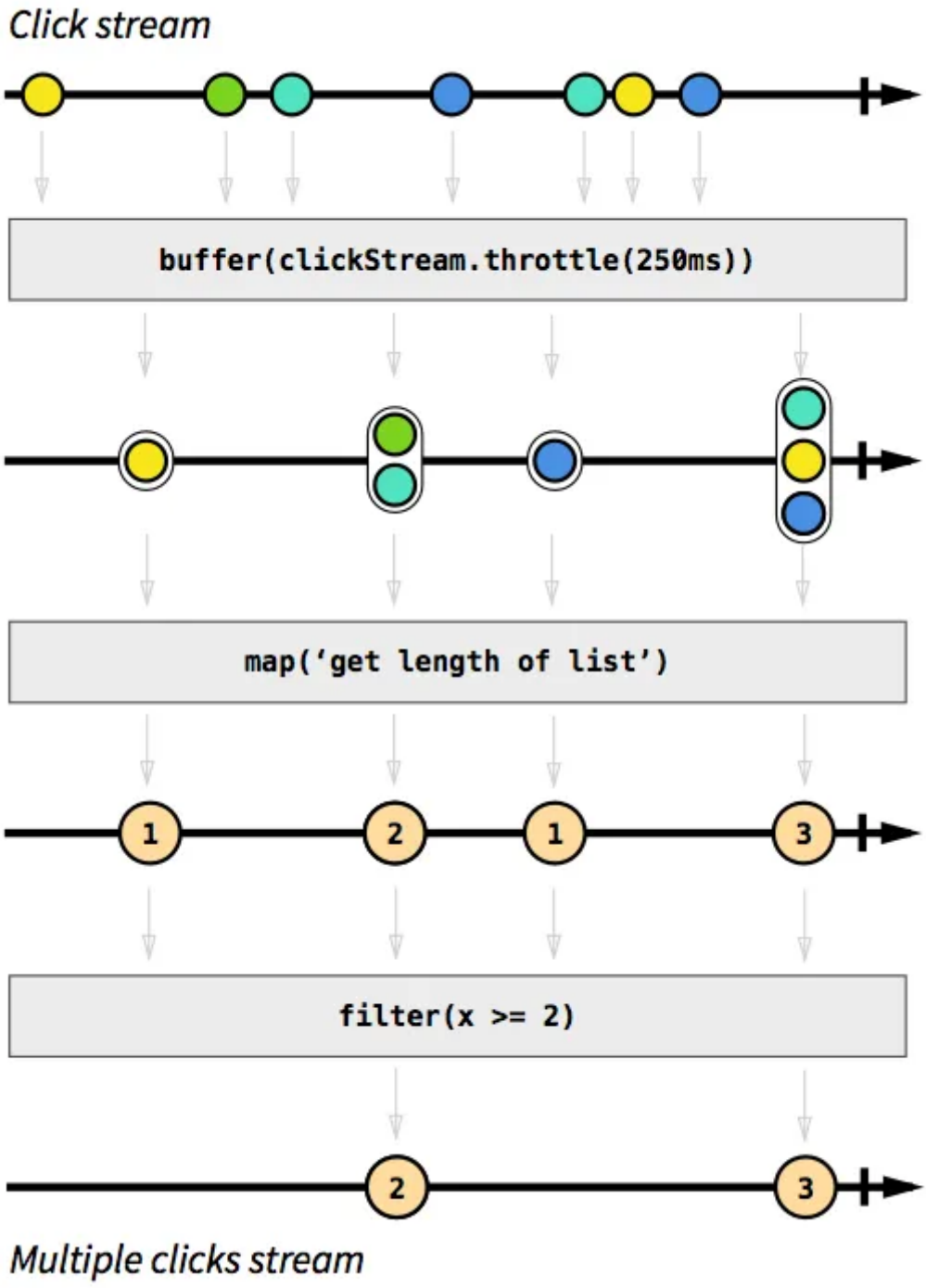
frp_future.scala hosted with ❤️ by GitHub view raw

Futures provide an abstraction that lets you easily execute long-running code concurrently and lets you handle the results as if they were available already. Once the long-running piece of code completes, the provided callback is executed — it *reacts* to the completion.

While Futures are a great way to handle simple, one off tasks that need to be executed concurrently, they can get messy if you want to do something more complicated. For instance, imagine implementing a spread-sheet application with Futures. Although you can combine Futures in scala with for-comprehensions and compose them in other ways, things will get messy pretty soon.

In contrast to Futures, most popular reactive libraries such as Rx view events as data streams. (They are also called “observables”, but we’ll stick to “streams” because that seems more intuitive to me.) They allow you to transform those event streams and glue together resulting actions. While Futures can be used once (the callback is called once the code execution completes), event streams can be used to fire an action everytime an event happens. So event streams are similar to Futures, but more powerful because they add the dimension of time.

Let’s look at an example to make things clearer. Let’s say you want to fire an event everytime a user double clicks. Visualizing this as a stream looks as follows:



(Borrowed from Andre Staltz's highly recommended introduction to Reactive Programming)

When viewing click events as a stream, you can use a simple declarative API to transform this stream into another stream of events for double clicks. Now simply subscribe to this stream with a callback that executes whatever it is that you want to execute when a user double clicks.

There is a lot more to reactive programming and its APIs that clearly goes beyond the scope of this article and that others have already explained better than I could. Check out Andre Staltz's Missing introduction to Reactive Programming to see a complete example of Rx in JavaScript or see a list of Tutorials on the ReactiveX website.

Reactive Systems

Now, how does this all connect to reactive systems as described in the reactive manifesto? Does the manifesto want you to embrace reactive programming and only write code in a reactive style? Well, not really. The manifesto talks more about *reactive systems* than about reactive programming. This sometimes gets mingled together and leads to confusion.

Reactive systems conform to certain archtectural design principles. Those design principles are meant to lead to systems that are responsive, scalable and fault-tolerant despite the growing requirements of today.

The primary means to achieve this end is *message passing*. While *reactive applications* (as in reactive programming) focus on events, *reactive systems* focus on messages. The reactive manifesto describes this distinction as follows:

> *A message is an item of data that is sent to a specific destination. An event is a signal emitted by a component upon reaching a given state. In a message-driven system addressable recipients await the arrival of messages and react to them, otherwise lying dormant. In an event-driven system notification listeners are attached to the sources of events such that they are invoked when the event is emitted. This means that an event-driven system focuses on addressable event sources while a message-driven system concentrates on addressable recipients.*

This focus on messages lets you scale easily and leads to *location transparency*. When your individual pieces of code only handle messages, it doesn't really matter if you send these messages to the same machine or to a machine on the other end of the world.

Reactive systems can be made up of *reactive applications*, but they don't have to be. It might generally be a good idea to default to a reactive programming style for reactive systems, but you can also conform to the design principles of reactive systems without it.

Functional Reactive Programming

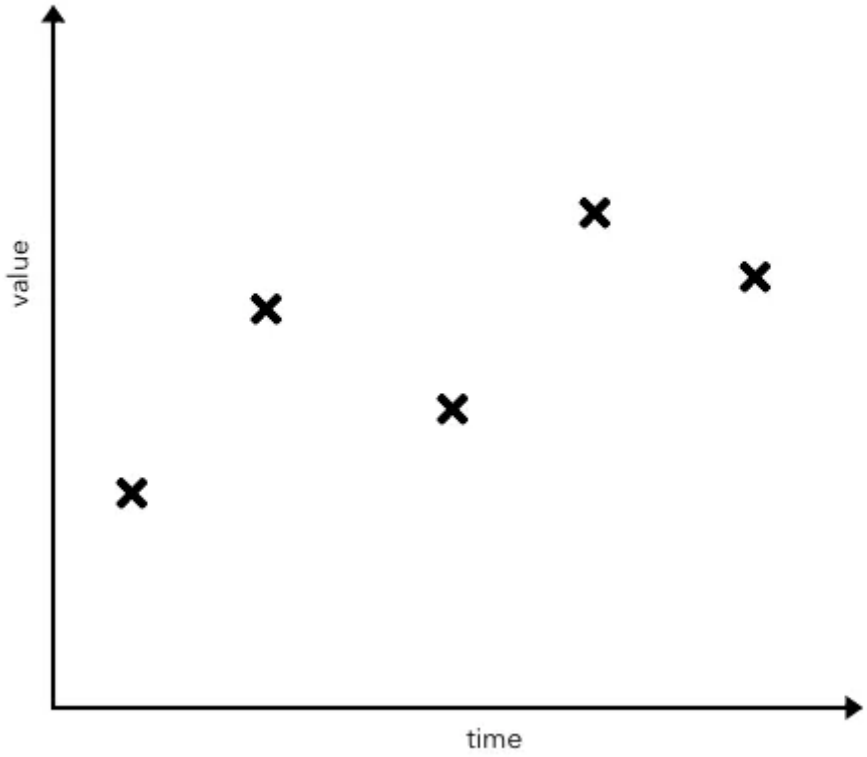
So finally, how does this relate to Functional Reactive Programming? ReactiveX and similar libraries for reactive programming get sometimes labelled "Functional Reactive Programming", but actually this is not quite right. ReactiveX is reactive and it employs many elements known from functional programming such as anonymous functions and methods such as map, filter and many others. However,

Functional Reactive Programming has been clearly defined as something else. As the documentation of ReactiveX puts it:

> *[ReactiveX] 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. One main point of difference is that functional reactive programming operates on values that change continuously over time, while ReactiveX operates on discrete values that are emitted over time.*

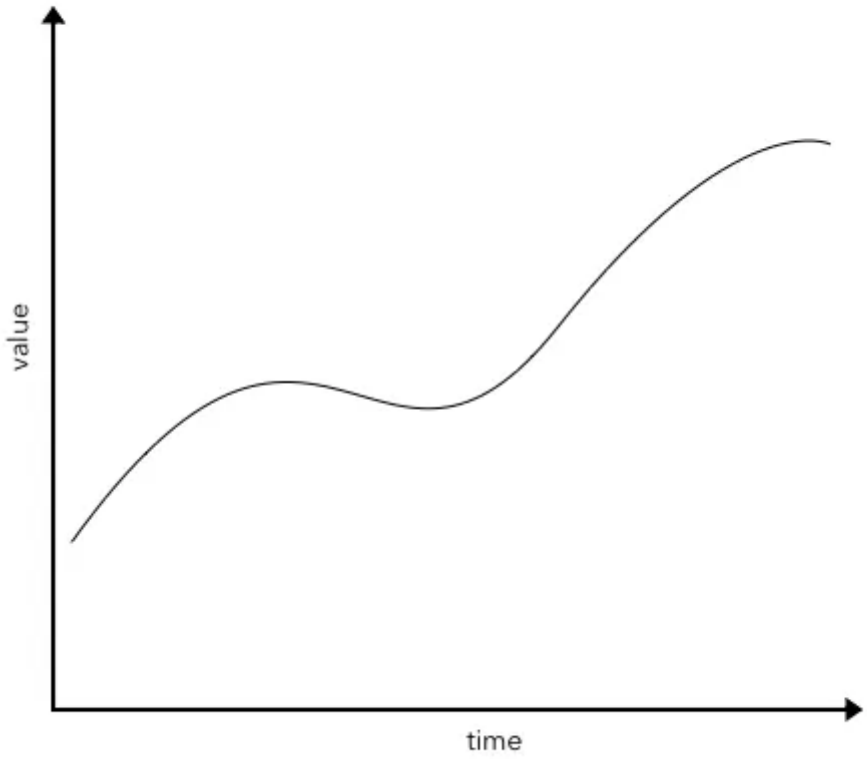
Functional Reactive Programming was defined originally more than 20 years ago by Conal Elliott and Paul Hudak in their paper Functional Reactive Animation. They made a distinction between "behaviours" and "events". Events are basically like the streams we looked at earlier. They are "discrete values that are emitted over time".

If you put them in a chart, they will look like this:



Events

Behaviours, however, are *continuous values that change over time*. This is a subtle but important difference. If you put a behaviour in a chart, it will look like this:



Behaviour

A behaviour always has a value. An event only has a last occurence (and maybe an associated value). So, for example, the position of a mouse is a behaviour because you can ask for its current position. Mouse clicks, however, are a stream — you cannot ask for the current value of a mouse click, only for its last occurence.

ReactiveX only models discrete events but no continuous values. So how come it gets away with it?

It is possible to abandon the distinction altogether. When asking for the value of an event, you can simply take the value of its last occurence. When having a continuous behaviour like a mouse movement, you can sample it and treat it like an event stream.

However, this can leed to a bunch of problems. For instance, it is possible to combine event streams by simply adding the events from the one stream to the other, but this doesn't make sense for continuous values. (Maybe you can sum up the values of two behaviours if this makes sense in your application, but that's a different thing than adding the events of one stream to the timeline of the other

stream.) If you don't make a distinction in the API, there is no way for the API to prevent you from doing something stupid.

If you're interested in a more extensive discussion on why the distinction makes sense, I recommend this article.

To get back to our original example of a worksheet with a cell summing up the values of two other cells: The values of the cells are behaviours. They always have a current value and the value of the cell calculating the sum depends on the values of the two other cells. On the other hand however, if you change the value of a cell, this change is an event.

If you now wanted to write a worksheet application, you could go about it in two ways: You could attach a callback to the change-event to update the value of the sum. However, if another value depended on that sum, you would need to attach another callback to that as well. You can see how this can become unwieldly pretty quick.

On the other hand, you could model the cell values as behaviours, where the value of the sum is a function of the two other values it depends on. This seems like a more natural approach to solve this problem that would lead to code that is more elegant.

If you want to learn more on how to implement Functional Reactive Programming in Scala, check out my articles on how to write your own little framework from scratch.

I hope this article gave you a better understanding of all the buzz words surrounding functional reactive programming. Obviously, I could only scratch the surface in this article and there is much more to functional programming, reactive programming and functional reactive programming. If you want to dig deeper, here are a few articles I can recommend:

- Functional Reactive Programming in Scala from Scratch by myself




- The introduction to reactive programming you've been missing: https://gist.github.com/staltz/868e7e9bc2a7b8c1f754
- An Introduction to Functional Reactive Programming: https://blog.danlew.net/2017/07/27/an-introduction-to-functional-reactive-programming/ (the view on functional reactive programming differs a bit from mine, but nevertheless a great article)

If you have any suggestions or feedback, let me know in the comments!

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

Follow

Written by Timo Stöttner

161 Followers · Writer for ITNEXT

Data Scientist, Software Developer and Tech Enthusiast

More from Timo Stöttner and ITNEXT