Manual

Table of Contents

Introduction

The manual serves as an introduction of the concepts in *REScala*. The full API is covered in the scaladoc especially for Signals and Events. More details can be found in [7, 3]. The manual introduces the concepts related to functional reactive programming and event-based programming from a practical perspective.

- The chapter Basic REScala covers how to get started and integrate *REScala* into a program, and
- The chapter Common combinators presents *REScalas* most common features fir composing signals and events.
- The chapter Combinators describes other combinators of *REScalas*.
- If you encounter any problems, check out the chapter Common Pitfalls.
- The readers interested in a more general presentation of these topics can find thee essential references in the section related work.

Setup

Create a `build.sbt` file in an empty folder with the following contents:

```
scalaVersion := "2.12.6"
resolvers += Resolver.bintrayRepo("stg-tud", "maven")
libraryDependencies += "de.tuda.stg" %% "rescala" % "0.24.0"
```

Install sbt and run `sbt console` inside the folder, this should allow you to follow along the following examples.

The code examples in the manual serve as a self contained Scala REPL session, all code is executed and results are annotated as comments using tut. Most code blocks can be executed on their own when adding this import, but some require definitions from the prior blocks. To use all features of *REScala* the only required import is:

```
import rescala.default._
```

Basic REScala

Because most code is imperative, you need to know the following imperative parts of *REScala* for starters, before making use of the functional features. This chapter is about using Var and Evt, the imperative subtypes of Signal and Event.

Var, set, now

A `Var[T]` holds a value of type `T`. `Var[T]` is a subtype of `Signal[T]`. See also the chapter about Signals. In contrast to declarative signals, `Var`s can be read and written to.

```
val a = Var(0)
val b = Var("Hello World")
val c = Var(List(1,2,3))
val d = Var((x: Int) => x * 2)
```

Vars enable the framework to track changes of input values. Vars can be changed directly, via set and transform, which will trigger a propagation:

```
a.set(10)
println(a.now)
// 10

a.transform( val => val + 1 )
println(a.now)
// 11

c.transform( list => 0 :: list )
println(c.now)
// List(0, 1, 2, 3)
```

Evt, fire

Imperative events are defined by the `Evt[T]` type. `Evt[T]` are a subtype of `Event[T]`. The value of the parameter `T` defines the value that is attached to the event. If you do not care about the value, you can use an `Evt[Unit]`. If you need more than one value to the same event, you can use tuples. The following code snippet shows some valid events definitions:

```
val e1 = Evt[Int]()
val e2 = Evt[Unit]()
val e3 = Evt[(Boolean, String, Int)]()
```

Events can be fired with the method `fire`, which will start a propagation.

```
e1.fire(5)
e2.fire(())
e3.fire((false, "Hallo", 5))
```

Now, observe, remove

The current value of a signal can be accessed using the `now` method. It is useful for debugging and testing, and sometimes inside onclick handlers. If possible, use observers or even better combinators instead.

```
println((a.now, s.now, t.now))
// (0, 2, false)
```

Handlers are code blocks that are executed when the event fires. The `observe` operator attaches the handler to the event. When a handler is registered to an event, the handler is executed every time the event is fired. The handler is a first class function that receives the events value as a parameter.

```
val e = Evt[String]()
val o = e.observe({ x =>
  val string = "hello " + x + "!"
  println(string)
})

e.fire("annette")
// hello annette!

e.fire("tom")
// hello tom!
```

If multiple handlers are registered, all of them are executed when the event is fired. Applications should not rely on the order of handler execution.

```
val e = Evt[Int]()
val o1 = e observe { x => println(x) }
val o2 = e observe { x => println(f"n: $x") }

e.fire(10)
// n: 10
// 10

e.fire(10)
```

```
// n: 10
// 10
```

Note that events without arguments still need an argument in the handler.

```
val e = Evt[Unit]()
e observe { x => println("ping") }
e observe { _ => println("pong") }
```

Scala allows one to refer to a method using the partially applied function syntax. This approach can be used to directly register a method as an event handler.

```
def m1(x: Int) = {
  val y = x + 1
  println(y)
}

val e = Evt[Int]
val o1 = e observe m1 _
e.fire(10)
```

Handlers can be unregistered from events with the `remove` operator. When a handler is unregistered, it is not executed when the event is fired. If you create handlers, you should also think about removing them, when they are no longer needed.

```
val e = Evt[Int]()
val handler1 = e observe println

e.fire(10)
// n: 10
// 10

handler1.remove()
```

Signal Expressions

Signals are defined by the syntax `Signal{`*sigexpr*`}`, where *sigexpr* is a side effect-free expression. A signal that carries integer values has the type `Signal[Int]`.

Inside a signal expression others signals should be accessed with the `()` operator. In the following code, the signal `c` is defined to be `a + b`. When `a` or `b` are updated, the value of `c` is updated as well.

```
val a = Var(2)
val b = Var(3)
val c = Signal { a() + b() }
println((a.now, b.now, c.now))
// (2,3,5)
```

```
a set 4
println((a.now, b.now, c.now))
// (4,3,7)

b set 5
println((a.now, b.now, c.now))
// (4,5,9)
```

The signal `c` is a dependent / derivative of the vars `a` and `b`, meaning that the values of `s` depends on both `a` and `b`.

Here are some more example of using signal expressions:

```
val a = Var(0)
val b = Var(2)
val c = Var(true)
val s = Signal{ if (c()) a() else b() }

def factorial(n: Int) = Range.inclusive(1,n).fold(1)(_ * _)

val a = Var(0)
val s: Signal[Int] = Signal {
  val tmp = a() * 2
  val k = factorial(tmp)
  k + 2
}
```

Example

Now, we have introduced enough features of *REScala* to give a simple example. The following example computes the displacement `space` of a particle that is moving at constant speed `SPEED`. The application prints all the values associated to the displacement over time.

```
val SPEED = 10
val time = Var(0)
val space = Signal{ SPEED * time() }
val o1 = space observe ((x: Int) => println(x))
// 0

while (time.now < 5) {
  Thread sleep 20
  time set time.now + 1
}
// 10
// 20
// 30
// 40
// 50

o1.remove()
```

The application behaves as follows. Every 20 milliseconds, the value of the `time` var is increased by 1 (Line 9). When the value of the `time` var changes, the signal expression at Line 3 is reevaluated and the value of `space` is updated. Finally, the current value of the `space` signal is printed every time the value of the signal changes.

Note that using `println(space.now)` would also print the value of the signal, but only at the point in time in which the print statement is executed. Instead, the approach described so far prints *all* values of the signal.

Common Combinators

Combinators express functional dependencies among values. Intuitively, the value of a combinator is computed from one or multiple input values. Whenever any inputs changes, the value of the combinator is also updated.

Latest, Changed

Conversion between signals and events are fundamental to introduce time-changing values into OO applications – which are usually event-based.

This section covers the basic conversions between signals and events. Figure 1 shows how basic conversion functions can bridge signals and events. Events (Figure 1, left) occur at discrete point in time (x axis) and have an associate value (y axis). Signals, instead, hold a value for a continuous interval of time (Figure 1, right). The `latest` conversion functions creates a signal from an event. The signal holds the value associated to an event. The value is hold until the event is fired again and a new value is available. The `changed` conversion function creates an event from a signal. The function fires a new event every time a signal changes its value.

Event-Signal

Figure 1: Basic conversion functions.

The `latest` function applies to a event and returns and a signal holding the latest value of the event `e`. The initial value of the signal is set to `init`.

```
latest[T](e: Event[T], init: T): Signal[T]
```

Example:

```
val e = Evt[Int]()
val s: Signal[Int] = e.latest(10)
assert(s.now == 10)

e.fire(1)
assert(s.now == 1)

e.fire(2)
assert(s.now == 2)

e.fire(1)
assert(s.now == 1)
```

The `changed` function applies to a signal and returns an event that is fired every time the signal changes its value.

`changed[U >: T]: Event[U]`

Example:

```
var test = 0
val v =  Var(1)
val s = Signal{ v() + 1 }
val e: Event[Int] = s.changed
val o1 = e observe ((x:Int)=>{test+=1})

v.set(2)
assert(test == 1)

v.set(3)
assert(test == 2)
```

Map

The reactive `r.map f` is obtained by applying `f` to the value carried by `r`. The map function must take the parameter as a formal parameter. The return type of the map function is the type parameter value of the resulting event. If `r` is a signal, then `r map f` is also a signal. If `r` is an event, then `r map f` is also an event.

```
val s = Var[Int](0)
val s_MAP: Signal[String] = s map ((x: Int) => x.toString)
val o1 = s_MAP observe ((x: String) => println(s"Here: $x"))

s set 5
// Here: 5
```

```
s set 15
// Here: 15

val e = Evt[Int]()
val e_MAP: Event[String] = e map ((x: Int) => x.toString)
val o1 = e_MAP observe ((x: String) => println(s"Here: $x"))

e.fire(5)
// Here: 5

e.fire(15)
// Here: 15
```

Fold

The `fold` function creates a signal by folding events with a given function. Initially the signal holds the `init` value. Every time a new event arrives, the function `f` is applied to the previous value of the signal and to the value associated to the event. The result is the new value of the signal.

```
fold[T,A](e: Event[T], init: A)(f :(A,T)=>A): Signal[A]
```

Example:

```
val e = Evt[Int]()
val f = (x:Int,y:Int)=>(x+y)
val s: Signal[Int] = e.fold(10)(f)

e.fire(1)
e.fire(2)
assert(s.now == 13)
```

Or, And

The event `e_1 || e_2` is fired upon the occurrence of one among `e_1` or `e_2`. Note that the events that appear in the event expression must have the same parameter type (`Int` in the next example). The or combinator is left-biased, so if both e_1 and e_2 fire in the same transaction, the left value is returned.

```
val e1 = Evt[Int]()
val e2 = Evt[Int]()
val e1_OR_e2 = e1 || e2
val o1 = e1_OR_e2 observe ((x: Int) => println(x))

e1.fire(1)
// 1
```

```
e2.fire(2)
// 2
```

The event `e && p` (or the alternative syntax `e filter p`) is fired if `e` occurs and the predicate `p` is satisfied. The predicate is a function that accepts the event parameter as a formal parameter and returns `Boolean`. In other words the filter operator filters the events according to their parameter and a predicate.

```
val e = Evt[Int]()
val e_AND: Event[Int] = e filter ((x: Int) => x>10)
val o1 = e_AND observe ((x: Int) => println(x))

e.fire(5)

e.fire(3)

e.fire(15)
// 15

e.fire(1)

e.fire(2)

e.fire(11)
// 11
```

{::comment} ## dropParam

The `dropParam` operator transforms an event into an event with `Unit` parameter. In the following example the `dropParam` operator transforms an `Event[Int]` into an `Event[Unit]`.

```
val e = Evt[Int]()
val e_drop: Event[Unit] = e.dropParam
val o1 = e_drop observe (_ => println("*"))

e.fire(10)
// *

e.fire(10)
// *
```

The typical use case for the `dropParam` operator is to make events with different types compatible. For example the following snippet is rejected by the compiler since it attempts to combine two events of different types with the `||` operator.

```
scala> /* WRONG - DON'T DO THIS */
     | val e1 = Evt[Int]()
```

```
e1: rescala.default.Evt[Int] = rescala.interface.RescalaInterfaceRequireSerializer#Evt:51

scala> val e2 = Evt[Unit]()
e2: rescala.default.Evt[Unit] = rescala.interface.RescalaInterfaceRequireSerializer#Evt:51

scala> val e1_OR_e2 = e1 || e2  // Compiler error
<console>:17: warning: a type was inferred to be `AnyVal`; this may indicate a programming e
       val e1_OR_e2 = e1 || e2  // Compiler error
                         ^
e1_OR_e2: rescala.reactives.Event[AnyVal,rescala.parrp.ParRP] = (or rescala.interface.Rescal
```

The following example is correct. The `dropParam` operator allows one to make the events compatible with each other.

```
val e1 = Evt[Int]()
// e1: rescala.default.Evt[Int] = rescala.interface.RescalaInterfaceRequireSerializer#Evt:5

val e2 = Evt[Unit]()
// e2: rescala.default.Evt[Unit] = rescala.interface.RescalaInterfaceRequireSerializer#Evt:

val e1_OR_e2: Event[Unit] = e1.dropParam || e2
// e1_OR_e2: rescala.default.Event[Unit] = (or e1_OR_e2:17 rescala.interface.RescalaInterfa
```

{:/comment}

Combinators

Count Signal

Returns a signal that counts the occurrences of the event. Initially, when the event has never been fired yet, the signal holds the value 0. The argument of the event is simply discarded.

`count(e: Event[_]): Signal[Int]`

```
val e = Evt[Int]()
val s: Signal[Int] = e.count

assert(s.now == 0)

e.fire(1)

e.fire(3)

assert(s.now == 2)
```

Last(n) Signal

The `last` function generalizes the `latest` function and returns a signal which holds the last `n` events.

```
last[T](e: Event[T], n: Int): Signal[List[T]]
```

Initially, an empty list is returned. Then the values are progressively filled up to the size specified by the programmer. Example:

```
val e = Evt[Int]()
val s: Signal[scala.collection.LinearSeq[Int]] = e.last(5)
val o1 = s observe println
// Queue()

e.fire(1)
// Queue(1)

e.fire(2)
// Queue(1, 2)

e.fire(3);e.fire(4);e.fire(5)
// Queue(1, 2, 3)
// Queue(1, 2, 3, 4)
// Queue(1, 2, 3, 4, 5)

e.fire(6)
// Queue(2, 3, 4, 5, 6)
```

List Signal

Collects the event values in a (growing) list. This function should be used carefully. Since the entire history of events is maintained, the function can potentially introduce a memory overflow.

```
list[T](e: Event[T]): Signal[List[T]]
```

LatestOption Signal

The `latestOption` function is a variant of the `latest` function which uses the `Option` type to distinguish the case in which the event did not fire yet. Holds the latest value of an event as `Some(val)` or `None`.

```
latestOption[T](e: Event[T]): Signal[Option[T]]
```

Example:

```
val e = Evt[Int]()
val s: Signal[Option[Int]] = e.latestOption()

assert(s.now == None)

e.fire(1)

assert(s.now == Option(1))

e.fire(2)

assert(s.now == Option(2))

e.fire(1)

assert(s.now == Option(1))
```

Fold matcher Signal

The `fold Match` construct allows to match on one of multiple events. For every firing event, the corresponding handler function is executed, to compute the new state. If multiple events fire at the same time, the handlers are executed in order. The acc parameter reflects the current state.

```
val word = Evt[String]
val count = Evt[Int]
val reset = Evt[Unit]
val result = Events.foldAll(""){ acc => Events.Match(
  reset >> (_ => ""),
  word >> identity,
  count >> (acc * _),
)}

val o1 = result.observe(r => println(r))
//

count.fire(10)

reset.fire()

word.fire("hello")
// hello

count.fire(2)
// hellohello
```

```
word.fire("world")
// world

update(count -> 2, word -> "do them all!", reset -> (()))
// do them all!do them all!
```

Iterate Signal

Returns a signal holding the value computed by `f` on the occurrence of an event. Differently from `fold`, there is no carried value, i.e. the value of the signal does not depend on the current value but only on the accumulated value.

`iterate[A](e: Event[_], init: A)(f: A=>A): Signal[A]`

Example:

```
var test: Int = 0
val e = Evt[Int]()
val f = (x:Int)=>{test=x; x+1}
val s: Signal[Int] = e.iterate(10)(f)

e.fire(1)

assert(test == 10)

assert(s.now == 11)

e.fire(2)

assert(test == 11)

assert(s.now == 12)

e.fire(1)

assert(test == 12)

assert(s.now == 13)
```

Change Event

The `change` function is similar to `changed`, but it provides both the old and the new value of the signal in a tuple.

`change[U >: T]: Event[(U, U)]`

Example:

```
val s = Var(5)
val e = s.change
val o1 = e observe println

s.set(10)
// Diff(Value(5), Value(10))

s.set(20)
// Diff(Value(10), Value(20))
```

ChangedTo Event

The `changedTo` function is similar to `changed`, but it fires an event only when the signal changes its value to a given value.

```
changedTo[V](value: V): Event[Unit]
```

```
var test = 0
val v =  Var(1)
val s = Signal{ v() + 1 }
val e: Event[Unit] = s.changedTo(3)
val o1 = e observe ((x:Unit)=>{test+=1})

assert(test == 0)

v set(2)

assert(test == 1)

v set(3)

assert(test == 1)
```

Flatten

The `flatten` function is used to “flatten” nested reactives.

It can, for instance, be used to detect if any signal within a collection of signals fired a changed event:

```
val v1 = Var(1)
val v2 = Var("Test")
val v3 = Var(true)
val collection: List[Signal[_]] = List(v1, v2, v3)
val innerChanges = Signal {collection.map(_.changed).reduce((a, b) => a || b)}
val anyChanged = innerChanges.flatten
val o1 = anyChanged observe println
// res105: rescala.reactives.Observe[rescala.parrp.ParRP] = res105:17

v1.set(10)
// 10

v2.set("Changed")
// Changed

v3.set(false)
// false
```

Common Pitfalls

In this section we collect the most common pitfalls for users that are new to reactive programming and *REScala*.

Accessing values in signal expressions

The `()` operator used on a signal or a var, inside a signal expression, returns the signal/var value *and* creates a dependency. The `now` operator returns the current value but does *not* create a dependency. For example the following signal declaration creates a dependency between `a` and `s`, and a dependency between `b` and `s`.

```
val s = Signal{ a() + b() }
```

The following code instead establishes only a dependency between `b` and `s`.

```
val s = Signal{ a.now + b() }
// <console>:17: warning: Using ‘now‘ inside a reactive expression does not create a depend
//        val s = Signal{ a.now + b() }
//                          ^
// s: rescala.default.Signal[Int] = s:17
```

In other words, in the last example, if `a` is updated, `s` is not automatically updated. With the exception of the rare cases in which this behavior is desirable, using `now` inside a signal expression is almost certainly a mistake. As a rule of dumb, signals and vars appear in signal expressions with the `()` operator.

Attempting to assign a signal

Signals are not assignable. Signal depends on other signals and vars, the dependency is expressed by the signal expression. The value of the signal is automatically updated when one of the values it depends on changes. Any attempt to set the value of a signal manually is a mistake.

Side effects in signal expressions

Signal expressions should be pure. i.e. they should not modify external variables. For example the following code is conceptually wrong because the variable `c` is imperatively assigned form inside the signal expression (Line 4).

```
var c = 0                 /* WRONG - DON'T DO IT */
// c: Int = 0

val s = Signal{
  val sum = a() + b();
  c = sum * 2
}
// s: rescala.default.Signal[Unit] = s:18

// ...
println(c)
// 4
```

A possible solution is to refactor the code above to a more functional style. For example, by removing the variable `c` and replacing it directly with the signal.

```
val c = Signal{
  val sum = a() + b();
  sum * 2
}
// c: rescala.default.Signal[Int] = c:17

// ...
println(c.now)
// 4
```

Cyclic dependencies

When a signal `s` is defined, a dependency is establishes with each of the signals or vars that appear in the signal expression of `s`. Cyclic dependencies produce a runtime error and must be avoided. For example the following code:

```
val a = Var(0)             /* WRONG - DON'T DO IT */
// a: rescala.default.Var[Int] = a:15

val s = Signal{ a() + t() }
// <console>:17: error: overloaded method value + with alternatives:
//   (x: Double)Double <and>
//   (x: Float)Float <and>
//   (x: Long)Long <and>
//   (x: Int)Int <and>
//   (x: Char)Int <and>
//   (x: Short)Int <and>
//   (x: Byte)Int <and>
//   (x: String)String
//  cannot be applied to (Boolean)
//        val s = Signal{ a() + t() }
//                            ^

val t = Signal{ a() + s() + 1 }
// <console>:17: error: overloaded method value + with alternatives:
//   (x: Double)Double <and>
//   (x: Float)Float <and>
//   (x: Long)Long <and>
//   (x: Int)Int <and>
//   (x: Char)Int <and>
//   (x: Short)Int <and>
//   (x: Byte)Int <and>
//   (x: String)String
//  cannot be applied to (Unit)
//        val t = Signal{ a() + s() + 1 }
//                            ^
```

creates a mutual dependency between `s` and `t`. Similarly, indirect cyclic dependencies must be avoided.

Objects and mutability

Vars and signals may behave unexpectedly with mutable objects. Consider the following example.

```
/* WRONG - DON'T DO THIS */
class Foo(init: Int) {
  var x = init
}
val foo = new Foo(1)
val varFoo = Var(foo)
val s = Signal{ varFoo().x + 10 }
```

```
println(s.now)
// 11

foo.x = 2

println(s.now)
// 11
```

One may expect that after increasing the value of `foo.x` in Line 9, the signal expression is evaluated again and updated to 12. The reason why the application behaves differently is that signals and vars hold *references* to objects, not the objects themselves. When the statement in Line 9 is executed, the value of the `x` field changes, but the reference hold by the `varFoo` var is the same. For this reason, no change is detected by the var, the var does not propagate the change to the signal, and the signal is not reevaluated.

A solution to this problem is to use immutable objects. Since the objects cannot be modified, the only way to change a filed is to create an entirely new object and assign it to the var. As a result, the var is reevaluated.

```
class Foo(val x: Int){}
val foo = new Foo(1)
val varFoo = Var(foo)
val s = Signal{ varFoo().x + 10 }

println(s.now)
// 11

varFoo set (new Foo(2))

println(s.now)
// 12
```

Alternatively, one can still use mutable objects but assign again the var to force the reevaluation. However this style of programming is confusing for the reader and should be avoided when possible.

```
/* WRONG - DON'T DO THIS */
class Foo(init: Int) {
  var x = init
}
val foo = new Foo(1)
val varFoo = Var(foo)
val s = Signal{ varFoo().x + 10 }

println(s.now)
// 11
```

```
foo.x = 2

varFoo set foo

println(s.now)
// 11
```

Functions of reactive values

Functions that operate on traditional values are not automatically transformed to operate on signals. For example consider the following functions:

```
def increment(x: Int): Int = x + 1
```

The following code does not compile because the compiler expects an integer, not a var as a parameter of the `increment` function. In addition, since the `increment` function returns an integer, `b` has type `Int`, and the call `b()` in the signal expression is also rejected by the compiler.

```
val a = Var(1)           /* WRONG - DON'T DO IT */
// a: rescala.default.Var[Int] = a:15

val b = increment(a)
// <console>:17: error: type mismatch;
//  found   : rescala.default.Var[Int]
//     (which expands to)  rescala.reactives.Var[Int,rescala.parrp.ParRP]
//  required: Int
//        val b = increment(a)
//                          ^

val s = Signal{ b() + 1 }
// s: rescala.default.Signal[Int] = s:16
```

The following code snippet is syntactically correct, but the signal has a constant value 2 and is not updated when the var changes.

```
val a = Var(1)
val b: Int = increment(a.now) // b is not reactive!
val s = Signal{ b + 1 } // s is a constant signal with value 2
```

The following solution is syntactically correct and the signal `s` is updated every time the var `a` is updated.

```
val a = Var(1)
val s = Signal{ increment(a()) + 1 }
```

Essential Related Work

{: #related }

A more academic presentation of *REScala* is in [7]. A complete bibliography on reactive programming is beyond the scope of this work. The interested reader can refer to [1] for an overview of reactive programming and to[8] for the issues concerning the integration of RP with object-oriented programming.

REScala builds on ideas originally developed in EScala [3] – which supports event combination and implicit events. Other reactive languages directly represent time-changing values and remove inversion of control. Among the others, we mention FrTime [2] (Scheme), FlapJax [6] (Javascript), AmbientTalk/R [4] and Scala.React [5] (Scala).

Acknowledgments

Several people contributed to this manual, among the others David Richter, Gerold Hintz and Pascal Weisenburger.

References

{: #ref} [1] E. Bainomugisha, A. Lombide Carreton, T. Van Cutsem, S. Mostinckx, and W. De Meuter. *A survey on reactive programming.* ACM Comput. Surv. 2013.

[2] G. H. Cooper and S. Krishnamurthi. *Embedding dynamic dataflow in a call-byvalue language.* In ESOP, pages 294–308, 2006.

[3] V. Gasiunas, L. Satabin, M. Mezini, A. Ńũnez, and J. Noýe. *EScala: modular event-driven object interactions in Scala.* AOSD '11, pages 227–240. ACM, 2011.

[4] A. Lombide Carreton, S. Mostinckx, T. Cutsem, and W. Meuter. *Loosely-coupled distributed reactive programming in mobile ad hoc networks.* In J. Vitek, editor, Objects, Models, Components, Patterns, volume 6141 of Lecture Notes in Computer Science, pages 41–60. Springer Berlin Heidelberg, 2010.

[5] I. Maier and M. Odersky. *Deprecating the Observer Pattern with Scala.react.* Technical report, 2012.

[6] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. *Flapjax: a programming language for ajax applications.* OOPSLA '09, pages 1–20. ACM, 2009.

[7] G. Salvaneschi, G. Hintz, and M. Mezini. *REScala: Bridging between object-oriented and functional style in reactive applications.* AOSD '14, New York, NY, USA, Accepted for publication, 2014. ACM.

[8] G. Salvaneschi and M. Mezini. *Reactive behavior in object-oriented applications: an analysis and a research roadmap.* AOSD '13, pages 37–48, New York, NY, USA, 2013. ACM.