REScala Reference Manual

Guido Salvaneschi
Technical University of Darmstadt
salvaneschi@informatik.tu-darmstadt.de

November 2013

Contents

1 Signals and Vars

A signal is language concept for expressing functional dependencies among values in a declarative way. Intuitively, a reactive value can depend on variables – sources of change without further dependencies – or on other reactive values. When any of the dependency sources changes, the expression defining the reactive value is automatically recomputed by the language runtime to keep the reactive value up-to-date.

Consider the following example:

```
1 var a = 2
2 var b = 3
3 var c = a + b
4 println(a,b,c) //-> (2,3,5)
5 a = 4
6 println(a,b,c) //-> (2,4,5)
7 c = a + b
8 println(a,b,c) //-> (4,3,7)
```

Line 3 specifies the value of `c` as a function of `a` and `b`. Since Line 3 defines a statement, the relation $c = a + b$ is valid after the execution of Line 3. Clearly, when the value of `a` is updated, the relation $c = a + b$ is not valid anymore (Line 6). To make sure that the relation still holds, the programmer needs to recompute the expression and reassign `c`, like in line 7.

Reactive programming and REScala provide abstraction to express *constraints* in addition to statements. In REScala the programmer can specify that the constraint $c := a + b$ *always* holds during the execution of a program, and every time `a` o `b` change, the value of `c` is automatically recomputed.

For example:

```
1 val a = Var(2)
2 val b = Var(3)
3 val c = Signal{ a() + b() }
4 println(a.getVal,b.getVal,c.getVal) //-> (2,3,5)
5 a()= 4
6 println(a,b,c) //-> (2,4,5)
7 println(a.getVal,b.getVal,c.getVal) //-> (4,3,7)
```

In the code above, the `Signal` in Line 3 defines the constraint $c := a + b$. When one of the variables involved in the constraint is updated (Line 6), the expression in the constraint is recomputed behind the scenes, and the value of `a` is automatically updated.

As the reader may have noticed, expressing constraints in REScala requires to conform some syntactic conventions.

Defining Vars Programmers express reactive computations starting from vars. Vars wrap normal Scala values. For example `Var(2)` create a var with an `[Int]` value and initializes it to the value 2. Vars are parametric types. A var that carries integers has the type `Var[Int]`. The following are all valid vars declarations.

```
1 val a = Var(0)
2 val b = Var("Hello World")
3 val c = Var(false)
4 val d: Var[Int] = Var(30)
5 val e: Var[String] = Var("REScala")
6 val f: Var[Boolean] = Var(false)
```

Assigning Vars Vars can be directly modified with the `()=` operator. For example `v()=3` replaces the current value of the `v` var with `3`.

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

Signal expressions When, inside a signal expression defining a signal `s`, a var or a signal is called with the `()` operator, the var of the signal are added to the values `s` depends on. In that case `s` *is a dependency* of the vars and the signals in the signal expression. All the following code snippets define valid signal declarations.

```
val a = Var(0)
val s: Signal[Int] = Signal{ a() + 1 }
```

```
val a = Var(0)
val b = Var(0)
val s = Signal{ a() + b() } // Multiple vars is a signal expression
```

```
val a = Var(0)
val b = Var(0)
val c = Var(0)
val s = Signal{ a() + b() }
val t = Signal{ s() * c() + 10 } // Mix signals and vars in signal expressions
```

```
val a = Var(0)
val b = Var(0)
val c = Var(0)
val s = Signal{ a() + b() }
val t = Signal{ s() * c() + 10 }
val u = Signal{ s() * t() } // A signal that depends on other signals
```

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

```
def factorial(n: Int) = ...
val a = Var(0)
val s: Signal[Int] = Signal{ // A signal expression can be any code block
  val tmp = a() * 2
  val k = factorial(tmp)
  k + 2  // Returns an Int
}
```

Reading reactive values The current value of a signal or a var can be accessed using the `getVal` method. For example:

```
val a = Var(0)
val b = Var(2)
val c = Var(true)
val s: Signal[Int] = Signal{ a() + b() }
val t: Signal[Boolean] = Signal{ !c() }

val x: Int = a.getVal
val y: Int = s.getVal
val x: Boolean = t.getVal
```

1.0.1 Example: speed

The following example computes the displacement `space` of a particle that is moving at constant speed `SPEED`. The application prints all the values of the displacement.

```
1  val SPEED = 10
2  val time = Var(0)
3  val space = Signal{ SPEED * time() }
4
5  space.changed += ((x: Int) => println(x))
6
7  while (true) {
8    Thread sleep 20
9    time() = time.getVal + 1
10 }
11
12 ── output ──
13 10
14 20
15 30
16 40
17  ...
```

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 executed and the value of `space` is updated. Finally, the current value of the `space` signal every time the value of the signal changes. Technically, this is achieved by converting the `space` signal to an event that is fired every time the signal changes its value (Line 5). The conversion is performed by the `changed` operator. The `+=` operator attaches an handler to the event returned by the `changed` operator. When the event fires, the handler is executed.

Line 5 is equivalent to the following code:

```
1 val e: Event[Int] = space.changed
2 val handler:  (Int => Unit) =  ((x: Int) => println(x))
3 e += handler
```

2 Events

2.1 Imperative events

2.1.1 Defining Events

REScala supports imperative events. Imperative events are defined by the `ImperativeEvent[T]` type. The value of the parameter `T` defines the value that is attached to an event.

Fir example the following code snippet defines and imperative events whose attached value is an integer:

```
1 val e = new ImperativeEvent[Int]()
```

2.1.2 Registering Handlers

Handlers can be defined attaching a code block to the event. The `+=` operator attaches the handler to the event.

```
1 val e = new ImperativeEvent[Int]()
2
3 e += { println() }
```

- Other ways to register a handler ? Alternative syntax for handlers

When a handler is registered to an event, the handler is executed every time the event is fired. If multiple handlers are registered, all of them are executed when the event is fired. Applications should not rely on any specific execution order for handler execution.

```
val e = new ImperativeEvent[Int]()

e += { println() }
```

The signature of the event handlers must conform the signature of the events.

2.1.3 Firing Events

Events can be fired with the same syntax of a method call. For example the event `e` in the following example is fired by the `e(10)` call.

```
val e = new ImperativeEvent[Int]()
e(10)
```

2.1.4 Registering Handlers

Handlers can be unregistered from events by the `-=` operator.

```
val e = new ImperativeEvent[Int]()

e += { println() }


e += { println() }
```

2.2 Declarative Events

REScala supports declarative events, which are defined as a combination of other events. For this purpose it offers operators like $e_1||e_2$ (occurrence of one among e_1 or e_2), $e_1\&\&p$ (e_1 occurs and the predicate p is satisfied), $e_1.map(f)$ (the event obtained by applying f to e_1). Event composition allows to express the application logic in a clear and declarative way. Also, the update logic is better localized because a single expression models all the sources and the transformations that define an event occurrence.

3 Conversion Functions

RESCALA provides functions that interface signals and events. Some of those functions can be called on a signal passing an event as the first parameter or can be called on an event passing a signal as the first parameter. While the behavior is the same, the signature of the function is obviously different. For simplicity, in those cases, here we only document the signature of the function called on the signal.

fold

Creates a signal by folding events with a given function.

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

```
val e = new ImperativeEvent[Int]()
val f = (x:Int,y:Int)=>(x+y)
val s: Signal[Int] = e.fold(10)(f)
e(1)
e(2)
assert(s.getValue == 13)
```

iterate

Returns a value computed by `f` on the occurrence of an event.
`iterate[A](e: Event[_], init: A)(f: A=>A) :Signal[A]`

hold

Returns a signal holding the latest value of the event `e`.
`hold[T](e: Event[T], init: T): Signal[T]`

Holds the latest value of an event as `Some(val)` or `None`.
`holdOption[T](e: Event[T]): Signal[Option[T]]`

last

Returns a signal which holds the last `n` events.
`last[T](e: Event[T], n: Int): Signal[List[T]]`

list

Collects the event values in a reactive list.
`list[T](e: Event[T]): Signal[List[T]]`

reset

On the event, sets the signal to one generated by the factory.
`reset[T,A](e: Event[T], init: T)(f: (T)=>Signal[A]): Signal[A]`

switch

Switches the value of the signal on the occurrence of `e`.
`switchTo[U](e :Event[U])(f: U=>T): Signal[T]`
`switchTo(e: Event[T]): Signal[T]`

Switches to a new signal once, on the occurrence of `e`.
`switchOnce(e: Event[_])(op: =>T): Signal[T]`
`switchOnce(e: Event[_], newSignal: Signal[T]): Signal[T]`

Switches between signals on the event `e`.
`toggle(e: =>Event[_])(op: =>T): Signal[T]`
`toggle(e: =>Event[_], other: Signal[T]): Signal[T]`

snapshot

Returns a signal updated only when `e` fires.
`snapshot(e: Event[_]): Signal[T]`

4 Technicalities

To work with REScala you need to properly import the reactive abstractions offered by the language. The following imports are normally sufficient for all REScala functionalities:

```
1 import react._
2 import react.events._
3 import macro.SignalMacro.{SignalM => Signal}
```

Note that signal expressions are currently implemented as macros, i.e. the body of a signal expression is macroexpanded. To use macros for signal expressions, the macro `SignalM` is imported and renamed to `Signal` (Line 3).

5 Common Pitfalls

In this section we collect some mistakes that are common to users that are new to reactive programming and REScala.

5.1 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 `getVal` 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`.

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

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

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

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

5.2 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.

5.3 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).

```
1 var c = 0                  /* WRONG — DON'T DO IT */
2 val s = Signal{
3   val sum = a() + b();
4   c = sum * 2
5 }
6  ...
7 foo(c)
```

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
}
 ...
foo(c.getVal)
```

5.4 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 */
val s = Signal{ a() + t() }
val t = Signal{ a() + s() + 1 }
```

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

5.5 Objects and mutability

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

```
1  class Foo(init: Int){            /* WRONG — DON'T DO IT */
2    var x = init
3  }
4  val foo = new Foo(1)
5
6  val varFoo = Var(foo)
7  val s = Signal{ varFoo().x + 10 }
8  // s.getVal == 11
9  foo.x = 2
10 // s.getVal == 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 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(x: Int){}
val foo = new Foo(1)

val varFoo = Var(foo)
val s = Signal{ varFoo().x + 10 }
// s.getVal == 11
varFoo()= newFoo(2)
// s.getVal == 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.

```
class Foo(init: Int){
  var x = init
}
val foo = new Foo(1)

val varFoo = Var(foo)
val s = Signal{ varFoo().x + 10 }
// s.getVal == 11
foo.x = 2
varFoo()=foo
// s.getVal == 11
```

5.6 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=>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 */
val b = increment(a)
val s = Signal{ b() + 1 }
```

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 = increment(a.getVal)
val s = Signal{ b + 1 }
```

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 }
```

6 Related Work

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 [7] 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).

7 Acknowledgments

Several people contributed to this manual with their comments. Among the others Gerold Hintz and Pascal Weisenburger.

References

[1] E. Bainomugisha, A. Lombide Carreton, T. Van Cutsem, S. Mostinckx, and W. De Meuter. A survey on reactive programming. *ACM Comput. Surv. (To appear)*, 2013.

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

[3] V. Gasiunas, L. Satabin, M. Mezini, A. Núñez, and J. Noyé. 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 and M. Mezini. Reactive behavior in object-oriented applications: an analysis and a research roadmap. In *Proceedings of the 12th annual international conference on Aspect-oriented software development*, AOSD '13, pages 37–48, New York, NY, USA, 2013. ACM.