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.

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*}.

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.

Reading reactive values The current value of

2 Events

3 Imperative events

3.0.1 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]()
```

3.0.2 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.

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

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

3.0.3 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.

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

3.0.4 Registering Handlers

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

```
1 val e = new ImperativeEvent[Int]()
2
3 e += { println() }
4
5
6 e += { println() }
```

4 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.

5 Conversion Functions

The RESCALA interface between signals and events exposed by the `Signal` trait.

– Creates a signal by folding events with a given function.
```
fold[T,A](e: Event[T], init: A)(f :(A,T)=>A): Signal[A]
```
– Returns a value computed by `f` on the occurrence of an event.
```
iterate[A](e: Event[_], init: A)(f: A=>A) :Signal[A]
```
– 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]]
```
– Returns a signal which holds the last `n` events.
```
last[T](e: Event[T], n: Int): Signal[List[T]]
```
– Collects the event values in a reactive list.
```
list[T](e: Event[T]): Signal[List[T]]
```
– 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]
```
– 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]
```
– Returns a signal updated only when `e` fires.
```
snapshot(e: Event[_]): Signal[T]
```

– Open interface between signals and events.
```
switch(e: Event[T])(fact: Factory[T,A]): Signal[A]
Factory[T,A].apply(eVal: T): (Signal[A], Factory[T,A])
```

6 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).

7 Common Pitfalls

In this section we collect some mistakes that are common to 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 `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.

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).

```
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.

```
1 val c = Signal{
2   val sum = a() + b();
3   sum * 2
4 }
5  ...
6 foo(c.getVal)
```

Objects and mutability Consider a signal `s` with a signal expression *sigexpr*. When one of the reactives the signal depends on is updated, the entire signal expression is evaluated again.

TODO

8 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).

9 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.