REScala Users Manual

Guido Salvaneschi
with
Gerold Hintz, Pascal Weisenburger

Technische Universität Darmstadt
salvaneschi@informatik.tu-darmstadt.de

Abstract Intro

Keywords: Functional-reactive Programming, Scala, Event-driven Programming

1 Signals and Vars

A signal 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.

2 Events

3 Imperative events

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]()
```

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.

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)
```

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

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

e += { println() }


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 Related Work

REScala builds on ideas originally developed in EScala [**?**] – 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 [**?**] (Scheme), FlapJax [**?**] (Javascript), AmbientTalk/R [**?**] and Scala.React [**?**] (Scala).

7 Acknowledgments

This work has been supported by the German Federal Ministry of Education and Research (Bundesministerium für Bildung und Forschung, BMBF) under grant No. 16BY1206E and by the European Research Council, grant No. 321217.

References