An Introduction to Reactive Programming (2)

Prof. Guido Salvaneschi
Outline

• Analysis of languages for reactive applications
• Details of reactive frameworks
• Advanced conversion functions
• Examples and exercises
• Related approaches
REACTIVE APPLICATIONS: ANALYSIS
Software Taxonomy

• A **transformational** system
  – Accepts input, performs computation on it, produces output, and terminates
  – Compilers, shell tools, scientific computations

• A **reactive** system
  – Continuously interacts with the environment
  – Updates its state
How to implement Reactive Systems?

- Observer Pattern
  - The *traditional* way in OO languages

- Language-level events
  - Event-based languages

- Signals, vars, events and their combination
  - Reactive languages
OBSERVER PATTERN: ANALYSIS
The *(good? old)* Observer Pattern

```java
boolean highTemp;
boolean smoke;

dummy void Init()
{
  tempSensor.register(this);
  smokeSensor.register(this);
}

don't need void notifyTempReading(TempEvent e) {
  highTemp = e.getValue() > 45;
  if (highTemp && smoke) {
    alert.start();
  }
}

void notifySmokeReading(SmokeEvent e) {
  smoke = e.getIntensity() > 0.5;
  if (highTemp && smoke) {
    alert.start();
  }
}
```
Observer for change propagation

- Main advantage:

Decouple the code that changes a value $X$ from the code that updates the values depending on $X$

- “Source” doesn’t know about “Constraint”
- Temp/Smoke sensors do not know about fire detector
The *(good? old)* Observer Pattern

• Events are often used to enforce data dependency *constraints*

  – boolean highTemp := (temp.value > 45);
The example

\[
\text{val } c = a + b
\]

\[
\text{val } a = 3
\]

\[
\text{val } b = 7
\]

\[
a = 4
\]

\[
b = 8
\]
The Example: Observer

trait Observable {
  val observers = scala.collection.mutable.Set[Observer]()
  def registerObserver(o: Observer) = { observers += o }
  def unregisterObserver(o: Observer) = { observers -= o }
  def notifyObservers(a: Int,b: Int) = { observers.foreach(_.notify(a,b)) }
}

trait Observer {
  def notify(a: Int,b: Int)
}

class Sources extends Observable {
  var a = 3
  var b = 7
}

class Constraint(a: Int, b: Int) extends Observer {
  var c = a + b
  def notify(a: Int,b: Int) = { c = a + b }
}

val s = new Sources()
val c = new Constraint(s.a,s.b)
s.registerObserver(c)
s.a = 4
s.notifyObservers(s.a,s.b)
s.b = 8
s.notifyObservers(s.a,s.b)
The *(good? old)* Observer Pattern

Long story of criticism...

• Inversion of *natural* dependency order
  – “Sources” updates “Constraint” but in the code
    “Constraint” calls “Sources” (to register itself)

• Boilerplate code

```scala
trait Observable {
  val observers = scala.collection.mutable.Set[Observer]()
  def registerObserver(o: Observer) = { observers += o }
  def unregisterObserver(o: Observer) = { observers -= o }
}

tempSensor.register(this);
smokeSensor.register(this);
```

Guido Salvaneschi: introduction to reactive programming
The *(good? old)* Observer Pattern

- Reactions do not compose, return void
  - How to define new constraints based on the existing ones

```scala
class Constraint(a: Int, b: Int) ... {
    var c = a + b
    def notify(a: Int, b: Int) = {
        c = a + b
    }
}

class Constraint2(d: Int) ... {
    var d = c * 7
    def notify(d: Int) = {
        d = c * 7
    }
}

+ = ???
```
The *good*? old) Observer Pattern

- Scattering and tangling of triggering code
  - **Fail to update** all functionally dependent values.
  - Values are often update too much *(defensively)*

```scala
val s = new Sources()
val c = new Constraint(s.a, s.b)
s.registerObserver(c)
s.a = 4
s.notifyObservers(s.a, s.b)
s.b = 8
s.notifyObservers(s.a, s.b)
```

Change(a)

```
change(a) view.update()
```

Change(b)

```
change(b) view.update()
```

Change(c)

```
change(c) view.update()
```
The *(good? old)* Observer Pattern

- Imperative updates of state

```java
class Constraint(a: Int, b: Int) extends Observer {
    var c = a + b
    def notify(a: Int, b: Int) = { c = a + b }
}
```

- No separation of concerns

```java
class Constraint(a: Int, b: Int) extends Observer {
    var c = a + b
    def notify(a: Int, b: Int) = { c = a + b }
}
```

---

Guido Salvaneschi: introduction to reactive programming
EVENT-BASED LANGUAGES: ANALYSIS
Event-based Languages

- Language-level support for events
  - C#, Ptolemy, REScala, ...
    
  ```scala
  val e = Evt[Int]()
  e += { println(_) }
  e(10)
  ```

- Imperative events

  ```scala
  val update = Evt[Unit]()
  ```

- Declarative events, ||, &&, map, ...

  ```scala
  val changed[Unit] = resized || moved || afterExecSetColor
  val invalidated[Rectangle] = changed.map( _ => getBounds() )
  ```
Event-based Languages

```scala
val update = Evt[Unit]()
val a = 3
val b = 7
val c = a + b  // Functional dependency

update += (_ =>{
  c = a + b
})

a = 4
update()

b = 8
update()
```
Event-based Languages

• More composable
  – Declarative events are composed by existing events (not in the example)

• Less boilerplate code
  – Applications are easier to understand

• Good integration with Objects and imperative style:
  – Imperative updates and side effects
  – Inheritance, polymorphism, ...
Event-based Languages

- Dependencies still encoded manually
  - Handler registration
- Updates must be implemented explicitly
  - In the handlers
- Notifications are still error prone:
  - Too rarely / too often

```java
class Connector(val start: Figure, val end: Figure) {
    start.changed += updateStart
    end.changed += updateEnd
    ...
    def updateStart() { ... }
    def updateEnd() { ... }
    ...
```
Reactive Languages

• Functional-reactive programming (FRP) -- Haskell
  – **Time-changing values** as dedicated language abstractions.
    \[\text{Functional reactive animation, Elliott and Hudak. ICFP '97}\]

• More recently:
  – FrTime  \[\text{Embedding dynamic dataflow in a call-by-value language, Cooper and Krishnamurthi, ESOP’06}\]
  – Flapjax  \[\text{Flapjax: a programming language for Ajax applications. Meyerovich et al. OOPSLA’09}\]
Reactive Languages and FRP

• Signals
  – Dedicated language abstractions for time-changing values

```scala
val a = Var(3)
val b = Var(7)
val c = Signal{ a() + b() }
println(c.now) > 10
a() = 4
println(c.now) > 11
```

• An alternative to the Observer pattern and inversion of control
/* Create the graphics */
title = "Reactive Swing App"
val button = new Button {
    text = "Click me!"
}
val label = new Label {
    text = "No button clicks registered"
}
contents = new BoxPanel(Orientation.Vertical) {
    contents += button
    contents += label
}

/* The logic */
listenTo(button)
val nClicks = 0
reactions += {
    case ButtonClicked(b) =>
        nClicks += 1
        label.text = "Number of button clicks: " + nClicks
        if (nClicks > 0)
            button.text = "Click me again"
    }

val nClicks = button.clicked.fold(0) {(x, _) => x + 1}

label.text = Signal { ( if (nClicks() == 0) "No" else nClicks() ) + " button clicks registered" }

button.text = Signal { "Click me" + (if (nClicks() == 0) "!" else " again ")}

contents = new BoxPanel(Orientation.Vertical) {
    contents += button
    contents += label
}
Reactive Languages

• Easier to understand
  – Declarative style
  – Local reasoning
  – No need to follow the control flow
to reverse engineer the constraints

• Dependent values are automatically consistent
  – No boilerplate code
  – No update errors (no updates/update defensively)
  – No scattering and tangling of update code

• Reactive behaviors are composable
  – In contrast to callbacks, which return void
NOW...

Signals allow a good design. But they are *functional* (only).

Functional programming is great! But...

The sad story:

- The world is **event-based**, ...
- Often **imperative**, ...
- And mostly **Object-oriented**

```scala
val a = Var(3)
val b = Var(7)
val c = Signal{ a() + b() }
val d = Signal{ 2 * c() }
val e = Signal{ "Result: " + d() }
```
Reactive Languages

• In practice, both are supported:
  – Signals (continuous)
  – Events (discrete)

• Conversion functions
  – Bridge signals and events
  – Allow interaction with objects state and imperative code

\[
\begin{align*}
\text{Changed} & \quad : \quad \text{Signal}[T] \rightarrow \text{Event}[T] \\
\text{Latest} & \quad : \quad \text{Event}[T] \rightarrow \text{Signal}[T]
\end{align*}
\]
ADVANCED INTERFACE FUNCTIONS
Fold

• Creates a signal by folding events with a function f
  – Initially the signal holds the `init` value.

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

- Variant of latest.
  - The Option type for the case the event did not fire yet.
  - Latest value of an event as Some(value) or None

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

```scala
val e = Evt[Int]()
val s: Signal[Option[Int]] = e.latestOption(e)
assert(s.now == None)
e(1)
assert(s.now == Option(1))
e(2)
assert(s.now == Option(2))
e(1)
assert(s.now == Option(1))
```
Last

• Generalizes **latest**
  – Returns a signal which holds the last $n$ events
  – Initially an empty list

• $\text{last}[T](e: \text{Event}[T], n: \text{Int}): \text{Signal}[\text{List}[T]]$

```scala
val e = Evt[Int]()
val s: Signal[List[Int]] = e.last(5)

assert(s.now == List())
e(1)
assert(s.now == List(1))
e(2)
assert(s.now == List(2,1))
e(3);e(4);e(5)
assert(s.now == List(5,4,3,2,1))
e(6)
assert(s.now == List(6,5,4,3,2))
```
List

• Collects the event values in a (ever growing) list
• Use carefully… potential memory leaks

• list[T](e: Event[T]): Signal[List[T]]
Iterate

• Repeatedly applies \( f \) to a value when \( e \) occurs
  – The return signal is constant, based on init
  – \( F \) is similar to a handler
• \( \text{iterate}[A](e: \text{Event}[_], \text{init}: A)(f: A=>A) :\text{Signal}[A] \)

```scala
var test: Int = 0
define e = Evt[Int]()
define f = (x:Int)=>{test=x; x+1}
define s: Signal[Int] = e.iterate(10)(f)
e(1)
assert(test == 10)
e(2)
assert(test == 11)
e(1)
assert(test == 12)
e(1)
assert(s.now == 10)
e(1)
assert(s.now == 10)
```

Guido Salvaneschi: introduction to reactive programming
Count

• Returns a signal that counts the occurrences of e
  – Initially, the signal holds 0.
  – The argument of the event is discarded.

• `count(e: Event[_]): Signal[Int]`

```scala
val e = Evt[Int]()
val s: Signal[Int] = e.count
assert(s.now == 0)
e(1)
e(3)
eassert(s.now == 2)
```
Snapshot

• Returns a signal updated only when \( e \) fires.
  – Other changes of \( s \) are ignored.
  – The signal is updated to the current value of \( s \).
  – Returns the signal itself before \( e \) fires

• snapshot\([V]\)(e : Event[_], s: Signal[V]): Signal[V]

```scala
val e = Evt[Int]()
val v = Var(1)
val s1 = Signal{ v() + 1 }
val s = e.snapshot(s1)
    s!? assert(s.now == 2)
e(1)
assert(s.now == 2)
v.set(2) // s1 == 3
assert(s.now == 2)
e(1)
assert(s.now == 3)
```
Change

- Similar to changed
  - changed[U]: Event[U]
  - Provides both the old and the new value in a tuple
  - change[U]: Event[Diff[U]]

```scala
val s = Signal{ ... }
val e: Event[Diff[Int]] = s.change
e += (x: Diff[Int]=> {
    val (old,nnew): (Int,Int) = x.pair
    ...
})
```
ChangedTo

• Similar to changed
  – The event is fired only if the signal holds the given value
  – The value of e is discarded
• \(\text{changedTo}[V](\text{value: } V): \text{Event}[\text{Unit}]\)

```java
var test = 0
val v = Var(1)
val s = Signal{ v() + 1 }
val e: Event[Unit] = s.changedTo(3)
e += ((x:Unit)=>{test+=1})
assert(test == 0)
v set 2
assert(test == 1)
v set 3
assert(test == 1)
```

Guido Salvaneschi: introduction to reactive programming
Toggle

- Switches between signals on the occurrence of e.
  - The value attached to the event is discarded
  - `toggle[T](e : Event[_], a: Signal[T], b: Signal[T]): Signal[T]

```scala
val e = Evt[Int]()
val v1 = Var(1)
val s1 = Signal{ v1() + 1 }
val v2 = Var(11)
val s2 = Signal{ v2() + 1 }
val s = e.toggle(s1, s2)
```

```
assert(s.now == 2)
e(1)
assert(s.now == 12)
v2.set(12)
assert(s.now == 13)
v1.set(2)
assert(s.now == 13)
e(1)
v1.set(3)
assert(s.now == 4)
v2.set(13)
assert(s.now == 4)
```
switchOnce

• Switches to a new signal provided as a parameter once, on the occurrence of e

switchOnce[T]
(e: Event[_], original: Signal[T], newSignal: Signal[T]): Signal[T]

```scala
val e = Evt[Int]()
val v1 = Var(0)
val v2 = Var(10)
val s1 = Signal{ v1() + 1 }
val s2 = Signal{ v2() + 1 }
val s3 = e.switchOnce(s1,s2)

assert(s3.now == 1)
v1.set(1)
assert(s3.now == 2)
e(1)
assert(s3.now == 11)
e(2)
v2.set(11)
assert(s3.now == 12)
```
Note on the interface

• We showed the “non OO” signature for most of the interface functions
  – In practice, the signature is in OO style
  – One of the parameters is the receiver of the method

• For example

  \[\text{snapshot}(e, s) \quad // \quad \text{snapshot}[V](e :\ Event[\_], s: \ Signal[V]): \ Signal[V]\]

  – Must be called as:

  \[e.\text{snapshot}(s) \quad // \quad e.\text{snapshot}[V](s: \ Signal[V]): \ Signal[V]\]
DETAILS ON THE REACTIVE MODEL
Implementation: Challenges

• In-language reactive abstractions
  – DSL/Compiler
  – Build the dependency model

• Language runtime
  – Dependency graph
    • Evaluation
    • Change propagation
    • Model maintenance

val e, f, g = Var(1)
val d = Var(true)
c = Signal { f() + g() }
b = Signal { e() * 100 }
a = Signal {
  if (d) c
  else b
}
DSL Implementation

• Building the graph
  – Var(3) -> leaf
  – Var(4) -> leaf
  – “a() + b()” saved in a closure
  – Signal{...} -> dependent node

• Signal expression evaluation
  – Reactive values -> edges
  – Signal = result of the evaluation

```scala
val a = Var(3)
val b = Var(4)
val c = Signal { a() + b() }
```
Pull vs. Push Models

E.g., REScala, Rx, bacon.js
Glitches

Temporary *spurious* values due to propagation order.

- Update order `abdc`
- `a()=2`  `b<-4, d<-7, c<-6, d<-10`

- Effects:
  - `d` redundantly evaluated 2 times
  - First value of `d` has *no meaning*
  - `e` erroneously fired two times
Glitch Freedom

• Ensured by updates in topological order
  – Nodes are assigned to levels $L_n$
  – Levels are updates in order
  – E.g., “abcde” or “acbde”

• Technical solutions:
  – Priority queue
  – Nodes wait for children before evaluating
Dynamic dependencies

- Dependencies based on runtime conditions
  - In case $c==\text{true}$, $d$ must change:
    - If $a$ changes
    - Not if $b$ changes
  - $d$ depends on $a$ or $b$
    based on current value of $c$
  - Change dependencies at runtime

```scala
val a = Var(3)
val b = Var(7)
val c = Var(false)
val d = Signal{
  if c() a()
  else b()
}
val e = Signal { 2 * d() }
```
(Lack of) Dynamic dependencies

- Easier implementation
- Redundant evaluations
  - d is executed upon b assignments
  - even if the d does not change

```-scala
case class Signal{
  def if function:
    a() else function
  def b()
}
while true
  b() = ... // system time
```
About Loops

• Reject loops
  – Responsibility to the programmer (REScala, Flapjax)
  – Loops rejected by the compiler

• Accept loops: which semantics?
  – Delay to the next propagation round
  – Fix point semantics
    • Time consuming?
    • Termination?

val x = Signal { y() + 1 }
val y = Signal { x() + 1 }
EXAMPLES AND EXERCISES
Example: Interface Functions

- **Count mouse clicks**

  ```scala
  val click: Event[(Int, Int)] = mouse.click
  val nClick = Var(0)
  click += { _ =>
    nClick.transform(_ + 1) // nClick() = nClick.now + 1
  }
  ```

- **Better with interface functions**

  ```scala
  val click: Event[(Int, Int)] = mouse.click
  val nClick: Signal[Int] = click.fold(0)( (x, _) => x+1 )
  ```

- **Even better: use `count`!**

  ```scala
  val click: Event[(Int, Int)] = mouse.click
  val nClick: Signal[Int] = click.count()
  ```

Conciseness vs. Generality
Example: Interface Functions

• Hold the position of the last click in a signal

```scala
val clicked: Event[Unit] = mouse.clicked
val position: Signal[(Int,Int)] = mouse.position
var lastClick = Var(0,0)
clicked += { _ =>
  lastClick() = position()
}
```

• Better with interface functions

```scala
val clicked: Event[Unit] = mouse.clicked
val position: Signal[(Int,Int)] = mouse.position
val lastClick: Signal[(Int,Int)] = position snapshot clicked
```
Mean Over Window

- Events collect *Double* values from a sensor
- Mean over a shifting window of the last $n$ events
- Print the mean only when it changes

**OLD**

```
e(2) e(1) e(3) e(2) e(1) e(3) e(4) e(1) e(3) e(4)
```

**NEW**

```
e(3) e(4)
```

$n$
Mean Over Window

- Mean over a shifting window of the last n events
- Print the mean only when it changes

```scala
val e = Evt[Double]

val window = e.last(5)
val mean = Signal {
  window().sum / window().length
}
mean.changed += {
  println(_)
}

e(2); e(1); e(3); e(4); e(1); e(1)
```

```
2.0
1.5
2.0
2.5
2.2
2.0
```
Example: Interface Functions

/* Compose reactive values */
val mouseChangePosition = mouseMovedE || mouseDraggedE
val mousePressedOrReleased = mousePressedE || mouseReleasedE
val mousePosMoving: Signal[Point] = mouseChangePosition.latest(new Point(0, 0))
val pressed: Signal[Boolean] = mousePressedOrReleased.toggle(Signal{false}, Signal{true})
/* Compose reactive values */
val mouseChangePosition = mouseMovedE || mouseDraggedE
val mousePressedOrReleased = mousePressedE || mouseReleasedE
val mousePosMoving: Signal[Point] = mouseChangePosition.latest(new Point(0, 0))
val pressed: Signal[Boolean] = mousePressedOrReleased.toggle(Signal{ false }, Signal{ true })
Example: Time Elapsing

- We want to show the elapsing time on a display
- (second, minute, hour, day)

```
(0,0,0,0)   (1,2,0,0)
(1,0,0,0)   ... 
(2,0,0,0)   (59,59,0,0)
  ...     (0,0,1,0)
(59,0,0,0)  ... 
(0,1,0,0)   (59,59,23,0)
(1,1,0,0)   (0,0,0,1)
(2,1,0,0)   ....
  ...     
(59,1,0,0) 
(0,2,0,0)
```
object TimeElapsing extends App {

println("start!")

val tick = Var(0)
val second = Signal{ tick() % 60 }
val minute = Signal{ tick()/60 % 60 }
val hour = Signal{ tick()/(60*60) % (60*60) }
val day = Signal{ tick()/(60*60*24) % (60*60*24) }

while(true){
  Thread.sleep(0)
  println((second.now, minute.now, hour.now, day.now))
  tick.set(tick.now + 1)
}
}

But day is still circular.
At some point day==0 again

Also, conceptually hard to follow
object AdvancedTimeElapsing extends App {
    println("start!")
    val tick = Evt[Unit]()

    val numTics = tick.count
    val seconds = Signal{ numTics() % 60 }
    val minutes = Signal{ seconds.changedTo(0).count.apply % 60 }
    val hours = Signal{ minutes.changedTo(0).count.apply % 24 }
    val days = hours.changedTo(0).count

    while(true){
        Thread.sleep(0)
        println((seconds.now, minutes.now, hours.now, days.now))
        tick(()); // tick.fire()
    }
}
Exercise: draw dependency graph

val tick = Evt[Unit]()
val numTics = tick.count
val seconds = Signal{ numTics() % 60 }
val minutes = Signal{ seconds.changedTo(0).count() % 60 }
val hours = Signal{ minutes.changedTo(0).count() % 24 }
val days = hours.changedTo(0).count

• Which variables are affected by a change to tick?
Example: Smashing Particles

• Particles
  – Get bigger
  – Move bottom-right
val toDraw = ListBuffer[Function1[Graphics2D, Unit]]()
type Delta = Point

class Oval(center: Signal[Point], radius: Signal[Int]) {
  toDraw += ((g: Graphics2D) =>
    {g.fillOval(center.now.x, center.now.y, radius.now, radius.now)})
}

val base = Var(0)
val time = Signal{base() % 200} // time is cyclic :)

val point1 = Signal{ new Point(20+time(), 20+time())}
new Oval(point1, time)
val point2 = Signal{ new Point(40+time(), 80+time())}
new Oval(point2, time)
...

• Signals are used inside objects!

override def main(args: Array[String]){
  while (true) {
    frame.repaint
    Thread.sleep 20
    base() = base.now + 1
  }
}
QUESTIONS?
Training with RP - Resources

- Examples in the lecture slides
  - Observer
  - Reactive programming
- Homework assignments
- REScala examples (online, RP and OO version)
- REScala manual (online)
List<String> l = Arrays.asList("a1", "c2", "b1", "c2");
l.stream()
  .filter(s -> s.startsWith("c"))
  .map(String::toUpperCase)
  .sorted()
  .collect(Collectors.toList)

gedataFromNetwork()
  .skip(10)
  .take(5)
  .map({ s -> return s + " transformed" })
  .subscribe({ println "onNext => " + it })

TwitterUtils.createStream(...)
  .filter(__.getText.contains("Spark"))
  .countByWindow(Seconds(5))