An Introduction to Reactive Programming

Prof. Guido Salvaneschi
Outline

• Intro to reactive applications
• The Observer pattern
• Event-based languages
• Reactive languages
INTRO TO REACTIVE APPLICATIONS
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

Guido Salvaneschi: introduction to reactive programming
Use of State

• Transformational systems:
  – Express transformations as incremental modifications of the internal data structures

  State is not necessary to describe the system

• Reactive systems:
  – Represent the current state of interaction
  – Reflect changes of the external world during interaction

  State is essential to describe the system
Reactive Applications

Interactive Applications
UI

Monitoring / Control Systems
Reactive Applications

• Many other examples
  – Web applications
  – Mobile apps
  – Distributed computations
    • Cloud
  – ...

• Typical operations
  – Detect events/notifications and react
  – Combine reactions
  – Propagate updates/changes
Reactive Applications
Why should we care?

• Event handling:
  – 30% of code in desktop applications
  – 50% of bugs reported during production cycle
Reactive Programming

Now...

- Reactive applications are extremely common
- Can we design new language features to specifically address this issue?

• Think about the problems solved by exceptions, visibility modifiers, inheritance, ...
REACTIVE PROGRAMMING
Reactive Programming

Definition... ?

“Programming language abstractions (techniques and patterns) to develop reactive applications”

For example, abstractions to:

- Represent event streams
- Automatically propagate changes in the state
- Combine events
  ...

Guido Salvaneschi: introduction to reactive programming
Reactive Programming

- Haskell: Fran, Yampa
- FrTime, Flapjax, REScala, Scala.react, ...
- Angular.js, Bacon.js, Reactive.js, ...
- Microsoft Reactive Extensions (Rx)

- Books 2014-17
...becoming Very Popular

Java 8

```
List<String> l = Arrays.asList("a1", "c2", "b1", "c2");
l.stream()
  .filter(s -> s.startsWith("c"))
  .map(String::toUpperCase)
  .sorted()
  .collect(Collectors.toList)
```

ReactiveX

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

Spark

```
val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
  .map(word => (word, 1))
  .reduceByKey(_ + _)
  .count
  .saveAsTextFile("hdfs://...")
```
Reactive Extensions (Rx)

Java: RxJava
JavaScript: RxJS
C#: Rx.NET
C#(Unity): UniRx
Scala: RxScala
Clojure: RxClojure
C++: RxCpp
Lua: RxLua
Ruby: Rx.rb
Python: RxPY
Groovy: RxGroovy
JRuby: RxJRuby
Kotlin: RxKotlin
Swift: RxSwift
PHP: RxPHP
Reactive Programming

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

```
Observer
+notify()

Subject
+observerCollection
+registerObserver(observer)
+unregisterObserver(observer)
+notifyObservers()

notifyObservers()
for observer in observerCollection
  call observer.notify()

ConcreteObserverA
+notify()

ConcreteObserverB
+notify()
```
The Observer Pattern

• What about Java Swing?
  – javax.swing
public class Beeper extends JPanel implements ActionListener {

    JButton button;

    public Beeper() {
        super(new BorderLayout());
        button = new JButton("Click Me");
        button.setPreferredSize(new Dimension(200, 80));
        add(button, BorderLayout.CENTER);
        button.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e) {
        Toolkit.getDefaultToolkit().beep();
    }

    private static void createAndShowGUI() {
        // Create the GUI and show it.
        JFrame frame = new JFrame("Beeper");       //Create and set up the window.
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JComponent newContentPane = new Beeper();  //Create and set up the content pane.
        newContentPane.setOpaque(true);
        frame.setContentPane(newContentPane);    //Display the window.
        frame.pack();
        frame.setVisible(true);
    }

    public static void main(String[] args) {
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                createAndShowGUI();
            }
        });
    }
}
boolean highTemp;
boolean smoke;

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

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();
    }
}
EVENT-BASED LANGUAGES
Event-based Languages

Language-level support for events

• Events as object attributes
  – Describe changes of the object's state
  – Part of the interface

• Event-based languages are *better*!
  – More concise, clear programming intention, ...
  – C#, Ptolemy, EScala, EventJava, ...
public class Drawing {
    Collection<Figure> figures;

    public event NoArgs Changed();

    public virtual void Add(Figure figure) {
        figures.Add(figure);
        figure.Changed += OnChanged;
        OnChanged();
    }

    public virtual void Remove(Figure figure) {
        figures.Remove(figure);
        figure.Changed -= OnChanged;
        OnChanged();
    }

    protected virtual void OnChanged() {
        if (Changed != null) {
            Changed();
        }
    }

    ...
}
EVENTS IN SCALA
REScala

- www.rescala-lang.com
  - An advanced event-based system
  - Abstractions for time-changing values
  - Bridging between them

- Philosophy: foster a more declarative and functional style without sacrificing the power of OO design
- Pure Scala
Adding Events to Scala

• C# events are recognized by the compiler
  – Scala does not support events by itself, but...

• Can we introduce events using the powerful Scala support for DSLs?

• Can we do even better than C#?
  – E.g., event composition?
REScala events: Summary

• Different types of events: Imperative, declarative, ...

• Events carry a value
  – Bound to the event when the event is fired
  – Received by all the handlers

• Events are parametric types.
  – Event[T], Evt[T]

• All events are subtype of Event[T]
Imperative Events

• Valid event declarations

   val e1 = Evt[Unit]()
   val e2 = Evt[Int]()
   val e3 = Evt[String]()
   val e4 = Evt[Boolean]()

   val e5: Event[Int] = Evt[Int]()

   class Foo
   val e6 = Evt[Foo]()
Imperative Events

• Multiple values for the same event are expressed using tuples

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

val e4 = Evt[(Boolean,String,Int)]()

val e5: Evt[(Int,Int)] = Evt[(Int,Int)]()
```
Handlers

• Handlers are executed when the event is fired
  – The += operator registers the handler.
• The handler is a first class function
  – The attached value is the function parameter.

```scala
var state = 0
val e = Evt[Int]()
e += { println(_) }
e += (x => println(x))
e += ((x: Int) => println(x))
e += (x => { // Multiple statements in the handler
  state = x
  println(x)
})
```
Handlers

- The signature of the handler must conform the event
  - E.g., Event[(Int,Int)] requires (Int,Int) => Unit
  - The handler:
    - receives the attached value
    - performs side effects.

```scala
val e = Evt[(Int,String)]()
e += (x => {
  println(x._1)
  println(x._2)
})
e += (x: (Int,String) => {
  println(x)
})
```
Handlers

- Events without arguments still need a Unit argument in the handler.

```scala
val e = Evt[Unit]()
e += { x => println("Fired!") }
e += { (x: Unit) => println("Fired!") }
```
Methods as Handlers

• Methods can be used as handlers.
  – *Partially applied functions* syntax
  – Types must be correct

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

val e = Evt[Int]
e += m1_
e(10)
```
Firing Events

• Method call syntax
• The value is bound to the event occurrence

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

e1(10)
e2(false)
e3((10,"Hallo"))
```
Firing Events

- Registered handlers are executed every time the event is fired.
  - The actual parameter is provided to the handler

```scala
val e = Evt[Int]()
e += { x => println(x) }
e(10)
e(11)

-- output ----
10
11
```
Firing Events

• All registered handlers are executed
  – The execution order is non deterministic

```scala
val e = Evt[Int]()
e += { x => println(x) }
e += { x => println("n: "+ x)}
e(10)
e(11)

-- output ----
10
n: 10
11
n: 11
```
Firing Events

• The .remove operator unregisters a handler via its handle
• The += operator also returns the handle that will be used for unregistration

val e = Evt[Int]()
val handler1 = { x: Int => println(x) }
val handler2 = { x: Int => println("n: " + x) }

val h1 = e += handler1
val h2 = e += handler2
e(10)
h1.remove
e(10)
h2.remove
e(10)

-- output ----
10
n: 10
n: 10
Imperative Events

- Events can be referred to generically

\[
\text{val e1: Event[Int] = Evt[Int]()}\]
DECLARATIVE EVENTS
The Problem

- Imperative events are fired by the programmer
- Conceptually, certain events depend on other events

- Examples:
  - `mouseClickE -> museClickOnShape`
  - `mouseClose, keyboardClose -> closeWindow`

- Can we solve this problem enhancing the language?
Declarative Events

- Declarative events are defined by a combination of other events.

- Some valid declarations:

``` scala
val e1 = Evt[Int]()
val e2 = Evt[Int]()

val e3 = e1 || e2
val e4 = e1 && ((x: Int) => x > 10)
val e5 = e1 map ((x: Int) => x.toString)
```
OR events

• The event e1 || e2 is fired upon the occurrence of one among e1 or e2.
  – The events in the event expression have the same parameter type

```java
val e1 = Evt[Int]()
val e2 = Evt[Int]()
val e1_OR_e2 = e1 || e2
val e1_OR_e2 += ((x: Int) => println(x))
e1(10)
e2(10)

-- output ----
10
10
```
Predicate Events

• The event e && 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 and returns Boolean.
  – && filters events using a parameter and a predicate.

```scala
val e = Evt[Int]()
val e_AND: Event[Int] = e && ((x: Int) => x>10)
e_AND += ((x: Int) => println(x))
e(5)
e(15)
-- output ----
15
```
Map Events

• The event e map f is obtained by applying f to the value carried by e.
  – The map function takes the event parameter as a formal.
  – The return type of map is the type parameter of the resulting event.

```scala
val e = Evt[Int]()
val e_MAP: Event[String] = e map ((x: Int) => x.toString)
e_MAP += ((x: String) => println("Here: " + x))
e(5)
e(15)
```

-- output ----
Here: 5
Here: 15
EXAMPLES OF RESCALA EVENTS
Example: Figures

abstract class Figure {
  val moved[Unit] = afterExecMoveBy
  val resized[Unit]
  val changed[Unit] = resized || moved || afterExecSetColor
  val invalidated[Rectangle] = changed.map(_ => getBounds())

  val afterExecMoveBy = new Evt[Unit]
  val afterExecSetColor = new Evt[Unit]

  def moveBy(dx: Int, dy: Int) { position.move(dx, dy); afterExecMoveBy() }
  def resize(s: Size) { size = s }
  def setColor(col: Color) { color = col; afterExecSetColor() }
  def getBounds(): Rectangle
}

Guido Salvaneschi: introduction to reactive programming
Example: Figures

class Connector(val start: Figure, val end: Figure) {
    val h1 = start.changed += updateStart _
    val h2 = end.changed += updateEnd _
    ...
    def updateStart() { ... }
    def updateEnd() { ... }
    ...
    def dispose {
        h1.remove
        h2.remove
    }
}
Example: Figures

- Inherited events
  - May be overridden
  - Are late bound

```scala
abstract class Figure {
  val moved[Unit] = afterExecMoveBy
  val resized[Unit]

  ...
}

class RectangleFigure extends Figure {
  val resized = afterExecResize || afterExecSetBounds
  override val moved = super.moved || afterExecSetBounds

  ...
  val afterExecResize = new Evt[Unit]
  val afterExecSetBounds = new Evt[Unit]

  ...
  def resize(s: Size) { ... ; afterExecResize() }
  def setBounds(x1: Int, y1: Int, x2: Int, y2: Int) { ... ; afterExecSetBounds }

  ...
}
```
Example: Temperature Sensor

```scala
class TemperatureSensor {
  val tempChanged[Int] = new Evt[Int]

  ... 

  def run {
    var currentTemp = measureTemp()
    while(!stop) {
      val newTemp = measureTemp()
      if (newTemp != currentTemp) {
        tempChanged(newTemp)
        currentTemp = newTemp
      }
      sleep(100)
    }
  }
}
```
REACTIVE LANGUAGES
Events and Functional Dependencies

Events are often used for functional dependencies

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

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

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

a = 4
update()
b = 8
update()
Constraints

- What about expressing functional dependencies as constraints?

```scala
val a = 3
val b = 7
val c = a + b // Statement
println(c)
> 10
a= 4
println(c)
> 10

val a = 3
val b = 7
val c := a + b // Constraint
println(c)
> 10
a= 4
println(c)
> 11
```
EMBEDDING REACTIVE PROGRAMMING IN SCALA
Reactive Values

- **Vars**: primitive reactive values
  - Updated “manually”

- **Signals**: reactive expressions
  - The constraints “automatically” enforced

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

• Change propagation model
  – Dependency graph
  – Push-driven evaluation

```scala
val a = Var(3)
val b = Var(7)
val c = Signal{ a() + b() }
val d = Signal { 2 * c() }
...```

Guido Salvaneschi: introduction to reactive programming
SIGNALS AND VARS
Vars

• Vars wrap normal Scala values

• Var[T] is a parametric type.
  – The parameter T is the type the var wraps around
  – Vars are assigned by the "()=" operator

```scala
val a = Var(0)
val b = Var("Hello World")
val c = Var(false)
val d: Var[Int] = Var(30)
val e: Var[String] = Var("REScala")
val f: Var[Boolean] = Var(false)

a()= 3
b()="New World"
c()=true
```
Signals

• Syntax: `Signal{sigexpr}`
  – Sigexpr should be **side-effect free**

• Signals are parametric types.
  – A signal `Signal[T]` carries a value of type `T`
Signals: Collecting Dependencies

- A **Var** or a **Signal** called with () in a signal expression is added to the dependencies of the defined signal.

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

![Diagram showing dependencies between variables and signal.](image)
Signals: Examples

```scala
val a = Var(0)
val b = Var(0)
val c = Var(0)

val r: Signal[Int] = Signal{ a() + 1 }  // Explicit type in var decl
val s = Signal{ a() + b() }            // Multiple vars in a signal expression
val t = Signal{ s() * c() + 10 }       // Mix signals and vars in signal expressions
val u = Signal{ s() * t() }            // A signal that depends on other signals
```
Signals: Examples

```scala
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
}
```
Signals

- Accessing reactive values: now
  - Often used to return to a *traditional* computation

```scala
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.now
val y: Int = s.now

val z: Boolean = t.now
println(z)
```
EXAMPLES OF SIGNALS
Example

Guido Salvaneschi: introduction to reactive programming
Example: Observer

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

```
title = "Reactive Swing App"
val label = new ReactiveLabel
val button = new ReactiveButton

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
}
```
Example: Smashing Particles

```scala
class Oval(center: Signal[Point], radius: Signal[Int]) {
  ...
}
val base = Var(0)  // Increases indefinitely
val linearTime = base()
val cyclicTime = Signal{linearTime() % 200}
val point1 = Signal{
  new Point(20+ cyclicTime (), 20+ cyclicTime ())
} new Oval(point1, cyclicTime )
...  // 4 times
```
BASIC CONVERSION FUNCTIONS
REScala design principles

• Signals (and events) are objects fields
  – Inheritance, late binding, visibility modifiers, ...

• Conversion functions bridge signals and events
Basic Conversion Functions

- Changed :: Signal[T] -> Event[T]
- Latest :: Event[T] -> Signal[T]
Example: Changed

```scala
val SPEED = 10
val time = Var(0)
val space = Signal{ SPEED * time() }

while (true) {
    Thread.sleep 20
    time() = time.now + 1
}

space.changed += ((x: Int) => println(x))

-- output --
10
20
30
40
...
```
val senseTmp = Evt[Int]() // Fahrenheit
val threshold = 40

val fahrenheitTmp = senseTmp.latest(0)
val celsiusTmp = Signal{ (fahrenheitTmp() – 32) / 1.8 }

val alert = Signal{ if (celsiusTmp() > threshold) "Warning" else "OK" }
Quiz 1

val v1 = Var(4)
val v2 = Var(2)
val s1 = Signal{ v1() + v2() }
val s2 = Signal{ s1() / 3 }

assert(s2.now == 2)
v1()=1
assert(s2.now == 1)
var test = 0
val v1 = Var(4)
val v2 = Var(2)
val s1 = Signal{ v1() + v2() }
s1.changed += ((x: Int)=>{test+=1})

assert(test == 0)
v1()=1
assert(test == 1)
Quiz 3

```scala
val e = Evt[Int]()
val v1 = Var(4)
val v2 = Var(2)
val s1 = e.latest(0)
val s2 = Signal{v1() + v2() + s1()}

assert(s2.now == 6)
e(2)
assert(s2.now == 8)
e(1)
assert(s2.now == 7)
```
TRUBLESHOOTING
Common pitfalls

• Establishing dependencies
  – () creates a dependency.
    Use only in signal expressions
  – now returns the current value

• Signals are not assignable.
  – Depend on other signals and vars
  – Are automatically updated

\[
\begin{align*}
  \text{val } a &= \text{Var}(2) \\
  \text{val } b &= \text{Var}(3) \\
  \text{val } c &= \text{Signal}\{ a.\text{now} + b() \}
\end{align*}
\]
Common pitfalls

• Avoid side effects in signal expressions

```scala
var c = 0
val s = Signal{
  val sum = a() + b();
  c = sum * 2
}
...
foo(c)
```

```scala
val c = Signal{
  val sum = a() + b();
  sum * 2
}
...
foo(c.now)
```

• Avoid cyclic dependencies

```scala
val a = Var(0)
val s = Signal{ a() + t() }
val t = Signal{ a() + s() + 1 }
```
Reactive Abstractions and Mutability

• Signals and vars hold references to objects, not the objects themselves.

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

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

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

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

```scala
class Foo(x: Int) //Immutable
val foo = new Foo(1)

val varFoo = Var(foo)
val s = Signal{
  varFoo().x + 10
}
assert(s.now == 11)
varFoo() = new Foo(2)
assert(s.now == 12)
```
QUESTIONS?