An Introduction to Reactive Programming (2)

Guido Salvaneschi

Software Technology Group

Outline

- Analysis of languages for reactive applications
- Details of reactive frameworks
- Advanced conversion functions
- Examples and exercises
- Related approaches

REACTIVE APPLICATIONS: ANALYSIS

How to implement Reactive Systems ?

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

- Language-level events
  - Event-based languages

- Signals, vars, events and combinations of.
  - Reactive languages

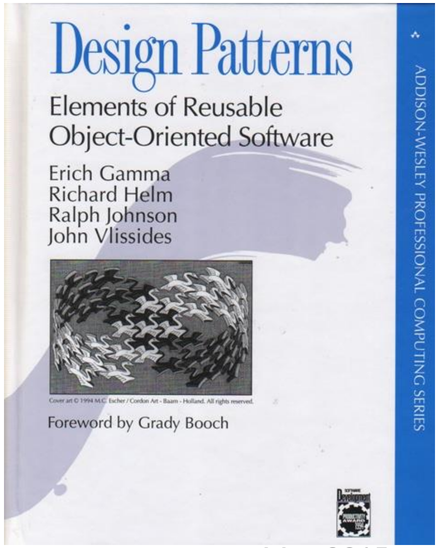
OBSERVER PATTERN: ANALYSIS

Observer for change propagation

- Main advantage:

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

  - "Sources" 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

val c = a + b

val a = 3

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

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

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

The (*good*? old) Observer Pattern

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

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

+

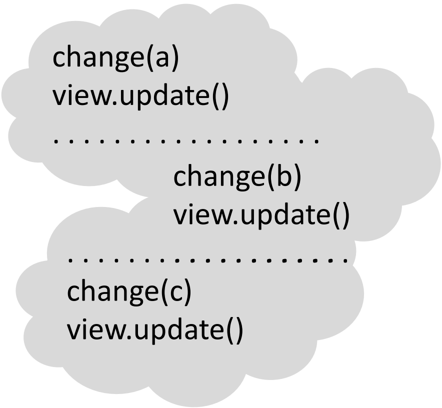
```
class Constraint2(d: Int) … {
   var d =  c * 7
   def notify(d: Int) = {
      d = c * 7
} }
```

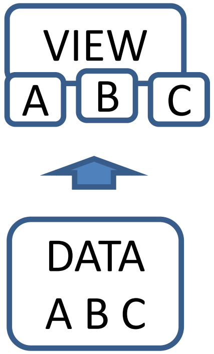
= ??

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**)

```
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

- Imperative updates of state

```
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

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

Update logic
+
Constraint definition

EVENT-BASED LANGUAGES: ANALYSIS

Event-based Languages

- Language-level support for events
  - C#, Ptolemy, REScala, …

```
val e = new ImperativeEvent[Int]()
e += { println(_) }
e(10)
```

  - Imperative events

```
val update = new ImperativeEvent[Unit]()
```

  - Declarative events, ||, &&, map, …

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

Event-based Languages

```
val update = new ImperativeEvent[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

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

REACTIVE LANGUAGES: ANALYSIS

Reactive Languages

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

- More recently:
  - FrTime *[Embedding dynamic dataflow in a call-by-value language, Cooper and Krishnamurthi, ESOP'06]*
  - Flapjax *[Flapjax: a programming language for Ajax applications. Meyerovich et al. OOPSLA'09]*
  - Scala.React *[I.Maier et al, Deprecating the Observer Pattern with Scala.React. Technical report, 2012]*

Reactive Languages and FRP

- Signals
  - Dedicated language abstractions for **time-changing** values

```
val a = Var(3)
val b = Var(7)
val c = Signal{ a() + b() }

println(c.get)
> 10
a()= 4
println(c.get)
> 11
```

- An alternative to the Observer pattern and inversion of control

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).

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

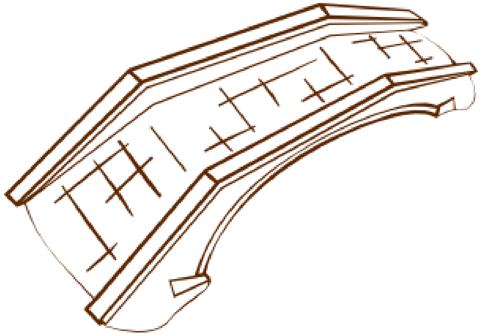
Functional programming is great! But…

The sad story:

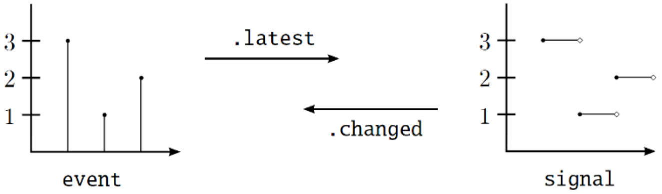
- The world is **event-based**, …
- Often **imperative, …**
- And mostly **Object-oriented**

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



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



ADVANCED INTERFACE FUNCTIONS

Fold

- Creates a signal by folding events with a function f
  - Initially the signal holds the **init** value.
- 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)
assert(s.get == 10)
e(1)
e(2)
assert(s.get == 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]]

```
val e = new ImperativeEvent[Int]()
val s: Signal[Option[Int]] = e.latestOption(e)
assert(s.get == None)
e(1)
assert(s.get == Option(1))
e(2)
assert(s.get == Option(2))
e(1)
assert(s.get == Option(1))
```

Last

- Generalizes **latest**
  - Returns a signal which holds the last **n** events
  - Initially an empty sequence
- last[T](e: Event[T], n: Int): Signal[Seq[T]]

```
val e = new ImperativeEvent[Int]()
val s: Signal[Seq[Int]] = e.last(5)
assert(s.get == Seq())
e(1)
assert(s.get == Seq(1))
e(2)
assert(s.get == Seq(1,2))
```

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

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 acc when e occurs
  - f is applied to an accumulator produced by the previous iteration (acc=init in the first iteration)
  - The value of e is ignored. The returned signal holds f(acc)
- iterate[A](e: Event[_], init: A)(f: A=>A) :Signal[A]

```
var test: Int = 0
val e = new ImperativeEvent[Int]()
val f = (x:Int)=>{test=x; x+1}
val s: Signal[Int] = e.iterate(10)(f)
```

```
e(70)
assert(test == 10)
assert(s.get == 11)
e(80)
assert(test == 11)
assert(s.get == 12)
e(15)
assert(test == 12)
assert(s.get == 13)
```

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]

```
val e = new ImperativeEvent[Int]()
val s: Signal[Int] = e.count
assert(s.get == 0)
e(1)
e(3)
assert(s.get == 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]

```
val e = new ImperativeEvent[Int]()
val v =  Var(1)
val s1 = Signal{ v() + 1 }
val s = e.snapshot(s1)
```

S !?

```
assert(s.get == 2)
e(1)
assert(s.get == 2)
v.set(2)  // s1 == 3
assert(s.get == 2)
e(1)
assert(s.get == 3)
```

Change

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

```
val s = Signal{ ... }
val e: Event[(Int,Int)] = s.change
e += (x: (Int,Int)=> {
   ...
})
```

ChangedTo

- Similar to changed
  - The event is fired only if the signal holds the given value
  - The value of e is discarded
- changedTo[V](value: V): Event[Unit]

```
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)
```

test !?

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]

```
val e = new ImperativeEvent[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)
```

S !?

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

switchTo

- Switches the signal on the occurrence of the event e.
  - The final result is a constant signal
  - The value of the retuned signal is carried by the event e.
- switchTo[T](e : Event[T], original: Signal[T]): Signal[T]

```
val e = new ImperativeEvent[Int]()
val v =  Var(1)
val s1 = Signal{ v() + 1 }
val s2 = s1.switchTo(e)
```

```
assert(s2.get == 2)
e(1)
assert(s2.get == 1)
e(100)
assert(s2.get == 100)
v.set(2)
assert(s2.get == 100)
```

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]

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

```
assert(s3.get == 1)
v1.set(1)
assert(s3.get == 2)
e(1)
assert(s3.get == 11)
e(2)
v2.set(11)
assert(s3.get == 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

```
IFunctions.snapshot(e,s)    // snapshot[V](e : Event[_], s: Signal[V]): Signal[V]
```

  - Can be called as:

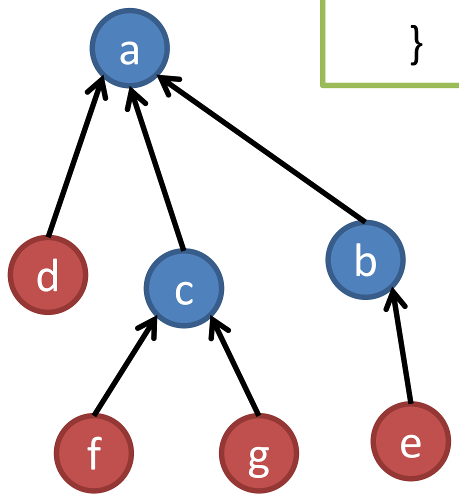
```
e.snapshot(s)    // e.snapshot[V](s: Signal[V]): Signal[V]
s.snapshot(e)    // s.snapshot[V](e : Event[_]): Signal[V]
```

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

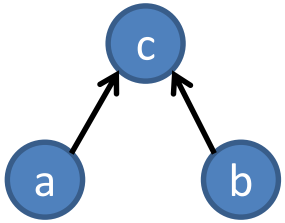
```
val e, f, g = Var(1)
val d = Var(true)

c = Signal { f() + g() }
b = Signal { e() * 100 }
a = Signal {
        if (d) c
        else b
    }
```



DSL Implementation

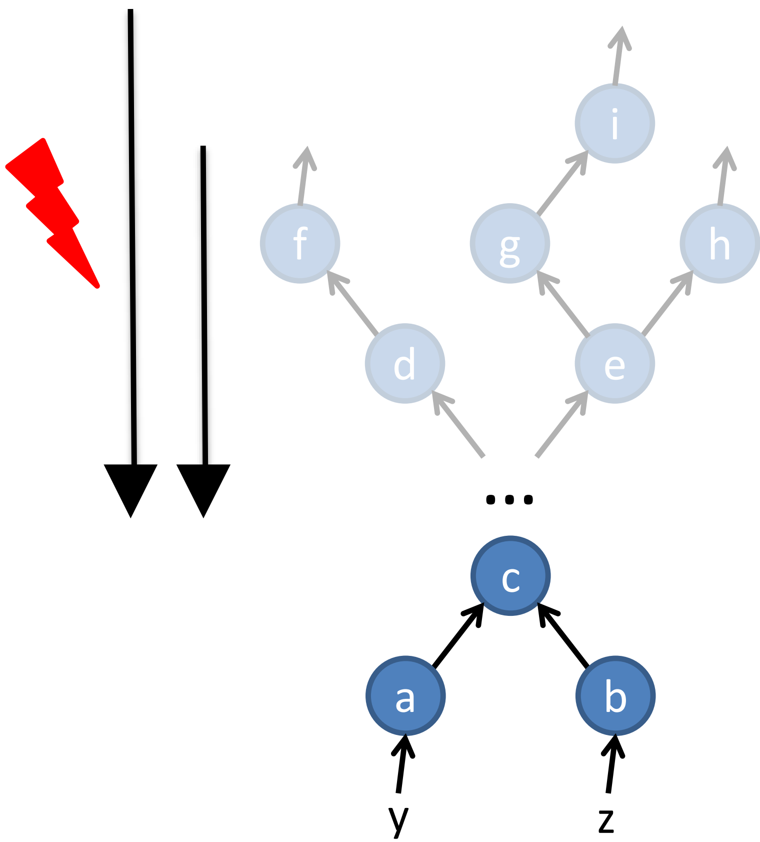
- Building the graph
  - Var(3) -> leaf
  - Var(4) -> leaf
  - "a() + b()" saved in a closure
  - Signal{...} -> dependent node

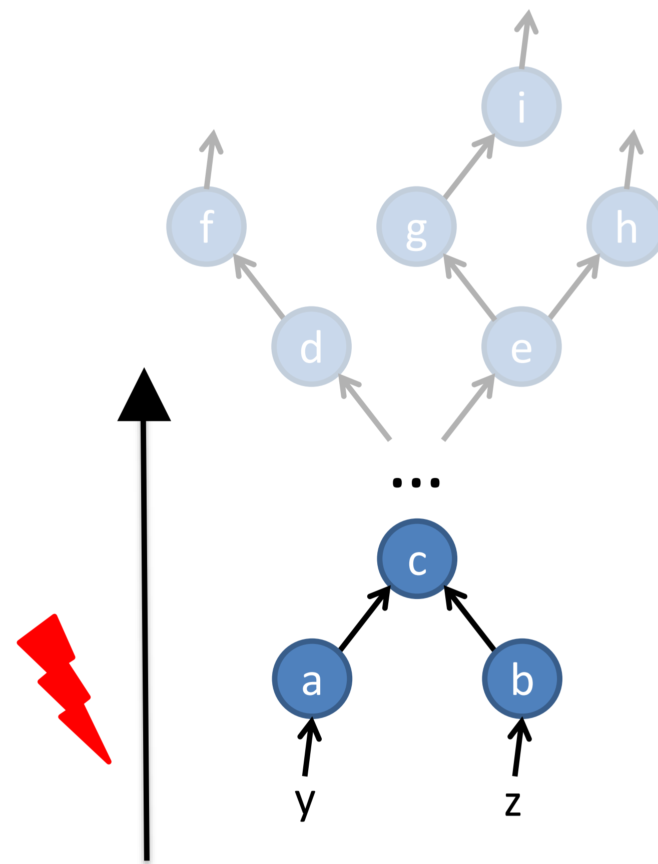
```
val a = Var(3)
val b = Var(4)
val c = Signal { a() + b() }
```



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

Pull vs. Push Models





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

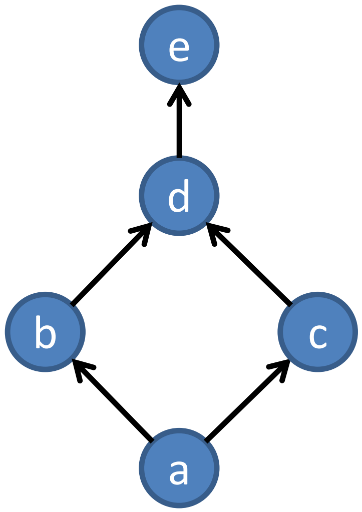
Glitches

Temporary *spurious* values due to propagation order.

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

```
val a = Var(1)
val b = Signal{ a()*2 }
val c = Signal{ a()*3 }
val d = Signal{ b() + c() }
val e = d.changed
```

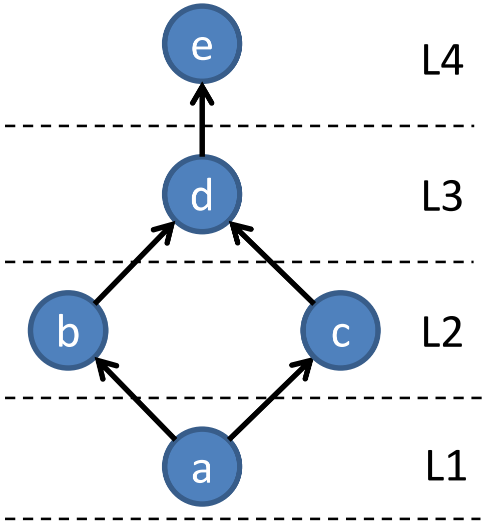
- Effects:
  - d redundantly evaluated 2 times
  - First value of d has *no meaning*
  - e erroneously fired two times



Glitch Freedom

- Ensured by updates *in topological order*
  - Nodes are assigned to levels **Ln**
  - Levels are updates in order
  - E.g., "<u>abcde</u>" or "<u>acbde</u>"

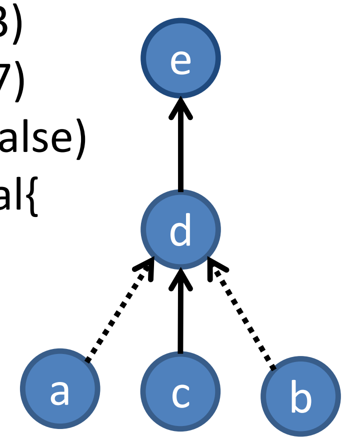
- Technical solutions:
  - Priority queue
  - Nodes wait for children



Dynamic dependencies

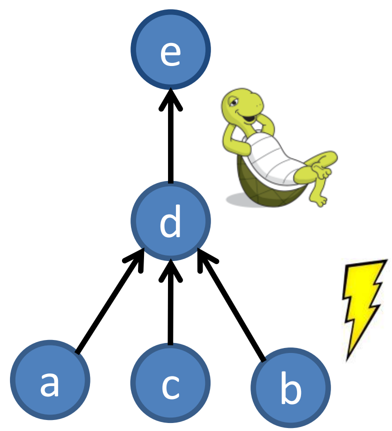
- Dependencies based on runtime conditions
  - In case *c==true*, *d* must change:
    - If *a* changes
    - Not if *b* changes
  - *d* depends on *a* or *b* based on current *c*
  - Change dependencies at runtime

```
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

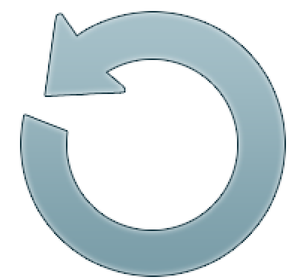



```
val a = Var(3)
val t = Var(7)
val c = Var(true)
val d = Signal{
  if c()
    a()
  else
    b()
}
while(true){
  b()= … // system time
}
```

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

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

- Better with interface functions

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

- Even better: use *count*!

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

Conciseness
vs.
Generality

Example: Interface Functions

- Keep the position of the last click in a signal

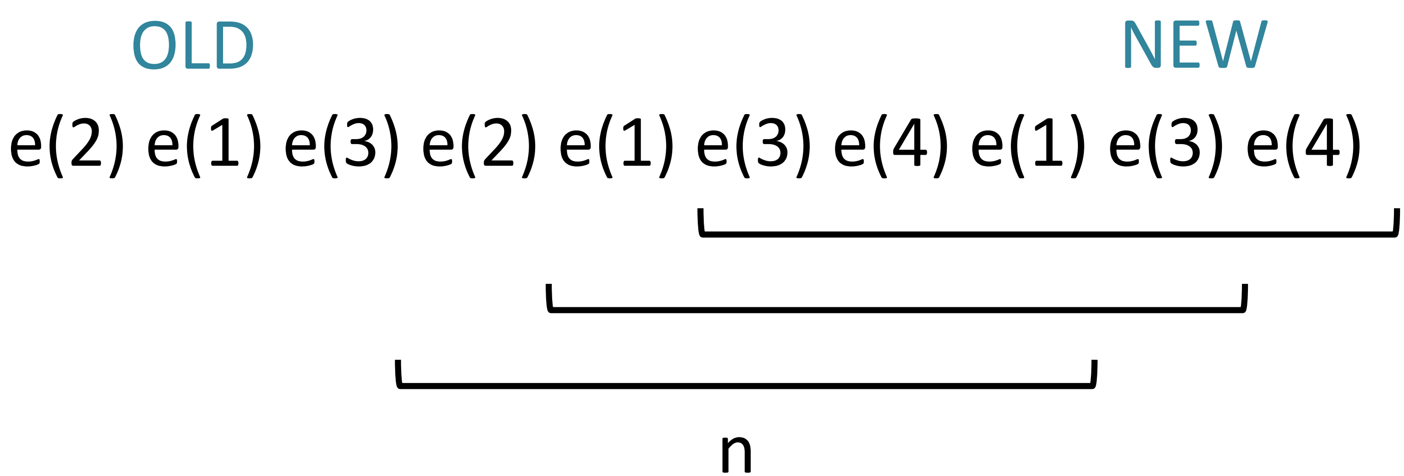
```
val clicked: Event[Unit] = mouse.clicked
val position: Signal[(Int,Int)] = mouse.position
var lastClick = Var(0,0)
clicked += { _ =>
  lastClick()= position()
}
```

- Better with interface functions

```
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



Mean Over Window

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

```
val e = new ImperativeEvent[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)
```

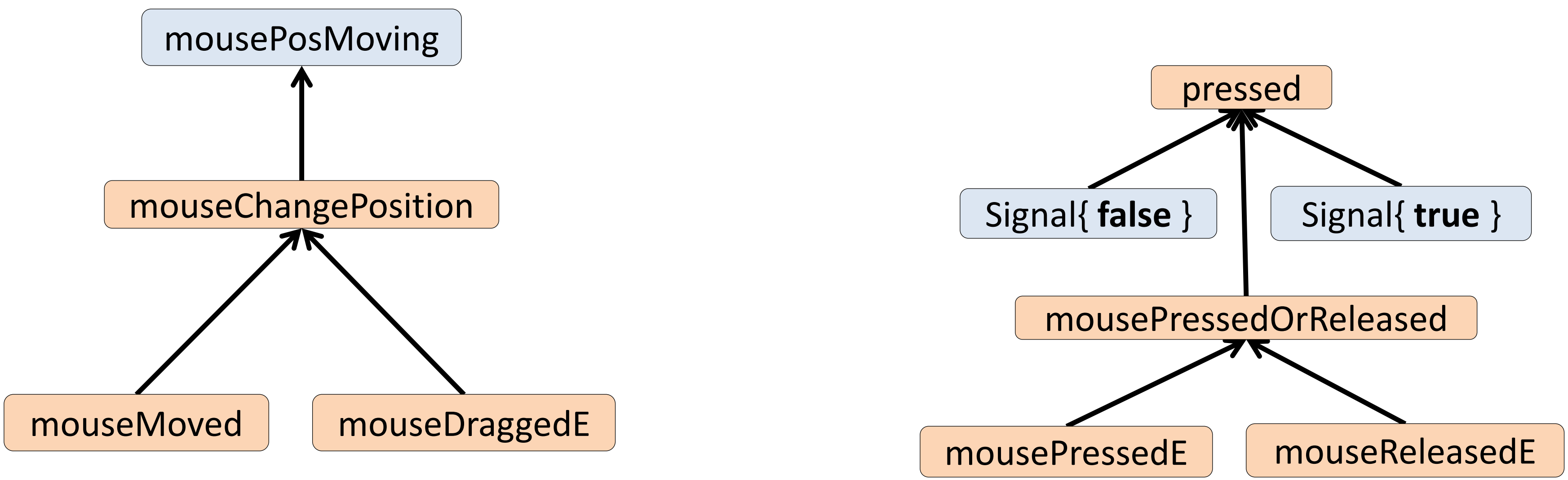
```
2.0
1.5
2.0
2.5
2.2
2.0
```

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 } )
```

Dependency Graph

```
/* 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)
(1,0,0,0)
(2,0,0,0)
…
(59,0,0,0)
(0,1,0,0)
(1,1,0,0)
(2,1,0,0)
…
(59,1,0,0)
(0,2,0,0)
(1,2,0,0)
…
(59,59,0,0)
(0,0,1,0)
…
(59,59,23,0)
(0,0,0,1)
….

Time Elapsing: First Attempt

```
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.get, minute.get, hour.get, day.get))
    tick.set(tick.get + 1)
  }
}
```

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

Also, conceptually hard to follow

Time Elapsing

Use

s.changedTo(v)

- Fires and event if s holds v

e.count

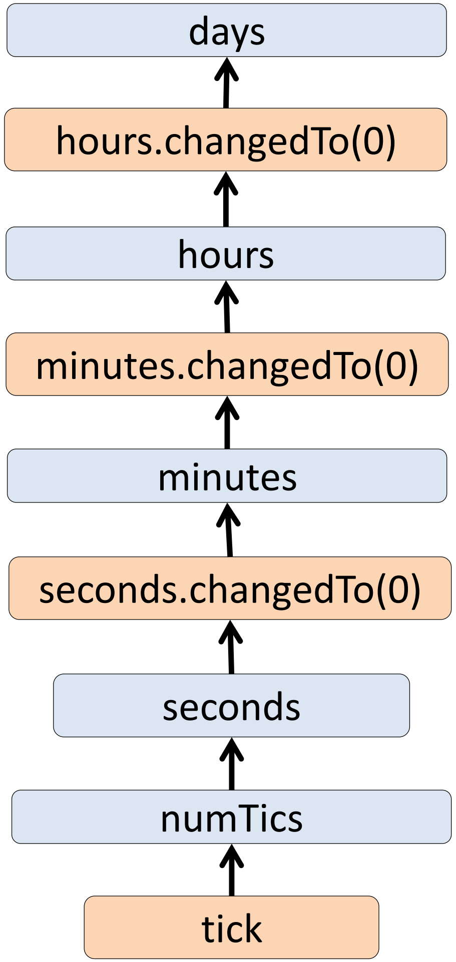
- Counts the occurrences of e

```
object AdvancedTimeElapsing extends App {
  println("start!")
  val tick = new ImperativeEvent[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

  while(true){
    Thread.sleep(0)
    println((seconds.get, minutes.get, hours.get, days.get))
    tick()
  }
}
```

Exercise: draw dependency graph

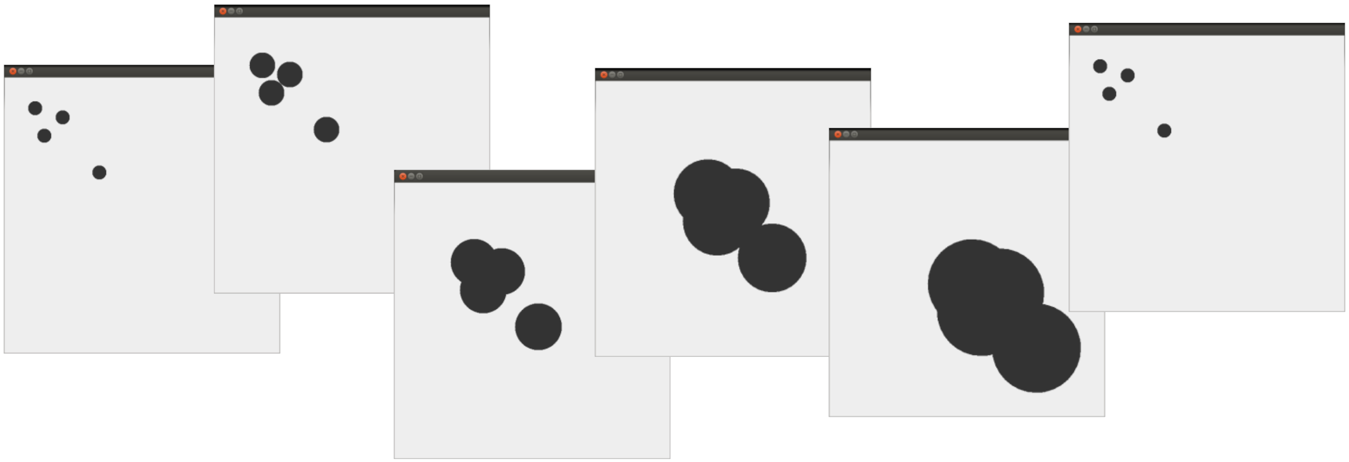
```
val tick = new ImperativeEvent[Unit]()
val numTics = tick.count
val seconds = Signal{ numTics() % 60 }
val minutes = Signal{ seconds.changedTo(0).count() % 60 }
val hours = Signal{ minutes.changedTo(0).count() % 24 }
val days = hours.changedTo(0).count
```

- Which variables are affected by a change to tick ?

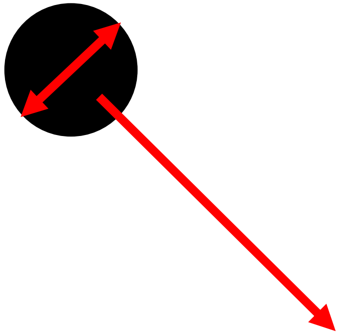


(Some nodes omitted)

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.get.x,center.get.y, radius.get, radius.get)})
 }

 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)
  …
```

```
override def main(args: Array[String]){
  while (true) {
     frame.repaint
     Thread sleep 20
     base()= base.get + 1
 }}
```

- Signals are used inside objects!

Training with RP - Resources

- Examples in the lecture slides
  - Observer
  - Reactive programming
- Homework assignments
- REScala examples (online, RP and OO version)
- REScala manual (online)

QUESTIONS?