

# An Introduction to Reactive Programming (2)

Guido Salvaneschi

Software Technology Group

# Outline

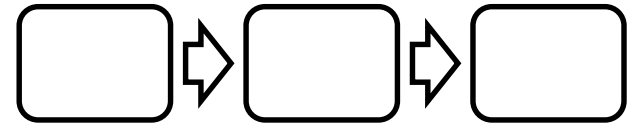
- Analysis of languages for reactive applications
- Details of reactive frameworks
- Advanced conversion functions
- Examples and exercises

# REACTIVE APPLICATIONS: ANALYSIS

# Software Taxonomy

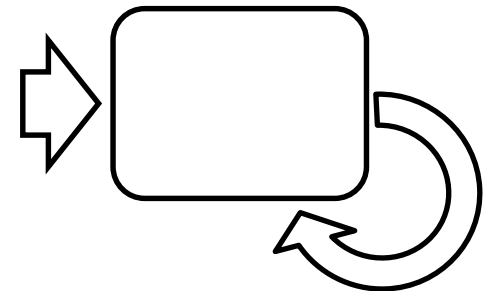
- A **transformational** systems

- 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
- Editors, Web apps, embedded software



# Use of State

- Transformational systems:
  - Express transformations as incremental modifications of the internal data structures
  - Represent the state of iterations in loops

Use of state is not essential

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

State is essential to describe the system

# How to implement Reactive Systems ?

- Observer Patter
  - The *traditional* way in OO languages
- Language-level events
  - In event-based languages
- Signals, vars, events and combinations of.
  - Reactive languages

# OBSERVER PATTERN: ANALYSIS

# 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 }  
}
```

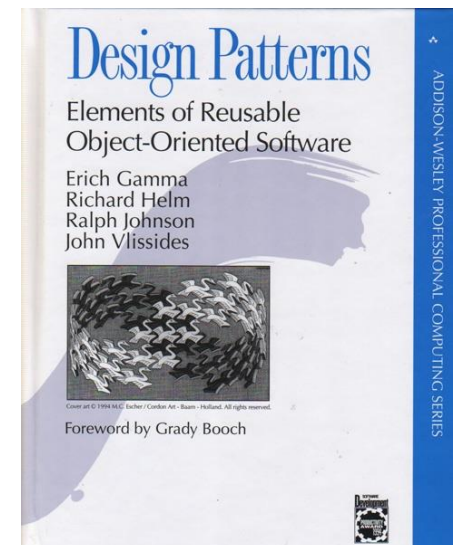
```
val s = new Sources()  
val c = new Constraint(s.a,s.b)  
s.registerObserver(c)  
s.a = 4  
s.notifyObservers(s.a,s.b)  
s.b = 8  
s.notifyObservers(s.a,s.b)
```

# 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”



# 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

```
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  
  }  
}
```

+

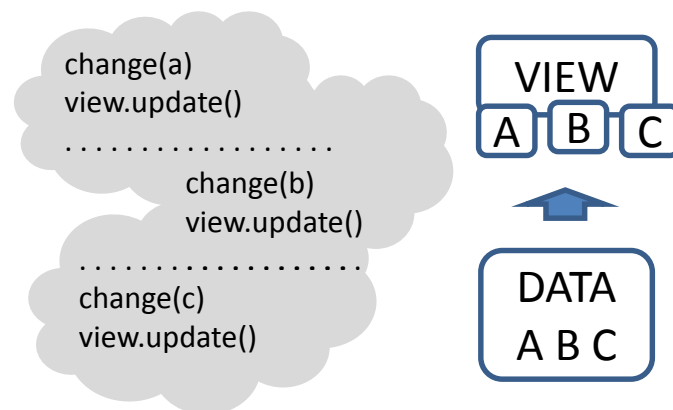
```
class Constraint2(d: Int) ... {  
  var d = c * 7  
  def notify(d: Int) = {  
    d = c * 7  
  }  
}
```

= ??

# 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, ||, &&, dropParam, 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
- 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() }
```

- An alternative to the Observer pattern and inversion of control

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

```

/* 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"
}

```

```

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

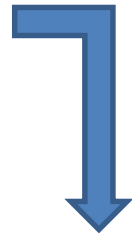
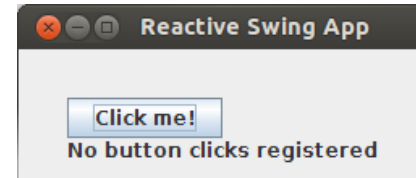
val nClicks = button.clicked.fold(0) {(x, _) => x + 1}

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
}

```



# 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...

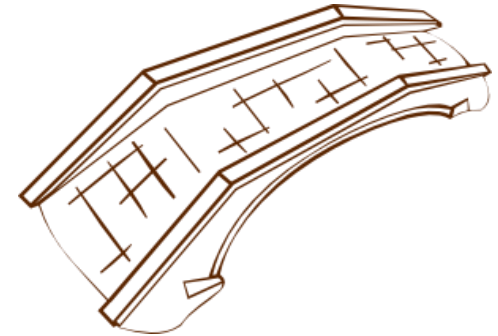
The sad story:

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



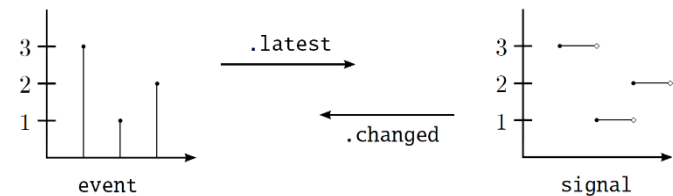
# Reactive Languages

- 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]

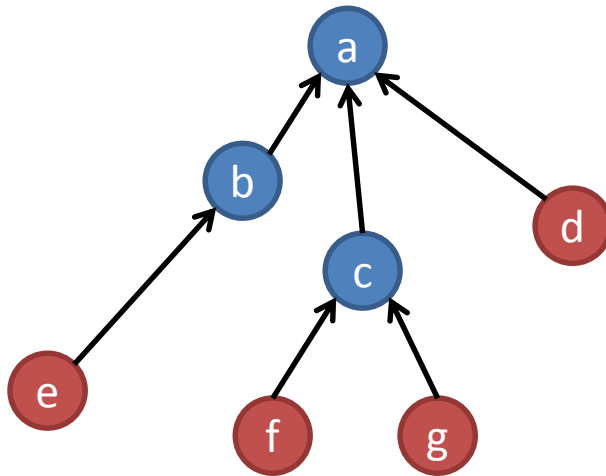
**Latest** :: Event[T] -> Signal[T]



# **DETAILS ON THE REACTIVE MODEL**

# Implementation

- Change propagation model
  - Topologically ordered dependency graph
  - Push-driven evaluation
  - Track dynamic dependencies



```
val e = Var(1)
val f = Var(2)
val g = Var(4)
val d = Var(true)
```

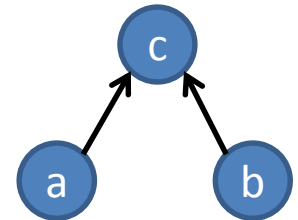
```
c = Signal { f() + g() }
b = Signal { e() * 100 }
a = Signal {
  if (d) c
  else b
}
```

# DSL implementation

Intuitively:

- `Var(3)` creates a leaf node
- `Var(4)` creates a leaf node
- The expression “`a() + b()`” is saved in a closure
  - To be evaluated later when a leaf changes
- `Signal{...}` creates another node
- The closure is evaluated
  - In the evaluation, the reactive values are detected.
  - The associated edges in the graph (i.e. the references from the leaves) are created
  - The result of the evaluation is assigned to the signal

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

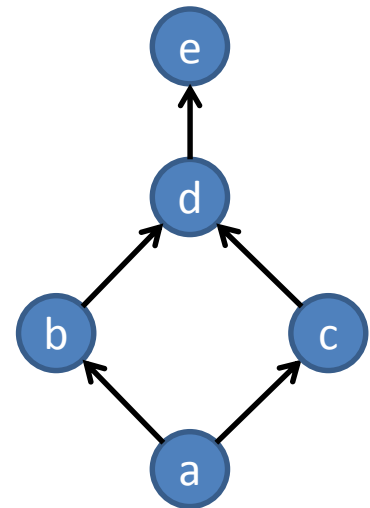


# Glitches

- Glitch: a temporary *spurious* value due to the propagation order.

- Consider the update order abdc
- $a()=2$      $b<-4$ ,  $d<-7$ ,  $c<-6$ ,  $d<-10$
- d is redundantly evaluated 2 times
- The first value of d has no meaning
- e is erroneously fired two times  
the first one with the spurious value

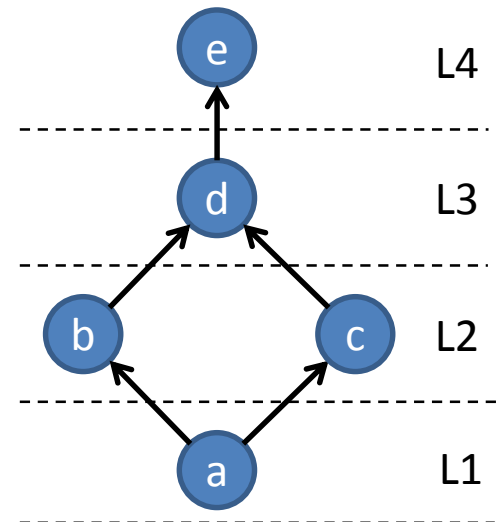
```
val a = Var(1)
val b = Signal{ a()*2 }
val c = Signal{ a()*3 }
val d = Signal{ b() + c() }
val e = d.changed
```



# Glitch Freedom

- Ensured by updates *in topological order*
  - Nodes are assigned to levels  $L_n$  based on their position in the graph
  - Levels are updates in order
- In this case “abcde” or “acbde”
- In practice, nodes to evaluate are in a priority queue ordered by level

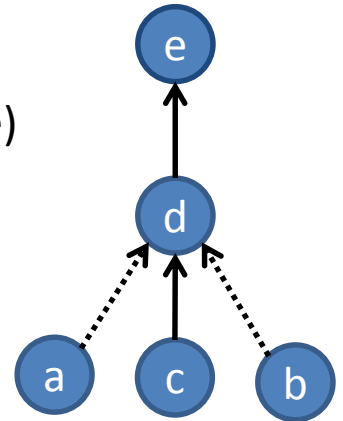
```
val a = Var(1)
val b = Signal{ a()*2 }
val c = Signal{ a()*3 }
val d = Signal{ b() + c() }
val e = d.changed
```



# Dynamic dependencies

- In some cases, dependencies are a consequence of a dynamic condition
  - When `c==true`, `d` must update if `a` changes but not if `b` changes
  - `d` depends on `a` or `b` based on the value of `c`
  - Reactive frameworks reroute the 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 * c() }
```

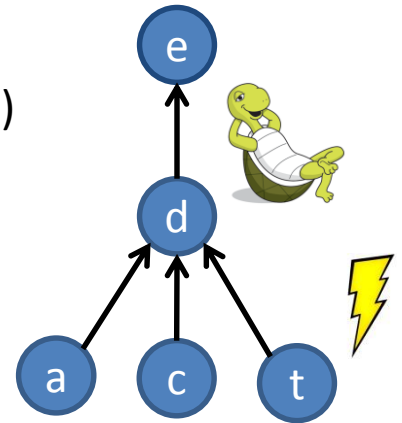




# Dynamic dependencies

- What happens if dynamic dependencies are fix ?
- Redundant evaluations
  - d is executed every time t is assigned even if the value of d does not change

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



# Loops: Options

- Reject loops
  - The programmer has the responsibility (REScala, Flapjax)
  - Loops are 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 }
```

# 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.getVal == 10)
e(1)
e(2)
assert(s.getVal == 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.getVal == None)
e(1)
assert(s.getVal == Option(1))
e(2)
assert(s.getVal == Option(2))
e(1)
assert(s.getVal == Option(1))
```

# Last

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

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

```
assert(s.getVal == List())  
e(1)  
assert(s.getVal == List(1))  
e(2)  
assert(s.getVal == List(2,1))
```

```
e(3);e(4);e(5)  
assert(s.getVal == List(5,4,3,2,1))  
e(6)  
assert(s.getVal == 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
  - F is similar to a handler
- `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(1)
```

```
assert(test == 10)
```

```
assert(s.getVal == 10)
```

```
e(2)
```

```
assert(test == 11)
```

```
assert(s.getVal == 10)
```

```
e(1)
```

```
assert(test == 12)
```

```
assert(s.getVal == 10)
```



# 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.getValue == 0)
e(1)
e(3)
assert(s.getValue == 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{ v1() + 1 }  
val s = e.snapshot(s1)
```

S !?

```
assert(s.getValue == 2)  
e(1)  
assert(s.getValue == 2)  
v.setVal(2)  
assert(s.getValue == 2)  
e(1)  
assert(s.getValue == 3)
```

# Change

- Similar to changed
  - `changed[U >: T]: Event[U]`
  - Provides both the old and the new value in a tuple
  - `change[U >: T]: 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 setVal 2
assert(test == 1)
v setVal 3
assert(test == 1)
```

test !?

# Reset

- Factory uses the event value to build the new signal.
  - Initially, the init value is used
  - Then the value of the event

`reset[T,A](e: Event[T], init: T)(factory: (T)=>Signal[A]): Signal[A]`

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

```
val v1 = Var(0)
```

```
val v2 = Var(10)
```

```
val s1 = Signal{ v1() + 1 }
```

```
val s2 = Signal{ v2() + 1 }
```

```
def factory(x: Int) = x%2 match {
```

```
  case 0 => s1
```

```
  case 1 => s2
```

```
}
```

```
val s3 = e.reset(100)(factory)
```

```
assert(s3.getVal == 1)
```

```
v1.setVal(1)
```

S3 !?

```
assert(s3.getVal == 2)
```

```
e(101)
```

```
assert(s3.getVal == 11)
```

```
v2.setVal(11)
```

```
assert(s3.getVal == 12)
```

# 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.getValue == 2)  
e(1)  
assert(s.getValue == 12)  
v2.setVal(12)  
assert(s.getValue == 13)  
v1.setVal(2)  
assert(s.getValue == 13)  
e(1)  
v1.setVal(3)  
assert(s.getValue == 4)  
v2.setVal(13)  
assert(s.getValue == 4)
```

# switchTo

- Switches the signal on the occurrence of the event e.
  - The result is signal depending on the value of e
  - 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.getVal == 2)
e(1)
assert(s2.getVal == 1)
e(100)
assert(s2.getVal == 100)
v.setVal(2)
assert(s2.getVal == 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.getVal == 1)
v1.setVal(1)
assert(s3.getVal == 2)
e(1)
assert(s3.getVal == 11)
e(2)
v2.setVal(11)
assert(s3.getVal == 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]
```

# 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 )
```

# Example: Interface Functions

- Keep the position of the last click in a signal

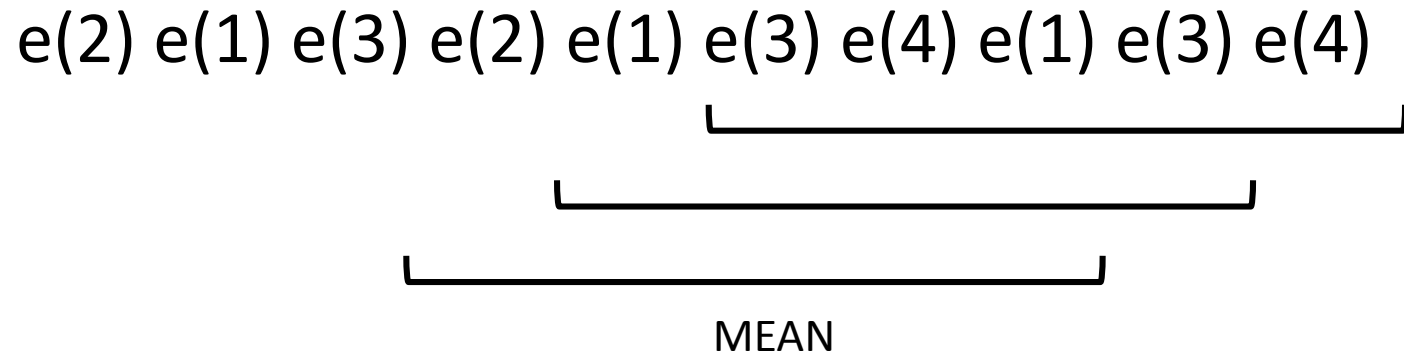
```
val clicked: Event[Unit] = mouse.clicked
val position: Signal[(Int,Int)] = mouse.position
val lastClick: Signal[(Int,Int)] = Signal{ lastClickPos() }
var lastClickPos = Var(0,0)
clicked += { _ =>
  lastClickPos()= position()
}
```

- 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(_)}
2.0
1.5
2.0
2.5
2.2
2.0
```

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

# 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

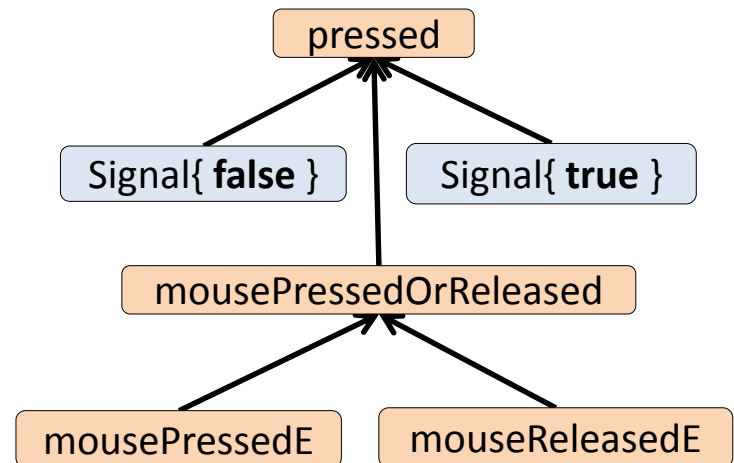
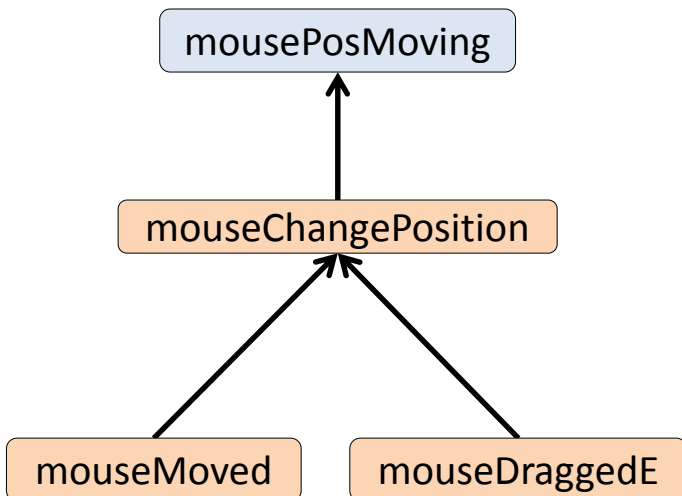
```
/* Compose reactive values */
```

```
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)	(1,2,0,0)
(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)	

# 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.getVal, minute.getVal, hour.getVal, day.getVal))
```

```
    tick.setVal(tick.getVal + 1)
```

```
  }
```

```
}
```

But day is still circular.

At some point day==0 again

Also, conceptually hard to follow

# Time Elapsing

```
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.getVal, minutes.getVal, hours.getVal, days.getVal))  
    tick()  
  }  
}
```

Use

s.changedTo(v)

- Fires and event if s holds v

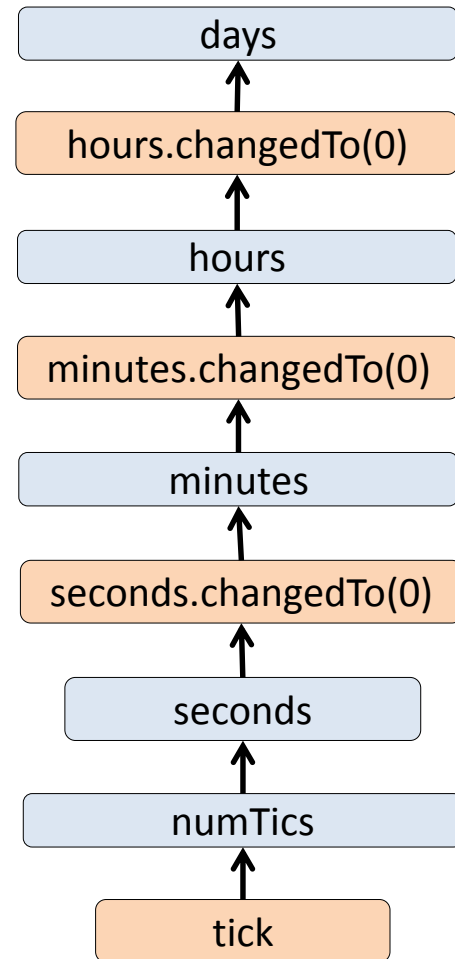
e.count

- Counts the occurrences of e

# 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
```

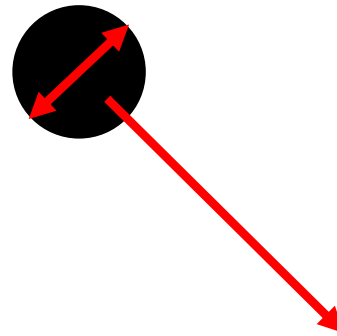
- Which variables are affected by a change to tick ?



# 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.getVal.x,center.getVal.y, radius.getVal, radius.getVal)})
}
```

```
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)
```

```
...
```

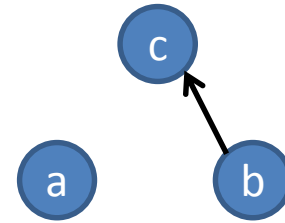
```
override def main(args: Array[String]){
  while (true) {
    frame.repaint
    Thread sleep 20
    base()= base.getVal + 1
  }}
}
```

- Signals are used inside objects!

# TRUBLESHOOTING

# Common pitfalls

- Establishing dependencies
  - () creates a dependency.  
Use only in signal expressions
  - getVal returns the current value




```
val a = Var(2)
val b = Var(3)
val c = Signal{ a.getVal + b() }
```

- Signals are **not** assignable.
  - Depend on other signals and vars
  - Are automatically updated




# Common pitfalls

- Avoid side effects in signal expressions

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

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

- Avoid cyclic dependencies

```
val a = Var(0)  WRONG!  
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.

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

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

S !?

```
class Foo(init: Int){  
  var x = init  
}  
val foo = new Foo(1)  
val varFoo = Var(foo)  
val s = Signal{  
  varFoo().x + 10  
}  
assert(s.getVal == 11)  
foo.x = 2  
varFoo().x = 2  
assert(s.getVal == 11)
```

```
class Foo(x: Int) //Immutable  
val foo = new Foo(1)  
val varFoo = Var(foo)  
val s = Signal{  
  varFoo().x + 10  
}  
assert(s.getVal == 11)  
varFoo().x = 2  
assert(s.getVal == 12)
```

# QUESTIONS?