Functional reactive Scala (Fresca) for 2D painting

Michel Ganguin

Under the supervision of Dr. S. McDirmid
Semester Project
Programming Methods Laboratory
EPFL
Professor M. Odersky

February 21, 2007

Contents

1 Introduction **2**

2 Background **2**
2.1 Signal concept 2
2.2 Examples of signals in scala 3
2.3 Examples of events in scala 4

3 Functional reactive core in Scala **4**
3.1 The Signal trait 4
3.2 The Constant class 5
3.3 The IOSignal class 5
3.4 The EventStream trait 5
3.5 The EventSource class 6
3.6 The Prelude object 6

4 Functional reactive API for swing **6**

5 Functional reactive API for Java2D **7**
5.1 Shape classes and related traits 7
5.1.1 Shape class 7
5.1.2 GlobalShape class - A special shape 7
5.1.3 String2D class 7
5.1.4 AWTShape class 8
5.1.5 Ellipse class 8
5.1.6 Rectangle class 8
5.1.7 RoundRectangle class 8
5.1.8 Line class 9
5.1.9 Color trait 9
5.1.10 FillColor trait 9
5.1.11 Transform trait and helpers 10
5.1.12 Stroke trait 10
5.2 Paint trait 10
5.2.1 PaintBoard class 11
5.3 Mouse trait 12
5.4 Time object 13
5.5 How to write extensions 13
5.5.1 Add a new shape 13
5.5.2 Add a new shape specialization 15

6 Conclusion **16**

7 Further work **16**

8 Appendix **17**
8.1 ClockDemo - An analog clock 17
8.2 PaintDemo - A pong Game 18

1 Introduction

Functional reactive programming (FRP) tries to abstracts values that changes over time by using signals, and makes binding an action or a function to a signal possible instead of providing an ”onChange” function. This makes source code more intuitive and concise. One domain that greatly benefit from FRPis GUI programming, since it is heavily based on events (mouse click, button press, ...) and timing (animation, mouse position, ...). In this work, I have focused on writing a functional reactive Scala API to Java2D [1].

2 Background

Two major programming languages have implemented FRP:
Haskell (Yampa [2],Fruit [3], Fran [4]) and Scheme (FrTime [5, 6]), as well as some ohters, such as Ocamlrt [7], a javascript implemention of FrTime: Flapjax [8], ...

2.1 Signal concept

There are two kinds of ”signals”, continous and discrete. A continous signal always has a value. On the other hand, a discrete signal, an event, has a value only when it occurs, or even no value at all. From now on, continous signals will be called ”signals” and discrete signals ”events”.

Generally speaking, a signal is a quantity. Some signals are derived from others.

Some examples of signals:

- time
- temperature
- position
- speed (derived from time and position)
- ...

An event is an abstract concept that combines two or more signals with a punctual constraint on exactly one of them.

Some examples of events:

- a motor’s temperature at $200km/h$ (temperature with speed constraint)
- your time of arrival at home (time with position constraint)
- your current location (position with time constraint)
- ...

Another thing about events: the probability that two of them occur at the same time is zero. There is no reason to combine events like signals, because this creates an event that never happens! It is possible to combine an event with signals to create another event, though. It occurs at the same time than

the initial event with a value that is a combination of its and all the signals' values.

The difference between signals and events isn't always obvious. A signal changing value, it can be seen as an event; when an event carries a value that is accessible as "value at last occurrence", it acts like a signal. Thus, it is probably always possible to switch from one to the other.

In this implementation, a signal is created with an initial value and its behaviour is like an event when the value changes. An event is created without an initial value, but with the kind (type) of it. The value isn't "held" after the event's occurrence. An event can mutate to a signal by holding its last occurence's value (the signal value is undefined before the event's first occurence).

2.2 Examples of signals in scala

To create a signal and to make the value change:

```
val aNumber = IOSignal(10)
aNumber.current = 15
```

This creates a signal that carries an Int value.

Now to use it: in scala it is not possible to use a signal as a function parameter (as parameters are evaluated before the function is). Otherwise, one could write:

```
Console.println(aNumber)
```

As expected, this does not work. aNumber is printed once, even if the carried Int changes. The new value is never printed. Instead, using the scala "for" statement, its desugarized form foreach, or the onChange method, it becomes possible:

```
for (val n <- aNumber) Console.println(n)
aNumber.foreach(n => Console.println(n))
aNumber.onChange(n => Console.println(n))
```

The function is passed as a parameter to the signal and executed every time the value changes. It is then possible to transform a signal:

```
val aNumberX2 = for (val n <- aNumber) yield 2*n
val aNumberX2 = aNumber.map(n => 2*n)

val aNumberIsOdd = for (val n <- aNumber) yield
                       if (n%2 == 0) true else false
```

Or, with another signal:

```
val sndNb = IOSignal(25)

val sum = for (val n1 <- aNumber; val n2 <- sndNb)
            yield n1 + n2
val sum = aNumber.flatmap(n1 =>
            sndNb.map(n2 => n1 + n2))
```

2.3 Examples of events in scala

To create an event and to change its value:

```
val intEvent = new EventSource[Int]
intEvent.fire(10) //event occurs with 10 as value
```

Like for signals (except flatmap as mentioned above about event's combination), to use the event:

```
for (val i <- intEvent) Console.println(i)
val intEventX2 = for (val i <-intEvent) yield 2*i
```

Or in a more readable way (but event's value is ignored):

```
on(mouse.clicked)(Console.println("clicked"))
```

The or (|) method can be used to "wait for" two different events:

```
val intEvent1 = new EventSource[Int]
val intEvent2 = new EventSource[Int]
for (val ev <- (intEvent1 | intEvent2) ) yield ev
```

3 Functional reactive core in Scala

3.1 The Signal trait

This is the trait that provides methods to manipulate signals:

sig.current Returns sig's current value.

sig.current(anObserver) Return sig's current value and bind an action to sig (it is simpler to use foreach instead)

sig.map(function) To apply a function to sig's value (only registers the function, no evaluation).

sig.flatMap(function) To apply a function to sig's values (only registers the function, no evaluation). To create a combination of two or more signals.

sig.foreach(function) To apply a function to sig's value and evaluate it.

sig.onChange(function) Is the same as foreach.

sig.redMap(initValue, function) To create a new signal that depends on its last value and on sig's value. (see PaintDemo's click count signal for example).

sig.trigger(eventStream) To create a new signal corresponding to sig but triggered on eventStream events.

sig.delayed To create a new signal which value is next to the last.

sig.retained To create a new signal equal to sig but augmented by the `previous` method that returns its previous value.

sig.clean To create a new signal, the change of which is triggered only when the new value differs from the old one (according to Scala's !=).

3.2 The Constant class

This is a very simple signal that never changes. Objects Constant and LazyConstant contains an apply method to instantiate a Constant. With Constant, the value is evaluated at creation time, whereas with LazyConstant it is evaluated at use. Actions associated with Constants are executed once.

Example:

```
// creates a new constant
val color1 = Constant(new java.awt.Color.RED)
// create a new lazy constant, something can be
// uninitialized at this time
val color2 = LazyConstant(something.getColor())
...
// but must be initialized now
color2 foreach (...)
```

3.3 The IOSignal class

This is a signal which value can change. Objects IOSignal and LazyIOSignal contains an apply method to instanziate an IOsignal. Like for Constant, the difference is when the initial value is evaluated. The mixed-in trait Source contains the method "current_=(value)" to change the signal's value.

Example (i is printed 3 times):

```
val sig = IOSignal(42)
sig foreach (i => Console.println("" + i))
sig.current = 54
sig.current = 23
```

3.4 The EventStream trait

This is the trait that provides methods to manipulate events:

ev.suscribe(eventObserver) To bind an action to ev (it is simpler to use foreach instead).

ev.map(function) To apply a function to the ev's value (only registers the function, no evaluation).

ev.foreach(function) To apply a function to ev's value and evaluate it.

ev.|(aSecondEventStream) To create a new EventStream that fires when any of both EventStream fires.

ev.hold To transform an EventStream into a Signal by holding last ev's value (value is Scala's _ before first occurrence).

3.5 The EventSource class

This class extends EventStream and is used to create a new event. It provides the fire method that executes every of the event's bound actions.

Example of two consecutive events of Int (i is printed twice):

```
val ev = new EventSource[Int]
ev.foreach(i => Console.println("" + i))
ev.fire(3)
ev.fire(5)
```

3.6 The Prelude object

In this object, some implicit methods are defined for Constant signals, for handling String, Int, or Boolean signals, as well as two methods to make events easier to use.

Constant implicit method to convert any value to a constant signal.

Int implicit binary operators (+, /, *, >, >=, <, <=, ==, !=) to create a new Int signal which value is "leftvalue op rightvalue".

String implicit binary operators + to create a new String signal which value is the concatenation of signals' values.

Boolean implicit binary operators (||,&&) to create a new signal which value is "leftvalue op rightvalue" and tertiary operator (boolSig ? aSig ! anotherSig) to create a new signal which value is the aSig's value if boolSig is true, the anotherSig's value otherwise.

on(ev)(action) To execute action when ev events occurs.

onceOn(ev)(action) To execute action only when ev event occurs for the first time.

4 Functional reactive API for swing

Ingo Maier has studied and coded this part; here, I will show some examples of how to use his API. There will not be many explanations, only what is needed to create a window in which we can paint.

To create an application:

```
object MyApp extends UIApplication {
  def initUI(): Frame = {
    val title: Signal[String] = ...
    val child: Widget = ...
    ...
    // new Frame with OperationOnClose
    new Frame(title, child) with ExitOnClose
  }
}
```

And an example of a Widget: TextField

```
val text: IOSignal[String] = ...
val tf = new TextField(text)
// signal text is updated when the content of
// the TextField is modified
```

In the next section there will be another Widget, PaintBoard, in which we can paint.

5 Functional reactive API for Java2D

5.1 Shape classes and related traits

5.1.1 Shape class

The Shape class is the superclass of all shapes that can be drawn with the Paint trait. It is a signal ,too: it extends Signal[Unit] with Source[Unit]. Unit, because there are subclasses with too many differences (ex: String2D vs. AWTShape). Direct subclasses have to be case classes (see how to write extensions for more details) and have to "trigger" a changement when a contained signal changes (**this**.current =**this**.current). In trait Paint, repaint is triggered by all the Shape instances.

```
case class AShape(sig:Signal[...]) extends Shape {
  sig foreach (s => current = current)
  ...
}
```

5.1.2 GlobalShape class - A special shape

This is not really a shape, it represents the entire component which is painted into. The Color trait specifies a background color (and erases all previously-drawn shapes!), thus it may have to be drawn first. The Transform trait specifies a transformation that is applied to all shapes drawn after this one. There may have more than one GlobalShapes with Transform, as transformations are concatenated. Own transformations of other shapes are concatenated, too.

To set the background color:

```
val gs = new GlobalShape with Color {
    val color = Constant(java.awt.Color.BLUE)
}
```

5.1.3 String2D class

String2D is a Shape defined by a text signal *str* and coordinates signals (x, y) that are the bottom left corner where the text has to be drawn. Graphic's method drawString is used to paint this shape.

5.1.4 AWTShape class

AWTShape is a wrapper class for java.awt.Shape signals. To be drawn, all shapes that inherit java.awt.Shape have to extend this class. Graphic's methods draw and fill are used to paint this shape.

5.1.5 Ellipse class

An ellipse defined by specified double coordinates and the size of the bounding rectangle. (x, y), w and h respectively represent the upper left coordinates, width and height of the ellipse. This is a subclass of AWTShape.

An ellipse that follows the mouse position:

```
val x = mouse.pos.map(p => p.getX())
val y = mouse.pos.map(p => p.getY())
val w = Constant(10.)
val h = Constant(20.)
val rr = new Ellipse(x, y, w, h) with FillColor{
    val fillColor = Constant(java.awt.Color.BLACK)
}
```

For example, the ball of the PaintDemo application is an Ellipse.

5.1.6 Rectangle class

A rectangle defined by specified double coordinates and size. (x, y), w and h respectively represent the upper left coordinates, width and height of the rectangle. This is a subclass of AWTShape.

A rectangle that follows the mouse position:

```
val x = mouse.pos.map(p => p.getX())
val y = mouse.pos.map(p => p.getY())
val w = Constant(10.)
val h = Constant(20.)
val rr = new Rectangle(x, y, w, h) with FillColor{
    val fillColor = Constant(java.awt.Color.BLACK)
}
```

5.1.7 RoundRectangle class

A rectangle with rounded corners defined by specified double coordinates, size and corner size. (x, y), w and h respectively represent the upper left coordinates, width and height of the rectangle. $arcw$, $arch$ represent width and height of the corners arcs. This is a subclass of AWTShape.

A rounded rectangle that follows the mouse position:

```
val x = mouse.pos.map(p => p.getX())
val y = mouse.pos.map(p => p.getY())
val w = Constant(10.)
val h = Constant(20.)
val arcw = Constant(3.)
val arch = Constant(3.)
```

```
val rr = new RoundRectangle(x, y, w, h, arcw, arch)
                                                with Color{
    val color = Constant(java.awt.Color.BLACK)
}
```

For example, the bar of the PaintDemo application is a RoundRectangle.

5.1.8 Line class

A line segment specified with double coordinates signals. $(x1, y1)$ and $(x2, y2)$ specify the start end end points of the segment. This is a subclass of AWTShape.

A line from component's origin to mouse position:

```
val x1 = mouse.pos.map(p => p.getX())
val y1 = mouse.pos.map(p => p.getY())
val x2 = Constant(0.)
val y2 = Constant(0.)
val line = new Line(x1, x2, y1, y2) with Color{
    val color = Constant(java.awt.Color.BLACK)
}
```

For example, the ClockDemo application uses Lines to draw a clock.

5.1.9 Color trait

To draw a shape (draw the outline of an AWTShape, draw a String2D, set the background color for a GlobalShape), use the Color trait. Color requires a Shape; with this trait mixed in, the shape will be drawn using color's signal value. An AWTShape or a GlobalShape that do not mix this trait in will not be drawn, whereas a String2D will be drawn black. If GlobalShape isn't the first shape drawn and it mixes Color in, everything drawn before will be erased.

```
new Shape with Color {
    val color = Constant(new java.awt.Color.RED)
}
```

5.1.10 FillColor trait

To fill a shape, use the FillColor trait. FillColor requires an AWTShape; with this trait mixed in, the shape will be filled (Graphic2D's method fill is called) with the specified color signal. A shape that does not mix this trait in will not be filled.

```
new Shape with FillColor {
    val fillColor = Constant(
                new java.awt.Color.RED)
}
```

5.1.11 Transform trait and helpers

To apply a geometric transformation to a shape, use the Transform trait. Transform requires a Shape; with this trait mixed in, the specified transformation matrix signal will be applied to the shape. So as not to interfere with other shapes, the inverted tranformation is applied once the shape is drawn, except for GlobalShape (can be used to transform an entire "scene").

```
new Shape with Transform {
    //identity matrix, useless, but concise example
    val matrix = Constant(
            new java.awt.geom.AffineTransform())
}
```

It is not very easy to handle transformation matrices directly, thus there are some helper methods in Transform object:

Transform.rotate(theta) Returns a matrix signal based on theta Double signal which represents a clockwise rotation of theta radians with the component origin as rotation's center.

Transform.rotate(theta, x, y) Same as above, but with x and y signals to specify the rotation's center.

Transform.scale(x, y) Returns a matrix signal based on x and y signals which represent scale factors along x and y axis.

Transform.shear(x, y) Returns a matrix signal based on x and y signals which represent multipliers by which coordinates are shifted as factors of the original xy coordinate

Transform.translate(x, y) Returns a matrix signal based on x and y signals which represent translation distances in x-axis and y-axis direction

For examples, see the ClockDemo application.

5.1.12 Stroke trait

To modify the default stroke of a shape's outline, use the Stroke trait. Stroke requires an AWTShape; with mixin this trait mixed in, the outline will be drawn with the specified stroke signal's value.

```
new Shape with Stroke {
    val stroke = Constant(
                new java.awt.BasicStroke(2.f))
}
```

5.2 Paint trait

The Paint trait can draw shape signals. As mentioned in previous subsection, the shapes has their own properties (color, fillcolor, ...) and are not affected by others (except in the special case of GlobalShape). The only specialty is the

order in which to draw the shapes. This trait takes care of redrawing (by calling repaint()) when a shape signal changes: no need to do this "by hand".

When mixing in the Paint trait, there are three things to do:

1. Tell Paint which JComponent to draw on, **val** par =theJComponent
2. Override the paint methods of this JComponent to call the frpaint method with the Graphics context
3. Tell the Paint trait which shapes to draw, **val** shapes =listOfShapes. They are drawn in sequence, head of list first.

```
new Widget with Paint {
  type Native = JComponent
  protected var _native = new JComponent() {
      override def paint(g: Graphics) = {
        // not mandatory but often usefull
        super.paint(g)
        frpaint(g)
      }
    }
  override protected def initNative = {
    //to be sure that _native is initialized
    //before calling super.initNative
    _native
    super.initNative
  }

  // specify on which element paint has to draw on
  val par = _native

...

  // specify which shapes paint has to draw
  val shapes = myShapeList
}
```

5.2.1 PaintBoard class

PaintBoard is a Widget that mixes Paint in. It is abstract; the only abstract value is the list of shapes to draw (val shapes).

Usage example:

```
new PaintBoard {
  // definition of the shapes
  ...

  val shapes = AboveDefinedShapeList
}
```

Shapes can, of course, be defined outside the class, but if they are dependent on something related to the PaintBoard, like its size or to the mouse (**new** PaintBoard **with** Mouse ...), they have to be within the class.

5.3 Mouse trait

Mouse is not really Java2D-related; I put it here because there often - always? - is a need to have mouse signals in Java2D applications.

As, in Java, a mouse is bound to a Component, Mouse requires an frp.swing.-UIElement (which contains the native java.awt.Component with the required addMouse*Listener methods). Once mixed in, field "mouse" becomes a new instance of Mouze.

```
val myUIElement = new MyUIElement with Mouse
myUIElement.mouse
```

Mouse button values in Mouse object:

```
Mouse.BUTTON1
Mouse.BUTTON2
Mouse.BUTTON3
Mouse.ANYBUTTON
```

Provided signals and events:

mouse.focus Boolean signal that determines if the UIElement has focus or not. It is button-independent. Initial value is false.

mouse.pos Signal that carries the current mouse position as a java.awt.Point. It is button-independent. Initial value is (0, 0).

mouse.dragpos Method that returns a signal carrying the current mouse position as a java.awt.Point while a mouse button is pressed. The optional argument specifies which one has to be pressed; without arguments, it is the same as with Mouse.ANYBUTTON. Initial value is (0, 0).

mouse.pressed Method that returns a Boolean signal determining whether a button is pressed or not. The optional argument specifies which one to observe; without arguments, it is the same as with Mouse.ANYBUTTON. Initial value is false.

mouse.clicked Method that returns an EventStream of java.awt.event.Mouse-Event which is fired when a button is clicked. The optional argument specifies which one to observe; without arguments, it is the same as with Mouse.ANYBUTTON.

mouse.wheel EventStream of Int's which is fired when the mouse wheel moves. The value determines the move's direction (negative for up and positive for down) and length (absolute value). It is button-independent.

Make sure that focus, pos and wheel are fields and the others are methods! Example: print something when button one is clicked

```
val mouse1click = mouse.clicked(Mouse.BUTTON1)
on(mouse1click)(
    Console.println("Button 1 is clicked"))
```

5.4 Time object

Like Mouse, Time is not really Java2D-related, but still useful.

This object provides two signals, milliseconds and seconds, that are triggered by a GUI safe (run in DispatchThread) timer.

Milliseconds carries a Long value representing the milliseconds elapsed since the Epoch (00:00:00 UTC, January 1, 1970). Seconds is derived from milliseconds.

To obtain these signals, simply do:

```
Time.milliseconds
Time.seconds
```

Example: creation of a boolean signal that changes every second (and which value tells whether seconds are odd or even)

```
Time.seconds map (s => (s%2) == 0 )
```

5.5 How to write extensions

5.5.1 Add a new shape

If the new shape is a java.awt.Shape it is easy. Create a new class that extends AWTShape and provide it with a retained shape signal (retained only because it is necessary the erase the previous shape when repainting it.).

Simplest possible one, with one signal:

```
class NewAWTShape(sst: Signal[SomeType]) extends
    AWTShape(sst.map(st =>
        new TheJavaShape(st)).retained)
```

A little bit more complicated example (this is this source code of Line class):

```
class Line(x1: Signal[Double], x2: Signal[Double],
           y1: Signal[Double], y2: Signal[Double])
    extends AWTShape((for (val xx1 <- x1;
                           val xx2 <- x2;
                           val yy1 <- y1;
                           val yy2 <- y2) yield
      new Line2D.Double(xx1,xx2,yy1,yy2)).retained)
```

If the new shape is not a java.awt.Shape, it is more complicated. Create a new case class that extends Shape, and for every signal of the new shape "update" the Shape signal.

```
case class String2D(str: Signal[String],
                    x: Signal[Double],
                    y: Signal[Double])
                            extends Shape {
  str foreach (s => current=current)
  x foreach (xx => current=current)
  y foreach (yy => current=current)
}
```

This was the simple part. Our shape is created and now we have to tell the Paint trait how to draw it. If we know the exact size of the shape we can use a retained signal to repaint only the shape; with String2D it becomes complicated as the size is dependent on the font and String str's length. To do this, add a new case to the match of the setShapes method in the Paint trait. Actual code of setShapes:

```
private def setShapes(ls: List[Shape]): Unit = {
  shapes foreach (sh => sh match {
    case s: AWTShape with Transform => sh
      foreach (s => par.repaint())
    case AWTShape(ash) =>

      // repaint triggered by sh and not ash
      // to be sure to take care about Color,
      // FillColor, Stroke,... modifications
      sh foreach (s => par.repaint( {
        val r = ash.previous.getBounds().union(
                        ash.current.getBounds())

        // need to grow by 1 pixel because of
        // rounding from Double to Int
        r.grow(1, 1)

        r
      }))
    case _ => sh foreach (s => par.repaint())
  })
}
```

And last thing in frpaint: (fragment of frpaint code)

```
case s:String2D =>
  //if string comes with a transformation, apply it
  if (s.isInstanceOf[String2D with Transform]) {
  val sr = s.asInstanceOf[String2D with Transform]
    g2.transform(sr.matrix.current)
  }

  // if String2D comes with a color signal use it
  // else use black as default color
  if (s.isInstanceOf[String2D with Color]) {
    val sc = s.asInstanceOf[String2D with Color]
    g2.setColor(sc.color.current)
  } else {
    g2.setColor(java.awt.Color.BLACK)
  }

  // draw the shape
  g2.drawString(s.str.current,
        s.x.current.asInstanceOf[Int],
        s.y.current.asInstanceOf[Int])

  // if there was a transformation unapply
```

```
  // it to not affect later drawing
  if (s.isInstanceOf[String2D with Transform]) {
    val sr =s.asInstanceOf[String2D with Transform]
    g2.transform(sr.matrix.current.createInverse())
  }
```

You have to test each specialization that can be applied to the shape, and if necessary unapply the modification once drawn (as for Transform in this example). It is done with instance checks and casts because specializations can be cummulated. With pattern matching, we should to have a number of cases which is the size of the powerset of the number of applicable specializations.

5.5.2 Add a new shape specialization

To add a new shape specialization, two things has to be done. The creation of the trait, and the modification of frpaint method in Paint trait to tell how to apply the specialization. First the trait, define the abstract val and "update" Shape when the val changes. Here is the code of the Color trait:

```
trait Color requires Shape {
  val color: Signal[java.awt.Color]

  color foreach (m => current = current)
}
```

But this does not work! Two problems comes up (Scala language limitations or errors. Used version is 2.3.3):

1. Compile time error: color does not exist!
   Workaround: create a dumb val (see code)
2. Runtime error: dumb val may be initialized before abstract val color
   Workaround: wait for color initialization

```
trait Color requires Shape {
  val color: Signal[java.awt.Color]

  // dumb val because without it color
  // is undefined (color not found)
  private val dumbc =
    //wait until color is initialized
    Time.milliseconds.current(new Observer {
      def changed: Unit =
        if (color == null)
          current(this)
        else
          color foreach (m =>
            Color.this.current=Color.this.current)
    })
}
```

Now that the specialization trait is OK, let's add the code into the frpaint method of the Paint trait. This must be must for every single shapes (in this case for GlobalShape, String2D and AWTShape):

```
shapes foreach (s => s match {
  case s: GlobalShape =>
    ...
    if (s.isInstanceOf[GlobalShape with Color]) {
      val sc=s.asInstanceOf[GlobalShape with Color]
      g2.setBackground(sc.color.current)
      import scala.compat.Math.MAX_INT
      g2.clearRect(0, 0, MAX_INT, MAX_INT)
    }
    ...
  case s:String2D =>
    ...
    if (s.isInstanceOf[String2D with Color]) {
      val sc = s.asInstanceOf[String2D with Color]
      g2.setColor(sc.color.current)
    } else {
      g2.setColor(java.awt.Color.BLACK)
    }
    ...
  case as: AWTShape =>
    ...
    if (as.isInstanceOf[AWTShape with Color]) {
      val asc=as.asInstanceOf[AWTShape with Color]
      g2.setColor(asc.color.current)
      g2.draw(asc.sh.current)
    }
    ...
})
```

As you can see there is not real duplication of code since color has a different meaning foreach case of Shape.

6 Conclusion

Functional reactive programming offers a powerful way to program event-based programs such as user interfaces or robots [9]. Object oriented programming with self types ("requires") offers a nice way to specialize objects.

In my opinion, programming GUIs becomes more funny! Side effects are minimized! Source code becomes shorter and easier to read!

7 Further work

API: complete the Java2D support. String2D does not yet support font, Shape does not yet support renderinghint, clip, composite, ... ; parametrize the API to use other drawing APIs or even 3D painting.

Fresca in general: write an API to use signals over the network; implement integrals and derivations of signals.

Write a lot of beautiful programs using Fresca!

8 Appendix

8.1 ClockDemo - An analog clock

```
object ClockDemo extends UIApplication{
  def initUI(): Frame = {
    val text = Constant("A Clock")

    val paintArea = new PaintBoard() with Mouse {
      native.setPreferredSize(new java.awt.Dimension(50, 50))

      val gs = new GlobalShape with Color {
        val color = Constant(java.awt.Color.LIGHT_GRAY)
      }

      val clock = new Ellipse(0., 0., 50., 50.) with Color {
        val color = Constant(java.awt.Color.BLACK)
      }

      //Time.seconds is GMT
      val hourtheta = Time.seconds.map(s =>
                        (2*Math.Pi/12*((s/3600+1)%12)))
      val hours = new Line(25., 25., 25., 15.) with Color
                                  with Transform with Stroke{
        val matrix = Transform.rotate(hourtheta, 25., 25.)
        val color = Constant(java.awt.Color.RED)
        val stroke = Constant(new java.awt.BasicStroke(2.f))
      }

      val minutetheta = Time.seconds.map(s =>
                          (2*Math.Pi/60*((s/60)%60)))
      val minutes = new Line(25., 25., 25., 5.) with Color
                                                with Transform {
        val matrix = Transform.rotate(minutetheta, 25., 25.)
        val color = Constant(java.awt.Color.BLACK)
      }

      val secondtheta = Time.seconds.map(s =>
                          2*Math.Pi/60*(s%60))
      val seconds = new Line(25., 25., 25., 5.) with Color
                                                with Transform {
        val matrix = Transform.rotate(secondtheta, 25., 25.)
        val color = Constant(java.awt.Color.BLACK)
      }

      //tell painter to draw these shapes (from head to tail)
      val shapes = gs :: clock :: hours ::
                    minutes :: seconds :: Nil
    }

    new Frame(text, paintArea) with ExitOnClose
  }
}
```

8.2 PaintDemo - A pong Game

This is the code of an one-player pong game that uses signals.

```
object PaintDemo extends UIApplication{

  def initUI(): Frame = {
    val text = IOSignal("Enjoy␣the␣game")
    val title = for (val s <- Time.seconds) yield
            "Time:␣" + s/3600%24 + ":" + s/60%60 + ":" + s%60

    val paintArea = new PaintBoard() with Mouse {
      native.setPreferredSize(new java.awt.Dimension(200,200))

      val mouse1pressed = mouse.pressed(Mouse.BUTTON1)

      // mouse pos as a string
      val textpos = mouse.pos.map(p =>
                      "(" + p.x + ",␣" + p.y + ")")

      // count distance parcoured with mouse in pixels
      val dist = mouse.pos.delayed.recMap(0., (p, d:Double) =>
                  d + p.distance(mouse.pos.current))

      // recMap example, counts clicks
      val click = mouse1pressed.recMap(0, (p, c:Int) =>
                                  if (p) c + 1 else c).clean

      // print the counted click on Console
      click foreach (d => Console.println("click␣count:␣"+d))

      // String Shape with rotation on click: distance
      // parcoured with mouse
      val sdist =
        new String2D(dist.map(d => "dist:␣"+d),10.,100.)
                                                  with Transform{
          val matrix =
            Transform.rotate(click.map(i =>
                          Math.Pi/10.+i/10.), 10., 100.)
      }

      // String Shape: text from TextField
      val stext = new String2D(text, 10., 40.)

      // String shape: mouse position (x,y)
      val smousepos = new String2D(textpos, 10., 60.)

      // 50 Hz as timerate (and refreshrate)
      val hz50 = Time.milliseconds.map(ms => ms/20).clean

      //bar
      //bar width
      val bar_w = 40.
      //bar height
      val bar_h = 5.
```

```
// pos of mouse when moved and when dragged
val dragandpos = IOSignal(mouse.pos.current)
mouse.pos foreach (p => dragandpos.current = p )
mouse.dragpos foreach (p => dragandpos.current = p )

//bar x position: follow mousepos but with min/max
//to be always visible
val bar_x = dragandpos.map(p => {
    val barcenter = bar_w / 2.
    val xmax = native.getSize().getWidth() - bar_w
    val x = p.getX() - barcenter
    Math.max(0., Math.min(xmax, x))
  })

//bar y position: 20 pixels from bottom of component
//FIXME should be dependent on component size signal
//(not yet implemented) instead of time signal
val bar_y = hz50.map(t =>
              native.getSize().getHeight() - 20.)

//bar: Round rect dependent of above values
//with color and fillcolor
val bar = new RoundRectangle(
  bar_x, bar_y, bar_w, bar_h, 2., 2.) with Color
                                      with FillColor{
  val fillColor = Constant(java.awt.Color.BLACK)
  val color = Constant(java.awt.Color.BLACK)
}

// pause signal false when component focused true else
val pause = mouse.focus map (f => !f)

// new boolean signal initial value game is starting
val gameover = IOSignal(false)

// gameover reset to false when "restart" is written
// in TextField
text foreach (t => if (t == "restart")
                     gameover.current = false)

// Ball
// ball direction as a signal with initial value pi/6
val dir = IOSignal(Math.Pi / 6.)

// increase speed by a factor 3 if a mousebutton
// is pressed (and hold)
val speed = mouse1pressed.map(p => if (p) 3 else 1)

// ball size vary between 5 and 15 pixel
//(initial 10px, 50 times +/- .1)
val ball_w = hz50.recMap(10., (s, i: Double) =>
  if (s%100<50) i + .1 else i - .1)
val ball_h = ball_w
```

```
// ball x and y coord, last pos + speed in dir
// direction, dir bounces if the ball touch an edge
// or the bar, gameover becomes true if y pos reaches
// bottom. Ball does move only when not gameover and
// not pause
val ball_x: Signal[Double] =
  hz50.recMap(10., (t, x: Double) => {
  if (gameover.current) 10.
  else if (pause.current) x
  else {
    val tmpx = x + speed.current * Math.cos(dir.current)
    val xmax =
      native.getSize().getWidth() - ball_w.current
    val newx = Math.max(0., Math.min(xmax, tmpx))
    if ( (newx == xmax) || (newx == 0.) )
      dir.current = (-dir.current + Math.Pi)%(2*Math.Pi)
    newx
  }
})
val ball_y: Signal[Double] =
  hz50.recMap(10., (t, y: Double) => {
  if (gameover.current) 10.
  else if (pause.current) y
  else {
    var newy = y + speed.current *Math.sin(dir.current)
    val ymax =
      native.getSize().getHeight() - ball_h.current
    newy = Math.max(0., Math.min(ymax, newy))
    if (newy == ymax) {
      text.current = "Game Over"
      gameover.current = true
    }
    if ( /*(newy == ymax) ||*/ (newy == 0.) )
      dir.current = (-dir.current) % (2*Math.Pi)
    if(bar.sh.current.contains(
        ball_x.current, newy + ball_h.current) ||
      bar.sh.current.contains(
        ball_x.current + ball_w.current,
        newy + ball_h.current)) {
      newy = bar_y.current - ball_h.current
      dir.current = (-dir.current) % (2*Math.Pi)
    }
    newy
  }
})

// create ellipse with above created signals
// with fillcolor dependent on mouse1 button status
val ball = new Ellipse(ball_x, ball_y, ball_w, ball_h)
                                          with FillColor{
  val fillColor = mouse1pressed map (p =>
        if (p) java.awt.Color.RED
        else java.awt.Color.GREEN)
```

```
      }

      val gs = new GlobalShape with Color {
        val color = Constant(java.awt.Color.LIGHT_GRAY)
      }

      //tell painter to draw these shapes (from head to tail)
      val shapes = gs :: sdist :: stext :: smousepos ::
                   bar :: ball :: Nil

    }

    // creates and returns the frame with time as title, with
    // exit as behaviour when frame is closed and that
    // contains the paintArea and the TextField
    new Frame(title, PageBox(paintArea, new TextField(text)))
                                                  with ExitOnClose
  }
}
```

References

[1] Sun Java 2D API. `http://java.sun.com/products/java-media/2d/`.

[2] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *PLDI 00*, 2000.

[3] Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Haskell Workshop 01*, 2001.

[4] Conal Elliott and Paul Hudak. Functional reactive animation. In *ICFP 97*, 1997.

[5] G. Cooper and S. Krishnamurthi. Frtime: Functional reactive programming in plt scheme.

[6] Gregory H. Cooper Daniel Ignatoff and Shriram Krishnamurthi. Crossing state lines: Adapting object-oriented frameworks to functional reactive languages. In *Functional and Logic Programming Symposium*, 2006.

[7] Christopher T. King. Objective caml reactive toolkit. `http://users.wpi.edu/~squirrel/ocamlrt/`.

[8] Brown PLT team. Flapjax - functional reactive ajax. `http://www.flapjax-lang.org/`.

[9] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *Summer School on Advanced Functional Programming 2002, Oxford University*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer-Verlag, 2003.