RESCala: "Animal" case study observations

Gerold Hintz

1 Advantages of Signals & Events (vs events-only / signal-only)

1.1 Main point: detection of changes

We need the combination of events and signals to model processes which depend on the change of a value.

Example: The germination of a plant is defined through a process of aging, growing, and reaching a maximum size. This can be expressed very concise with signals:

```
class Plant extends BoardElement {

  val age = world.time.hour.changed.iterate(0)(_ + 1)
  val grows = age.changed && { _ % Plant.GrowTime == 0}
  val size = grows.iterate(0)(acc => math.min(Plant.MaxSize, acc + 1))
  val expands = size.changedTo(Plant.MaxSize)

  expands += {_ =>
    germinate() // spawn a new plant in proximity to this one
  }
}
```

The equivalent event-based code has to do a manual check on every update:

```
class Plant extends BoardElement {

  var age = 0
  var size = 0
  val grows = new ImperativeEvent[Unit]
  val expands = grows && (_ => size == Plant.MaxSize)

  expands += {_ =>
    germinate() // spawn a new plant in proximity to this one
 }

 tickHandler = {_: Unit =>
    age += 1
    if(age % Plant.GrowTime == 0){
```

```
      val oldSize = size
      size = math.min(Plant.MaxSize, size + 1)
      if(size != oldSize)
        grows()
    }
  }
}
```

The equivalent signal-only code would have to perform a manual check for a value change as well. In the absence of events (in particular the `changed` event), the relationship would have to be defined as a pure functional dependancy, rather than through an explicit `grow` event. The code in the `tick` function has to do a lot of ugly manual checks. In addition, the reaction to a very specific condition (size has reached a maximum value), gets cluttered into the `tick` method, which should not have anything to do with that.

```
class Plant(override implicit val world: World) extends BoardElement {

  val age = Var(0)
  val size = Signal { math.min(Plant.MaxSize, age() / Plant.GrowTime) }

  def tick {
     // we have to store the old size now, otherwise we could not detect changes
    val oldSize = size.getValue

    age() = age.getValue + 1

    if(size.getValue != oldSize){ // did the value change
       if(size.getValue == Plant.MaxSize) // did the value reach MaxSize
        germinate() // spawn a new plant in proximity to this one
    }
  }
}
```

2 Shortcomings of Signal code

2.1 Handlers on late bound events

Sometimes we want to define an event on a signal which is late bound. This works (but we have to make the event lazy). However, we can not *register an event handler* on this event. When the object `Animal` gets instanciated, the signal `isDead` is still unbound.

```
abstract class Animal {

  val isDead: Signal[Boolean] // this value is abstract

  lazy val dies = isDead changedTo true // we can do this
  dies += {_ => world.board.clear(position.getValue)} // we can not do this
}

class Carnivore extends Animal {
  isDead = Signal { energy() > 10} // subclass substitutes concrete signal
```

```
}
```

2.2 Overriding signal values

We can override any signal value in a subclass. However, as a signal is a `val` member, we can not access the super-value. Consider the class Animal with member energyDrain. In the subtype Female, we want to override this value, by multipliying this signal with a given factor. However

```
abstract class Animal {
    val energyDrain = Signal {...}
}

trait Female extends Animal {
  val isPregnant = Signal {...}
  val factor = Signal { if(isPregant()) 1.2 else 1 } // multiply energy drain by 1.2 if
      pregnant.
  override val energyDrain = Signal { super.energyDrain() * factor() } // problem:
      super can not be used on val members!
 }
```