Functional Reactive Programming

- Reactive programming is about reacting to sequences of events that happen in time
- Functional view: Aggregate an event sequence into a signal
    - A signal is a value that changes over time
    - It is represented as a function from time domain to the value domain
    - Instead of propagating updates to mutable state one by one, we define new signals in terms of existing ones
- **Example**: Mouse Positions
    - **Event-based View**: Whenever the mouse moves, an event ***MouseMoved(toPos: Position)*** is fired.
    - **FRP View**: A signal, ***mousePosition: Signal[Position]*** which at any point in time represents the current mouse position

Fundamental Signal Operations

- There are two fundamental operations over signals:
    1. Obtain the value of the signal at the current time. In our library this is expressed by () application:

        mousePosition() *// the current mouse position*
    2. Define a signal in terms of other signals. In our library, this is expressed by the Signal constructor:

```
def inRectangle(LL: Position, UR: Position): Signal[Boolean] =
  Signal {
    val pos = mousePosition()
     LL <= pos && pos <= UR
  }
// checks whether the current mouse position is in the rectangle
// provided by its lower left corner and upper right corner
```

Constant Signals

- Signal Syntax can define a signal that has no dependencies and always defines the same value:

```
val sig = Signal(3)      // the signal that is always 3
```

Variable Signals

- Values of type Signal are immutable
- But our library also defines a subclass Var of Signal for signals that can be changed
- Var provides an “update” operation, which allows to redefine the value of a signal from the current time on:

```
val sig = Var(3)
sig.update(5)  // from now on, sig returns 5 instead of 3
```

Update Syntax

- In Scala, calls to update can be written as assignments using some syntactic sugar. For instance, for an array arr: ***arr(i) = 0*** is translated to ***arr.update(i, 0)*** which calls an update method which has the following signature:

  def update(idx: Int, value: T): Unit

  where T is the type parameter of the class defined for the array: Array[T]
- Generally, an indexed assignment like $f(E_1, \ldots E_n) = E$ is translated to:

  $f.update(E_1, \ldots E_n, E)$
- This works also if n = 0: ***f() = E*** is shorthand for ***f.update(E)***. Hence, ***sig.update(5)*** can be abbreviated to ***sig() = 5***

Signals and Variables

- Signals of type Var look a bit like mutable variables, where ***sig()*** is dereferencing and ***sig() = newValue*** is update.
- But there is a crucial difference: We can map over signals, which gives us a relation between two signals that is maintained automatically at all future points in time. No such mechanism exits for mutable variables; we have to propagate all updates manually
- **Example**: Repeat the BankAccount example of last section with signals. Add a signal balance to BankAccounts. Define a function consolidated which produces the sum of all balances of a given list of accounts. What savings were possible compared to the publish/subscribe implementation?

```
class BankAccount {
  val balance = Var(0)
  def deposit(amount: Int): Unit =
    if (amount > 0) {
      val b = balance()
      // without this constant definition, the system throws a cyclic
      // definition error - balance is a function over time
      balance() = b + amount
    }
  def withdraw(amount: Int): Unit =
    if (0 < amount && amount <= balance()) {
      val b = balance()
      balance() = b – amount
    }
    else
      throw new Error("insufficient funds")
}

object accounts {
  def consolidated(accts: List[BankAccount]): Signal[Int] =
    Signal(accts.map(_.balance()).sum)
}
```

The solution with signals is much shorter than the one with publish/subscribe one.

- Note that there's an important difference between the variable assignment v = v + 1 and the signal update s() = s() + 1. In the first case, the new value of v becomes the _old + value of v plus 1. In the second case, we try define a signal s to be at all points in time one larger than itself. This obviously makes no sense.

Exercise: *Consider the two code fragments below:*

1. ***val num = Var(1)***
   val twice = Signal(num() * 2)
   num() = 2
2. ***var num = Var(1)***
   val twice = Signal(num() * 2)
   num = Var(2)

Do they yield the same final value for twice()? – ***No***

In the first, num is a constant signal, so twice will modify when num is modified, so it will be 4. In the second, the num modification will create a new signal that will not be bound to twice, so it will remain unmodified, so it will be 2.