Functional Reactive Programming

Principles of Functional Programming

Martin Odersky

What is FRP?

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 to the value domain.
- Instead of propagating updates to mutable state, 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.

Origins of FRP

FRP started in 1997 with the paper *Functional Reactive Animation* by Conal Elliott and Paul Hudak and the *Fran* library.

There have been many FRP systems since, both standalone languages and embedded libraries.

Some examples are: *Flapjax*, *Elm*, *React.js*

Event streaming dataflow programming systems such as Rx are related but the term FRP is not commonly used for them.

We will introduce FRP by means of a minimal class, `frp.Signal` whose implementation is explained at the end of this module.

`frp.Signal` is modelled after Scala.react, which is described in the paper *Deprecating the Observer Pattern*.

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
```

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
     }
   ```

Signal Interpretation

Compare

```
def inRectangle(LL: Position, UR: Position): Boolean =
    val pos = mousePosition()
    LL <= pos && pos <= UR
```

with

```
def inRectangle(LL: Position, UR: Position): Signal[Boolean] =
  Signal {
    val pos = mousePosition()
    LL <= pos && pos <= UR
  }
```

`inRectangle` creates a signal that, *at any point in time* is equal to the test whether `mousePosition` *at that point in time* is in the box `[LL..UR]`.

Constant Signals

The `Signal(...)` syntax can also be used to define a signal that has always the same value:

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

Computing Signals

The idea of FRP is quite general. It does not prescribe whether signals are continuous or discrete and how a signal is evaluated.

There are several possibilities:

1. A signal could be evaluated on demand, every time its value is needed.
2. A continuous signal could be sampled at certain points and interpolated in-between.
3. Updates to a discrete signal could be propagated automatically to dependent signals.

The last possibility is the functional analogue to the observer pattern.

We will pursue this one in the rest of this unit.

Time-Varying Signals

How do we define a signal that varies in time?

- We can use externally defined signals, such as `mousePosition` and map over them.
- Or we can use a `Signal.Var`.

Variable Signals

Expressions of type `Signal` cannot be updated.

But our library also defines a subclass `Signal.Var` of `Signal` for signals that can be changed.

`Signal.Var` provides an "update" operation, which allows to redefine the value of a signal from the current time on.

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

Aside: Update Syntax

In Scala, calls to `update` can be written as assignments.

For instance, for an array `arr`

```
arr(i) = 0
```

is translated to

```
arr.update(i, 0)
```

This calls an `update` method which can be thought of as follows:

```
class Array[T]:
  def update(idx: Int, value: T): Unit
  ...
```

Aside: Update Syntax

Generally, an indexed assignment like $\texttt{f}(\texttt{E}_1, \ldots, \texttt{E}_\texttt{n}) = \texttt{E}$

is translated to $\texttt{f.update}(\texttt{E}_1, \ldots, \texttt{E}_\texttt{n}, \texttt{E})$.

This works also if $n = 0$: $\texttt{f}() = \texttt{E}$ is shorthand for $\texttt{f.update}(\texttt{E})$.

Hence,

```
sig.update(5)
```

can be abbreviated to

```
sig() = 5
```

Signals and Variables

Signals of type `Signal.Var` look a bit like mutable variables, where

```
sig()
```

is dereferencing, and

```
sig() = newValue
```

is update.

But there's a crucial difference:

We can apply functions to signals, which gives us a relation between two signals that is maintained automatically, at all future points in time.

No such mechanism exists for mutable variables; we have to propagate all updates manually.

Example

Repeat the `BankAccount` example of the 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?

Signals and Variables (2)

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

Do they yield the same final value for `twice()`?

O yes
O no

Exercise

Consider the two code fragments below

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

Do they yield the same final value for `twice()`?

O yes
X no