Imperative Event Handling: The Observer Pattern

Principles of Functional Programming

Martin Odersky

The Observer Pattern

The Observer Pattern is widely used when views need to react to changes in a model.

Variants of it are also called

- publish/subscribe
- model/view/controller (MVC).

Publisher and Subscriber Traits

```
trait Subscriber:
  def handler(pub: Publisher): Unit

trait Publisher:
  private var subscribers: Set[Subscriber] = Set()

  def subscribe(subscriber: Subscriber): Unit =
    subscribers += subscriber

  def unsubscribe(subscriber: Subscriber): Unit =
    subscribers -= subscriber

  def publish(): Unit =
    subscribers.foreach(_.handler(this))
end Publisher
```

Observing Bank Accounts

Let's make `BankAccount` a `Publisher`:

```
class BankAccount extends Publisher:
  private var balance = 0

  def deposit(amount: Int): Unit =
    if amount > 0 then
      balance = balance + amount

  def withdraw(amount: Int): Unit =
    if 0 < amount && amount <= balance then
      balance = balance - amount

    else throw Error("insufficient funds")
```

Observing Bank Accounts

Let's make `BankAccount` a `Publisher`:

```
class BankAccount extends Publisher:
  private var balance = 0
  def currentBalance: Int = balance        // <---
  def deposit(amount: Int): Unit =
    if amount > 0 then
      balance = balance + amount
      publish()                            // <---
  def withdraw(amount: Int): Unit =
    if 0 < amount && amount <= balance then
      balance = balance - amount
      publish()                            // <---
    else throw Error("insufficient funds")
```

An Observer

A `Subscriber` to maintain the total balance of a list of accounts:

```
class Consolidator(observed: List[BankAccount]) extends Subscriber:
  observed.foreach(_.subscribe(this))

  private var total: Int = _
  compute()                // total is assigned in ‘compute()‘

  private def compute() =
    total = observed.map(_.currentBalance).sum

  def handler(pub: Publisher) = compute()
  def totalBalance = total
end Consolidator
```

Observer Pattern, The Good

- Decouples views from state
- Allows to have a varying number of views of a given state
- Simple to set up

Observer Pattern, The Bad

- Forces imperative style, since handlers are `Unit`-typed
- Many moving parts that need to be co-ordinated
- Concurrency makes things more complicated
- Views are still tightly bound to one state; view update happens immediately.

To quantify (Adobe presentation from 2008):

- $1/3^{rd}$ of the code in Adobe's desktop applications is devoted to event handling.
- $1/2$ of the bugs are found in this code.

How to Improve?

During the rest of this session we will explore a different way, namely *functional reactive programming*, in which we can improve on the imperative view of reactive programming embodied in the observer pattern.

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 $\texttt{f}(\texttt{E}_1, \ldots, \texttt{E}_n) = \texttt{E}$

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

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()`?

0 yes
0 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

A Simple FRP Implementation

Principles of Functional Programming

Martin Odersky

A Simple FRP Implementation

We now develop a simple implementation of `Signal`s and `Var`s, which together make up the basis of our approach to functional reactive programming.

The classes are assumed to be in a package `frp`.

Their user-facing APIs are summarized in the next slides.

Summary: The Signal and Var APIs

```
trait Signal[T+]:
  def apply(): T = ???

object Signal:
  def apply[T](expr: => T) = ???

  class Var[T](expr: => T) extends Signal[T]:
    def update(expr: => T): Unit = ???
```

Note that `Signal`s are covariant, but `Var`s are not.

Implementation Idea

Each signal maintains

- its current value,
- the current expression that defines the signal value,
- a set of *observers*: the other signals that depend on its value.

Then, if the signal changes, all observers need to be re-evaluated.

Dependency Maintenance

How do we record dependencies in `observers`?

- When evaluating a signal-valued expression, need to know which signal `caller` gets defined or updated by the expression.
- If we know that, then executing a `sig()` means adding `caller` to the `observers` of `sig`.
- When signal `sig`'s value changes, all previously observing signals are re-evaluated and the set `sig.observers` is cleared.
- Re-evaluation will re-enter a calling signal `caller` in `sig.observers`, as long as `caller`'s value still depends on `sig`.
- For the moment, let's assume that `caller` is provided "magically"; we will discuss later how to make that happen.

Implementation of Signals

The `Signal` trait is implemented by a class `AbstractSignal` in the `Signal` object. This is a useful and common implementation technique that allows us to hide global implementation details in the enclosing object.

```
opaque type Observer = AbstractSignal[?]
def caller: Observer = ???  // Magic, for now

abstract class AbstractSignal[+T] extends Signal[T]:
  private var currentValue: T = _
  private var observers: Set[Observer] = Set()

  def apply(): T =
    observers += caller
    currentValue

  protected def eval: () => T   // evaluate the signal
```

(Re-)Evaluating Signal Values

A signal value is evaluated using `computeValue()`

- on initialization,
- when an observed signal changes its value.

Here is its implementation:

```
protected def computeValue(): Unit =
  val newValue = eval()
  val observeChange = observers.nonEmpty && newValue != currentValue
  currentValue = newValue
  if observeChange then
    val obs = observers
    observers = Set()
    obs.foreach(_.computeValue())
```

Creating Signals

Signals are created with an `apply` method in the `Signal` object

```
object Signal:
  def apply[T](expr: => T): Signal[T] =
    new AbstractSignal[T]:
      val eval = () => expr
      computeValue()
```

Signal Variables

The `Signal` object also defines a class for variable signals with an `update` method

```
class Var[T](initExpr: => T) extends AbstractSignal[T]:
  protected var eval = () => initExpr
  computeValue()

  def update(newExpr: => T): Unit =
    eval = () => newExpr
    computeValue()
end Var
```

Who's Calling?

How do we find out on whose behalf a signal expression is evaluated?

The most robust way of solving this is to pass the caller along to every expression that is evaluated.

So instead of having a by-name parameter

```
expr: => T
```

we'd have a function

```
expr: Observer => T
```

and when evaluating a signal, `s()` becomes `s(caller)`

Problem: This causes a lot of boilerplate code, and it's easy to get wrong!

Calling, Implicitly

How about we make signal evaluation expressions implicit function types?

```
expr: Observer ?=> T
```

Then all caller parameters are passed implicitly.

In the following we will use the type alias

```
type Observed[T] = Observer ?=> T
```

New Signal and Var APIs

```
trait Signal[T+]:
  /** The current value of this signal */
  def apply: Signal.Observed[T]

object Signal:
  /** Create a signal that evaluates using ‘expr‘ */
  def apply[T](expr: Observed[T]): Signal[T] = ???

  class Var[T](expr: Observed[T]) extends AbstractSignal[T]
    /** Update the signal to use new expression ‘expr‘ from now on */
    def update(expr: Observed[T]): Unit = ???
```

Evaluating Signals from the Outside

At the root, it's the application that evaluates a signal.

So there's no other signal that is the "caller".

To deal with this situation, we define a special given instance `noObserver` in `Signal`.

```
given noObserver: Observer = new AbstractSignal[Nothing] with
  override def eval = ???
  override def computeValue() = ()
```

Since `noObserver` is the companion object of `Signal` it's always applicable when an `Observer` is needed.

But inside `Signal {...}` expressions, it's the implicitly provided observer that takes precedence since it is in the lexically enclosing scope.

Summary

We have given a quick tour of functional reactive programming, with some usage examples and an implementation.

This is just a taster, there's much more to be discovered.

In particular, we only covered one particular style of FRP: Discrete signals changed by events.

Some variants of FRP also treat continuous signals.

Values in these systems are often computed by sampling instead of event propagation.