



# A Simple FRP Implementation

Principles of Functional Programming

Martin Odersky

## A Simple FRP Implementation

We now develop a simple implementation of Signals and Vars, 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 Signals are covariant, but Vars 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 as Observer = new AbstractSignal[Nothing]:  
  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.