Deprecating the Observer Pattern

EPFL-REPORT-148043

Ingo Maier Tiark Rompf Martin Odersky

EPFL

{firstname}.{lastname}@epfl.ch

Abstract

Programming interactive systems by means of the observer pattern is hard and error-prone yet is still the implementation standard in many production environments. We present an approach to gradually deprecate observers in favor of reactive programming abstractions. Several library layers help programmers to smoothly migrate existing code from callbacks to a more declarative programming model. Our central high-level API layer embeds an extensible higher-order data-flow DSL into our host language. This embedding is enabled by a continuation passing style transformation.

General Terms Design, Languages

Keywords data-flow language, reactive programming, user interface programming, Scala

1. Introduction

Over the past decades, we have seen a continuously increasing demand in interactive applications, fueled by an ever growing number of non-expert computer users and increasingly multimedia capable hardware. In contrast to traditional batch mode programs, interactive applications require a considerable amount of engineering to deal with continuous user input and output. Yet, our programming models for user interfaces and other kinds of continuous state interactions have not changed much. The predominant approach to deal with state changes in production software is still the observer pattern [25]. We hence have to ask: is it actually worth bothering?

For an answer on the status quo in production systems, we quote an Adobe presentation from 2008 [43]:

- 1/3 of the code in Adobe's desktop applications is devoted to event handling logic
- 1/2 of the bugs reported during a product cycle exist in this code

Our thesis is that these numbers are bad for two reasons. First, we claim that we can reduce event handling code by at least a factor of 3 once we replace publishers and observers with more appropriate abstractions. Second, the same abstractions should help us to reduce the bug ratio in user interface code to bring it at least on par with the rest of the application code. In fact, we believe that event handling code on average should be one of the least error-prone parts of an application.

But we need to be careful when we are talking about event handling code or logic. With these terms, we actually mean code that deals with a variety of related concepts such as continuous data synchronization, reacting to user actions, programming with futures and promises [33] and any blend of these. Event handling is merely a common *means* to implement those matters, and the usual abstractions that are employed in event handling code are callbacks such as in the observer pattern.

To illustrate the precise problems of the observer pattern, we start with a simple and ubiquitous example: mouse dragging. The following example traces the movements of the mouse during a drag operation in a path object and displays it on the screen. To keep things simple, we use Scala closures as observers.

```
var path: Path = null
val moveObserver = { (event: MouseEvent) =>
  path.lineTo(event.position)
  draw(path)
}
control.addMouseDownObserver { event =>
  path = new Path(event.position)
  control.addMouseMoveObserver(moveObserver)
}

control.addMouseUpObserver { event =>
  control.removeMouseMoveObserver(moveObserver)
  path.close()
  draw(path)
}
```

The above example, and as we will argue the observer pattern as defined in [25] in general, violates an impressive line-up of important software engineering principles:

Side-effects Observers promote side-effects. Since observers are stateless, we often need several of them to simulate a state machine as in the drag example. We have to save the state where it is accessible to all involved observers such as in the variable `path` above.

Encapsulation As the state variable `path` escapes the scope of the observers, the observer pattern breaks encapsulation.

Composability Multiple observers form a loose collection of objects that deal with a single concern (or multiple, see next point). Since multiple observers are installed at different points at different times, we can't, for instance, easily dispose them altogether.

Separation of concerns The above observers not only trace the mouse path but also call a drawing command, or more generally, include two different concerns in the same code location. It is often preferable to separate the concerns of constructing the path and displaying it, e.g., as in the model-view-controller (MVC) [30] pattern.

Scalablity We could achieve a separation of concerns in our example by creating a class for paths that itself publishes events when the path changes. Unfortunately, there is no guarantee for data consistency in the observer pattern. Let us suppose we would create another event publishing object that depends on changes in our original path, e.g., a rectangle that represents the bounds of our path. Also consider an observer listening to changes in both the path and its bounds in order to draw a framed path. This observer would manually need to determine whether the bounds are already updated and, if not, defer the drawing operation. Otherwise the user could observe a frame on the screen that has the wrong size (a *glitch*).

Uniformity Different methods to install different observers decrease code uniformity.

Abstraction There is a low level of abstraction in the example. It relies on a heavyweight interface of a control class that provides more than just specific methods to install mouse event observers. Therefore, we cannot abstract over the precise event sources. For instance, we could let the user abort a drag operation by hitting the escape key or use a different pointer device such as a touch screen or graphics tablet.

Resource management An observer's life-time needs to be managed by clients. Because of performance reasons, we want to observe mouse move events only during a drag operation. Therefore, we need to explicitly install and uninstall the mouse move observer and we need to remember the point of installation (`control` above).

Semantic distance Ultimately, the example is hard to understand because the control flow is inverted which results in too much boilerplate code that increases the semantic distance between the programmers intention and the actual code.

Mouse dragging, which already comes in large varieties, is just an example of the more general set of input gesture recognition. If we further generalize this to event sequence recognition with (bounded or unbounded) loops, all the problems we mentioned above still remain. Many examples in user interface programming are therefore equally hard to implement with observers, such as selecting a set of items, stepping through a series of dialogs, editing and marking text – essentially every operation where the user goes through a number of steps.

1.1 Contributions and Overview

Our contributions are:

- We show how to integrate composable reactive programming abstractions into a statically typed programming language that solve the problems of the observer pattern. To our knowledge, Scala.React is the first system that provides several API layers allowing programmers to stepwise port observer-based code to a data-flow programming model.
- We demonstrate how an embedded, extensible data-flow language provides the central foundation for a composable variant of observers. It further allows us to easily express first-class events and time-varying values whose precise behavior change over time.
- The embedded data-flow language can make use of the whole range of expressions from our host language without explicit lifting. We show how this can be achieved by the use of delimited continuations in the implementation of our reactive programming DSL.

In the following, we start with the status quo of handling events with callbacks and gradually introduce and extract abstractions that eventually address all of the observer pattern issues we identified above. Ultimately, we will arrive at a state where we make efficient use of object-oriented, functional, and data-flow programming principles. Our abstractions fit nicely into an extensible inheritance hierarchy, promote the use of immutable data and let clients react to multiple event sources without inversion of control.

2. A general interface for composable events

The first step to simplify event logic in an application is to come up with a general event interface so that all event handling code can work with a uniform interface. A second aspect to this is reusability: if we can hide event propagation and observer handling behind a general interface, clients can easily publish events for their own data structures. Now, designing such a general interface is an easy task. We introduce a class `EventSource[A]`, which represents generic event sources. We can use an event source to *raise* or *emit* events at any time. Type parameter `A` denotes the type an event from a given source can have. Here is how we create an event source of integers and raise a number of events:

```
val es = new EventSource[Int]
es raise 1
es raise 2
```

We provide method `observe` which accepts a closure to react to events. The following prints all events from our event source to the console.

```
observe(es) { x =>
  println("Receiving␣" + x)
}
```

Sometimes, we want to get a handle of observers, e.g. to uninstall and dispose them prematurely. Therefore, method `observe` actually returns an observer that can be disposed with a single method call:

```
val ob = observe(es) { x =>
  println("Receiving␣" + x)
}
...
ob.dispose()
```

Note, that there is no need to remember the event source to uninstall the observer. To put the above together, we can now create a button control that emits events when somebody clicks it. We can use an event source of integers with an event denoting whether the user performed a single click, or a double click, and so on:

```
class Button(label: String) {
  val clicks: Events[Int] = new EventSource[Int] {
    // call "this raise x" for each system event
  }
}
```

Member `clicks` is publicly an instance of trait `Events` that extracts the immutable interface of `EventSource`:

```
abstract class Events[+A] {
  def subscribe(ob: Observer): Unit
  def message(ob: Observer): Option[A]
}
class EventSource[A] extends Events[A] {
  def emit(ev: A): Unit
  ...
}
```

We can now implement a quit button as follows:

```
object Application extends Observing {
  ...
  val quitButton = new Button("quit")
  observe(quitButton.clicks) { x => System.exit() }
}
```

A consequence from our event streams being first-class values is that we can abstract over them. Our application from above observes button click events directly. Instead, it could observe any given event stream, may it be button clicks, menu selections, or a stream emitting error conditions. What if, however, we want to quit on events from multiple sources? We could add the same observer to all of those streams, but that would be quite some duplication:

```
val quitButton = new Button("quit")
val quitMenu = new MenuItem("quit")
val fatalExceptions = new EventSource[Exception]
observe(quitButton.clicks) { x => System.exit() }
observe(quitMenu.clicks) { x => System.exit() }
observe(fatalExceptions) { x => System.exit() }
```

Now that we have a first-class event abstraction we need composition features! To improve on the example above, we merge multiple event streams into a single one here. For that purpose, we introduce a merging operator in class `Events[A]` with the following signature:

```
def merge[B>:A](that: Events[B]): Events[B]
```

This method creates a new event stream that emits all events from the receiver and the given stream[1]. We say that the newly created event stream *depends on* the arguments of the merge operator; together they form a part of a larger dependency graph as we will see shortly. The reactive framework automatically ensures that events are properly propagated from the arguments (the *dependencies*) to the resulting stream (the *dependent*).

We can now write a general application trait which can be reused by any UI application:

```
trait UIApplication extends Observing {
  ...
  val quit: Events[Any]
  observe(quit) { x =>
    ... // clean up, display dialog, etc
    System.exit()
  }
}

object MyApp extends UIApplication {
  ...
  val quit = (quitButton.clicks
        merge quitMenu.clicks
        merge fatalExceptions)
}
```

Notice that method `merge` is parametric on the event type of the argument stream. The type parameter `B` is bound to be a base type of `A`, denoted by `B>:A`. It means we can merge any two streams for which the can infer a common base type `B`. This enables us to safely declare trait `Events[+A]` to be covariant in its event type, indicated by the plus sign in front of the `A`. As a result we can merge events of unrelated types such as `quitButton.clicks` which raises events of type `Int` and `quitMenu.clicks` and `fatalExceptions`, which raise events of types, say, `Unit` and `Exception`. The compiler simply infers the least upper bounds of those types, which in this case, is `Any`, the base type of all Scala values.

In order to log a reason why the application should quit, we need to converge on a common event type for all involved

[1] There is a minor issue here: both streams could raise events at the same time. In that case, the left stream (the receiver) wins. We could introduce a more general merge operator that would emit both events but it would have the more complicated type `def merge[B](that: Events[B]): Events[Option[A], Option[B]]` and would be harder to use. Note that we will discuss the notion of simultaneous events below in more detail.

streams, e.g., String. We can extract quit messages from each of the merged event streams with the help of the map combinator which is defined in trait Events:

```
def map[B](f: A => B): Events[B]
```

It returns a stream of results which raises at the same time as the original stream but each event applied to the given function. We can now implement quit as follows:

```
val quit =
  (quitButton.clicks.map(x => "Ok")
      merge quitMenu.clicks.map(x => "Ok")
      merge fatalExceptions.map(x => x.getMessage))
```

There are many more useful combinators in class Events that have their origin in functional reactive programming (FRP) [13, 21, 48]. A somewhat Scala specific convenience combinator is

```
def collect[B](p: PartialFunction[A, B]): Events[B]
```

which maps and filters at the same time. The resulting event stream raises those events from the original stream applied to the given partial function p for which p is defined.

The map and filter combinators can both be implemented in terms of collect:

```
def map[B](f: A => B): Events[B] =
  collect { case x => f(x) }
def filter(p: A => Boolean): Events[A] =
  collect { case x if p(x) => x }
```

3. Reactors: composable observers without inversion of control

Using our new event abstraction, we can now define a class Control that exposes mouse events as event streams. Our initial mouse dragging example becomes:

```
var path: Path = null
var moveObserver = null
observe(control.mouseDown) { event =>
  path = new Path(event.position)
  moveObserver =
    observe(control.mouseMoves) { event =>
      path.lineTo(event.position)
      draw(path)
    }
}
observe(control.mouseUp) { event =>
  moveObserver.dispose()
  path.close()
  draw(path)
}
```

Since we have a uniform observe mechanism and first-class events, we can abstract over the events involved in a drag operation. We could, for example, wrap the above in a function with the following signature:

```
def installDragController(start: Events[Positional],
  move: Events[Positional], end: Events[Positional])
```

and let mouse events extend the Positional interface with a position member. Now, we could let users perform drag operations with a different pointer device and start or abort with key commands. For example:

```
def installDragController(pen.down, pen.moves,
  pen.up merge escapeKeyDown.map(x => pen.position.now))
```

Yet, the most important issue that makes the above code hard to understand still remains: its control flow is inverted. Ideally, we would want to directly encode a state machine which can be described informally as follows:

1. Start a new path, once the mouse button is pressed
2. Until the mouse is released, log all mouse moves as lines in the path
3. Once the mouse button is released, close the path

In order to be able to turn the above steps into code that is conceptually similar, we need to find a way to let clients *pull* values from event streams without blocking the current thread. In Scala.React, we can achieve this with a *reactor* and the next operator which is part of an embedded data-flow language. The following creates a reactor that implements our running example without inversion of control.

```
Reactor.once { self =>
  // step 1:
  val path = new Path((self next mouseDown).position)
  // step 2:
  self loopUntil mouseUp {
    val m = self next mouseMove
    path.lineTo(m.position)
    draw(path)
  }
  // step 3:
  path.close()
  draw(path)
}
```

Object Reactor defines the two methods

```
def once(body: Reactor=>Unit): Reactor
def loop(body: Reactor=>Unit): Reactor
```

that clients can use to create a reactor that executes its body once or repeatedly. The actual body of a reactor is given as an argument to these methods. Formally, a body is a function that accepts the reactor under construction as an argument (a self reference similar to Java's and Scala's built-in **this**) and evaluates to Unit, i.e., we are interested in the side-effects of the body, not the result it evaluates to. The body uses the given self reference to express an embedded data-flow program. Class Reactor contains two core data-flow methods:

```
def next[A](e: Events[A]): A
def delay: Unit
```

Method next simply suspends the current reactor until the given event stream emits a value. Once the stream raises an event e, it evaluates to e and continues the reactor's execution. Method delay suspends the current reactor and continues after all pending messages have been propagated by the reactive framework.

In the above example, we first create a new path and then wait for the next mouse down event to add a line to the path with the current mouse position. This covers step 1 from our informal description. Step 2 is covered by the following loop which uses method

```
def loopUntil[A](e: Events[A])(body: =>Unit): A
```

which can be expressed in terms of next and other combinators we will encounter further below. The loop iterates until mouseUp has raised an event during an iteration step. Above, we simply drop the result and close the path in step 3.

4. Signals: Time-varying values

In the previous section, we saw how we can move logic from observers into the event dependency graph and thus into the framework to increase composability and reduce boilerplate code. Thereby we have dealt with problems that we could naturally model as events such as mouse clicks, button clicks, menu selections, and exceptions. A large body of problems in interactive applications, however, deals with synchronizing data that changes over time. Consider the button from above which could have a time-varying label. We represent time-varying values by instances of trait Signal:

```
class Button(label: Signal[String])
```

Trait Signal is the continuous counterpart of trait Events and contains a mutable subclass:

```
abstract class Signal[+A]
class Var[A](init: A) extends Signal[A] {
  def update(newValue: A): Unit = ...
}
```

Again, the base class (Signal) is covariant on its type parameter denoting the type of values a signal can hold.

4.1 Signal Expressions

The principal instrument to compose signals are not combinator methods, as for event streams, but *signal expressions*. Here is an example how one can build the sum of two integer signals:

```
val a = new Var(1)
val b = new Var(2)
val sum = Signal{ a()+b() }
observe(sum) { x => println(x) }
a()= 7
b()= 35
```

The above code will print 9 and 42. The Signal function invoked on the third line takes an expression (the *signal expression*) that continuously evaluates to the new signal's value. Signals that are refered to through the function call syntax such as a() and b() above are precisely those signals that the new signal will depend on. It is possible to obtain the current value of a signal without creating any dependenices by calling Signal.now. A call to now is valid inside as well as outside of a signal expression. To illustrate the difference between now and the function call syntax, consider the following snippet:

```
val b0 = b.now
val sum1 = Signal{ a()+b0 }
val sum2 = Signal{ a()+b.now }
val sum3 = Signal{ a()+b() }
```

All three sum signals depend on a, but only the last one also depends on b as mentioned above. Signal sum1 is different from sum2. Whenever sum2's expression is about to be reevaluated, the current value of b is obtained anew, while b0 in sum1 is a constant.

Signals are primarily used to create variable dependencies as seen above. There is not much more to it. Clients can build signals of any immutable data structure and any side-effect free operations inside signal expressions.

Here is how the public signal interface looks like:

```
trait Signal[+A] {
  def apply(): A
  def now: A
  def changes: Events[A]
}
```

The apply method allows us to use the function call syntax on signals. Every expression of the form e() is rewritten to e.apply() by the Scala compiler. This piece of syntactic sugar is part of the language to support first-class functions in a uniform way. Method changes gives us an event view on a signal, i.e., the resulting event raises the current value of the signal whenever it changes.

4.2 Constants

Constant signals are represented by a subclass of Signal[A]:

```
class Val[+A](value: A) extends Signal[+A]
```

We provide an implicit conversion from plain Scala values to Vals that allows us to use plain Scala values as constant signals:

```
implicit def coerceVal[A](v: A): Val[A] = new Val(v)
```

We can now write:

```
val b = new Button("Quit")
```

The last line is expanded by the compiler to

```
val b = new Button(Val("Quit"))
```

5. Data-flow Reactives

As a next step to improve our dragging example, we will separate the construction of the path from the drawing operations. We achieve this by using an extension of the data-flow language we know from reactors. Instead of performing side-effects, we build a *data-flow signal*:

```
val path: Signal[Path] =
  Val(new Path) once { self =>
    import self._
    val down = next(mouseDown)
    emit(previous.moveTo(down.position))
    loopUntil(mouseUp) {
      val m = next(mouseMove)
      emit(previous.lineTo(m.position))
    }
    emit(previous.close)
  }
```

Methods once and loop, as we know them from reactors are similarly defined for signals. Instead of creating a reactor, they create a new signal and are called on a signal that delivers the initial values for the new signal. In the example above, we create a signal that starts with an empty path and then proceeds once through the given data-flow body. Argument self refers to the signal under construction and is of type DataflowSignal[Path] which extends Signal[Path] and defines a set of data-flow methods in addition to those available for reactors. To keep things short, we first import all members from our self reference which lets us drop the self prefixes. We replaced all path mutations and drawing calls by calls to data-flow method emit, which changes the resulting path signal immediately. We refer to method previous in DataflowSignal in order to obtain the previous value of our path signal and modify its segments. We are using an immutable Path class above. Methods lineTo and close do not mutate the existing instance, but return a new path instance which extends or closes the previous one.

We can now implement the drawing operation in a simple external observer:

```
observe(path)(draw)
```

5.1 A data-flow language

We have now two variants of a data-flow language, one for reactors and one for signals. In order to keep languages consistent and extract common functionality, we can factor our existing abstractions into a class hierarchy as follows.

```
trait Reactive[+Msg, +Now] {
  def current(dep: Dependant): Now
  def message(dep: Dependant): Option[Msg]
  def now: Now = current(Dependent.Nil)
  def msg: Msg = message(Dependent.Nil)
}
trait Signal[+A] extends Reactive[A,A]
trait Events[+A] extends Reactive[A,Unit]
```

Classes Signal and Events share a common base trait Reactive. We will therefore collectively refer to them as *reactives* in the following. Trait Reactive declares two type parameters: one for the message type an instance emits and one for the values it holds. For now, we have subclass signal which emits its value as change messages, and therefore its message and value types are identical. Subclass Event only emits messages and never holds any value. Its value type is hence Unit. Subclasses of trait Reactive need to implement two methods which obtain the reactive's current message or value and create dependencies in a single turn.

In order to build a data-flow reactive using the loop and once combinators, we implicitly convert a reactive to an intermediate class that provides those combinators[2]:

```
implicit def eventsToDataflow[A](e: Events[A]) =
  new EventsToDataflow(e)
implicit def signalToDataflow[A](s: Signal[A]) =
  new SignalToDataflow(s)
```

These intermediate classes are defined as follows:

```
trait ReactiveToDataflow[M, N,
    R <: Reactive[M,N],
    DR <: DataflowReactive[M,N,R]]
    extends Reactive[M, N] {
  protected def init: R

  def loop(body: DR => Unit): R
  def once(body: DR => Unit): R
}

class EventsToDataflow[A](initial: Events[A])
  extends Events[A]
  with ReactiveToDataflow[A, Unit, Events[A],
                          DataflowEvents[A]]

class SignalToDataflow[A](initial: Signal[A])
  extends Signal[A]
  with ReactiveToDataflow[A, A, Signal[A],
                          DataflowSignal[A]]
```

Trait ReactiveToDataflow extends Reactive and provides two additional type parameters to fix the precise type of reactives we are creating. The type related details of this design are out of the scope of this paper. It is a result from our experience we gathered during the redesign of Scala's collection library which is thoroughly described in [41].

The base type for data-flow reactives defines the data-flow language for reactives and is specified as follows:

```
trait DataflowReactive[M, N, R <: Reactive[M,N]]
      extends Reactive[M, N]  {
  def emit(m: M): Unit
  def switchTo(r: R): Unit
  def delay: Unit

  def next[B](r: Reactive[B,_]): B
  def nextVal[B](r: Reactive[_,B]): B
}
```

[2] This is similar to extension methods in LINQ [46] but kept outside of trait Reactive for a different reason: to fix the concrete type of data-flow reactives loop and once create while still allowing covariant Msg and Now type parameters.

next Waits for the next message from the given reactive `r`. It immediately returns, if `r` is currently emitting.

nextVal Waits for the next change in the given reactive `r`. It immediately returns, if `r` is currently changing, i.e., if `next(r)` would immediately return.

delay Suspends the current data-flow reactive and continues its execution the next propagation cycle.

emit Emits the given message `m` if `m` makes sense for the current data-flow reactive and its current value. The current value of the reactive is changed such that it reflects the changed content. The evaluation of the reactive continues the next propagation cycle.

switchTo Switches the behavior of the current data-flow reactive to the given reactive `r`. Immediately emits a message that reflects the difference between the previous value of the current reactive and `r`. Emits all messages from `r` until the next call to `emit` or `switchTo`. The evaluation of the reactive continues the next propagation cycle.

Note that the following data-flow signal

```
0 once { self =>
  self switchTo sig
  self emit 1
}
```

first holds the current value of `sig` and then, in the next propagation cycle, switches to `1`. It is equivalent to signal

```
sig once { self =>
  self emit 1
}
```

Since reactors share a subset of the above data-flow language, we can extract this subset into a common base trait for `Reactor` and `DataflowReactive`:

```
trait DataflowBase {
  def next[B](r: Reactive[B, _]): B
  def delay: Unit
}
```

Note that only instances of classes that immediately specify their base class's parameters are visible to common library users. Therefore, they generally do not see any of the more complicated types above.

5.2 Reactive combinators as data-flow programs

Our first reactive composition feature above where event stream and signal combinators. There are two related questions now. Do we have a sufficient set of reactive combinators to address all arising event handling problems (conveniently)? To be honest, we do not know and there is no substantial empirical data available on how FRP combinators are used. What if a client hence feels the need to implement additional combinators? He has to know about the inner workings of reactive event propagation and how data inside the dependency graph is kept coherent. Another reason why reactive combinators can be tricky to implement is because they are implemented with an observer based approach suffering from inversion of control. In summary, clients extending Scala.React would see all the awkward details we actually wanted to shield from clients. Luckily, our data-flow language proves to be a convenient tool to implement reactive combinators! Here is how we can implement some combinators in class `Events[A]` that are not trivially implemented in terms of other combinators. We have seen `collect` already:

```
def collect[B](p: PartialFunction[A, B]) =
  Events.loop[B] { self =>
    val x = self next outer
    if (p isDefinedAt x) self emit p(x)
    else self.delay
  }
```

Combinator `hold` creates a signal that continuously holds the previous value that the event stream raised:

```
def hold(init: A): Signal[A] =
  Val(init) loop { self =>
    self emit (self next this)
  }
```

Combinator `switch` creates a signal that behaves like the first given signal until this stream raises an event. From that point on, it switches to the second given signal:

```
def switch[A](before: Signal[A],
              after: =>Signal[A]): Signal[A] =
  before once { self =>
    self next this
    self switchTo after
  }
```

Combinator `take` creates a stream that raises the first `n` events from this stream and then remains silent.

```
def take(n: Int) = Events.once[A] { self =>
  var x = 0
  while(x < n) {
    self emit (self next outer)
    x += 1
  }
}
```

The use of `Events.once` ensures that the resulting event stream does not take part in event propagation anymore, once it has raised `n` events. A `drop` combinator can be implemented in a similar fashion.

The `merge` combinator is the only axiomatic combinator which is not implemented in terms of a data-flow reactive. Alternatively, we could provide a fork and join data-flow expression. In our experience so far, use cases that would justify its existence did not arise.

Trait `Signal[A]` contains two flatten combinators which are defined for signals of events and signals of signals. They return a signal or event that continuously behaves like the signal or event that is currently held by the outer signal. They can be implemented as follows:

```
def flattenEvents[B]
  (implicit witness: A => Events[B]) =
    Events.loop[B] {
      self switchTo witness(self next this)
    }

def flatten[B](implicit witness: A => Signal[B]) =
  witness(this.now) loop { self =>
    self switchTo witness(self next this)
  }
```

These can be generalized into a single generic combinator. Flattening a signal of reactives makes sense for any subclass of `Reactive`, not just `Signal` or `Events`.

```
def flatten[M, N, R <: Reactive[M,N],
      DR <: DataflowReactive[M,N,R]]
  (implicit c: A =>
           R with ReactiveToDataflow[M,N,R,DR]): R =
    c(now) loop { self =>
      self switchTo c(self next this)
    }
```

The implicit parameter is used to convert a current signal value to a `ReactiveToDataflow` in order to construct a data-flow reactive. This enables us to flatten a signal of any subtype `R` of `Reactive` to an instance of `R` that behaves like the reactive that is currently held by the signal. We will later see how to meaningfully extend trait `Reactive`.

5.3 Combinators versus data-flow operations

As an alternative to a data-flow formulation of the dragging problem, we can write it in a purely combinator-based style. A similar example can be found in [35], which we adopt to Scala.React in the following.

```
val moves = mouseDown map { md =>
  mouseMove map (mm => new Drag(mm))
}
val drops = mouseUp map { mu =>
  Events.Now(new Drop(mu))
}
val drags = (moves merge drops).flatten
```

Above, we use our previously defined `flatten` combinator to switch between different event streams. For this reason, `flatten` is sometimes called `switch`. In fact, our `switch` combinator from above is a simple form of `flatten`.

The above example is actually an instance of a more general problem: *event partitioning*. Event partitioning deals with the task of constructing event streams that depend on different streams at different time periods. Above, our resulting `drags` stream loops over three partitions.

1. It remains silent until the next mouse down event.
2. Between mouse down and mouse up events, it depends on mouse move events,
3. After the mouse up event, it raises a final drop event.

One way to deal with event partitioning is to build a higher order event stream as above and switch between different event streams with the `flatten` combinator. Useful instruments in our combinator toolbox are nesting `map` applications to create higher-order event streams and the asymmetric `merge` combinator to switch between (overlapping) event streams.

Our data-flow formulation of the same problem is equivalent but avoids dealing with higher order event streams. Whereas the higher order event stream defines the state transition of a state machine as a nested data structure, a data-flow program describes it as a flat program source.

Combinators and data-flow reactives can naturally complement each other. For instance, the implementation of data-flow method `loopUntil` in trait `DataflowBase` is a data-flow program which reuses the `switch` combinator:

```
def loopUntil[A](es: Events[A])(body: =>Unit): A = {
  val x = es switch (None, Some(es.msg.get))
  while(x.now == None) {
    body
  }
  x.now.get
}
```

5.4 Recursion

Dataflow reactive can be used to define recursive signals and events. Consider the following signal:

```
val counter = 0 loop { self =>
  self emit (self.now + 1)
}
```

This signal is ill-defined and will throw an exception because we are referring to the current value of the signal while evaluating its current value! We therefore define method `previous` in `DataflowSignal`, which let us access the old value of the signal under construction:

```
val counter = 0 loop { self =>
  self emit (self.previous + 1)
}
```

This signal is well-defined because `emit` statements introduce a delay. Otherwise the resulting signal would loop forever. Therefore, delays introduced by `emit` and `switchTo` have a safety aspect to them in that they prevent loops from looping infinitively. Note that the above signal in fact inspects the behavior of the reactive scheduler: it counts propagation cycles starting with the current one. We could, e.g., use it to implement a frame rate signal that updates its value every second:

```
val frameRate = Val(0) loop { self =>
  val c0 = counter.now
  self next Clock.inSeconds(1)
  self emit (counter.now-c0)
}
```

5.5 Extending the reactive hierarchy

So far we have dealt with signals and event streams as the only concrete type of reactives. The `Reactive` base interface is general enough to support a variety of subclasses. Examples are futures that eventually evaluate to some result. They emit exactly one message, the result of type `A` and have a continuous value of `Option[A]`:

```
trait Future[+A] extends Reactive[A,Option[A]]
```

The implementation of data-flow method `loopUntil` we have seen above can be slightly simplified. The `switch` combinator that creates the guard for the internal while loop

```
val x = es switch (None, Some(es.msg.get))
```

can be replaced by a future

```
val x = Future.fromEvent(es)
```

with the rest of the implementation unchanged.

Another example are incremental updates. We can think of a reactive text document or list that emit deltas to their previously held elements:

```
trait Document[+A]
  extends Reactive[DocDelta, String]
trait RList[+A]
  extends Reactive[ListDelta[A], List[A]]
```

Certain operations, such as concatenation of documents and mapping list elements can be optimized for incrementally changing reactives. Getting back to our dragging example, we can define a reactive path class that emits additional segments for each call to `lineTo` and alike. Path deltas can be represented by an abstract data type encoded as Scala case classes:

```
sealed class PathDelta
case class MoveTo(x: Int, y: Int) extends PathDelta
case class LineTo(x: Int, y: Int) extends PathDelta
case object Close extends PathDelta
```

The actual reactive subclass would then hold values of the immutable path we used above and emit `PathDelta` instances:

```
class RPath extends Reactive[PathDelta, Path]
```

We further provide an implicit conversion from `RPath` to an `RPathToDataflow` instance that can create `RPaths` using the data-flow language provided by `DataflowReactive`.

```
class DataflowRPath(init: RPath) extends RPath
  with DataflowReactive[PathDelta, Path,
                   RPath, DataflowRPath]
implicit def rpath2dataflowrpath(r: RPath) =
  new DataflowRPath(r)
```

We can build an `RPath` as follows:

```
val path: RPath = (new RPath) once { self =>
  val down = self next mouseDown
  self emit MoveTo(down.position)
  val up = self loopUntil mouseUp {
    val m = self next mouseMove
    self emit LineTo(m.position)
  }
  self emit Close
}
```

Alternatively, we could implement `lineTo` and `close` as data-flow methods in `DataflowRPath` and use them instead of emitting messages directly:

```
def lineTo(x: Int, y: Int) = emit(LineTo(x,y))
def close(x: Int, y: Int) = emit(Close)
```

6. Implementation

Scala.React proceeds in propagation cycles. The system is either in a propagation cycle or, if there are no pending changes to any reactive, idle. Its model of time is thus a discrete one. Every propagation cycle has two phases: first, all reactives are synchronized so that observers, which are run in the second phase, cannot observe inconsistent data. During a propagation cycle, the reactive world is paused, i.e., no new changes are applied and no source reactive emits new events or changes values. Consider the following reactor:

```
Reactor.once {
  val es: Events[Device] = connectDevice()
  (self next es).initialize()
}
```

If event stream `es` would be able to emit events before the reactor calls `next`, we could miss to initialize some connected device. Pausing the reactive world during a propagation cycle and queueing all incoming external events to be processed in a later cycle prevents this from happening.

6.1 Non-strict Implementation

Our first prototypes implemented a "mark and validate on demand" change propagation mechanism. When receiving external events, a reactive source would invalidate itself and notify their dependents which in turn would recursively do the same. In order to avoid glitches, we first finish invalidating dependents until we start reevaluating them, i.e., a single message propagation cycle has two consecutive phases.

Unfortunately this approach is not very scalable. Although it generally restricts message propagation to a fraction of the dependency graph, every invalidated reactive source always invalidates the transitive closure of its dependents. This is not always necessary, though. Non-injective signal expressions can be very common. Consider the following signals, for example:

```
val sum = Signal { x() * y() }
val coord = Signal { p().x }
val status = Signal {
  if(x() > 10000) "danger" else "ok"
}
```

If x and y are integer signals and one holds the value 0, changing the other does not change the value of signal sum. Member access is another case: if signal p holds points with x and y coordinates, changing p's value does not automatically change the value of coord. Depending of the depth of a dependency graph and the location of non-injective expressions, the above propagation scheme can become very inefficient. For an example that conveys the big picture and that is not some pathological case, consider a tree structure such as a control hierarchy or a document object model (DOM) in a GUI application. It is relatively common to propagate some data from the root to the leafs and possibly back again. An example are visual attributes such as CSS styles that controls or DOM nodes inherit from their parent. In the following example, we aggregate several style attributes into an immutable Pen class. The bounds of a tree node that displays geometric shapes depends on its parent pen's line width attribute.

```
class Pen(color: Color,
          lineWidth: Int,
          cap: CapStyle)

trait GeometryNode extends Node {
  def shape: Signal[Shape]
  val bounds = Signal {
    f(shape.bounds, parent().pen().lineWidth)
  }
}

trait Group extends Node {
  val bounds =
    children.foldLeft(Rectangle.Nil) { (a,b) =>
      (a union b).bounds
    }
}
```

We can now build a deep dependency graph with a long path from the root's Pen attribute down to the bounds attributes of each geometry node and up again for the bounds of each group node. Changing the pen of the root node will trigger a bounds calculation for most of the tree, no matter whether the line width attribute of the new pen is different from the old.

6.2 Push-driven Implementation

Our current implementation overcomes the inefficiencies of our first implementation with a purely push-based propagation approach. In order to prevent reactives from observing inconsistent data (also called *glitches*) during and between propagation cycles, we keep the dependency graph topologically sorted. To do so, we assign each reactive a level in the dependency graph. All source reactives have a level of 0, and every dependent reactive has the level of its highest dependency plus 1.

A propagation cycle proceeds as follows.

1. Enter all modified/emitting reactives into a priority queue with the priorities being the reactives' levels.
2. While the queue is not empty, take the reactive on the lowest level and validate its value and message. The reactive decides whether it propagates a message to its dependents. If it does so, its dependents are added to the priority queue as well.

The last step solves our propagation capping problem from above. A signal which evaluates to the same value as before, e.g., would not propagate a change message during the current cycle.

For a dependency graph with a fixed topology, i.e., where the level of reactives never change, this simple algorithm is sufficient to avoid data inconsistencies. We do, however, need to deal with conditionals, branches, and loops in signal expressions and in particular data-flow reactives that can drop previous dependencies and establish new ones from one propagation cycle to the next.

6.2.1 Dynamic Dependencies

Consider the following example:

```
val x = Var(2) // level 0
val y = Cache { f(x()) } // level 1
val z = Cache { g(y()) } // level 2
val result =
  Signal { if(x()==2) y() else z() } // level 2 or 3
```

Depending on the current value of signal x, signal result can have a topological level of 2 or 3. The most efficient solution would be to always assign to a signal a level that is higher than all levels of its *potential* dependencies – without evaluating the signal expression. Unfortunately, the dynamics of reactive dependencies is at odds with itself at this time. In general, we can neither statically determine all *possible* dependencies nor all possible levels of an expression signal. Therefore, we cannot know the level of an invalid signal before we actually evaluate its current value. The previously known level of a reactive merely serves as a reference value in the invalidation phase. If the level turns out greater than the previous one, we abort the current evaluation by throwing an exception, assign a higher level to the affected signal and reschedule it for validation on a higher level in the same propagation cycle. Because of this potential for redundant computation, programmers are advised to move expensive computations until after referencing a reactive's dynamic dependencies. Fortunately though, many combinators have enough information to predict the precise level of the resulting reactive and will thus never abort during evaluation. Moreover, signals and data-flow reactives typically "warm up" during their first iteration and do not need to be aborted in future propagation cycles.

Note that aborting the evaluation of a reactive is only safe because we disallow side-effects in signal expressions and reactive combinator arguments. Reactors, however, are allowed to perform side-effects, as this is their main purpose. This fact prevents us from aborting and rescheduling them, otherwise we could perform some side-effects more than once per propagation cycle. Luckily, our solution is straightforward. Since reactors do not have dependents, we can assign to them an infinite level (practically this means `Int.MaxValue` in Scala), i.e., observers can never even attempt to access inconsistent data since they are executed after all reactives have been validated.

6.2.2 Cycles

We must be careful not to run into cycles when computing the topological ordering of reactives. Consider the following recursive, but non-cyclic and hence well-defined signal definitions:

```
val c = Var(true)
val x = Signal { if(c()) 1 else y() }
val y = Signal { if(c()) x() else 1 }
```

Whenever signal `c` changes its value, a naïve implementation could lift signals `x` and `y` alternatively one level higher than the other, eventually resulting in an infinite loop or an arithmetic overflow[3]. Ideally, though, signals `x` and `y` would alternate between levels 1 and 2. A simple solution could be to reset the level of each reactive to zero prior to each propagation cycle. We would loose the warming up effect we discussed above, however. Reactives with multiple dependencies could consequently become very expensive, since their level would need to be recomputed every time their value is about to change. A slightly more elaborate implementation can avoid this inefficiency. In the middle of a propagation cycle, we actually have a cyclic dependency. For instance, when `c` changes from `true` to `false`, `x` is evaluated before `y` and lifted above `y`. Now `x` has `y` as a dependency and vice versa. We can efficiently detect such a cycle when `y` is entered into the invalidation queue with a level lower than `x` and perform a level reset only for the involved reactives. Luckily, situations like the above are very uncommon in our experience.

6.3 Signal expressions

The three key features that let us implement the concise signal expression syntax we are using throughout the paper are Java's thread local variables [31], Scala's call-by-name arguments and Scala's function call syntax. When we are constructing the following signal

```
Signal { a() + b() }
```

we are in fact calling the method

```
def Signal[A](op: =>A): Signal[A]
```

[3] Note that some languages, such as Lustre [26], simply disallow the given example. Without a dependently typed effect system or a compiler plugin in Scala, however, we cannot statically reject the above program, even if we would want to.

with the argument `{ a() + b() }`. The Scala compiler rewrites expressions `a()` and `b()` to `a.apply()` and `b.apply()`, which becomes important below. The arrow notation `=>A` makes `op` a call-by-name argument that evaluates to values of type `A`. This means the sum expression gets converted to a closure and is passed to the `Signal` method without evaluating it. This is how we capture signal expressions. The actual evaluation happens in the `Signal.apply` method which returns the current value of the signal while establishing signal dependencies. Method `Signal.apply` comes in different flavors, but the general concept remains the same: it maintains a thread local stack of dependent reactives that are used to create dependency sets. A signal that caches its values is either valid or has been invalidated in the current or a past propagation cycle. If it is valid, it takes the topmost reactive from the thread local stack without removing it, adds it to its set of dependents and returns the current valid value. If it is invalid, it additionally pushes itself onto the thread local stack, evaluates the captured signal expression, and pops itself from the stack before returning its current value.

We support lightweight signals that do not cache their values. They just evaluate the captured signal expression, without touching the thread local stack. The stack can then be accessed by signals that are called from the signal expression. The lightweight signal hence does not need to maintain a set of dependents or other state.[4]

6.4 Data-flow Reactives

Our data-flow DSL is implemented in terms of delimited continuations that we introduce in Scala 2.8 [44]. Trait `DataflowBase` contains most of the infrastructure to implement our data-flow language in terms of continuations:

```
trait DataflowBase {
  protected var continue =
    () => reset { mayAbort { body() } }

  def body(): Unit @suspendable
  def next[B](r: Reactive[B,_]): B @suspendable
  def nextVal[B](r: Reactive[_,B]): B @suspendable

  def delay: Unit @suspendable =
    shift { (k: Unit => Unit) =>
      continueLater { k() }
    }
}
```

Note that for brevity we have omitted the `@suspendable` annotations before. They are hints for the Scala to ensure that no CPS transformed code escapes a `reset`. Method `body` is implemented by subclasses and runs the actual data-flow program. Calls to `delay`, `next` or any extensions defined in subclasses (such as `emit` and `switchTo`) are implemented in terms of two helper methods:

```
def continueNow[A](op: (A=>Unit)=>Unit) =
```

[4] That is why we call signals that do not cache values *permeable*.

```
  shift { (k: A=>Unit) =>
    continue = { () => mayAbort { op(k) } }
    continue()
  }

def continueLater(k: =>Unit) = {
  continue = { () => k }
  engine nextTurn this
}
```

Method `continueNow` accepts a function `op` which takes the current continuation as an argument. A call to `shift`, which is part of the Scala standard library, captures the current continuation `k`, transforms it by applying `op` to it and stores the result in a variable for immediate and later use. The transformed continuation is wrapped in a call to `mayAbort`, which properly aborts and reschedules the current evaluation if the topological level of any accessed reactive is higher or equal to the current level. Method `continueLater` captures a given continuation in a variable and schedules this reactive for the next turn.

When validated during a propagation cycle, a data-flow reactive simply runs its current continuation saved in variable `continue`, which initially starts executing the whole body of the reactive.

In order to offer library clients a completely transparent data-flow language that let them reuse any existing kind of Scala expression, we extended our CPS transform implementation to correctly deal with loops and exceptions. We refer to Appendix A for the details.

We could use continuations for signal expressions as well. When discovering a level mismatch, instead of aborting and rescheduling the entire evaluation of the signal, we would reschedule just the continuation of the affected signal and reuse the result of the computation until the level mismatch was discovered, captured in the continuation closure. Unfortunately, this approach is rather heavyweight (though less heavyweight than using blocking threads) on a runtime without native CPS support. We therefore do not currently implement it and leave it for future work to compare the outcome with our current implementation. We are particularly interested in how often one actually needs to abort in realistic reactive applications, whether the performance overhead of CPS justifies its usage in this context in contrast to the memory overhead of the closures created by our CPS transform.

6.5 Side-effects

Local side-effects are allowed in any signal expression or reactive combinator argument such as the `Events.map` function argument or a data-flow body. *Local* in this context refers to side-effects that do not escape the scope of the combinator argument such as the `take` combinator implementation in Section 5. We disallow non-local side-effects since the evaluation of a reactive can abort because and retried on a higher level in the same propagation cycle. The following signal can therefore increase counter `i` twice when the level of the given signal `sig` changes:

```
def countChanges(sig: Signal[Any]) = {
  val i = 0
  Signal { sig(); i += 1; i }
}
```

When aborting a signal evaluation, the entire signal expression is reevaluated, working with a fresh set of local variables that are not affected by previous runs. For data-flow reactives, however, the situation is slightly more complicated. As they simulate state machines, we want to keep their state from one evaluation to the next. A reevaluation should therefore not execute any side-effecting expression twice. We ensure that this is indeed the case by always capturing the current continuation before a level mismatch can happen. Since a level mismatch can happen only during a call to `next`, `switchTo`, `message`, or `now`, and since these methods do not perform side-effects that affect the state of a data-flow reactive before a level mismatch abortion can happened, continuing with the captured continuation after an abortion ensures that we do not evaluate side-effecting expressions redundantly.

Internal or external side-effects in reactors are not problematic, since reactors always have maximum level, i.e., their evaluation never aborts.

6.6 Avoiding memory leaks in low-level observers

Internally, Scala.React's change propagation is implemented with observers. We do expose them to clients as a very lightweight way to react to changes in a reactive as we have seen in Section 2. Stepping back for a moment, one might be tempted to implement a `foreach` method in `Events` or even `Reactive` and use it as follows:

```
class View[A](events: Events[A]) {
  events foreach println
}
```

This can easily lead to a reference pattern as depicted in Figure 1.

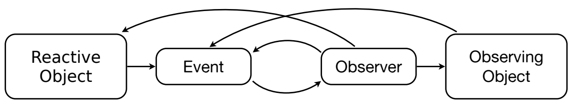

Figure 1. Common reference pattern in the standard observer scenario.

The critical reference here is the arc from the event stream to the observer, which prevents the object on the right to be disposed as usual. In a language that relies on runtime garbage collection such as Scala, we generally expect objects to be disposed once we don't hold a strong reference to them anymore. In the above scenario, however, we have a strong reference path from the event source to the observing object. For every observing object that we want to dispose before the reactive object, we would need to remember to call an explicit disposal method that unsubscribes the observing object's observers, otherwise the garbage collector cannot reclaim it.

The reference pattern in Figure 2 eliminates the leak potential.

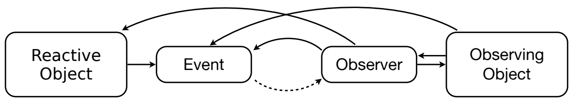

Figure 2. Observer reference pattern that eliminates the memory leak potential of the standard scenario from Figure 1.

Note the weak reference from the event source to the observer, depicted by a dashed arc. It eliminates any strong reference cycles between the observing and the publishing side. In order to prevent the observer from being reclaimed too early, we also need a strong reference from the observing object to the observer. It is important that we are always able force the programmer into creating the latter strong reference, otherwise we haven't gained much. Instead of remembering that she needs to call a dispose method, she would now need to remember to create an additional reference. Fortunately, we can use Scala's traits to achieve our desired reference pattern while reducing the burden of the programmer. The following trait needs to be mixed in by objects that want to observe events. API clients have no other possibility to subscribe observers.[5]

```
trait Observing {
  private val obRefs = new ListBuffer[Observer]()

  abstract class PersistentOb extends Observer {
    obRefs += this
  }

  protected def observe(e: Events[A])(op: A=>Unit) =
    e.subscribe(new PersistentOb {
      def receive() { op(e message this) }
    })
}
```

Instead of using a `foreach` method, we can now write the following:

```
class View[A](events: Events[A]) extends Observing {
  observe(events) { x => println(x) }
}
```

An instance of class `View` will print all arriving events and can now be automatically collected by the garbage collector, independently from the given event stream and without manually uninstalling its observer.

[5] We can directly store the first allocated persistent observer in a reference field. Only for the less common case of multiple persistent observers per `Observing` instance we do allocate a list structure to keep observer references around. This saves a few machine words for common cases.

7. Related Work

Some production systems, such as Swing and other components of the Java standard libraries, closely follow the observer pattern as formalized in [25]. Others such as Qt and C# go further and integrate uniform event abstractions as language extensions [32, 40, 47]. F# additionally provides first-class events that can be composed through combinators. The Rx.Net framework can lift C# language-level events to first-class event objects (called `IObservables`) and provides FRP-like reactivity as a LINQ library [36, 46]. Its precise semantics, e.g. whether glitches can occur, is presently unclear to us. Systems such as JavaFX [42], Adobe Flex [2] or JFace Data Binding [45] provide what we categorize as *reactive data binding*: a variable can be bound to an expression that evaluates to the result of that expression until it is rebound. In general, reactive data binding systems are pragmatic approaches that usually trade data consistency guarantees (glitches) and first-class reactives for a programming model that integrates with existing APIs. Flex allows embedded Actionscript [37] inside XML expressions to create data bindings. JavaFX's use of reactivity is tightly integrated into the language and transparent in that ordinary object attributes can be bound to expressions. Similar to Scala.React, JFace Data Binding establishes signal dependencies through thread local variables but Java's lack of closures leads to a more verbose data binding syntax. JFace is targeted towards GUI programming and supports data validation and integrates with the underlying Standard Widget Toolkit (SWT).

7.1 Functional Reactive Programming

Scala.React's composable signals and event streams originate in functional reactive programming (FRP) which goes back to Conal Elliot's Fran [21]. Its concepts have since been adopted by a variety of implementations with different programming interfaces and evaluation models. They all have in common that they provide a combinator-based approach reactive programming.

Fran integrates general ideas from synchronous data-flow languages [27] into the non-strict, pure functional language Haskell. It takes a monadic approach and encapsulates operations over time-varying values (called *behaviors* in Fran) and discrete events into data structures that are evaluated by an interpreter loop. Most of Scala.React's combinators methods in `Events` and `Signal` originate from Fran. Fran's evaluation model is pull-based, i.e., reactives are continuously sampled. An evaluation cycle hence consists of simple nested function calls. In [19], Fran-style FRP was revised to fit into Haskell's standard type classes such as `Monad` and `Functor`. It furthermore removes inefficiencies related to Fran's purely pull-based evaluation. It would be very interesting to find performance comparisons to Fran and other Haskell-based systems.

Fran's explicit notion of time and reactives and its stream-based implementation could lead to space-time leaks in FRP programs [20]. In order to overcome the potential of leaking reactives, Yampa for Haskell [15, 39] provides a first class notion of signal *functions* based on arrows [29], a generalization of monads. This leads to a programming style conceptually close to designing electric circuit diagrams. Fran and Yampa promote a purely functional, combinator-based programming style. Yampa, like Fran, is pull-based and evaluates reactives in recursive function calls.

FrTime [12, 13] integrates FRP-style reactivity into the strict, impure, dynamically typed functional language Scheme. Scala.React's propagation model originates in FrTime which also uses a push-driven evaluation model utilizing a topologically ordered dependency graph. Like in Scala.React, redundant evaluation is avoided both in breadth since FrTime propagates events only to dependents of invalid reactives and in depth, since propagation is capped when reaching non-injective signal operations and event filters. FrTime relies on Scheme's macro system to achieve concise signal composition syntax with no need to adapt existing data structures and operations to FrTime. As a consequence, a binary expression over signals (called *behaviors* in FrTime), e.g., creates a new signal. This makes the system formally very elegant but originally could lead to large dependency graphs. These issues have been addressed in [8], which successfully applies deforestation-like techniques to FrTime programs. Scala.React's approach using thread local variables for signal expressions is similar in spirit as it avoids storing continuations for each signal access.

Flapjax [35] is a library and compiler for FRP-like reactivity for web programming in Javascript and evolved out of FrTime. Flapjax comes with a compiler to allow concise signal expressions. When used as a library, it relies on explicit lifting of reactives and is then syntactically more heavyweight but integrates well into JavaScript. Like our approach, Flapjax integrates reactive programming in an object-oriented language. In contrast to Scala, Flapjax/-Javascript is dynamically typed, i.e., expressions which can be statically rejected by Scala.React can lead to delayed runtime errors in Flapjax as demonstrated in [35].

Frappé [14] is an FRP implementation for Java and uses a mixed push-pull implementation. Similar to JFace, its syntax is verbose because of Java's lack of closures and generics by the time it was written. Being based on Bean Properties it integrates with the Java standard library and thus supports observers.

SuperGlue [34] is a declarative object-oriented component language that supports a signal concept similar to that found in FRP and Scala.React. For signal connections, SuperGlue follows a unique approach that we find is closer to constraint programming such as in the Kaleidoscope language family [24] than to a combinator- or data-flow-based approach. SuperGlue provides guarded connection rules and rule overriding which is a simple form of Kaleidoscope's constraint hierarchies [7]. Its propagation model is similar to our previous non-strict model.

7.2 Adaptive Functional Programming

Adaptive functional programming (AFP) [1] is an approach to incremental computation in ML. It is related to FRP and Scala.React since it also captures computation for repeated reevaluation in a graph. AFP has been implemented in Haskell [9] using monads. Our CPS-based representation of data-flow programs is related to this effort and other monadic FRP implementations because any expressible monad has an equivalent formulation in continuation passing style [22, 23]. The AFP dependency graph and propagation algorithms are more elaborate than Scala.React's. This is mostly due to AFP's support for computation over incrementally changing data structures. In contrast, Scala.React and FRP currently supports reactivity mostly for plain values. Our effort to fit signals, event streams, and collections such as our reactive path structure into a standard reactive hierarchy with incremental change messages tries to approach the expressiveness of AFP.

7.3 Dataflow Programming

Our built-in data-flow language is mostly inspired by synchronous data-flow languages such as Signal [5], Lustre [26], and Esterel [6]. Unlike Scala.React, these languages usually provide strong guarantees on memory and time requirements and are compiled to state machines. As a tradeoff and in contrast to our system, most synchronous dataflow languages are first-order, i.e., they distinguish between stream processing functions, streams, and values that can be carried over streams. Work towards a higher-order synchronous data-flow language can be found in [11].

7.4 Complex Event Processing

Data-flow reactives share certain characteristics with complex event processing (CEP) systems such as TelegraphCQ [10], SASE [49], and Cayuga [18]. These systems use custom query languages similar to SQL [17] to recognize event patterns from multiple sources. Queries are usually stateful, similar to reactors and data-flow reactives, but are not designed to perform external side-effects. CEP systems are optimized to handle large data streams efficiently while our implementation is targeted towards general purpose event systems where the amount of event data is comparatively small.

7.5 Actors

Data-flow reactives and reactors share certain similarities with actors [3, 28], which are used as concurrency abstractions. Actors usually run concurrently and communicate with each other through message passing. Each actor has a mailbox, from which it sequentially processes incoming messages. Actors resemble data-flow reactives in that they handle messages without inversion of control but do so only from their own mailbox. Actors resemble event streams in that they can send events but, different from event streams, to specified actors. Actor communication is hence inherently directed and push-driven. In summary, while an actor sends messages to certain actors chosen by itself, it reacts to incoming messages from arbitrary actors. For data-flow reactives, the converse is true. They send messages to the public and react to messages from sources chosen by itself. Both actors and data-flow reactives simulate state machines, i.e., they encapsulate internal state. The major difference is that state transitions and data availability are synchronized among reactives, whereas actors behave as independent units of control.

8. Conclusion

We have demonstrated a new method backed by a set of library abstractions that allows a gradual transition from classical event handling with observers to reactive programming. The key idea is to use a layered API that starts with basic event handling and ends in an embedded higher-order dataflow language. In the introduction, we identified many software engineering principles that the observer pattern violates. To summarize, our system addresses those violations as follows:

Uniformity and abstraction We started with the introduction of first-class event streams and signals that were later generalized to reactives. The first-class status of reactives addresses the issue of abstraction and uniformity. Instead of relying on heavyweight event publishing components, our polymorphic reactives offer slim, uniform interfaces that represent isolated concepts. Low-level observers, reactors and data-flow reactives work the same way, regardless of the precise representation of an event stream or signal.

Encapsulation Reactors and data-flow reactives form single objects that can process complex event patterns such as in the dragging example without exposing external state.

Resource management Observer life-times are automatically restricted by our `Observing` trait. Our data-flow language ensures that internal observers are disposed when no longer needed.

Side-effects We can restrict the scope of side effects in reactors and data-flow reactives because our data-flow DSL captures the state of execution internally.

Composability Reactives can be composed in a functional way by means of reactive combinators or in a data-flow-oriented way by means of our data-flow DSL.

Separation of concerns Uniform reactives makes it straightforward to create signals and events of existing immutable data-structures and factor the interaction with external APIs into observers or reactors.

Scalablity Our message propagation algorithm ensures that no reactive can ever see inconsistent reactive data, regardless of the number of reactive dependencies. Base classes in the reactive hierarchy provide most of the implementation for a data-flow DSL that can be extended to suit the special needs of custom data structures.

Semantic distance The semantic distance between the programmer's intention and the resulting code is largely reduced by avoiding inversion of control.

We see a similar correspondence between imperative and functional programming in Scala and data-flow and functional reactive programming in Scala.React. Imperative programming is close to the (virtual) machine model and used to implement functional collections and combinators. Data-flow programming in our system is a simple extension to Scala's imperative core and can be readily used to implement reactive abstractions and combinators as we have shown. Programmers can always revert to reactors and low-level observers in case a data-flow oriented or combinatorial solution is not obvious.

Given our previously identified issues of the observer pattern for which we are now providing a gradual path out of the misery, we have to ask: Is the observer pattern becoming an anti-pattern?

Scala.React is a generalization of our previous non-strict reactive programming implementation which is used in a commercial game engine developed by Mimesis Republic[6] and can be downloaded from the author's website at `http://lamp.epfl.ch/~imaier`. It is under active development and contains a growing number of automated tests and examples. We will further soon make available a prototype for a reactive GUI framework.

Acknowledgments

We thank Sean McDirmid, who helped design our early prototypes. We moreover thank Stéphane Le Dorze, Daniel Lehenbauer, Pavel Curtis, and Antony Courtney for many valuable discussions on reactive programming and Donna Malayeri for proof-reading drafts of this paper.

References

[1] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. *ACM Trans. Program. Lang. Syst.*, 28(6):990–1034, 2006.

[2] Adobe Systems. Flex quick start: Handling data. `http://www.adobe.com/devnet/flex/quickstart/using_data_binding/`, 2010.

[3] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-01092-5.

[4] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992. ISBN 0-521-41695-7.

[6] `http://www.mimesis-republic.com/`

[5] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Sci. Comput. Program.*, 16(2):103–149, 1991.

[6] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Sci. Comput. Program.*, 19(2), 1992.

[7] A. Borning, R. Duisberg, B. Freeman-Benson, A. Kramer, and M. Woolf. Constraint hierarchies. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 48–60, 1987.

[8] K. Burchett, G. H. Cooper, and S. Krishnamurthi. Lowering: a static optimization technique for transparent functional reactivity. In *PEPM '07: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 71–80, 2007.

[9] M. Carlsson. Monads for incremental computing. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, 2002.

[10] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: continuous dataflow processing. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 668–668, 2003.

[11] J.-L. Colaço, A. Girault, G. Hamon, and M. Pouzet. Towards a higher-order synchronous data-flow language. In *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*, pages 230–239, New York, NY, USA, 2004. ACM.

[12] G. H. Cooper. *Integrating Dataflow Evaluation into a Practical Higher-Order Call-by-Value Language*. PhD thesis, Brown University, 2008.

[13] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *In European Symposium on Programming*, pages 294–308, 2006.

[14] A. Courtney. Frappé: Functional reactive programming in Java. In *PADL '01: Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages*, 2001.

[15] A. Courtney, H. Nilsson, and J. Peterson. The Yampa arcade. In *Haskell '03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 7–18, 2003.

[16] O. Danvy and A. Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[17] C. J. Date. *A guide to the SQL standard : a user's guide to the standard relational language SQL*. Addison-Wesley, Reading, Mass. [u.a.], 1987. ISBN 0-201-05777-8.

[18] A. J. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. M. White. Towards expressive publish/subscribe systems. In *EDBT*, pages 627–644, 2006.

[19] C. Elliott. Push-pull functional reactive programming. In *Haskell Symposium*, 2009.

[20] C. Elliott and C. Elliott. Functional implementations of continuous modeled animation (expanded version). In *In Proceedings of PLILP/ALP '98*, 1998.

[21] C. Elliott and P. Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997. URL `http://conal.net/papers/icfp97/`.

[22] A. Filinski. Representing monads. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1994.

[23] A. Filinski. Representing layered monads. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1999.

[24] B. N. Freeman-Benson. Kaleidoscope: mixing objects, constraints, and imperative programming. In *Proc. of OOPSLA/ECOOP*, pages 77–88, New York, NY, USA, 1990. ACM. ISBN 0-201-52430-X.

[25] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

[26] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. In *Proceedings of the IEEE*, pages 1305–1320, 1991.

[27] N. Halbwachs, A. Benveniste, P. Caspi, and P. L. Guernic. Data-flow synchronous languages, 1993.

[28] P. Haller and M. Odersky. Actors that Unify Threads and Events. Technical report, Ecole Polytechnique Federale de Lausanne, 2007.

[29] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, 1998.

[30] G. E. Krasner and S. T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, 1988. ISSN 0896-8438.

[31] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns (3rd Edition)*. Addison-Wesley Professional, 2006. ISBN 0321256174.

[32] J. Liberty. *Programming C#*. O'Reilly & Associates, Inc., 2003. ISBN 0596004893.

[33] B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. *SIGPLAN Not.*, 23(7), 1988.

[34] S. McDirmid and W. C. Hsieh. Superglue: Component programming with object-oriented signals. In *Proc. of ECOOP*, pages 206–229. Springer, 2006.

[35] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: a programming language for ajax applications. *SIGPLAN Not.*, 44(10):1–20, 2009.

[36] Microsoft Corporation. Reactive Extensions for .NET (Rx). `http://msdn.microsoft.com/en-us/devlabs/ee794896.aspx/`, 2010.

[37] C. Moock. *Essential ActionScript 3.0*. O'Reilly, 2007. ISBN 0596526946.

[38] L. R. Nielsen. A selective CPS transformation. *Electr. Notes Theor. Comput. Sci.*, 45, 2001.

[39] H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 51–64, 2002.

[40] Nokia Corporation. Qt - A cross-platform application and UI framework. `http://qt.nokia.com/`, 2010.

[41] M. Odersky and A. Moors. Fighting bit rot with types (experience report: Scala collections). In *FSTTCS 2009*, 2009.

[42] Oracle Corporation. JavaFX. `http://javafx.com/`, 2010.

[43] S. Parent. A possible future of software development. `http://stlab.adobe.com/wiki/index.php/Image:2008\_07\_25\_google.pdf`, 2008.

[44] T. Rompf, I. Maier, and M. Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In *ICFP*, pages 317–328, 2009.

[45] The Eclipse Foundation. JFace Data Binding. `http://wiki.eclipse.org/index.php/JFace_Data_Binding`, 2010.

[46] M. Torgersen. Querying in C#: how language integrated query (LINQ) works. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, 2007.

[47] K. Walrath, M. Campione, A. Huml, and S. Zakhour. *The JFC Swing Tutorial: A Guide to Constructing GUIs, Second Edition*. Addison Wesley Longman Publishing Co., Inc., 2004. ISBN 0201914670.

[48] Z. Wan and P. Hudak. Functional reactive programming from first principles. In *In ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–252, 2000.

[49] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 407–418, 2006.

A. CPS transforming loops

In [44] we have shown that first-class continuations can be implemented efficiently on the JVM and similar managed runtimes using a variant of the well-known CPS transform [4]. This result is a bit surprising, since CPS transforming a program replaces all method returns with function calls in tail position and requires allocating closure records to pass local state—a program structure very much at odds with stack-based architectures that do not support efficient tail calls. The way to make the CPS transform viable is to apply it only selectively [38], driven by the type system. In the system presented in [44], the control operators `shift` and `reset` [16] are declared with the following signatures:

```
def shift[A,B,C](fun: (A => B) => C): A @cps[B,C]
def reset[A,C](ctx: =>(A @cps[A,C])): C
```

The type system makes sure that `@cps` annotations are propagated outward from uses of `shift`. Thus, a function like

```
def foo(x: Int) = 2 * shift((k:Int=>Int) => k(x+1))
```

will have a return type of `Int @cps[Int,Int]`.

All expressions of type `A @cps[B,C]` will be translated to objects of type `Shift[A,B,C]`, where `Shift` is a generalized CPS monad:

```
class Shift[+A,-B,+C](val fun: (A => B) => C) {
  def flatMap[A1,B1,C1<:B](f: A=>Shift[A1,B1,C1]) =
    new Shift((k: A1=>B1) =>
                          fun((x:A) => f(x).fun(k)))
  def map[A1](f: A=>A1) = ...
}
```

The function `foo` defined above can then be transformed as follows:

```
def foo(x: Int) = new Shift((k:Int=>Int) =>
                                  k(x+1)).map(2 * _)
```

Being a type-driven transformation, it is easy to disallow capturing continuations in programming constructs that do not lend themselves to a straightforward CPS representation. One important example are imperative while-loops. The following example shows that delimited continuations make loops considerably more powerful:

```
var x = 0
while (x < 10) {
  shift { k => println("+"); k(); println("-") }
  x += 1
}
```

The code will print + when *entering* an iteration and - when *leaving* it. In general, while loops are no longer equivalent to directly tail-recursive functions. This is demonstrated by CPS-transforming the code above, which exhibits indirect recursion through several levels of function invocations:

```
var x = 0
def loop() = if (x < 10) {
  new Shift { k => println("+"); k(); println("-") }
  .flatMap { _ => x += 1; loop() }
} else {
  new Shift { k => k() }
}
loop()
```

Generalizing this translation to

```
def loop() = if (<condition>) {
  <body>.flatMap { _ => loop() }
} else {
  new Shift { k => k() }
}
loop()
```

is unfortunate, however, since no while-loop that contains CPS code can be implemented using branch instructions anymore. On the JVM, which lacks constant-space tail-calls, while loops with larger numbers of iterations would likely result in stack overflow errors. For the example above this is indeed the expected behavior, as demanded by the non tail-recursive continuation semantics. However, many actual uses of continuations inside while-loops might rather look like this:

```
var x = 0
while (x < 1000000) {
   if (x % 100000 == 0)
     shift { k => println("+"); k(); println("-") }
   x += 1
}
```

Due to the low number of actual continuations accessed (only 10 in total), throwing a stack overflow error here would not be acceptable.

Fortunately, we can put to work once more the initial choice of targeting a variant of the CPS monad instead of composing functions directly. We have glossed over the definition of `map` in class `Shift`, which is a (rather simple) static optimization to avoid building `Shift` objects that would be *trivial*, i.e. transforming expression `x` to `new Shift(k => k(x))`. Regarding the above example, a conditional with a `@cps` expression only in the `then` part will be transformed to build exactly such a trivial `Shift` object in the else part. Thus, all but 10 `Shift` objects will be trivial in the example. However, the situation is more complex as for `map` since whether one iteration is trivial or not depends on a *dynamic* condition.

Similar in spirit to `map`, we can further specialize for trivial `Shift` instances by restructuring `Shift` into a hierarchy of classes:

```
sealed abstract class Shift[+A,-B,+C] {
  def fun(f: A=>B): C
  // map, flatMap declared abstract
}
case class Shift0[+A,-B](x: A) extends Shift[A,B,B]{
  def flatMap[A1,B1,C1<:B](f: A=>Shift[A1,B1,C1]) =
                                                 f(x)
  def map[A1](f: A=>A1) = Shift0[A1,B,B](f(x))
}
case class Shift1[+A,-B,+C](fun0: (A => B) => C)
                              extends Shift[A,B,C] {
  def fun(f:A=>B) = fun0(f)
  // map, flatMap implemented as before
}
```

With this modified CPS monad implementation we can implement a more efficient translation of while-loops:

```
def loop() = if (<condition>) {
  <body> match {
    case Shift0(x) => loop()
    case ctx: Shift1 => ctx.flatMap { _ => loop() }
  }
} else {
  Shift0[Unit,Unit]()
}
loop()
```

In essence, we have inlined the dynamic dispatch and the definition of `Shift0.flatMap`. For the trivial case, this makes the recursive call a direct tail-call again and the standard, intraprocedural tail-recursion optimization performed by the Scala compiler will translate this call to a branch instruction.