Deprecating the Observer Pattern with Scala.React

EPFL-REPORT-176887

Ingo Maier 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 show how to integrate different reactive programming abstractions into a single framework that help migrate from observer-based event handling logic to more declarative implementations. Our central 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

We are seeing a continuously increasing demand in interactive applications, driven by a growing number of non-expert computer users. 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]. For an answer on whether it is actually worth bothering we quote an Adobe presentation from 2008 [37] on the status of current production systems:

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

We believe these numbers are bad not only because of the number of bugs is disproportional. We also believe that event handling code should account for a smaller fraction and that the way to achieve this is to provide better event handling abstractions. To illustrate the concrete problems of the observer pattern, we start with a simple and ubiquitous example: mouse dragging. The following example constructs a path object from mouse movements during a drag operation and displays it on the screen. For brevity, 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 generally defined in [25], encourages one to violate an impressive line-up of important software engineering principles:

Side-effects Observers promote side-effects on the API level.

Encapsulation Multiple observers are often needed to simulate a state machine as in the drag example. The state is stored in variables (such as `path` above) that are visible to all involved observers. For practical reasons, these variables are often placed in a broader scope were it can be misused by unrelated code.

Composability Multiple observers, dealing with a single concern such as a drag operation, form a loose collection of objects that are installed at different points at different times. Therefore, we cannot, e.g., easily add or remove drag behavior.

Resource management An observer's life-time needs to be managed explicitly. For 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 subject (point of installation).

Separation of concerns The observers from our example 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) [31] pattern.

Data consistency We can achieve a separation of concerns with a path that itself publishes events when it changes. Unfortunately, there is no guarantee for data consistency in the observer pattern. Suppose we create a rectangle that represents the bounds of our path, i.e., a publisher that depends on changes in our original path. Also consider an observer listening to changes in both the path and its bounds in order to draw a framed path. This observer needs to track explicitly 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, which is an example of a *glitch*.

Uniformity Different methods to install different observers decrease code uniformity.

Abstraction The observer pattern promotes the use of heavyweight interfaces. The example relies on a control class that often defines a much larger interface than a method to install mouse event observers. Therefore, we cannot abstract over event sources individually. For instance, we could let the user abort a drag operation by hitting any predefined key or use a different pointer device such as a touch screen or graphics tablet.

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

Mouse dragging 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

We present Scala.React, a library of composable, discrete reactive programming abstractions that provides several API layers to allow programmers to stepwise migrate from an observer-based to a reactive data-flow programming model that eventually addresses all of the issues raised above. Our contributions are in particular:

- We extend Scala with a typed higher-order reactive programming library that allows to combine reactive programming both in a functional as well as in imperative style. We show how these abstraction integrate with Scala's object system and how the trait concept simplifies binding observer-based interfaces to our reactive abstractions.
- We show how to incorporate a synchronous data-flow language similar to Esterel [6], but extended to a multi-valued domain and with higher-order features, into our reactive framework.
- We present an approach based on traits and weak references that automatically binds the life-time of observers to an enclosing object, eliminating a common source of memory leaks in observer-based and reactive programming.
- We show how to implement our language as a library in terms of a dependency stack, closures and delimited continuations. In contrast to previous similar systems, our implementation is agnostic of any host language feature, and treats every expression uniformly, no matter whether built-in or user-defined, including function application, conditionals, loops, exceptions and custom control flow operators, without special treatment by a macro system or explicit lifting.
- We present and analyze performance results of different implementation approaches in Scala.React and present the first performance comparison of two related reactive libraries on different, production quality platforms. In order to obtain meaningful results, we measure performance of reactive implementations *relative to* corresponding observer-based implementations on the same platform. We analyze existing functional reactive programs and propose favorable solutions using alternative abstractions in Scala.React.

Although our running example is drawn from user interface programming, Scala.React can also be used for a many other event processing domains, such as financial engineering, industrial control systems, automatic patient supervision, RFID tracking or robotics.

2. Event streams: a uniform interface for composable events

As a first step to simplify event logic we introduce a general event interface, addressing the issues of uniformity and abstraction from above. A third aspect to this is reusability: by hiding event propagation and observer handling behind a general interface, clients can easily publish events for their own data structures. We introduce a class `EventSource[A]`, which represents generic sources of events of type `A`. We can schedule an event source to emit a value at any time. Here

is how we create an event source of integers and emit two events:[1]

```
val es = new EventSource[Int]
es emit 1; es emit 2
```

We can print all events from our event source to the console as follows:

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

Method `observe` takes event stream `es` and a closure that accepts event values `x` from `es`. The resulting observer `ob` can be disposed by a single method call to `ob.dispose()`, which uninstalls `ob` from all sources. Unlike in the traditional observer pattern, there is no need to remember the event sources explicitly. 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] {
    // for each system event call "this emit x"
  }
}
```

Member `clicks` is publicly an instance of trait `Events` that extracts the immutable interface of `EventSource`, i.e., without an explicit `emit` method. We can now implement a quit button as follows:

```
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. Above, we observe button clicks directly. Instead, we 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? Adding the same observer to all of those streams would lead to 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 can add composition features in the style of functional reactive programming (FRP) [14, 20, 45]. In the example above, it would be better to merge multiple event streams into a single one and install a single observer. The `merge` operator in class `Events[A]` creates a new event stream that emits all events from the receiver and the given stream:

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

[1] Infix notation `a emit b` is shorthand for `a.emit(b)` in Scala and works for any method.

Propagation in Scala.React acts in turns, i.e., emits all pending changes before it proceeds to the next turn. It is impossible to start a turn before the previous one has finished, which is important to ensure the absence of glitches. Since propagation is also synchronous, we can have event streams emitting values simultaneously. The given merge operator is biased towards the receiver, i.e., if event streams `a` and `b` emit simultaneously, `a merge b` emits the event from `a`. We will discuss details of our propagation model in more detail below.

We say that the newly created event stream *depends on* the arguments of the merge operator; together they are 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*). Method `merge` is parametric on the event type of the argument stream. The type parameter `B` is bound to be a super type of `A`, denoted by `B>:A`. We can therefore declare trait `Events[+A]` to be covariant in its event type, indicated by the plus sign. As a result we can merge events of unrelated types such as `quitButton.clicks` which emits events of type `Int` and `quitMenu.clicks` and `fatalExceptions`, which emits 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.

We can now move most of the code into a common application trait which can be reused by any UI application:

```
trait UIApp extends Observing {
  val quit: Events[Any]
  def doQuit() {
    /* clean up, display dialog, etc */;
    System.exit()
  }
  observe(quit) { doQuit() }
}
```

We can ignore the `Observing` base type for now. Clients can now easily customize the event source `quit` and the quit action `doQuit`:

```
object MyApp extends UIApp {
  ...
  val quit = quitButton.clicks merge
    quitMenu.clicks merge fatalExceptions
}
```

Another possibility to adapt an observer based interface to a uniform event API in Scala.React are selective mixins. Toolkits sometimes factor out interfaces for components that support a certain set of listeners or use naming conventions instead. Many classes in Java Swing, for instance, contain a `addChangeListener` or `addActionListener` method per convention. We can factor out a common interface in Scala.React using Scala's traits:

```
trait HasAction { this: HasActionType =>
  def getAction(): Action
```

```
  def addActionListener(a: ActionListener)
  def removeActionListener(a: ActionListener)

  val actionPerformed: Events[Action] =
    new EventSource[Action] { source =>
      this.addActionListener(new ActionListener {
        def actionPerformed(e: ActionEvent) =
          source emit getAction
      })
    }
}
```

The above creates a trait `HasAction` that defines a set of abstract methods that it expects to be implemented. We can now take a Java Swing `JButton` and add an `actionPerformed` event stream by writing

```
class MyButton extends JButton with HasAction
val button = new MyButton
observe(button.actionPerformed) { println("clicked") }
```

Note that the Scala compiler finds implementations of methods defined in traits based on their signature. This allows us to conveniently bind `JButton` to an event stream because from the perspective of the compiler, class `JButton` implements the abstract methods from trait `HasAction` even though Swing does not know about our trait.

In our example, in order to log a reason why the application should quit, we need to converge on a common event type for all involved 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[A]`:

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

It returns a stream of results that emits 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 common FRP combinators in class `Events` as we will see later.

3. Reactors: observers without inversion of control

Since we have a uniform observation mechanism and first-class events, we can abstract over the events involved in a drag operation. Therefore, we can define a function that installs all necessary observers for our drag controller:

```
def makePath(start: Events[Pos], move: Events[Pos],
  end: Events[Pos]): Path = {
  val path: Path = new Path()
  var moveObserver: Observer = null
  observe(start) { pos =>
    path.clear()
    path.moveTo(pos)
    moveObserver = observe(moves) { pos => ... }
  }
  observe(end) { pos => moveObserver.dispose(); ... }
  path
}
```

Now, we can easily let users perform drag operations with a different pointer device and start or abort with key commands. For example:

```
makePath(pen.down, pen.moves,
  pen.up merge escapeKeyDown.map(x => pen.pos.now))
```

Yet, three important issues remain. The control flow is still inverted, we have to explicitly dispose the mouse move observer and have no easy means to dispose the drag behavior as a whole. Ideally, we 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

We can turn the above steps into code in a straightforward way, using Scala.React's imperative data-flow language and the concept of *reactors*. The following reactor implements our running example without inversion of control.

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

Factory method `Reactor.loop` creates a new reactor, taking a function as an argument which accepts the reactor under construction `self` as an argument. This self reference gives the function a handle to a data-flow DSL, from which we use three operators above. Method `await` and `awaitNext` are defined as

```
def await[A](e: Events[A]): A
def awaitNext[A](e: Events[A]): A
```

and suspend the current reactor until the given event stream `e` emits a value. Method `await` return immediately, if the given stream is currently emitting, whereas `awaitNext` always suspends first. Once the stream emits an event `e`, it evaluates to `e` and continues the reactor's execution.

Another useful operator is `pause`, returning nothing. It suspends the current reactor and continues after all pending messages have been propagated by the reactive framework.

Now, `awaitNext` is equivalent to `pause` followed by `await`. More on `pause` and Scala.React's turn-based scheduling will follow below.

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
```

It watches event stream `e` while iterating over `body` until `es` emits. In Scala, type `=>Unit` denotes a call-by-name argument that returns `Unit`. This is equivalent to a parameterless closure with more concise syntax. In the example, the loop continues following the path until the mouse is released. We drop the result of `loopUntil` and close the path in step 3.

The imperative data-flow language largely reduces semantic distance and eliminates the need to dispose intermediate observers – `loopUntil` takes care of the latter. Reactors also give us a single handle to the entire behavior induced by multiple observers, which allows us to dispose the whole drag behavior by a single call to `Reactor.dispose()`.

4. Signals: time-varying values

Until now, we have been dealing with problems that we can 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:

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

Class `Signal` has a covariant type parameter denoting the type of the values it can hold.

4.1 Signal Expressions

Signals can be composed using *signal expressions*. We can build the sum of two integer signals `a` and `b` as follows:

```
val sum = Signal{ a()+b() }
observe(sum) { x => println(x) }
```

The `Signal` function invoked on the first line takes an expression (the *signal expression*) that continuously evaluates to the new signal's value. Signals that are referred by function call syntax as in `a()` and `b()` above are the dependencies of the new signal. In order to create only a momentary dependency, clients can call method `Signal.now`. 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`, i.e., they are invalidated when `a` changes. Only the last signal, though, also gets invalidated by changes in `b`. In `sum2`, whenever its expression is about to be reevaluated, the current value of `b` is obtained anew, while `b0` in `sum1` is `b`'s value at contraction time.

Signals are primarily used to create variable dependencies as seen above. Clients can build signals of any immutable data structure and safely use any operations not causing global side-effects inside signal expressions (details will be covered in Section 7.6). Expressions of the form `e()` are rewritten to `e.apply()` by the Scala compiler in order to support first-class functions in a uniform way. Method `apply` therefore establishes signal dependencies. Constant signals can be created by using the `Val` method. We can create a button with a constant label by writing `new Button(Val("Quit"))`.

Trait `Signal[A]` also defines a `changes` method and trait `Events[A]` defines a `hold` method that can be used to convert between the two. They are defined as

```
def changes: Events[A]
def hold(init: A): Signal[A]
```

Given the mutual conversion between an event stream and a signal, one might ask why Scala.React supports them as different concepts. As we will see later, there is in fact a common base class, which defines a common set of operations. However, there is a practical reason to keep them separate. As opposed to signals, event streams dispose their value after a propagation turn and can therefore not support the `Signal.apply` interface.

5. Imperative data-flow reactives

While reactors allow us to address most of the observer pattern's issues, we often want to separate the concerns of constructing a path from drawing it. For this purpose, we extend our imperative data-flow language to signals and events. Here is how we build a path signal:

```
val path: Signal[Path] = Signal.flow(new Path) { self =>
  val down = self await mouseDown
  self()= self.previous.moveTo(down.position)
  self.loopUntil(mouseUp) {
    val e = self awaitNext mouseMove
    self()= self.previous.lineTo(e.position)
  }
```

```
  self()= self.previous.close()
}
```

Method `Signal.flow` is similar to `Reactor.flow`. Instead of creating a reactor, though, its creates a new signal and takes an initial value. In the example, 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 `SignalFlowOps[Path]`, which extends the data flow language of reactors. Compared to the reactor-based version, we have replaced all path mutations and drawing calls by operations of the form `self()= x`, which changes the resulting path signal immediately. We call method `self.previous` in order to obtain the previous value of our path signal and modify its segments. Note that the DSL exposed by `self` does not provide a method to obtain the current value, since we cannot obtain the current value while in the process of evaluating it. 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.

5.1 An imperative data-flow language

We have now two variants of an imperative data-flow language, one for reactors and one for signals. In order to keep these languages consistent and extract common functionality, we factor our existing abstractions into a class hierarchy as follows[2].

```
trait Reactive[+P, +V] {
  def valueNow: Option[V]
  def pulseNow: Option[P]
  def subscribe(dependent: Reactive[Any,Any])
  ...
}
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. We will refer to their data-flow versions as *flow reactives*. Trait `Reactive` declares two type parameters: one for the type of pulses an instance emits and one for the values it holds. While pulses vanish after they were propagated, values persist. For now, we have subclass `Signal` which emits its current value as pulses, and therefore its pulse and value types are identical. Subclass `Event` only emits pulses and never holds any value. Its value type is hence singleton type `Unit`.

Companion objects[3] `Events` and `Signal` define methods `flow`, which we have seen previously, and `loop`, which runs the body repeatedly. Here is how they are defined for signals:

```
def flow[A](init: A)
          (op: SignalFlowOps[A]=>Unit): Signal[A] =
  new FlowSignal(init) { def body = op(this) }
def loop[A](init: A)
          (op: SignalFlowOps[A]=>Unit): Signal[A] =
  flow(init){ self => while(!isDisposed) op(self) }
```

We see that the self argument for flow signals is of type `SignalFlowOps` which extends a base trait `FlowOps`, defining most data flow operations that available for reactives and reactors. We summarize them in the following.

```
def pause: Unit
```

Suspends the current flow reactive and continues its execution the next propagation turn. As most data flow operators react to their input instantaneously, this operator is usually used to break infinite cycles as we will see in later examples.

```
def halt: Unit
```

Suspends and disposes the current reactive.

```
def await[A](input: Reactive[A, Any]): A
```

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

```
def awaitNext[A](input: Reactive[A, Any]): A
```

Equivalent to `pause; await(input)`, i.e., it ignores the current pulse from `input` if present.

```
def par(left: => Unit)(right: => Unit): Unit
```

It first runs the left then the right branch until they suspend, finish or call `join`. If both branches finish, or at least one branch called `join`, this method immediately returns. If no branch called `join` and at least one branch is not finished yet, this method suspends until next turn when it will continue evaluating each branch where it stopped previously. It proceeds so until both branches are finished or at least one calls `join`.

```
def join: Unit
```

Halts the execution of the current branch and, if present, joins the innermost enclosing 'par' expression. Halts the current branch only if there is no enclosing `par` expression.

```
def loopUntil[A](r: Reactive[A, Any])
                (body: =>Unit): A
```

Repeatedly evaluates `body` until `r` emits. If the body is suspended and hasn't finished executing, the continuation of the body is thrown away. Returns the pulse of `r`.

```
def loopEndUntil[A](r: Reactive[A, Any])
                   (body: =>Unit): A
```

Similar to `loopUntil` but returns only after the body has finished evaluating.

```
def abortOn[A](input: Reactive[A, Any])
              (body: =>Unit)
```

Similar to `loopUntil` but evaluates `body` only once.

[2] The class hierarchy we present here is slightly simplified compared to the actual implementation and focusses on the conceptually interesting aspects.

[3] Code in companion objects in Scala are similar to static class members in Java.

```
def <<(p: P): Unit
```

This operator is defined for events only. It emits the given pulse `p`, overwriting the current pulse.

```
def update(v: V): Unit
```

This operator is defined for signals only. It sets the current value to `v`, overwriting the current value. The compiler also lets us write `self()= x` instead of `self.update(x)`, as we have seen above.

5.2 Reactive combinators as imperative data-flow programs

Given our new imperative data-flow language, clients can now implement reactive combinators without intimate knowledge about the implementation details of Scala.React and without reverting to low-level techniques such as observers and inversion of control. Our data-flow language hides those details from them. To exercise the expressiveness of our data flow language, we will show how we can implement the most important built-in combinators in class `Events[A]` that are not trivially implemented in terms of other combinators. Event though most combinators are implemented differently for efficiency reasons, notice how we can easily read semantic subtleties from the data flow formulations such as when a combinator starts acting on its inputs.

The following `collect` combinator can be used to implement other combinators:

```
def collect[B](p: PartialFunction[A, B]) =
  Events.loop[B] { self =>
    val x = self await outer
    if (p isDefinedAt x) self << p(x)
    self.pause
  }
```

The resulting event stream emits those events from the original stream applied to partial function `p` for which `p` is defined. A `PartialFunction` can be written as a series of case clauses as in a pattern match expression. We refer the enclosing stream as `outer`. Notice that we use `await` and not `awaitNext`, i.e., the combinator processes events immediately, starting with the current one from `outer` if present.

Combinators `map` and `filter` can now 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 }
```

Combinator `hold` creates a signal that continuously holds the previous value that the event stream (**`this`**) emitted:

```
def hold(init: A): Signal[A] = Signal.loop(init) { self =>
  self << (self await outer)
  self.pause
}
```

Again, by using `await`, we are potentially dropping the initial value if `outer` is currently emitting.

Combinator `switch` creates a signal that behaves like given signal `before` until the receiver stream emits an event. From that point on, it behaves like given signal `after`:

```
def switch[A](before: Signal[A],
              after: =>Signal[A]): Signal[A] =
  Signal.flow(before.now) { self =>
    abortOn(outer) {
      self << (self await before)
      self.pause
    }
    val then = after
    while(!then.isDisposed) {
      self << (self await then)
      self.pause
    }
  }
```

We continuously emit values from stream `before` until `outer` aborts it. We then evaluate the call-by-name argument `after`, denoted by the arrow notation `=>Signal[A]`, and continue to emit events from that stream.

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

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

The use of `Events.flow` ensures that the resulting event stream does not take part in event propagation anymore, once it has emitted `n` events. A `drop` combinator can be implemented in a similar fashion. Operator `scan` continuously applies a binary function `op` to values from an event stream and an accumulator and emits the result in a new stream:

```
def scan[B](init: B)(op: (B,A)=>B): Events[B] = {
  var acc = init
  Events.loop[B] { self =>
    val x = self await outer
    acc = op(acc, x)
    self << acc
    self.pause
  }
}
```

Trait `Events[A]` contains two flatten combinators, which are defined for events of events and events 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 for nested events, and similarly for signals, as follows:

```
def flatten[B](implicit isEvents: A => Events[B]) = {
```

```
  var inner: Events[B] = Events.Never
  Events.loop[B] { self =>
    val es = loopUntil(outer) {
      self << (self await inner)
      self.pause
    }
    inner = isEvents(es)
  }
}
```

It repeatedly waits for values in the current inner stream until the outer stream emits. Once that happens, the new inner stream is stored and the process starts again. We use the implicit argument as a witness to ensure that `flatten` can only be called for which emitted values of type `A` are indeed of type `Events[B]`. Implicits arguments of this kind are automatically supplied by the compiler. The inner stream variable is initialized with `Event.Never` which creates a stream that never emits.

The `merge` combinator can be implemented as follows:

```
def merge[B >: A](that: Events[B]): Events[B] =
  Events.flow[B] { self =>
    par {
      while(!that.isDiposed) {
        self << (self await that)
        self.pause
      }
    } {
      while(!outer.isDiposed) {
        self << (self await outer)
        self.pause
      }
    }
  }
```

The branches are switched because events from `outer` take precedence. Each branch terminates once the corresponding event stream is disposed. As a result and because of the absence of a `join`, the `par` statement returns as soon as both the outer the parameter streams have been disposed, disposing the resulting stream.

6. Routers: imperative event coordination

So far, we have been dealing with abstractions that specify their dependencies at the point of creation. Whether through combinator-based composition, signal expressions or flow reactives, once we create a new reactive, all its dependencies are fully specified by its implementation. In contrast to observer-based event handling, where clients establish data flow in different places through side-effects, data flow in reactive programming is more localized and visible, as argued in more detail by Courtney in [16]. Even though implementations are not necessarily functional, e.g., in the context of flow reactives, dependencies are specified in a functional way. This is not a necessity, however, for Courtney's argument to remain true. We will demonstrate this at the example of another reactive abstraction in Scala.React, the *router*.

Consider the task of routing mouse clicks through a control hierarchy in a GUI framework. A mouse click arriving from the system is usually dispatched to the control closest to the leafs but still containing the coordinates of the click. A functional way to model this is to let every control contain an event stream of mouse events, filtering the system event stream based on the control's bounds. However, this makes dispatching an $\Theta(n)$ operation, with n being the number of controls in the UI, when it could be an $O(d)$ operation, with d being the maximum depth of the control hierarchy, amortized usually $O(\log(n))$. The problem with the given functional implementation is that every filtered event stream has only local knowledge, when there is the global knowledge that there will be precisely one control receiving the mouse event.

We can simplify this problem to a demultiplexer, i.e., routing a single source event to one of n outputs based on the event value. A domain where demultiplexers are common are reactive trading systems, where an event stream of market data is often partitioned and sent to different targets, depending on the type of security, for example. A functional reactive demultiplexer implementation would look as follows:

```
val source = EventSource[Int]
val outs: Array[Events[Int]] = Array.tabulate(n) { i =>
  source filter { x => x == i }
}
```

This has the aforementioned problem of an $\Theta(n)$ running time. Using a Scala.React router, we can implement it as an $O(1)$ operation:

```
val source = EventSource[Int]
val router = new Router {
  val outs = Array.tabulate(n) { i => EventSource[Int] }
  def react() {
    val x = self await source
    outs(x) = x
    self.pause
  }
}
```

This implementation of a router uses the data flow DSL to wait for the `source` to emit and then redirects the value to the correct output.

Routers are not only a way to implement certain data dependencies more efficiently, they are also useful for programmers that are more familiar with an imperative programming style. As such, they can be considered a bridge between observer-based programming and a more declarative reactive programming style that establishes dependencies in a functional way.

7. Implementation

Scala.React proceeds in discrete steps, called *(propagation) turns*. It is driven by two customizable core compo-

nents, the propagator and the scheduler. Both together form a domain which exposes the public API. External updates, such as a signal assignment, are forwarded to the thread-safe scheduler interface which schedules a revalidation request for the next turn. The scheduler is also responsible for initiating a turn when one or more requests are available. The precise details of when and how this happens is up to the respective scheduler implementation.

Once a turn is initiated by the scheduler, the propagator takes over, collects all requests, and validates all affected reactives. When done, it returns control to the scheduler, which usually performs some cleanup and waits for further incoming requests. This design allows us to hook into existing frameworks, which can make certain assumptions in which context client code is run. For example, Java Swing has a single thread policy, requiring (almost) all code to be run on a single event dispatcher thread. This thread continuously consumes messages from a system-wide event queue. We can let our propagator run on this thread by implementing a domain as follows:

```
object SwingDomain extends Domain {
  def schedule(op: =>Unit) =
    SwingUtilities.invokeLater(new Runnable {
      def run() { op }
    })
}
```

All classes from Scala.React that we have discussed so far are defined inside trait `Domain`, which is used as a module. Singleton object `SwingDomain` is an instance of that module and implements abstract method `schedule` which is invoked by the reactive engine to schedule new cycles and inject external updates. In this example, it pushes updates to the Swing event queue.

7.1 Change Propagation

Our change propagation implementation uses a push-based approach based on a topologically ordered dependency graph. When a propagation turn starts, the propagator puts all nodes that have been invalidated since the last turn into a priority queue which is sorted according to the topological order, briefly *level*, of the nodes. The propagator dequeues the node on the lowest level and validates it, potentially changing its state and putting its dependent nodes, which are on greater levels, on the queue. The propagator repeats this step until the queue is empty, always keeping track of the current level, which becomes important for level mismatches below. For correctly ordered graphs, this process monotonically proceeds to greater levels, thus ensuring data consistency, i.e., the absence of glitches. The basics of this implementation are similar to the one of FrTime and Flapjax, described in more detail in [13]. Our implementation differs in two important aspects. First, our implementation, which in contrast to FrTime or Flapjax, deals with every language construct uniformly, needs to deal with graphs whose topology can change in unpredictable ways. Second, we also support lazy validation of nodes, which changes the algorithm from above. We will discuss theses changes in detail below.

7.2 Strict vs Lazy Nodes

The above description of our propagation algorithm is only true for nodes that are strictly validated when notified of a change. We call those nodes strict nodes. A strict node can avoid propagating to its dependents if a change in a dependency does not necessarily result in the strict node to emit. Examples of such nodes are events created by `Events.filter` or signum signal `Strict{ if(x() >= 0) 1 else -1 }`. A further use case for strict nodes are stateful reactives, such as event streams created by `Events.count` or `Events.scan`. Such a reactive updates its internal state independently of whether another node is currently interested in it or not. Therefore, `Event.counts` creates an event stream that counts all events from the input stream. If it would validate only when queried, it would potentially miss some events. We find such semantics surprising even though some such as the event library for F# from [38] or EScala [26], implement stateful reactives this way.

Lazy nodes are notified by their dependencies eagerly, but validated only when queried. Examples of such nodes are events created by `Events.map` or `Events.merge`. Their semantics would be the same if they were strict nodes, since they neither accumulate state nor drop events based on their value, i.e., they do not have to evaluate their current event strictly. In contrast to strict nodes, lazy nodes can be used to create subgraphs that are evaluated only on demand. Lazy nodes take a slightly different path through the propagator than strict nodes. While a strict node always ends up on the queue when notified, a lazy node simply sets an internal bit that indicates that it has been invalidated if not set yet. The node is not inserted into the queue for strict evaluation, but always notifies its dependents when it gets invalidated, which happens at most once per turn. Since they evade the propagation queue, lazy nodes have a performance advantage over strict nodes as we will see in Section 9. Their disadvantage over strict nodes is that they have to notify their dependents regardless of whether they will actually emit or not, since they cannot determine their current value until evaluated. This is not an issue for some event combinators as mentioned above, but for expression signals, clients can choose between lazy and strict signals by using either the `Strict {...}` or `Lazy {...}` constructor. Note that we are using the standard `Signal{...}` constructor throughout the paper to indicate when details we discuss are independent of the evaluation strategy. Clients can override it to create lazy or strict signals per default.

Flow reactives are always strict nodes, since the framework generally needs to assume that they evaluate code with side-effects in reaction to an event. Consider the data-flow implementation of `Events.take` from Section 5.2 for an example.

7.3 Signal Expressions and Opaque Nodes

A signal of the form `Signal { a() + b() }` creates a graph node that captures its dependencies inside a closure. In contrast to other systems, we neither use an explicit lifting mechanism that lifts regular functions, such as the + operator in the example to the signal domain such as in Flapjax or Fran [19, 33] nor do we use a macro system to perform lifting automatically as in FrTime [13]. These previous implementations have in common that every language construct, such as function application, conditional expressions, loops and exceptions, needs to be treated specially before they can be used in a reactive system. We use an approach instead, that is completely agnostic of any language feature, which is extremely valuable in an evolving language which also supports user definable control flow operators.

In Scala.React, signals created with a closure as above, are completely opaque to the framework. It does not know anything about the signal's dependency structure in advance. The same is true for flow reactives or reactors. Such nodes are flagged as *opaque*. Nodes that are not opaque are generally those created using combinators such as `map` or `filter`.

When constructing signal `Signal { a() + b() }`, we are in fact calling method

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

with argument `{ a() + b() }`. The Scala compiler rewrites expressions `a()` and `b()` to method calls `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. It maintains a stack of dependent reactives that are used to establish dependencies. A signal s is either valid or has been invalidated in the current or a past propagation turn. If s is valid, `Signal.apply` takes the topmost reactive from the dependents stack without removing it, adds it to its set of dependents and returns its own current valid value. If s is invalid, it additionally pushes itself onto the dependent stack, evaluates the captured signal expression, and pops itself from the stack before returning its current value. This way, signals further downstream will add s to their dependencies.

For some language expressions, such as conditional operations, dependencies are dynamic. Consider signal

```
val s = Signal { if(c()) a() else b() }
```

Signal `s` should ideally ever depend only on two signals: on signal `c` and also on signal `a` if `c` is `true`, otherwise on `c` and `b`. Therefore, when a node notifies its dependents of a change, it removes all dependents that are flagged as opaque. In the example, once `a` notifies `s` of a change, it removes `s` from its dependent set. If, in the meanwhile, `c` has changed to `false`, `s` will be added to `b` but not to `a` again, since it does not evaluate `a.apply` in this case. Note that this is a slight compromise to the ideal case. If `c` changes, `s` is not automatically removed from `a` or `b`, but only next time `a` or `b` change. Dependency tracking for a dynamic opaque node can hence over-approximate until the next change occurs in a node that it logically does not depend on anymore. This approach, however, works with any control flow operation, either built-in or user-defined. We therefore trade a bit of redundant computation for the benefits of a single general implementation.

The technique we describe here is somewhat related to lowering for FrTime as described by Burchett in [9]. In contrast to Burchett's approach, we use it as our principal composition mechanism at runtime and not as an optimization pass at compile time after all dependencies are known from a macro-based lifting. Moreover, we keep track of dynamic dependencies, whereas in [9], a lowered signal depends on the union of its *potential* statically known dependencies, i.e., the conditional signal from above would depend on `a`, `b` and `c` at any time. Most importantly, our implementation is general, whereas lowering needs to be adapted for all kinds of language expressions. For example, in contrast to lowering, our approach handles runtime dependencies and higher-order programming without further adaptation. Consider the following function, which creates the reactive sum of a runtime list of integer signals:

```
def sum(xs: List[Signal[Int]]) = Signal {
  xs.foldLeft(0) { (sig, sum) => sig() + sum }
}
```

This cannot be lowered with the approach from [9] because the lowering operator needs to know all of the resulting dependencies., which is not possible in the example. The same is true for virtual methods for which the optimizer cannot predict the signal they return. Also consider the following example, dealing with higher-order signals:

```
class RPoint { val x, y: Signal[Int] }
val p1, p2: Signal[RPoint] = ...
val x = Signal {
  if(p1() != null) p1().x() else p2().x()
}
```

This chooses a coordinate signal from one of two points, based on a standard null pointer check. If we would need to evaluate the branches in the example eagerly in order to determine all of `x`'s potential dependencies, we would need to be careful to avoid a null pointer exception.

We find it therefore important to preserve the evaluation order of different language constructs, such as conditional evaluation in if expressions or short circuit semantics in boolean operations. A final consideration is that with our approach the creation of reactive nodes is transparent to the programmer, whereas when performed as an optimization, the number of nodes created is less predictable.

7.4 The imperative data-flow language

Our data-flow DSL is implemented in terms of Scala's delimited continuations [41]. The core CPS functionality is implemented in trait `FlowReactive`:

```
trait FlowReactive[P, V] extends Reactive[P, V] {
  private var _continue =
    () => reset { body(); doDispose() }

  def body()
  ...
}
```

A `FlowReactive` keeps track of its internal program counter by maintaining a reference to its continuation. Note that syntax `() => e` defines a closure with zero parameters and body `e`. The initial continuation runs method `body` which contains a flow reactive's implementation. While running, the body is capturing further continuations which are delimited by the surrounding call to `reset` from Scala's continuation support library. The initial continuation finally performs some cleanup, such as removing all dependents.

Trait `FlowReactive` also defines three helper methods, which are used to implement our data flow operators. Method `shiftAndContinue` captures the current continuation, transforms it, essentially inserting code between the current and the next expression, stores the continuation and then runs it. Storing it before running it is necessary because of potential level mismatches, which we will discuss in Section 7.6.

```
def shiftAndContinue[A](body: (A => Unit) => Unit) =
  shift { (k: A => Unit) =>
     _continue = () => body(k)
     _continue()
  }
```

Method `shift`, like `reset` from above, is part of the continuation library in Scala. It captures the current continuation `k`, passing it to the supplied function, which in this case stores and runs a transformed version of `k`.

Method `continueLater` captures a given continuation `k` and tells the framework to evaluate the current flow reactive in next turn. Evaluating a flow reactive simply means running the previously stored continuation:

```
def continueLater(k: =>Unit) = {
  _continue = () => k
  evalNextTurn(this)
}
```

Method `continueWith` runs a given continuation until the next suspension point and returns the remaining continuation. It stores a given function as a continuation, runs it and returns the remaining continuation. Note that the continuation itself usually modifies the stored continuation, so the resulting continuation will be different from the supplied function.

```
def continueWith(k: => Unit): (()=>Unit) = {
  _continue = () => k
  _continue()
  _continue
}
```

Our data flow operators are implemented in terms of these methods. Method `pause`, which suspends until in the next turn, simply captures and passes the current continuation to `continueLater`:

```
def pause = shift { (k: Unit => Unit) =>
  continueLater { k() }
}
```

Method `halt`, which stops the flow reactive entirely, captures the current continuation, throws it away, and performs some cleanup, such as removing all dependents:

```
def halt = shift { (k: Unit => Unit) => doDispose() }
```

Method `await` uses `shiftAndContinue` to insert code that checks whether its `input` is emitting and if so, continues with the value from the input. Otherwise, it (re)subscribes the flow reactive to the given input and suspends, waiting to get notified by the `input`.

```
def await[B](input: Reactive[B, Any]): B =
  shiftAndContinue[B] { k =>
    input.ifEmittingElse { p => k(p) }
                         { input subscribe this }
  }
```

Method `ifEmittingElse {then} {els}` runs the `then` branch in reaction to an emitted value or the `els` branch otherwise. Since a flow reactive is opaque and does not manage its dependencies on its own, it gets unsubscribed when notified by `input`. Moreover, as lazy nodes can notify their dependencies even though they might not emit, we have to repeatedly call `subscribe` inside the closure passed to `shiftAndContinue`. This approach ensures that our flow reactive is always unsubscribed after `await` has returned.

More complicated control flow can be implemented in terms of operators `par` and `join`. Method `join` replaces the current continuation with a function stored in a separate variable, which becomes important in the implementation of `par`.

```
var _join = () => ()
def join = shiftAndContinue[Unit] { k => _join() }
```

Method `par` repeatedly runs (and resumes in another turn after being suspended) two branches in succession, until both are finished evaluating, or returns once a branch calls `join`. We introduce a counter variable `latch` that we use to take care of these two return condition:

```
def par(left: => Unit)(right: => Unit): Unit =
  shiftAndContinue[Unit] { exitK =>
    var latch = 2
    val doJoin = { () => latch = 0 }
```

```
    def evalBranches(left: => Unit)(right: => Unit) ...

    evalBranches { reset { left;  latch -= 1 } }
                 { reset { right; latch -= 1 } }
}
```

We capture the continuation after the `par` statement, called `exitK`. We introduce a fresh latch variable that indicates how many branches have finished. We set the continuation variable used by `join` to a closure that sets `latch` to zero, indicating that both branches are forced to terminate. We then wrap both branches in separate `reset` calls to delimit the continuation scope in the branches. We also count down the `latch` variable at the end of each branch to indicate that they have finished. Evaluation of the branches is handled by local function `evalBranches`, which is defined as follows:

```
def evalBranches(left: => Unit)(right: => Unit) {
  val oldJoin = _join
  _join = doJoin
  val leftK = continueWith(left)
  if (latch > 0) {
    val rightK = continueWith(right)
    _join = oldJoin
    if (latch > 0) {
      _continue =
        () => evalBranches { leftK() } { rightK() }
    } else exitK()
  } else exitK()
}
```

In order to be able to deal with nested `par` statements, we first replace the old join closure with the one that captures the correct `latch` variable. Then we evaluate the left branch and if `latch` indicates that we should continue, we evaluate the second branch. For each branch, we save the remaining continuations in `leftK` and `rightK`. It is important that wrap each branch in a `reset` call above so they are delimited correctly and don't capture anything beyond the `par` statement. Then we reinstall the old join closure for potential `join` calls in a surrounding `par` statement to work correctly. If the current `par` statement hasn't terminated yet, we store a continuation that evaluates the rest of the branches. If we encounter that the `par` statement should terminate, we call `exitK`, the continuation starting after the `par` statement.

Other high-level data-flow expressions can be written in terms of the above operators. Loop operator `loopEndUntil` that loops until a given reactive emits, but always finishes the loop body first, can be implemented as follows:

```
def loopEndUntil[A](r: Reactive[A, Any])
                   (body: =>Unit): A = {
  var done = false
  par { await(r); done = true }
      { while (!done) body }
  r.getPulse
}
```

One branch waits for the input to emit and then sets variable `done` which is used as the loop condition in the second branch.

Method `abortOn` is a variation of this, but instead of maintaining a condition variable, we can use the `join` operator to abort the second branch immediately.

```
def abortOn[A](input: Reactive[A, Any])
              (body: =>Unit): Unit =
  par { await(input); join }
      { body; join }
```

Note that the order of branches for both `loopEndUntil` and `abortOn` determine a subtle semantic detail. The given order makes sure that the abortion condition holds in the same turn in which the input emitted. If the branches were exchanged, `loopEndUntil` might still start the body once more and `abortOn` might still run a slice of the loop body in the same turn in which the input emitted.

Now, we can implement `loopUntil` either using `par` directly, or use `abortOn` to abort the loop immediately:

```
def loopUntil[A](r: Reactive[A, Any])
                (body: =>Unit): A = {
  abortOn(r) { while (true) body }
  r.getPulse
}
```

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

7.5 Routers

A router ends up as a dependency node in the graph, one level higher than the maximum of its dependencies. They are managed the same way as other reactives. A source reactive such as an `EventSource` or a `Var` that is created inside a router is not on level 0 as usual, but on a level above the router. Source reactive constructors know where they are created by using an implicit argument. Method `Var`, for instance, actually accepts an additional implicit argument `owner` that is automatically passed in by the compiler when one is in scope. The default argument specifies the domain object as the owner. Inside a router, however, the router overwrites the implicit owner with itself, which creates the correct dependency structure.

7.6 Level Mismatches

Opaque nodes, such as expression signals or flow reactives, for which the framework does not know the dependency structure, can access arbitrary dependencies during their evaluation which can change from turn to turn. We therefore need to prepare for an opaque node n to access another node that is on a higher topological level. Every node that is *read* from during n's evaluation, first checks whether the current propagation level which is maintained by the propagator is greater than the node's level. If it is, it proceed

as usual, otherwise it throws a level mismatch exception containing a reference to itself, which is caught only in the main propagation loop. The propagator then hoists n by first changing its level to a level above the node which threw the exception, reinserting n into the propagation queue (since it's level has changed) for later evaluation in the same turn and then transitively hoisting all of n's dependents.

For simple expression signals, this approach can result in redundant evaluation since a signal is evaluated from the beginning. The following signal can therefore increase counter `i` twice when the level of the given signal `sig` changes:

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

Expression signals should therefore never perform external side-effects.

For flow reactives, our CPS-based implementation ensures that the current continuation is always captured before a level mismatch can happen. When reevaluated later in the same turn, they continue at the point where a mismatch exception was thrown. Therefore, no branch is evaluated redundantly. Without this important invariant, we wouldn't be able to use local variables inside flow reactives. We can therefore safely rewrite the above signal to

```
def countChanges(sig: Signal[Any]) = {
  var i = 0
  Signal.loop(0) { self =>
    self awaitNext sig()
    i += 1
    self()= i
  }
}
```

For reactors, as their purpose is to perform non-local side-effects, it is important that their evaluation never aborts. Since reactors cannot have dependents, they can always have maximum topological level, so this requirement is easily satisfied.

We could use continuations for signal expressions as well. When discovering a topological 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 topological mismatch was discovered, captured in the continuation closure. Unfortunately, this approach is too heavyweight on a runtime without native CPS support.

7.7 Avoiding a memory leak potential of observers

Internally, Scala.React's change notification is implemented in terms of observers. We do expose them to clients as a very lightweight way to react to changes 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 to print out all changes in event stream `es`:

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

This usually leads to a reference pattern as shown in Figure 1. The critical reference path that goes from the reactive

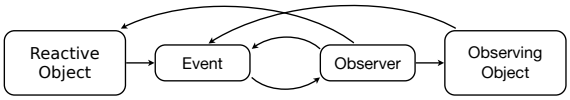

Figure 1: Common observer reference pattern.

object to the observing object is often not visible, since observing objects typically abstract from their precise dependencies. For every observing object we want to dispose before its dependencies are garbage collected, we would need to switch to explicit memory management. This constitutes a common potential for memory leaks in observer-based programming [8] and is an instance of the issue of explicit resource management we identified in the introduction. A common solution is to break all reference cycles by replacing the strong reference from event to observer by a weak one as depicted in Figure 2. This only partially solves the

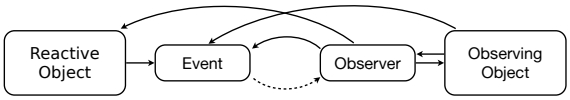

Figure 2: Reference pattern in Scala.react

problem, though. It is now the client's responsibility to establish a strong reference from the observing object. Failing to do so can lead to a premature reclamation of the observer.

We propose a novel implementation based on Scala's traits that moves this responsibility into the framework. Any observing object needs to mix in the following `Observing` trait. It is important to note that clients have no other possibility to create observers so we guarantee that no object involved is disposed prematurely.

```
trait Observing {
  private val obRefs = new Set[Observer]

  abstract class Observer extends react.Observer {
    obRefs += this
    override def dispose() {
      super.dispose()
      obRefs -= this
    }
  }

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

Method `observe` actually creates an observer that automatically adds itself to an observer set of the enclosing object during constructionMethod `dispose` is still supported, but must also remove the observer reference from the enclosing object. Instead of using `foreach`, we can now write the following:

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

An instance of class `View` can now be automatically collected by the garbage collector once clients do not hold any references to it anymore, independently from the given event stream and without manually uninstalling its observer.

7.7.1 Weak Forward References

In fact, all forward references, i.e., from nodes to their dependents, are held in weak references. Recreating them in dynamic graphs was a major source of garbage in previous implementations of Scala.React. Currently, we ensure that there is only ever at most a single weak reference object to any node in the dependency graph. Note that we cannot use a HashMap for pooling weak references, since a HashMap creates additional garbage, defeating our goal. Instead, every node that can have dependencies stores a reference to a weak reference to itself when needed.

This reduces the number of objects for nodes with multiple dependencies for static graphs. More importantly, it largely reduces the garbage created in dynamic graphs with opaque nodes and also reduces the work that needs to be done by the weak reference support infrastructure.

8. Further Examples

8.1 Unifying blocking and non-blocking APIs

Some frameworks provide two different APIs for similar functionality, one blocking and one non-blocking. Usually, while the blocking interface is a simple blocking function call, the non-blocking interface relies on callbacks. In Java Swing, e.g., dialog windows can be modal or modeless. Modal dialogs block the event dispatcher of the whole application, until they are closed. Modeless dialogs process events concurrently. Converting between modal and modeless dialogs during development is a cumbersome task, since the code dealing with user input has to be changed. Using our imperative data flow language, this problem can be eliminated completely. To open a simple dialog that expects a positive or negative answer, we can define a common method `ask` that expects a `self` reference as in a flow reactive to obtain a handle to the data flow DSL and a message string. A blocking implementation simply uses a standard Swing option pane:

```
def ask(self: DataflowOps, msg: String) =
  JOptionPane.showInputDialog(null, msg,
              JOptionPane.YES_NO_OPTION)
```

whereas the non-blocking version uses a method `openDialog` that redirects answers in a modeless Swing dialog to an event stream:

```
def ask(self: DataflowOps, msg: String) =
  self await openDialog(msg)

def openDialog(msg: String): Events[String] = {
  val dialog = new JDialog(null: JFrame, msg)
  val es = EventSource[String]
  // bind es with Swing listeners
  es
}
```

We can now simply write

```
Reactor { self =>
  val answer = ask(self, "Ready␣to␣submit␣request?")
  ...
}
```

and switch between blocking and non-blocking implementations without changing client code.

8.2 Reactive Painting

We can use the dynamic dependency creation using dependent stacks in contexts other than signal expressions. Consider the task of painting a GUI control whose visual representation depends on a number of signals. An example is the fish eye demo, based on an example from the FrTime distribution. It paints a grid of dots that change their size depending on their proximity to the mouse pointer. The size of the grid, as well as the distance between dots if configurable, represented by integer signals. The size of each dot is represented by a two dimensional array of integer signals. Our Java Swing drawing code simply looks as follows:

```
def draw(g: Graphics2D) {
  val d = dist()
  val g = gridSize()
  for (x <- 0 until g) {
    for (y <- 0 until g) {
      val size = sizes(x)(y)
      val s = size()
      g.fillOval(x * d - s/2, y * d - s/2, s, s)
    }
  }
}
```

This draws the grid and automatically establishes dependencies to signals `dist`, `gridSize` and each `size` signal. Note that the `size` signal can change when the grid size changes. Dependencies are automatically adapted, since we will iterate over different `size` signals when grid size change.

Method `draw` is an implementation of a template method in a custom Swing component class `RCanvas`. `RCanvas` wraps calls to `draw` in a call to `observeDynamic` from trait `Observing`. It is a variant of the `observe` method we have seen, but establishes dependencies using the dependent stack instead of specifying them explicitly. A dynamic observer created this

way works like an expression signal but like all observers are leaf nodes that never abort, so it safe to call drawing code when evaluating.

8.3 Calculator

An FRP implementation of a simple calculator is one of the standard demos that can be found on the Flapjax website[4]. The event handling logic of the Flapjax example is non-trivial, using 16 reactive combinators, and creates as many reactive nodes, some of which are higher-order. The corresponding event handling code for a calculator in idiomatic Scala.React is as follows:

```
val digits: Events[Int] = ...
val ops: Events[Int]
val operators: Array[Double=>Double=>Double] = ...
val display: Signal[Double] = {
  var f = { x:Double => x }
  Signal.loop(0.0) { self =>
    val op = self.loopUntil(ops) {
      self()= 10*self.previos + (self await digits)
    }
    f = operators(op)(self.previos)
    self()= f(now)
    self.loopUntil(digits) {
      f = operators(self awaitNext ops)(self.previos)
    }
  }
}
```

Entered digits arrive at stream `digits`. Binary operators are stored as curried functions in array `ops`, with operator inputs arriving at `ops` indexing into the array. The `display` signal holds the contents of the display. It starts by receiving digits updating the display until an operator is entered. Then it stores that operator in an internal function and proceeds receiving operator inputs until the next digit is received. This code is equally dense to the Flapjax implementation and less than half of its size. Its imperative nature directly reflects the state machine character of this example and creates only a single reactive node at runtime.

9. Evaluation and Discussion

We will now show further examples and benchmark results. As it is not very meaningful to compare absolute performance across languages and platforms, we show how our examples perform relative to a baseline implementation in terms of observers on the same platform. This gives us an indication – and by no means evidence – how viable in terms of performance a reactive implementation is compared to an observer-based one. All measurements were done on a MacbookPro Late 2011, i7 2.4 Ghz, 8GB RAM on Mac OS X 10.7 with Scala 2.9.1 (with optimizations enabled) on Oracle's Java 1.6 update 31. Flapjax examples were run in Chrome 15.0 with the latest Flapjax version as of April 7th, 2012 from [39], optimized with Google's closure compiler. All numbers are steady state performance after a proper number of warmup iterations. We follow the general guideline and measure across multiple invocations of the VM [27].

[4] `http://www.flapjax-lang.org/try/index.html?edit=calc.html`

9.1 Reactive Chains and Fans

This simple benchmark measures the time it takes to propagate a single event along a linear chain and a fan of reactives compared to a very simple observer implementation. The reactive dependency graphs are depicted in Figure 3. This test gives an indication of the overhead of the different lifting mechanisms and how much time the framework spends to ensure data consistency. We implement the same functionality for EScala [26], Scala.React and Flapjax [33]. For the observer implementation, we have implemented a basic `Observable` class that stores registered observers in an array and notifies them when changed. For the linear chain, one observable always notifies a single other observable. For the fan, one observable notifies n other observables, which then notify one other observable each that accumulates the result. For EScala, which supports first-class events but no signals, we derive events from a source using the `map` combinator. We run 4 variants for Scala.React, composing events with `map`, which returns a lazy reactive, with `collect`, which returns a strict reactive, and finally composing signals using the `Lazy` and `Strict` constructors from Section 7.3. We run 2 variants for Flapjax, one using events composed with `map` and one using behaviors (Flapjax's time-varying value abstraction) using Flapjax's built-in lifting mechanism.

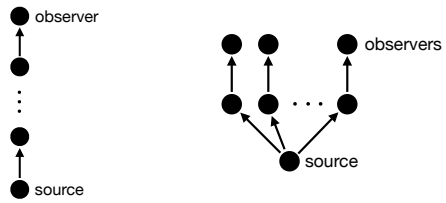

Figure 3: Dependency graphs for reactive chains and fans.

9.1.1 Results

Results can be found in Figures 4 and 5. We are mostly interested in relative performance as shown in Figure 5 but include absolute numbers for reference. Naturally, the observer-based implementation beats every other implementation on the same platform by quite some margin.

Even though we deal with one more level of indirection in Scala.React compared to Flapjax or EScala, by using weak forward references, our implementations compared to the corresponding implementations in Flapjax and EScala perform well. In EScala, the task to ensure data consistency is fundamentally easier, since it only supports events and no higher-order combinators, no dynamic graphs and all depen-

dencies are known in advance. Yet, our general implementation is slightly faster for event streams.

Performance for signals in Scala.React is worse than using event combinators due to the use of a dependent stack and continuous resubscription as discussed in Section 7.3. Fortunately, the overhead seems negligible for lazy signals. By contrast, strict signals have a fairly high overhead, we assume due to the fact that they evaluate the paths through the graph piecewise, instead of in one large call chain initiated by the installed observers that accumulate the result. We admit, however, that it is difficult to pin down the exact reason due to runtime optimizations by the VM.

Our corresponding propagation and lifting mechanisms perform well when compared to Flapjax. Scala.React is about factor 3 faster for events chains, and from factor 5 to about factor 7 for signal chains (both in terms of absolute and relative performance). For fans, Scala.React is relatively faster by factor 4 to 10.

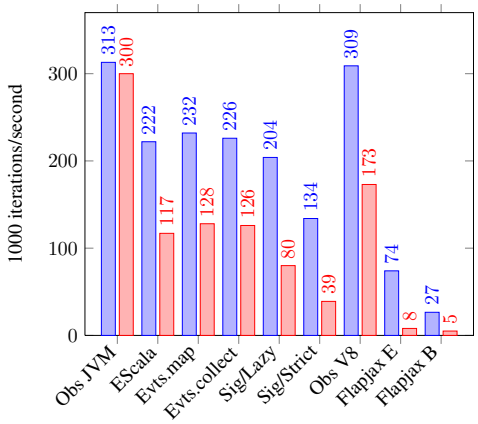

Figure 4: Propagation performance of a single event through a linear chain of 100 nodes (blue/left) or a fan with a fanout of 100 (red/right) in 1000 events propagated per second.

9.2 Dragging and General Sequencing

We compare the performance of our imperative dataflow implementation to an alternative implementation using an FRP-style combinator approach, which can be found in [33]. In Scala.React, an equivalent FRP implementation looks as follows.

```
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
```

This creates two event streams `moves` and `drops` of type `Events[Events[T]]` with `T` being a base type of `Drag` and `Drop`. We then merge the two streams and use our previously defined `flatten` combinator to switch between them[5]. Once the

[5] In some FRP implementations `flatten` is called `switch`

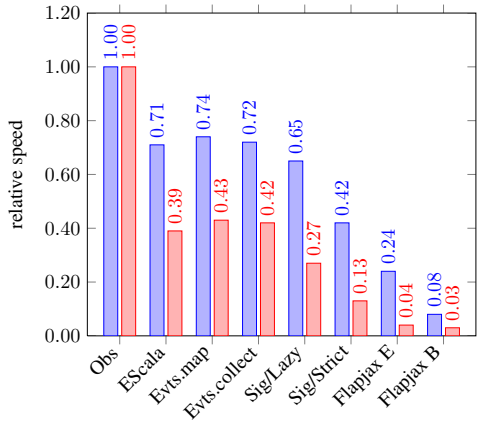

Figure 5: Relative performance to the respective observer-based implementation from Figure 4

mouse up event is received, the singleton stream `Events.Now`, which emits a single event immediately, shuts down the mouse move events in the flattened result.

One problem with this FRP dragging implementation is the repeated creation of inner event streams in the `moves` stream. Even though event streams created by the `map` combinator are lazy in Scala.React, and `flatten` makes sure that it only ever reacts to changes in the *current* inner stream, every mouse down event nevertheless creates one new dependent node for the `mouseMove` stream. Therefore, for every mouse down event, there will be more work to do, until the garbage collector decides to collect those dangling dependents. Note that in contrast to Flapjax, in Scala.React the garbage collector can indeed collect those dependents because the are only weakly referenced by `mouseMove`, as discussed in Section 7. This problem is actually an amplification of the problem of explicit resource management with observers that we enumerated in the introduction.

In the present case, we can improve, yet not entirely fix the code by pulling out the instantiation of the inner event stream:

```
val inner = mouseMove map { mm => new Drag(mm) }
val moves = mouseDown map { md => inner }
...
```

Unfortunately, we still have a dependent for all mouse moves, not only those between pairs of down and up events. Moreover, in case the inner stream would depend on the event `md` from the outer stream, pulling out the inner stream is not possible without introducing external mutable variables. For example, if we want drag event coordinates to be relative to the mouse down coordinates, our code would look as follows:

```
var pos0 = 0
val inner = mouseMove map { mm =>
  new Drag(pos0 - mm.pos)
```

```
}
val moves = mouseDown map { md =>
  pos0 = md.pos
  inner
}
...
```

If we want to stay purely functional, we would have to use a different set of combinators, potentially writing our own. We conclude that dealing with event sequences in a functional reactive way can become complicated. In particular, dealing with higher order event streams efficiently is an important issue that programmers need to be aware of.

Our data-flow formulation of the same problem, which is very similar to our running example and therefore won't repeat here, avoids this problem entirely without using higher order event streams. Dependents and internal callbacks are deactivated as soon as possible. Performance issues aside, we also argue that the source code is easier to understand, at least for programmers not used to a functional style.

One can discuss how valuable our measurements we present below are for a dragging example where the event load is inherently limited. Note, however, that this problem is just one instance of correlating a sequence of events, which is common in many other domains with higher event loads such as RFID tracking, fraud detection, patient monitoring or industrial control systems. We mainly focus on the dragging example because there is an FRP implementation available that was written by a third party.

9.2.1 Results

Results can be found in Figure 6. They give the relative running times in relation to a corresponding observer-based implementation on the same platform. One iteration simulates a mouse down, followed by 10 mouse moves, followed by 1 mouse up event, followed by n mouse move events, with n varying along the horizontal axis. This means we increase the number of events that should not be considered for a drag operation.

The original FRP implementation in Scala.React, repeatedly creating new dangling dependents improves over time in relation to the observer implementation. As the garbage collector keeps the number of those unnecessary dependents nearly constant, the absolute running time stays at a nearly constant low, relatively independent to the event load.

The two fastest implementations are the improved FRP and flow implementations in Scala.React approaching factor 0.7 for high event loads. For low event loads, the flow implementation is slightly slower because it creates more closures internally and repeatedly un- and resubscribes dependents. This overhead becomes less apparent for high event loads, also because the improved FRP implementation still maintains one unnecessary dependent between mouse up and down events.

For Flapjax, we only give results for an improved implementation, since the original version creates too many dangling references that are never disposed due to the lack of weak references in Javascript. The improved Flapjax implementation has a much higher overhead compared to the corresponding observer implementation than the two Scala.React implementations.

We do not provide results for EScala because it lacks support for higher-order event streams, imperative dataflow or similar facilities that would allow us to implement recognizing event sequences conveniently.

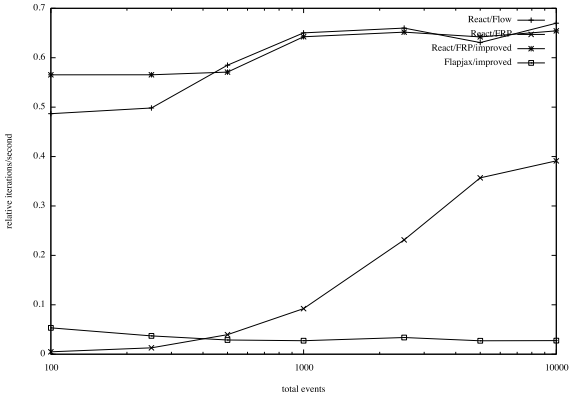

Figure 6: Dragging performance of different implementations relative to an observer-based implementation on the same platform.

9.3 Demultiplexer

Figure 7 shows that the router implementation of a demultiplexer has constant speed with respect to the number of outputs. The FRP implementations in Scala.React and EScala are generally slower even for smaller number of outputs. The EScala implementation runs faster than Scala.React using `filter`, since the filtering operation in EScala is lazy whereas in Scala.React, it is strict. We have discussed the benefits of a strict filter implementation in Section 7.2.

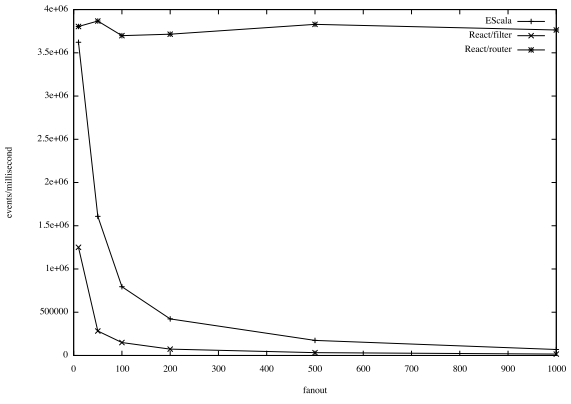

Figure 7: Demultiplexing a single event to a number of outputs.

10. Related Work

Scala.React combines ideas from imperative data-binding systems, functional reactive programming and synchronous data-flow languages [29].

Our imperative data-flow language is inspired by synchronous data-flow languages such as Signal [4], Lustre [28], and in particular Esterel [6]. Like Scala.React, they are glitch-free but unlike our work, they are used to implement realtime systems and lack features from general purpose languages so that they can be compiled to state machines. They work with built-in types only and lack higher-order features, which makes them less expressive than Scala.React. Colaço et al. [12] discuss some work towards a Lustre with higher-order features. Our data flow language kernel is similar to Esterel's, but implemented in terms of continuations in multi-valued domains. See [5] for an overview of different Esterel kernels.

Imperative data-binding systems such as JavaFX [36] or Adobe Flex [2] provide a way to bind expressions to variables. This is similar to programming with signals in Scala.React. Without a first-class notion of time-varying values and events, however, those systems often lack important abstraction mechanisms and need to revert to inversion of control for complex event logic. Our dependency management with dependent stacks is similar to JFace Data Binding [43] but does not use thread-locals since our turn-based scheduling makes sure that there is only a single propagation thread active at a time.

First-class signals and event streams and the combinator-based component of Scala.React originate from functional reactive programming (FRP) which goes back to Conal Elliott's Fran [20]. Earlier designs suffered from memory leaks and have since been revised [19]. Scala.React combines ideas from a variety of FRP implementations such as FrTime, Flapjax and Frappé [14, 15, 33] that differ in programming interfaces, evaluation models and semantics details. Our discrete, push-based evaluation model is similar to FrTime's [13], also used by Flapjax. Our approach shares some similarities with FrTime after a lowering optimization pass ([9]), as discussed in detail in Section 7.3. Unlike FrTime or Flapjax, however, Scala.React supports not only strict but also lazy evaluation of nodes. Some FRP implementations allow side-effects internally in order to integrate with existing imperative toolkits. Unlike any FRP system, Scala.React embraces imperative reactive programming in its API that allows clients to implement complex event logic without inversion of control or use of higher-order abstractions as discussed in Section 9. Unlike all FRP system known to us, Scala.React integrates with the full range of the host language's features *without specific adaptation* through special purpose combinators or compilation. This is particularly attractive in an evolving and extensible language such as Scala as elaborated in Section 7.3. Some FRP implementations, such as FrTime, use weak forward references similar to Scala.React. Our approach to bind the life time of observers to an observing object, however, is novel to our knowledge. F# has been extended in [38] with FRP-style and imperative data-flow programming abstractions. Unlike Scala.React, this work does not support time varying values or synchronous semantics. It defines a garbage collection scheme that is an alternative to our `Observing` trait approach and results in stateful reactive combinators such as `take`, `scan` or `hold` whose semantics vary depending on whether observers are installed or not.

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 [34, 44]. In contrast to FRP or Scala.React, it is not glitch-free.

Ptolemy [40] and EScala [26] support an event notion through a combination of implicit invocation (II) and aspect-oriented programming [42]. As opposed to our work, the original motivation behind II is to decouple system components by embracing inversion of control. In contrast to Scala.React, EScala is a language extension to Scala instead of a library, and blurs the aforementioned distinction by supporting basic built-in event combinators. Clients can explicitly trigger events but also use AOP point cuts to trigger events implicitly at different points in the control flow. Since EScala exclusively support events as reactive abstractions and all dependencies are static and known at compile-time, the task to ensure data consistency is fundamentally easier. It propagates in two phases. It first collects reactions for every node and then executes them. Moreover, EScala propagates lazily, i.e., as in F#'s event implementation [38] stateful combinators have different semantics depending on whether they are leaf nodes or not.

In contrast to Scala.React, neither Ptolemy or EScala support signals, imperative data-flow and rely on inversion of control for more complex event logic. For a deeper comparison of the systems in this research area, we refer to [26]. In many of the aforementioned OOP systems, such as EScala, Flapjax and Frappé, forward references are strong, i.e., they suffer from the problems we mention in Section 7.7.

SuperGlue [32] is a declarative object-oriented component language that supports a signal concept similar to that found in Scala.React. For signal connections, SuperGlue differs by following a declarative approach closer to constraint programming as in the Kaleidoscope language family [24]. For example, SuperGlue provides guarded connection rules and rule overriding which is a simple form of Kaleidoscope's constraint hierarchies [7].

Adaptive functional programming (AFP) [1] is an approach to incremental computation in ML. It is related to Scala.React as it also captures computation for repeated reevaluation in a graph. AFP has been ported to Haskell [10] using monads. Our CPS-based representation of data-flow programs is related to this effort and other monadic implementations because any expressible monad has an equivalent

formulation in continuation passing style [22, 23]. The AFP dependency graph and propagation algorithms are much more heavyweight than Scala.React's being based on time stamps and a selective replay mechanism.

Our imperative data-flow reactives share certain characteristics with complex event processing (CEP) systems such as TelegraphCQ [11], SASE [46], Cayuga [18]. These systems use custom query languages similar to SQL [17] to recognize event patterns from multiple sources. Queries can often be stateful, similar to imperative data-flow reactives, but are not designed for external side-effects but to compose complex events out of simpler ones. CEP systems are usually optimized for large-scale stream processing while our implementation is targeted towards general purpose event systems. EventJava [21] is a CEP system that extends Java with an event notion and declarative expressions for event correlation. EventJava's events are special methods that are either called directly or based on their enclosing type, i.e., the programming model is fundamentally different from Scala.React and embraces a certain degree of inversion of control. It further differs from our work in that it requires its own compiler and is stream-oriented and asynchronous. There are many more similar CEP or publish/subscribe systems that we cannot mention due to space reasons. They are usually implemented as a language extension and favor a certain programming paradigm and reactive abstraction, while Scala.React provides a wider range of abstractions that support reactive programming in multiple paradigms.

Our imperative data-flow reactives (and reactors) share certain similarities with actors [3, 30]. Actors run concurrently and communicate with each other through messages. In contrast to our reactors and reactives, actors are asynchronous. State transitions and data availability are synchronized among reactives, whereas actors behave as independent units of control, i.e., there is no notion of simultaneity and order of message arrival across actors is unspecified (apart from a notion of fairness). Actors communicate with each other directly, whereas reactives broadcast to a dynamic set of dependents.

11. Conclusion

We have demonstrated a new method backed by a set of library abstractions that allows transitioning from classical event handling with observers to reactive programming abstractions. The key idea is to integrate basic event handling with observers, function reactive programming and an embedded higher-order data flow language into a single framework. Programmers can choose between abstractions that suit their need. When a more declarative solution is not obvious or less efficient, they can always revert to lower-level observers or imperative abstractions such as routers. Different aspects of our system address different software engineering principles we identified in the introduction:

Uniformity/Abstraction First-class polymorphic events and signals, generalized to reactives, offer lightweight, reusable and uniform interfaces. Low-level observers, reactors and flow reactives work the same way, regardless of the precise representation of their dependencies.

Side-effects/Encapsulation Reactors and flow reactives encapsulate complex event logic such as in the dragging example without exposing external state.

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

Composability Reactives can be composed with signal expressions, functional combinators or imperative data-flow. That way we can obtain a handle to an entity representing a single feature instead of multiple loosely coupled objects.

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

Consistency Scala.React's synchronous propagation model ensures that no reactive can ever see inconsistent reactive data, regardless of the number or shape of reactive dependencies.

Semantic distance The semantic distance is vastly reduced by avoiding inversion of control even for complex event logic.

Scala.React and code examples from above can be downloaded from `http://lamp.epfl.ch/~imaier`.

References

[1] Umut A. Acar, Guy E. Blelloch, and Robert 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] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

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

[5] G. Berry. The constructive semantics of pure esterel, 1999.

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

[7] Alan Borning, Robert Duisberg, Bjorn Freeman-Benson, Axel Kramer, and Michael Woolf. Constraint hierarchies. In *OOPSLA*, pages 48–60, 1987.

[8] Bryan Cattle. If only we'd used ants profiler earlier... http://www.codeproject.com/KB/showcase/IfOnlyWedUsedANTSProfiler.aspx, 2007.

[9] Kimberley Burchett, Gregory H. Cooper, and Shriram Krishnamurthi. Lowering: a static optimization technique for transparent functional reactivity. In *PEPM*, pages 71–80, 2007.

[10] Magnus Carlsson. Monads for incremental computing. In *ICFP*, 2002.

[11] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. TelegraphCQ: continuous dataflow processing. In *SIGMOD*, pages 668–668, 2003.

[12] Jean-Louis Colaço, Alain Girault, Grégoire Hamon, and Marc Pouzet. Towards a higher-order synchronous data-flow language. In *EMSOFT*, pages 230–239, 2004.

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

[14] Gregory H. Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *ESOP*, pages 294–308, 2006.

[15] Antony Courtney. Frappé: Functional reactive programming in Java. In *PADL*, 2001.

[16] Antony Courtney. Functionally modeled user interfaces. In *Interactive Systems. Design, Specification, and Verification*. 2003.

[17] Christopher 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.

[18] Alan J. Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker M. White. Towards expressive publish/subscribe systems. In *EDBT*, pages 627–644, 2006.

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

[20] Conal Elliott and Paul Hudak. Functional reactive animation. In *ICFP*, 1997.

[21] Patrick Eugster and K. R. Jayaram. Eventjava: An extension of java for event correlation. In *ECOOP*, 2009.

[22] Andrzej Filinski. Representing monads. In *POPL*, 1994.

[23] Andrzej Filinski. Representing layered monads. In *POPL*, 1999.

[24] Bjorn N. Freeman-Benson. Kaleidoscope: mixing objects, constraints, and imperative programming. In *OOPSLA/ECOOP*, pages 77–88, 1990.

[25] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[26] Vaidas Gasiunas, Lucas Sabatina, Mira Menzini, Angel Nú nez, and Jacques Noyé. Escala: Modular event-driven object interactions in scala. In *AOSD*, 2011.

[27] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. In *OOPSLA*, OOPSLA '07, 2007.

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

[29] Nicolas Halbwachs, Albert Benveniste, Paul Caspi, and Paul Le Guernic. Data-flow synchronous languages, 1993.

[30] Philipp Haller and Martin Odersky. Actors that Unify Threads and Events. Technical report, Ecole Polytechnique Federale de Lausanne, 2007.

[31] Glenn E. Krasner and Stephen 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.

[32] Sean McDirmid and Wilson C. Hsieh. Superglue: Component programming with object-oriented signals. In *ECOOP*, pages 206–229. Springer, 2006.

[33] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: a programming language for ajax applications. *OOPSLA*, pages 1–20, 2009.

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

[35] Martin Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

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

[37] Sean Parent. A possible future of software development. http://stlab.adobe.com/wiki/index.php/Image:2008_07_25_google.pdf, 2008.

[38] Tomas Petricek and Don Syme. Collecting hollywood's garbage: avoiding space-leaks in composite events. In *ISMM*, 2010.

[39] Brown PLT. Flapjax github repository. http://github.com/brownplt/flapjax.

[40] Hridesh Rajan and Gary T. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP*, 2008.

[41] Tiark Rompf, Ingo Maier, and Martin Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In *ICFP*, pages 317–328, 2009.

[42] Friedrich Steimann, Thomas Pawlitzki, Sven Apel, and Christian Kästner. Types and modularity for implicit invocation with implicit announcement. *TOSEM*, 2010.

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

[44] Mads 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.

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

[46] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *SIGMOD*, pages 407–418, 2006.