Abstract
In contrast to batch processing systems, interactive systems require substantial programming effort to continuously synchronize with their environment. We quickly review existing programming systems that address this task. We present scala.react, our approach to embedded reactive programming in Scala which combines many advantages of previous systems. We show how our implementation makes use of dynamic dataflow graphs, delimited continuations and implicit parameters in order to achieve desired semantic guarantees and concise syntax.

1. Introduction
In contrast to batch processing systems, interactive systems require substantial programming effort to continuously synchronize with their environment. There is a large body of research as well as practical approaches to address this problem with special purpose programming language abstractions. We will quickly give an overview of the most and prominent related approaches.

1.1 Synchronous dataflow languages
Synchronous dataflow languages [2, 3, 12] are high-level approaches to reactive programming in the narrow sense, i.e., where the environment cannot wait as, for instance, in real-time system. Synchronous in this context means that dataflow is associated with time which enables programs to relate different events and deal with the notion of simultaneity. Time in those systems is usually discrete, i.e., it is possible to access previous values and events and thusly encode recursion as in feedback loops. Common Lustre [12] examples for instance make heavy use of feedback loops. In contrast to Lustre, Esterel is an imperative language enriched with dataflow constructs, e.g., to suspend computation until a signal emits.

1.2 Functional Reactive Programming
Functional reactive programming (FRP) integrates ideas from Lustre into functional programming languages [6–10, 14]. In contrast to synchronous dataflow languages, signals are first class. FRP distinguishes between event streams and behaviors, which represent time varying values. Functional reactive programs are written in a combinator-based style. Viewing event streams as collections of time/value pairs, many combinators from functional collections such as map or filter, immediately make sense for event streams.

1.3 Dynamic Databinding
Dynamic databinding is a more pragmatic approach to programming interactive systems. Examples are JavaFX [16] or Flex [1] which extend imperative programming with mechanisms to persistently bind a variable to expressions which are reevaluated when a dependency changes. Databinding systems usually lack first-class reactive abstractions and a dataflow language. Interaction with existing toolkit happens through simple imperative observers. Moreover, semantics are only loosely defined. In JavaFX, e.g., a variable can be concurrently bound to multiple expressions which makes the value of the variable jump back and forth between bound expressions. A notion of simultaneity is usually not supported and therefore glitches can occur, i.e., inconsistent data that does not yet reflect the new state can be observed.

1.4 Contributions
We present scala.react, an embedded reactive programming DSL for Scala which combines the following aspects from previous approaches into a single framework:
- First-class notion of events and time varying values
- Declarative reactivity in the spirit of Lustre and FRP
- Imperative dataflow programming inspired by Esterel (enabled by delimited continuations)
- Support for simple observers in order to integrate with existing event-based libraries

2. First-class Reactives
The two main abstractions in scala.react are event streams and time varying values.

2.1 Event streams
Event streams are represented by instances of trait Events which has one mutable subclass:

```scala
trait Events[+A] {
  def subscribe(ob: Observer): Unit
  def message(ob: Observer): Option[A]
}
observe(quitButton.clicks) { x => System.exit() }
ob = observe(es) { x => println("Receiving: "$ + x) }
Method observe takes an event source es and a closure that accepts event values x from stream 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 [11], 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 since we don’t want every client to inject click events. 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 quit = quitButton.clicks merge quitMenu.clicks merge fatalExceptions
In order to improve the situation, we add composition features in the style of functional reactive programming (FRP) [6, 10, 20]. 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\(^1\):
def merge[B=A](that: Events[B]): Events[B]

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 and we use the common base type constraint trick in order to be able to declare Events covariant on its message type.

Clients can now easily customize the event source quit and the quit action doQuit:
object MyApp extends UIApplication {
    val quit: Events[Option[A]]
    def doQuit() {
        /* clean up, display dialog, etc */
        System.exit()
    }
    observe(quit) { doQuit() }
}

2.2 Signals

A large body of problems in interactive applications 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 again contains a mutable subclass:

trait Signal[+A] {
    def apply(): A
    def now: A
    def changes: Events[A]
}
class Var[A](init: A) extends Signal[A] {
\(^1\)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. An unbiased merge operator needs to have a more complicated type such as def merge[B](that: Events[B]): Events[(Option[A], Option[B])], emitting pairs of optional event values with at least one option = None. We will discuss the notion of simultaneous events below in more detail.
We will therefore collectively refer to signals and event streams as *reactives*. Trait Reactive declares two type parameters: one for the message type an instance emits and one for the values it holds. For now, we have subclass Signal which emits its value as change messages, and therefore its message and value types are identical. Subclass Event only emits messages and never holds any value. Its value type is hence Unit. Subclasses of trait Reactive need to implement two methods which obtain the reactive’s current message or value and create dependencies in a single turn.

### 3. Imperative Dataflow

We can express many simple event relationships very concisely with FRP-style combinators. Some complex event patterns such as sequencing, however, are difficult to express in FRP and existing examples usually employ higher-order reactives and switching or flattening such as the Flapjax dragging example in [14]. Here is a version adapted to scala.react:

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

For the eyes of a non-functional programmer, this example can be quite daunting. It is important to understand that moves and drops are nested event streams of type Events[Events[Drag]] and Events[Events[Drop]]. We merge them, which keeps their nested structure, until we flatten them (in some FRP implementations flatten is called switch) which essentially creates a simple event stream that behaves like a state machine.

Sequencing and looping, however, can be expressed quite naturally imperatively (as already indicated by Halbwachs in [12]). For this purpose, we provide a dataflow DSL not unlike Esterel. Here is the dragging example as an imperative dataflow program:

```scala
val drags = Events.loop { self =>
  self next mouseDown
  val mu = self.loopUntil(mouseUp) {
    self emit Drag(self next mouseMove)
  }
  self emit Drop(mu)
}
```

This creates an event stream that repeatedly loops through the given body. The body closure takes a self reference as an argument which provides us with a dataflow language. We call the next dataflow operator to wait for a mouse down event, then loop until we receive a mouse up event. Inside the inner loop and at the end, we call emit to let the enclosing event stream emit events. We can use the same dataflow language to create a signal that logs events.
val path = Val(new Seq) once { self =>
    val down = self next mouseDown
    val mu = self.loopUntil(mouseUp) {
        self emit (self.previous + Drag(self next mouseMove))
    }
    self emit (self.previous + Drop(mu))
}

Here, we start with a constant signal of an empty sequence. For illustration purposes, we log a single drag operation using the `once` combinator on the initial signal, which does not repeatedly loop but runs the given dataflow code only once. The signal dataflow language extends the event dataflow language by the `previous` operator, which provides the previous value of the signal under construction, in order to populate an immutable collection and let the signal emit it.

In order to build a data-flow reactive using the `loop` and `once` combinators on an initial reactive as in the previous example, we implicitly convert a reactive to an intermediate class that provides those combinators:

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

These intermediate classes are defined as follows:

```scala
trait ReactiveToDataflow[M, N, R <: Reactive[M, N]] extends Reactive[M, N] {
    protected def init: R
    def loop(body: DR => Unit): R
    def once(body: DR => Unit): R
}
```

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

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

Trait `ReactiveToDataflow` extends `Reactive` and provides two additional type parameters to fix the precise type of reactives we are creating. The type related details of this design are out of the scope of this paper. It is a result from our experience we gathered during the redesign of Scala’s collection library which is thoroughly described in [15]. The base type for data-flow reactives defines the data-flow language for reactives and is specified as follows:

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

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

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

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

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

Note that the following signal

```scala
Val(0) once { self =>
    self switchTo sig
    self emit 1
}
```

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

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

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

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

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

### 3.1 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. Here is how we can implement some of the built-in combinators in class Events[A] that are not trivially implemented in terms of other combinators. The following collect combinator can be used to implement other combinators:

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

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. Combinators map and filter can now both be implemented in terms of collect:

```scala
def map[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:

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

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:

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

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

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

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

```scala
def accumulate[B](init: B)(op: (B,A)=>B): Events[B] = {
  var acc = init
  Events.loop[B] { self =>
    val x = self next outer
    acc = op(acc, x)
    self emit acc
  }
}
```

It continuously applies a function op to values from an event stream and an accumulator and emits the result in a new stream. Trait Signal[A] contains two flatten combinators, which are defined for signals of events and signals of signals. They return a signal or event that continuously behaves like the signal or event that is currently held by the outer signal. They can be implemented as shown in Figure 1.

These can be generalized into a single generic combinator as shown in Figure 2. Flattening a signal of reactives makes sense for any subclass of Reactive, not just Signal or Events. The implicit parameter is used to convert a current signal value to a ReactiveToDataflow in order to construct a data-flow reactive. This enables us to flatten a signal of any subtype R of Reactive to an instance of R that behaves like the reactive that is currently held by the signal.

The merge combinator is the only axiomatic combinator that is not implemented in terms of an imperative data-flow program.

4. Implementation

Scala.React's change propagation engine proceeds in discrete time steps, or cycles. The engine is either idle or in a cycle, i.e., propagating changes. Any external update of a reactive schedules a revalidation request for the next cycle (and ensures that a next cycle will be initiated). When exactly cycles are scheduled and on which thread they are run can be configured by clients. A common scenario in UI programming is to let an engine run on the user interface thread. An engine that is run on the Swing event dispatcher, e.g., is created as follows:

```scala
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. Contents of our new module can be brought into current scope by a simple import statement import SwingDomain... Note that Scala's path dependent types ensure that reactives from different domain instances have incompatible types and cannot interact in the way we have described above. Details on different scheduling mech-
Figure 1: Flattening operators.

Figure 2: Generalized flattening operator.

anisms and how to establish dependencies between reactives from different domains will be subject of a separate paper.

Our change propagation implementation uses a standard, push-based approach based on topologically ordered dependency graphs. When a change propagation cycle starts, Scala.React starts with the lowest topological order, briefly level, and continuously validates reactives with the same level before it proceeds to higher levels. For this to work, reactives on lower levels must never depend on reactives on higher levels. This invariant can be temporarily violated during validation, however, when dependencies change. In this case we need to abort the validation of the current reactive, increase its level and reschedule its validation for that level. Details of a very similar implementation can be found in [5]. We will restrict ourselves in the following to a few novel aspects in Scala.React's implementation.

4.1 Signal expressions
The three key features that let us implement the concise signal expression syntax we are using throughout the paper are Java’s thread local variables [13], Scala’s call-by-name arguments and Scala’s function call syntax. When we are constructing signal 
\[ \text{Signal} \{ a() + b() \} \], we are in fact calling method

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

with call-by-name argument \( \{ a() + b() \} \) in order to capture the signal expression. The actual evaluation happens in the `apply` method which returns the current value of the signal while establishing signal dependencies. Method `Signal.apply` comes in different flavors, but the general concept remains the same: it maintains a thread local stack of dependent reactives that are used to create dependency sets. A signal that caches its values is either valid or has been invalidated in the current or a past propagation cycle. If it is valid, it takes the topmost reactive from the thread local stack without removing it, adds it to its set of dependents and returns the current valid value. If it is invalid, it additionally pushes itself onto the thread local stack, evaluates the captured signal expression, and pops itself from the stack before returning its current value.

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

4.2 The imperative data-flow language
Our data-flow DSL is implemented in terms of Scala’s delimited continuations [18]. Trait `DataflowBase` contains most of the implementation infrastructure for our data-flow language:\(^{3}\)

```
trait DataflowBase {
  protected var continue = () => reset { mayAbort { body() } }
  def mayAbort(op: =>Unit) = {
    engine nextTurn
    this
  }
}
```

Method `body` is implemented by subclasses and runs the actual data-flow program. Calls to `delay`, `next` or any extensions defined in subclasses (such as `emit` and `switchTo`) are implemented in terms of two helper methods:

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

\(^{3}\)Note that for brevity we have omitted the `@suspendable` annotations before
Method `continueLater` takes a continuation as an argument, captures it in a variable and schedules this reactive for the next turn. Continuation `k` is wrapped in a call to `mayAbort`, which properly aborts and reschedules the evaluation of `k` if the graph topology has changed because new dependencies were established. We can use `continueLater` to implement the delay data-flow operator as follows:

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

Method shift from the Scala standard library for continuations, captures the current continuation `k` and passes it to the closure given to shift. In the case of `delay`, we simply defer the execution of `k` to a later cycle by a call to `continueLater`.

Method `continueNow` is essentially used as a shift variant immune to topological changes. It accepts a function `op` which takes the current continuation as an argument. A call to `shift` captures the current continuation `k`, transforms it by applying `op` and stores the result in a variable for immediate and later use. The transformed continuation is again wrapped in a call to `mayAbort` in order to handle potential topological changes. We use `continueLater` and `continueNow` to implement operator `switchTo` in class `DataflowReactive`, a subclass of `DataflowBase`:

```scala
def switchTo(r: R): Unit @suspendable =
  continueNow { (k: (B=>Unit)) =>
    delegate = r
    trackDelegate()
    continueLater { trackDelegate(); k() }
  }
```

Type `R` denotes the type of reactive a certain `DataflowReactive` can switch to. We first capture the current reactive continuation in `continueNow`, store the reactive we switch to in field `delegate` and check for changes in the delegate. Since a `switchTo` always involves a delay, we call `continueLater` which not only defers the continuation but also prepends a check for changes in the delegate. Method `trackDelegate` is implemented by subclasses of `DataflowReactive` and contains code specific to caching mechanisms. Events, e.g., store current messages, whereas signal store values. Note that we use `continueNow` instead of `shift`, since `trackDelegate` can change the graph topology. Data-flow combinator `next` is implemented as follows:

```scala
def next[B](r: Reactive[B, _]): B @suspendable =
  continueNow { (k: (B=>Unit)) =>
    r message this match {
      case Some(x) => k(x)
      case None => trackDelegate()
    }
  }
```

A call to `r message this` returns `Some(x)`, if `r` is currently emitting `x`. It returns `None` if it is currently not emitting. The `this` reference denotes the receiver of the next call, i.e., `self` in `self next stream`. Again, we capture the current continuation in `continueNow`, and then invoke it if the given reactive is currently emitting. Otherwise we check the delegate for messages. Since `next` returns the value emitted by `r`, continuation `k` has an argument type different from `Unit` this time.

High-level data-flow expressions such as `loopUntil` can be written in terms of the above operators and existing combinators (which are also implemented in terms of data-flow operators as we have seen):

```scala
def loopUntil[A](es: Events[A]){
  (body: =>Unit @suspendable): A @suspendable = {
    loopUntil(A) {
      if (es.now == None) { body }
      x.now.get
    }
  }
```

When validated during a propagation cycle, a data-flow reactive simply runs its current continuation saved in variable `continue`, which initially starts executing the whole body of the reactive. 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 rather heavyweight (though less heavyweight than using blocking threads) on a runtime without native CPS support. We therefore do not currently implement it and leave it for future work to compare the outcome with our current implementation. Our current approach is similar to lowering in FrTime [4].

### 4.3 Avoiding a memory leak potential of observers

Internally, Scala.React’s change propagation is implemented in terms of observers. We do expose them to clients as a very lightweight way to react to changes in a reactive as we have seen above. 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 stream events:

```scala
case View[A](events: Events[A]){
  events foreach println
}
```
This usually leads to a reference pattern as depicted in Figure 3a. The observing object – the view in the example above – creates an observer and installs it at an event stream or signal that reflects changes in a reactive object.

The critical reference path goes from the reactive object to the observing object. This path is often not visible to clients, since observing objects typically abstract from their precise dependencies. For every observing object that we want to dispose before its reactive dependencies are garbage collected, we would hence need to switch to explicit memory management. This constitutes a common potential for memory leaks on observer-based programming and is a variation of the issue of explicit resource management we identified in the introduction.

The reference pattern in Figure 3b eliminates the leak potential. Note the weak reference from the event source to the observer, depicted by a dashed arc. It eliminates any strong reference cycles between the observing and the publishing side. In order to prevent the observer from being reclaimed too early, we also need a strong reference from the observing object to the observer. It is important that we always force a client into creating the latter strong reference, otherwise we haven’t gained much. Instead of keeping in mind that she needs to call a dispose method, she would now need to remember to create an additional reference. Fortunately, we can use Scala’s traits to achieve our desired reference pattern while reducing the burden of the programmer. The following trait needs to be mixed in by objects that want to observe events. API clients have no other possibility to subscribe observers.4

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

  trait PersistentObserver extends Observer {
    override def dispose() {
      super.dispose(); obRefs -= this
    }
  }

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

Method observe actually creates an observer that automatically adds itself to an observer set of the enclosing object during construction. Method 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:

```scala
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.

Note that our approach to avoiding memory leaks is an alternative to [17], which presents a garbage collection scheme that modifies the semantics of stateful reactivies. For instance, in [17], a signal obtained from `Events.hold` will only be updated if an observer is installed, whereas in scala.react, it always holds the latest event from the given stream.

5. Conclusion

We have shown how to combine reactive programming with different levels of abstractions into a single embedded DSL framework. Scala.react supports simple observer-based programming, to imperative synchronous dataflow a la Esterel and combinator based reactive programming in the style of FRP. The fundamental abstractions that enable us to combine these programming styles are first class reactivies such as event streams and signals. Delimited continuations allow for concise syntax of imperative reactive programs, closures and uniform function application allow for concise signal expressions, implicit conversions and mixin composition allow us to tie abstractions together and factor complex functionality into common base traits.

Acknowledgments

We thank the anonymous reviewers for their valuable comments.

References


---

4 We can directly store the first allocated persistent observer in a reference field. Only for the less common case of multiple persistent observers per `Observing` instance we do allocate a list structure to keep observer references around. This saves a few machine words for common cases.
Figure 3: Common observer reference pattern.


