Containers and Aggregates, Mutators and Isolates for Reactive Programming

Aleksandar Prokopec
EPFL, Switzerland
aleksandar.prokopec@gmail.com

Philipp Haller
Typesafe Switzerland
philipp.haller@typesafe.com

Martin Odersky
EPFL, Switzerland
martin.odersky@epfl.ch

Abstract

Many programs have an inherently reactive nature imposed by the functional dependencies between their data and external events. Classically, these dependencies are dealt with using callbacks. Reactive programming with first-class reactive values is a paradigm that aims to encode callback logic in declarative statements. Reactive values concisely define dependencies between singular data elements, but cannot efficiently express dependencies in larger datasets. Orthogonally, embedding reactive values in a shared-memory concurrency model convolutes their semantics and requires synchronization. This paper presents a generic framework for reactive programming that extends first-class reactive values with the concept of lazy *reactive containers*, backed by several concrete implementations. Our framework addresses concurrency by introducing *reactive isolates*. We show examples that our programming model is efficient and convenient to use.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Data-Flow Languages

General Terms Algorithms, Languages

Keywords reactive programming, reactive signals, reduction tree, reactive collections, isolates, reactive mutations

1. Introduction

Functional programming is a programming model of choice for declarative, stateless, functional-style computations whose output depends only on well-defined inputs available when the program starts. However, in applications like discrete event simulations, user interfaces, game engines or operating system kernels encoding state into a functional programming model can be cumbersome. Moreover, many programs are reactive in nature – their inputs change or become available during the execution of the program. *Reactive programming* addresses these issues by introducing callback functions that are called when inputs become available.

Classical reactive programs are sprinkled with callbacks. It is hard to reason about properties of such programs, as they lack the structure to identify information flow – their algorithmic structure is obscured by unstructured callback declarations. We identify various callback patterns and express them as higher-order functions parametrized by specific logic, making them equivalent to corresponding sets of callbacks.

This approach is similar in spirit to the methodology seen in functional programming. The arbitrary use of recursion is abandoned there in favour of structured higher-order primitives such as left and right folds, maps and filters. For reactive programming the goal is to isolate patterns of callbacks that should form the basis for writing programs. To paraphrase Erik Meijer [6] in a slightly different context – callbacks are the GOTO of reactive programming.

In this paper we describe a reactive programming framework called Reactive Collections based on efficient functional composition of reactive values and data structures. Specific data structures support incremental queries that are updated along with the data structure. Special care is taken to show that event propagations run within optimal bounds. In particular:

- We describe reactive data-types and their functional composition, focusing on manipulating mutable data.
- We show reactive aggregates that efficiently fold reactive values by means of aggregation trees.
- We show generic data containers for reactive programming, and discuss their event propagation complexities.
- We propose reactive isolates for concurrent reactive programming, used to express and encapsulate concurrency.

Sections 2, 3 and 4 study reactive values and their usage, generalize them to reactive containers and introduce reactive

Scala '14, July 28-29, 2014, Uppsala, Sweden.

http://dx.doi.org/10.1145/2637647.2637656

isolates. Section 5 contains the evaluation for our model. Section 6 discusses related work and Section 7 concludes.

Note that there is no motivation section in this paper. Many reactive programming abstractions are shown and we motivate each of them with usage examples as we go. One abstraction will lead naturally to another and higher abstractions will be built in terms of simpler ones, intuitively evolving a more elaborate reactive programming stack.

2. Reactive values

We will build abstractions and design patterns for reactive programming by means of a specific data-type called `Reactive[T]`. A reactive of type `T` denotes a data-type that may produce values of type `T`. We call such values *events*.

By itself, this data-type is not very useful, as there is no way for the outside world to interact with the events it produces. We therefore introduce the type `Reactor[T]` and extend reactives with a method `subscribe` – instances of reactors can be subscribed to events that reactives produce.

```
trait Reactor[T] {
  def react(value: T): Unit
  def unreact(): Unit }
trait Reactive[T] {
  def subscribe(r: Reactor[T]): Subscription }
```

The above constitutes the basis of what is called the *observer pattern*. After `subscribe` is given a reactor, that reactor's `react` method is called each time the reactive produces a new value. The return type `Subscription` is a value used to `unsubscribe` that specific reactor.

We now turn to a concrete reactive – `Reactive.Never`, which never produces an event. When a reactor subscribes to it, its `unreact` is called immediately.

```
trait Never[T] extends Reactive[T] {
  def subscribe(r: Reactor[T]) = {
    r.unreact(); Subscription.empty } }
```

Another reactive called `Reactive.Emitter` produces events when its `emit` method is called. Emitters are useful when bridging the gap between existing polling-based APIs and reactives. We factor out useful `Emitter` functionality into a separate reactive called `Default` (here simplified).

```
class Default[T] extends Reactive[T] {
  val reactors = Buffer[Reactor[T]]()
  def subscribe(r: Reactor[T]) = {
    reactors += r
    Subscription { reactors -= r } }
  def reactAll(value: T) =
    for (r <- reactors) r.react(value)
  def unreactAll() =
    for (r <- reactors) r.unreact() }
class Emitter[T] extends Default[T] {
  def emit(value: T) = reactAll(value)
  def close() = unreactAll() }
```

2.1 Composing reactive values

The facilities provided so far are sufficient to express reactive programs, but using them in large codebases results in what is known as the *callback hell*, where the programmer can no longer make sense of the semantics of the program.

Assume we have a reactive of mouse events and need to react to only left clicks. Instead of sprinkling the callback code with conditionals checking which button was pressed, we construct a new reactive value that only produces left clicks.

```
val leftClicks =
  clicks.filter(ev => ev.button == Mouse.Left)
```

The `filter` method returns both a reactive and a reactor. It subscribes to its parent reactive as if it were a reactor and produces only events that pass the user-specified predicate:

```
class Reactive[T] { self =>
  def filter(p: T => Boolean) =
    new Default[T] with Reactor[T] {
      def react(value: T) =
        if (p(value)) reactAll(value)
      def unreact() = unreactAll()
      self.subscribe(this) } }
```

A plethora of useful *reactive combinators* is defined in this way – `map`, `union`, `concat`, `until` or `after`, to name a few. The `after` operator ignores events of `this` reactive until an argument reactive produces an event, and `until` does the opposite. We single out the `scanPast` combinator [1] that scans over the past events, much like the functional `scanLeft` combinator scans values of a list.

```
def scanPast[S](z: S)(op: (S, T) => S) =
  new Default[S] with Reactor[T] {
    var last: S = z
    def react(value: T) {
      last = op(last, value); reactAll(last) }
    def unreact() = unreactAll()
    self.subscribe(this) }
```

The `scanPast` combinator be used to express other combinators like the total event count:

```
def total = scanPast(0) { (n, ev) => n + 1 }
```

Detecting scroll events using callbacks is cumbersome, but reactive combinators make it straightforward. Scrolling is essential when displaying zoomed parts of a picture, web page or a large document. A reactive `scrollEvents` of scrolls produces differences in coordinates on the screen that are triggered by dragging the mouse. Dragging starts when the user presses a mouse button and stops once it is released.

Assume first a `MouseEvent` has the last cursor position `xy` and information on whether a button is `down`. First, we need to extend mouse events with the information if the drag has just `started` – we use `Stat` objects for this purpose.

```
class Stat(xy: (Int, Int), down: Boolean,
    started: Boolean)
```

We will scan the reactive of mouse events into a reactive of status updates `stats`. Next, we will filter only those statuses for which the mouse button is pressed, as only those comprise dragging – we call these statuses `drags`. To

go from dragging to scrolling means to produce a reactive of scrolls that hold that last and current dragging position. The final step is just `map`ping a sequence of scrolls into coordinate diffs. The complete example is shown below:

```
class Scroll(last: (Int, Int), cur: (Int, Int))
def scrollEvents(ms: Reactive[MouseEvent]) = {
  val zstat = Stat((0, 0), false, false)
  val stats = ms.scanPast(zstat) { (st, ev) =>
    Stat(ev.xy, ev.down, !st.down && ev.down) }
  val drags = stats.filter(st => st.down)
  val zscroll = Scroll((0, 0), (0, 0))
  val scrolls = drags.scanPast(zscroll) {
        (scr, drag) =>
    if (drag.started) (drag.xy, drag.xy)
    else (scr.cur, drag.xy) }
  scrolls.map(scr => scr.cur - scr.last) }
```

A reactive of the current screen position then becomes:

```
scrollEvents(mouse).scanPast((0, 0)) {
  (pos, diff) => pos + diff }
```

That said, the application rendering logic might want to read the current scroll position at any time, and not as the events arrive by subscribing to this reactive. For this we introduce a special type of a reactive – we discuss this next.

2.2 Reactive signals

Reactive signals are a special kind of reactive values that cache the last event. The `apply()` method allows accessing this last event, i.e., the signal's *value*:

```
trait Signal[T] extends Reactive[T] {
  def apply(): T }
```

When a signal produces an event, we say that its value changed. A `Signal.const` never changes its value.

```
def const[T](v:T) = new Never[T] with Signal[T]
  { def apply() = v }
```

Some reactive combinators return signals instead of reactives. In fact, the result of the `scanPast` method is already a signal in disguise, so we change its return value. Lifting a reactive into a signal with an initial value `z` then becomes:

```
def signal(z: T): Signal[T] = scanPast(z) {
  (last, current) => current }
```

Some reactive combinators are specific to reactive signals. The `reducePast` combinator is similar to `scanPast`, but it takes the current value of the signal as the initial value:

```
def reducePast(op: (T, T) => T) =
  scanPast(apply())(op)
```

The `past2` combinator produces the previous signal value:

```
def past2: Signal[(T, T)] = scanPast
  ((apply(), apply())) { (p, v) => (p._2, v) }
```

The `zip` combinator takes the current signal `self`, another signal `that`, and a merge function `f`, and produces a signal of zipped values. Unlike `union` that outputs events produced by either reactive, `zip` relies on the current value to merge the values of both signals when an event arrives:

```
def zip[S, R](that: Signal[S])(f: (T,S) => R) =
  new Default[R] with Signal[R] {
    val self = Signal.this
    var last = f(self.apply(), that.apply())
    var live = 2
    def apply() = last
    def decrease() {
      live -= 1; if (live == 0) unreactAll() }
    self.subscribe(new Reactor[T] {
      def react(v: T) {
        last = f(v, that()); reactAll(last) }
      def unreact() = decrease() })
    that.subscribe(new Reactor[S] {
      def react(v: S) {
        last = f(self(), v); reactAll(last) }
      def unreact() = decrease() }) }
```

Dependencies between reactives and signals induce a *dataflow graph* in which events flow from event sources like emitters seen earlier to reactives created by combinators. Combinators `zip` and `union` are the only combinators shown so far that merge nodes in this graph rather than branching out, but they work on only two signals. A more powerful combinator called `Signal.aggregate` merges any number of signals. Implementing reactive aggregates as a chain of `zip` calls results in a chain of updates of an $O(n)$ length:

```
def aggregate[T](ss: Signal[T]*)(f: (T,T)=>T) =
  ss.reduceLeft { (_ zip _)(f) }
```

Above we apply the `reduceLeft` operation [6] to `zip` the signals. This is unsatisfactory – if the first signal in `ss` changes, values of all the other signals need to be reduced again. A better scheme organizes signals into a balanced tree to ensure at most $O(\log n)$ updates when a signal changes. Signals from `ss` are leaves in this tree and inner nodes are `zip`ped consecutive pairs of signals. `Signal.aggregate` constructs the tree by levels starting from the leaves and going up to the root. We use the list `grouped` combinator that produces a new list of pairs of consecutive values.

```
def aggregate[T](ss: Signal[T]*)(f: (T,T)=>T) =
  if (ss.length == 1) ss(0) else {
    val pairs = ss.grouped(2).map { pair =>
      if (pair.length == 1) pair(0)
      else pair(0).zip(pair(1))(f) }
    aggregate(pairs)(f) }
```

The signal shown above works for an arbitrary set of input signals, but this set is fixed when the signal is created. It is therefore called a *static reactive aggregate*.

```
def invAggregate[T](ss: Signal[T]*)
  (f: (T, T) => T)(inv: (T, T) => T) =
  new Signal[T] with Default[T]
  with Reactor[(T, T)] {
    var last = ss.foldLeft(ss.head)(f(_, _()))
    var live = ss.length
    def apply() = last
    def react(v: (T, T)) {
      last = op(inv(last, v._1), v._2) }
    def unreact() {
      live -= 1; if (live == 0) unreactAll() }
    for (s <- ss) s.past2.subscribe(this) }
```

An $O(1)$ static aggregate `invAggregate` exists given an inverse `inv` of the associative merge function `f` [4]. In other words, elements of `T` and `f` must form a *group*. This aggregate uses the merge inverse to update its value.

2.3 Higher-order reactives

Higher-order reactive values are best motivated by an example. Let's assume we are developing a game engine in which the player moves through open and closed virtual areas. We want to express the light intensity as a function of the player position and the time of day. In the outside areas the light depends on the time of the day, in the inside it is constant:

```
val out: Int => Double = h => sin(2*Pi*h/24)
val in: Int => Double = h => 0.5
val light: Reactive[Int => Double] =
  position.map(p => if (isIn(p)) in else out)
val intensity: Reactive[Double] =
  hours.zip(light) { (h, f) => f(h) }
```

This is unsatisfactory because it constraints all `light` contributions to be functions of the current hour. This is reflected in the type `Reactive[Int => Double]` of `light`. If we want to change the intensity inside so that it depends on the number of windows or candles, we have a problem.

The advantage of the `Reactive` type is that it encodes events it produces, but not inputs it depends on, as is the case with functions. We encode both `out` and `light` differently:

```
val out: Reactive[Double] =
  hours.map(h => sin(2 * Pi * h / 24))
val in: Reactive[Double] = const(0.5)
val light: Reactive[Reactive[Double]] =
  position.map(p => if (isIn(p)) in else out)
```

The `light` type is now `Reactive[Reactive[Double]]` – we call `light` a *higher-order reactive* since the events it produces are other reactive values, similar to how higher-order functions include other functions as part of their type. The question is: how do we pull the light intensity updates `Double` from the nested reactive? We could do the following:

```
val intensity = new Emitter[Double]
light.subscribe { (r: Reactive[Double]) =>
  r.subscribe(i => intensity emit i) }
```

That is clumsy – we even forgot to unsubscribe from the nested reactives. What we want is to multiplex the events from the *current* nested reactive, much like the digital multiplexer circuit produces values based on the control signal. Here, the control signals are the events `Reactive[Double]` produced by the outer reactive. We rely on a combinator `mux` that allows declaring the `intensity` as follows:

```
val intensity: Reactive[Double] = light.mux
```

This allows replacing the inside light intensity `const(0.5)` with an arbitrary `Reactive[Double]` without affecting the rest of the codebase. The `mux` combinator exists only on higher-order reactives, but it is not the only higher-order combinator. Recall the dragging example from Section 2.1 – higher-order reactives allow expressing it more intuitively:

```
val down = mouse.past2
  .filter(e => !e._1.down && e._2.down)
val up = mouse.past2
  .filter(e => e._1.down && !e._2.down)
val drags = down.map(_ => mouse.until(up))
drags.map(ds => ds.map(_.xy).past2.map2(_ - _))
  .concat
```

Here we isolate reactives of mouse presses `down` and releases `up`. The mouse `drags` are then sequences of mouse events between the next mouse press `down` and before the subsequent mouse release `up`. The postfix `concat` operator above concatenates nested reactives of drags together into scrolls. Only after one sequence of drags `unreacts`, the events from the next nested sequence of drags are taken. Similarly, the postfix `union` operator applied to a higher-order reactive produces a reactive with the events from all the nested reactives, but imposes no ordering on the nested reactives.

2.4 Reactive mutators

Reactive programming models typically strive towards declarative style and shun mutable state. However, current APIs expose side-effecting operations and mutability. Future APIs might be entirely reactive or might partially expose mutable state, but in either case, having a way to deal with mutability is currently useful. Furthermore, allocating immutable event objects for reactives on the heap, as shown so far, can be costly. For example, a simple real-time 3D game engine typically requires eight 4×4 double precision matrices. Allocating these matrices in each frame requires allocating $60kB$ of memory per second. If the reactives that produce these matrices are used in combinators to produce other reactives these $60kB$ are multiplied as many times. Moreover, there are other types of objects that are even more expensive to allocate. Most automatic memory management systems are currently limited when delivering smooth performance in real time systems with excessive allocations.

The same mutable object should therefore be reused for multiple events. In the case of transformation matrices, we would like to recompute the elements of the model-view matrix at each frame depending on the viewing `angle`:

```
angle.mutate(modelviewMatrix) {
  (m, a) => setUsingAngle(m, a) }
```

All the mutation events must be produced by a single signal that encapsulates the mutable object. A signal that holds such a mutable object is called a *mutable signal*.

```
class Mutable[M](val m: M) extends Signal[M]
  with Default[M] { def apply() = m }
```

The `modelviewMatrix` is a signal that wraps a matrix object `m`. The object `m` can only be modified with `mutate`:

```
def mutate[M](s: Mutable[M])(f: (M, T)=>Unit) =
  { f(s.m); s.reactAll(s.m) }
```

Mutable signal declarations allow back edges in the dataflow graph and can cause infinite event propagations:

```
val fibonacci = new Mutable(Array[Int](0, 1))
val nextFibo = fibonacci.map(a => a.sum)
nextFibo.mutate(fibonacci) { (a, s) =>
  a(0) = a(1); a(1) = s }
```

As soon as one of the last two Fibonacci numbers changes, the next Fibonacci is computed. This updates the last two Fibonacci numbers and the process begins again. A reaction from a mutable signal triggers a feedback event and Fibonacci numbers are computed forever. Mutability allows infinite event propagation in this reactive programming model. A natural question arises – can we express mutability without mutable signals, or are mutable signals its necessary precondition? In other words, can we express a *mutable cell that produces events* without mutable signals? It turns out that mutability was hidden all the time – a `ReactCell` is a union of functionality in signals, which inspect the current value, and emitters, which produce events with `emit`.

```
def ReactCell[T](v:T): (Emitter[T],Signal[T]) =
  { val emitter = new Emitter[T]
    (emitter, emitter.signal(v)) }
```

We implement `ReactCell` separately and name the cell mutator `:=`, to honour the creators of ALGOL and Pascal.

```
class ReactCell[T](var value: T)
extends Default[T] {
  def apply() = value
  def :=(v: T) { value = v; reactAll(v) } }
```

One can consider the reactive cell a very limited form of a collection, i.e., a data container. This particular container consists of a single element at all times, but, unlike common containers, allows reactions whenever this element changes. We will see more powerful forms of such containers next.

3. Reactive containers

A data structure is a way of organizing data so that a particular operation can be executed efficiently. Efficiency here may refer to running time, storage requirements, accuracy, scalability, energy consumption, or a combination thereof.

How data structures are related to reactive programming is best shown using an example. Given an image raster, an image viewing application needs to know the changes in a particular part of the raster to refresh the visible part of the image on the screen. Assume that drawing a line changes the state of n pixels. The application can simultaneously display multiple parts of the image in r different windows – there are r reactive dependencies on the image raster. Finally, each such window shows m pixels of the image.

Reactives can produce events of any datatype – events could be immutable data structures. However, representing an image raster with any of the known immutable data structures is far from optimal. Mutable signals from Section 2.4 can store mutable data structures, but they are also unsatisfactory. A mutable signal event does not encode which part of the raster changed:

```
val img = new Mutable(new Raster(wdt, hgt))
for (i <- 0 until r)
  img.subscribe(ra => refreshWindow(ra, i))
```

This updates all the windows and results in $\Theta(r \cdot m)$ update operations each time n pixels change – it is potentially much larger than the pixel changeset. Instead of updating all the windows, we can do better by keeping a reactive of pixel coordinates that change, and filter them for each window:

```
type XY = (Int, Int)
val raster = new Raster(wdt, hgt)
val img = new Emitter[(Int, Int)]
def update(xy: XY, c: Color) {
  raster(xy) = c; img.emit(xy) }
for (i <- 0 until r)
  img.filter(xy => withinWindow(xy, i))
     .subscribe(xy => refreshPixel(xy, i))
```

This still requires $\Theta(r \cdot n)$ updates, because r `filter` operations are triggered for each raster update. An optimal number of updates is bound by $\Omega(n)$ and $O(r \cdot n)$.

```
val raster = new Raster(wdt, hgt)
val img = new Matrix[Emitter[Color]](wdt, hgt)
for (xy <- (0, 0) until (wdt, hgt))
  img(xy) = new Emitter[Color]
def update(xy: XY, c: Color) {
  raster(xy) = c; img(xy).emit(c) }
for (i <- 0 until r; xy <- bounds(i))
  img(xy).subscribe(c => refreshPixel(xy, i))
```

In the last example updating n raster pixels results in notifying exactly n emitters, and each of them notifies up to r reactors, but possibly less. This array of reactive cells constitutes a simple data-structure – a way of organizing data so that the reactions can propagate more efficiently.

Definition *(Reactive container)* Let r be the *total number of subscribers*, n the *update size* and m the *relevant event space size*. If a reactive value of data structure updates has event propagation running time bounds $\Omega(n)$ and $O(r \cdot n)$ we call it an *efficient reactive value*. A *reactive container* is a data structure with at least one efficient reactive value, and is represented with the `ReactContainer[T]` type.

3.1 Reactive associative containers

Associative containers, or maps, are data structures that store pairs of keys and values, and efficiently retrieve and change values mapped to specific keys. Sets are special cases of maps where values carry no information. A `ReactMap` is a map that exposes a reactive value `keys` of the modified keys.

```
trait ReactMap[K, V]
extends ReactContainer[(K, V)] {
  def keys: Reactive[K]
  def +=(k: K, v: V): Unit
  def apply(k: K): Option[V] }
def reactHash[K, V] = new ReactMap[K, V] {
  val data = new HashMap[K, V]
  val keys = new Emitter[K]
  def +=(k: K, v: V) {
    data += (k, v); keys.emit(k) }
  def apply(k: K) = data(k) }
```

This reactive map is lacking something. Imagine that we use the reactive map in an OS to map monitors to threads currently holding them. At any point there are r threads waiting for a monitor. When a monitor is assigned to a thread, the thread must be notified. We use the `keys` reactive to do this, with a $\Theta(r)$ propagation time for each notification:

```
val monitors = reactHash[Monitor, Thread]
def wait(thread: Thread, target: Monitor) =
  monitors.keys.filter(
    m => m == target && monitors(m) == thread
  ).subscribe(m => t.notify())
```

In the vocabulary of the previous section, this reactive container does not expose a reactive value that propagates update events in $\Omega(n)$ and $O(r{\cdot}n)$ bounds, where n is the update size and r is the total number of subscribers. In this case $n = 1$, since an update always changes a single mapping, so updating `monitors` should only trigger $0 \leq r_0 \leq r$ event propagations, where r_0 is the number of threads waiting for a particular monitor. The `filter` expresses subscribing to one specific monitor. Filtering is inefficient, so we add the `reactive.apply` method to `ReactMap`, to allow reacting to a particular key:

```
def wait(t: Thread, target: Monitor) =
  monitors.reactive(target)
    .filter(m => monitors(m) == t)
    .subscribe(m => t.notify())
```

Subscriptions in `wait` are only as efficient as `reactive.apply`. To ensure that `reactive.apply` event propagation time is $\Omega(1)$ and $O(r)$ we store an emitter for each key.

```
trait ReactMap[K, V]
extends ReactContainer[(K, V)] {
  def keys: Reactive[K]
  def +=(k: K, v: V): Unit
  def apply(k: K): Option[V]
  def reactive: Lifted }
trait Lifted[K] { def apply(k: K):Reactive[K] }
def reactHash[K, V] = new ReactMap {
  val data = new HashMap[K, V]
  val perkey = new HashMap[K, Emitter[K]]
  val keys = new Emitter[K]
  val reactive = new Lifted {
    def apply(k: K) = perkey(k) }
  def +=(k: K, v: V) {
    data += (k, v)
    perkey.getOrUpdate(k, new Emitter).emit(k)
    keys.emit(k) }
  def apply(k: K) = data(k) }
```

There is a correspondence between the `apply` method that returns the *scalar* value at the specified key, and the reactive variant of the `apply` method that returns the reactive of the specified key. We will refer to the process of converting a data structure query into a reactive signal as *reactive lifting*, and call these signals *reactive queries*. Making reactive queries efficient reactive values requires *data structure-specific knowledge*, as was shown with hash maps.

3.2 Reactive aggregates

Recall that Section 2.3 introduced the higher-order reactives – special reactive values that produce events of other reactive values. This increase in complexity surprisingly lead to shorter and more concise programs. Reactive containers can also be higher-order – they can contain reactive values and other reactive containers. Again, how this is useful is best motivated by a concrete example. Consider the game engine from Section 2.3 one more time. In the previous example we expressed the light intensity as a higher-order reactive that produces either the light intensity outside or inside. In a real game, the rules of computing scene brightness are more dynamic. Next to a fire even the darkest night appears bright. A flash of a lightning is short, but brighter than a sunny day. The tavern in the night grows darker as you move away, but wielding a sword against a rock gives sparkles of light.

We identify several dependencies above – position, activity, weather conditions, proximity to light sources and current time. As discussed in Section 2.3, expressing lights as functions requires encoding their input and violates separation of concerns. We encode each light source as a signal:

```
def pulse(t: Millis) = time.millis.until(at(t))
val outside = time.hours.map(lightAtHour)
  .zip(weather)(lightAtWeather)
val flash = pulse(90.ms).map(t=>exp(-t/30.ms))
val fire = player.position.map(p=>dist(fire,p))
val sparks = action.filter(isWielding)
  .map(a => pulse(90.ms)).union
```

New signals could be dynamically added to a container:

```
val lights = ReactSet.hashMap[Signal[Double]]
lights += outside += flash += fire += sparks
```

However, a set only allows querying if a signal is present and we need to compute the gross intensity using the screen blend mode. Based on insights from Sections 2.2 and 2.3 we can produce a signal by aggregating values of all the lights:

```
def blend(a: Double, b: Double) = 1-(1-a)*(1-b)
val lights = new ReactCell(
  List(outside, flash, fire, sparks))
val intensity =
  lights.map(ls => aggregate(ls)(blend)).mux
```

The `intensity` is a *dynamic reactive aggregate* – an aggregate whose value is not bound to a fixed set of signals. Reassigning to `lights` ensures that the `intensity` is appropriately recomputed. As argued in Section 2.2, this particular aggregate is inefficient – changing the set of the signals or any signal value requires $O(s)$ updates, where s is the total number of signals. We now show a dynamic reactive aggregate with $O(\log s)$ event propagation time.

Similar to the static aggregate from Section 2.2, a dynamic aggregate uses a balanced tree and assigns signals to leaves. When a signal changes, $O(\log s)$ values on the path to the root are updated. Correspondingly, when a new signal is added, the tree is rebalanced in $O(\log s)$ time. This commutative aggregation tree is shown in Figure 1.

```
class CommuteTree[S, T](
  val get: S => T, val z: T, val op: (T, T) => T
) extends Signal[T]
  with Default[T] with ReactContainer[S] {
  private val leaves = new HashMap[S, Leaf]
  private var root: Node = new Empty
  trait Node {
    def depth: Int
    def above: Inner
    def apply(): T
    def add(leaf: Leaf): Node
    def pushUp() {}
    def housekeep() {} }
  class Empty extends Node {
    def depth = 0
    def above = null
    def apply() = z
    def add(lf: Leaf) = lf }
  class Leaf(s: S) extends Node {
    def depth = 0
    var above = null
    def apply() = get(s)
    override def pushUp() =
      if (above != null) above.pushUp()
    def add(leaf: Leaf) = {
      val n = new Inner(1, this, leaf, null)
      this.above = n
      leaf.above = n
      n.housekeep(); n }
    def remove(): Node = {
      if (above == null) new Empty
      else {
        if (above.left == this)
          above.left = null
        else above.right = null
        above.fixUp() } } }
  def pushUp(x: S) = {
    val leaf = leaves(x)
    leaf.pushUp()
    reactAll(root()) }
  def +=(x: S) = {
    if (leaves(x) == null) {
      val leaf = new Leaf(x)
      root = root.add(leaf)
      leaves(x) = Some(leaf)
      reactAll(root()) } }
```

```
class Inner(
  var depth: Int, var left: Node,
  var right: Node, var above: Inner
) extends Node {
  var value: T = _
  def apply() = value
  override def pushUp() {
    value = op(left(), right())
    if (above != null) above.pushUp() }
  def fixHeight() =
    depth = 1 + max(left.depth, right.depth)
  override def housekeep() {
    value = op(left(), right()); fixHeight() }
  def add(leaf: Leaf) {
    if (left.depth < right.depth) {
      left = left.insert(leaf)
      left.above = this }
    else {
      right = right.insert(leaf)
      right.above = this }
    housekeep()
    this }
  def contractLeft(): Node = {
    if (above == null) {
      right.above == null
      right
    } else {
      if (above.left == this) above.left = right
      else above.right = right
      right.above = above
      right } }
  def fixUp(): Node = {
    val n = {
      if (left == null) contractLeft()
      else if (right == null) contractRight()
      else rebalance() }
    n.housekeep()
    if (n.above != null) n.above.fixUp()
    else n } }
def apply(): T = root.apply()
def -=(x: S) {
  val leaf = leaves(x)
  if (leaf != null) {
    root = leaf.remove()
    leaves(x) = None
    reactAll(root()) } } }
```

Figure 1. Commutative aggregation tree implementation

The commutative aggregation tree is a signal of events `T` and a reactive container of type `S` simultaneously. As a signal, it produces events when the aggregation is updated. As a container, it has `+=` and `-=` methods. The `get` parameter transforms container values `S` to aggregation values `T`. The `pushUp` method can be used to update the tree if the aggregation value of some container value changes – this method climbs the tree and updates the values of nodes on the path to the root. After checking if the value is already present, `+=` updates the root by adding a new leaf with `add`.

The tree consists of three types of nodes – an `Empty` node, a `Leaf` node and an `Inner` node. At any point the `root` might be an empty node or a balanced tree of inner and leaf nodes. Each node type has a value `apply` that returns the aggregation in the corresponding subtree. Empty nodes return the neutral element `z`, leaves call `get` on the corresponding value S and inner nodes return the cached aggregation in their subtree. Each node supports the `add` operation that adds a new leaf and returns the modified tree. Adding a leaf to an empty tree returns the leaf itself and adding to a leaf returns an inner node with two children. The tree maintains the following invariant – the absolute difference in left and the right subtree depths is less than or equal to 1. Also, the inner nodes have a non-`null` left and

right child. To maintain these invariants, adding to an inner node adds the new signal to the subtree with fewer elements. Adding a single signal increases the height of the subtree by at most one, so the tree is guaranteed to be balanced after the add returns. The fact that there are no tree rotations may be surprising, but this follows from the lack of ordering. The tree is called CommuteTree for a reason – it works correctly only given that the aggregation operator is a commutative monoid. It is easy to check that the screen blend mode is commutative, associative and has a neutral element 0.0.

Removing a signal from the tree with -= removes the corresponding leaf. First, the leaf nulls out its reference in the parent. Then, all inner nodes on the path from the leaf to the root are fixed in fixUp. The fixUp method climbs the tree and restores the invariants. If it detects that a child is null, it calls contractLeft to eliminate the inner node. Otherwise, it calls rebalance to restore the depth invariant.

A commutative reactive aggregate is a higher-order container that uses the commutative aggregation tree:

```
class CommuteAggregate[T]
  (val z: T, val op: (T, T) => T)
extends Signal[T] with Default[T]
  with ReactContainer[Signal[T]] {
  val tree = new CommuteTree(s => s(), z, op)
  def +=(s: Signal[T]) = {
    tree += s; s.subscribe(tree.pushUp(_)) }
  def -=(s: Signal[T]) = tree -= s
  def apply() = tree() }
```

We rewrite our previous example to use this aggregate:

```
val intensity = new CommuteAggregate(0, blend)
intensity += outside += flash += fire += sparks
```

The motivating example for reactive dynamic aggregates has the property that the aggregation operator is commutative. In general, programs can require non-commutative operators – the associative reactive aggregate MonoidAggregate is similar to the commutative aggregate and achieves the same event propagation time bounds, but is more complex – its MonoidTree relies on tree rotations to maintain the relative order. We omit the implementation for brevity, but note that its balancing strategy is similar to that of an AVL tree.

As an example of a non-commutative aggregation, assume we have a large document with thousands of paragraphs of text. Each paragraph is a signal of strings. We want to display a list of search term occurrences in real time, updated as the user modifies the document, so we implement the reactive search using monoid aggregates:

```
type Parag = Signal[String]
def search(term: String,
  doc: (Set[Parag], Reactive[Parag])) = {
  val results =
    new MonoidAggregate(Results.empty)(concat)
  for (s <- doc._1)
    results += s.searchFor(term)
  doc._2.subscribe {
    s => results += s.searchFor(term) }
  results }
```

The search encodes the document as a tuple of existing paragraphs and a stream of future ones. It is more natural to encode it as a reactive Signal[String] set and then construct a reactive aggregate. We study how to do this next.

3.3 Composing reactive containers

We saw how different reactive containers expose different reactive queries, but are there queries common to all containers? Containers seen so far had methods += and -= that allowed adding and removing elements. It is reasonable for them to react when elements are inserted and removed, so we extend the reactive containers with inserts and removes:

```
trait ReactContainer[T] {
  def inserts: Reactive[T]
  def removes: Reactive[T] }
```

These reactives allow expressing common container combinators. Container size, the count of elements satisfying a predicate, exists and contains are as simple as:

```
def count(p: T => Boolean) =
  new Signal[Int] with Default[Int] {
    var n = scalarCount(p)
    def apply() = n
    inserts.subscribe { x =>
      if p(x) { n += 1; reactAll(n) } }
    removes.subscribe { x =>
      if p(x) { n -= 1; reactAll(n) } } }
def size = count(x => true)
def exists(p: T => Boolean) = count(p).map(_>0)
def contains(y: T) = exists(x => x == y)
```

Operators above are aggregates with an operator that forms an Abelian group, so they have $O(1)$ event propagation time. For operators that do not have an inverse, the monoidFold uses the aggregation tree to produce a signal:

```
def monoidFold(z: T)(op: (T, T) => T) = {
  val tree = new MonoidTree(z, op)
  foreach(x => tree += x)
  inserts.subscribe { x => tree += x }
  removes.subscribe { x => tree -= x }
  tree }
```

Methods monoidFold and commuteFold, are catamorphisms for reactive containers – they produces a single reactive from a container of values. Their expressive power is akin to foldLeft in functional programming [6].

Container combinators so far return reactive values, and not reactive containers. We call such methods *reactive queries*. The other class of combinators returns reactive containers. We call them *reactive transformers*. For example, the map combinator is a transformer that takes an injective mapping function and returns a mapped container:

```
def map[S](f: T => S) = new ReactContainer[S] {
  val inserts = self.inserts.map(f)
  val removes = self.removes.map(f) }
```

Reactive transformers such as filter, union and scan are similarly expressed with inserts and removes. They

do not store the elements in memory – the resulting containers are *lazy*. This is efficient, but not extremely useful, as the resulting containers support no reactive queries other than `inserts` and `removes`. To force a lazy container into a container with specific reactive queries, we rely on the builder abstraction [7] [9]. All the containers so far had some form of methods `+=` and `-=` used the add and remove elements. We factor these methods into an *incremental builder*. A special transformer called `to` uses incremental builders to construct a container from `inserts` and `removes`.

```
def to[R](b: ReactBuilder[T, R]) = {
  foreach(x => b += x)
  inserts.subscribe { x => b += x }
  removes.subscribe { x => b -= x }
  b.container }
trait ReactBuilder[T, R] {
  def +=(elem: T): Unit
  def -=(elem: T): Unit
  def container: R }
```

Coming back to the example from Section 3.2, we can express reactive document search in a more concise fashion:

```
def search(t: String, doc: ReactSet[Parag]) =
  doc.map(p => p.searchFor(t))
     .to[MonoidAggregate[Results]]
```

4. Reactive isolates

Shared-memory multithreaded programming is difficult to master. Not only is it non-deterministic and prone to data races, but traditional synchronization also allows deadlocks.

A careful reader will notice that the presented abstractions work correctly exclusively in a single-threaded program. This is intentional – a signal or a reactive container must only be used by the thread that created it. Still, utilizing parallel computing resources such as multicore CPUs is crucial. Assume we want to implement a UI that needs to react to mouse and key presses. We could repetitively poll for input events, and forward them to a reactive emitter.

```
val mouse = new Reactive.Emitter[MouseEvent]
while (!terminated) pollMouseEvent() match {
  case Some(me) => mouse emit me }
```

While this might be appropriate in a game engine that repetitively updates frames anyway, it does not suffice when the input API blocks or is callback-based. In these situations, other events cannot propagate until a mouse event arrives. A reactive framework has to express concurrency – a mouse event must be able to arrive concurrently to a keyboard event.

Reactive isolates are control flow entities executed by at most one thread at a time. If two events like a mouse event and a keyboard event arrive at the same time, they are enqueued in the isolate's event queue and serially emitted to the reactive called `source` specific to each isolate. By extending the `Isolate[T]` type, we define a new isolate template. In the following example the `SimpleUI` isolate reacts to different types of `UIEvent`s.

```
class SimpleUI extends Isolate[UIEvent] {
  source.subscribe {
    case MouseEvent(xy, down) => println(xy)
    case KeyEvent(c) => if (c=='q') exit(0) } }
```

Declaring an isolate template does not start an isolate. Reactive Collections require an *isolate system* to start an isolate:

```
def isolate[T](newIsolate: =>Isolate[T])
  (implicit s: Scheduler): Channel[T]
```

An isolate system creates an *isolate frame* that encapsulates isolate state – e.g. its name, event queue and its scheduler. It then creates the isolate's *channel* – the entity which delivers events to the isolate's event queue. The event queue is an internal data structure inaccessible to the programmer. It enqueues and dequeues events in a thread-safe manner:

```
trait EventQueue[T] {
  def enqueue(event: T): Unit
  def listen(f: IsolateFrame[T]): Unit
  def dequeue(): T }
```

The channel is the only connection to the isolate accessible to the programmer. All events emitted to the channel eventually end up on the event queue. However, events cannot be sent to the channel directly – instead, reactive values can be attached to the channel. After they are attached, the channel can be sealed. An isolate *terminates* once its channel is sealed and all the attached reactives unreact.

```
trait Channel[T] {
  def attach(r: Reactive[T]): Channel[T]
  def seal(): Channel[T] }
```

The isolate system implementation decides on a particular event queue and a channel implementation. The *scheduler* decides where and when to execute an isolate. After its `schedule` method is called, it assigns the isolate frame to a thread whenever its event queue is non-empty.

```
trait Scheduler {
  def schedule[T](f: IsolateFrame[T]): Unit }
```

The scheduler can start an isolate on a pool of worker threads, on a UI event thread, on a dedicated thread, piggyback the caller thread or even spawn a new process.

The event queue, the channel and the scheduler encapsulate three different facets of concurrency in Reactive Collections. The event queue determines how to concurrently buffer incoming events [10], the channel specifies how to deliver events to the isolate and agree on its termination, and the scheduler determines where and when to execute an isolate. While the first two are specific to the choice of an isolate system, the third is specified by the programmer when starting an isolate.

5. Evaluation

We compared Reactive Collections against RxJava and Scala.React implementations on an Intel i5-2500 3.30 GHz

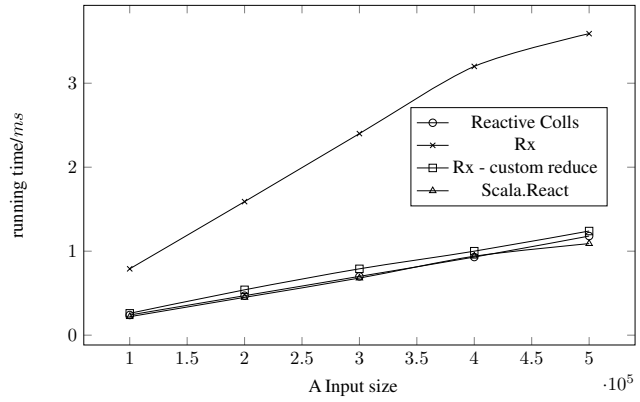

Figure 2. Performance of `scanPast`

quad-core CPU and JVM 1.7 on a simple `scanPast` microbenchmark. As shown in Figure 2, the Rx version was slightly slower than Reactive Collections due to the lack of primitive type specialization. Although this is currently not addressed, Scala allows future versions of Rx to resolve this. The immediate work-around in Rx is to implement a custom reduction observable using the `Observable.create` method. This results in roughly the same performance in Scala.React, RxJava and Reactive Collections.

To validate that Reactive Collections are practical in larger systems we implemented a simplistic 3D game engine, consisting of 8000 lines of user interface and game logic code written in Scala, GLSL and the Reactive Collections framework. Figure 3 shows the engine in action, achieving 50 FPS at high resolution. The first column shows the same scene at different times of the day where light angle, color and intensity are expressed as reactive values depending on the time of the day. By using mutable signals for input events and transformation matrices we were able to avoid most memory allocation and GC-related glitches.

Previously, we wrote the same codebase in a pure FRP style where events were propagated by topologically sorting the signals [3]. We rewrote the engine to the more asynchronous reactive model in this paper. The amount of refactoring required was minimal and consisted of creating several emitters for the phases in each frame. The rest of the game and UI logic surprisingly stayed identical. This seems to indicate that ordering may be overestimated in classical FRP systems and that a minimal amount of programmer interaction can ensure proper semantics.

6. Related work

Reactive programming is a programming paradigm focused around propagation of values and the flow of data. Functional reactive programming aims to express reactive dependencies declaratively, and was established by the work of Elliott and Hudak on Fran [2]. Elm [1], Rx [5] and other FRP frameworks have no concept of containers.

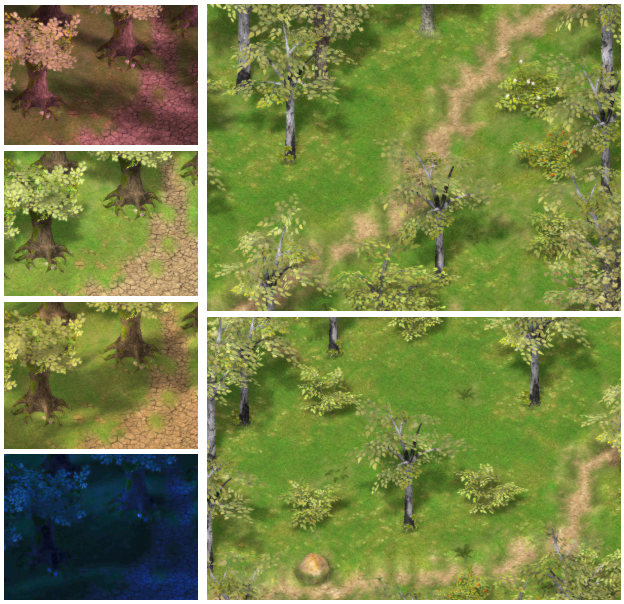

Figure 3. Reactive game engine screenshots

Rx [5] is a reactive programming framework for composing asynchronous and event-based programs using observable collections and LINQ-style query operators. There are several differences with respect to Reactive Collections. First of all, Rx observables can be shared between threads, which can be convenient. Conversely, in Reactive Collections, a reactive can only be used inside one thread (more generally, isolate), and events are propagated between isolates through separate entities called channels. Rx observables have a special `observeOn` combinator that forwards events to a custom event scheduler, which, in turn, may be multithreaded. Reactive Collections use the `isolate` method to bind specific sets of event propagations to a scheduler. Different sets of event propagations communicate through channel objects. Finally, Reactive Collections allow using mutable objects as events inside reactives. Rx also allows mutable objects in observables, but the programmer must guarantee that such events are never propagated to other threads.

Scala.React [4] [3] is a reactive framework written in Scala that aims to fully implement the FRP model. It introduces the concept of *opaque nodes* that define signals as arbitrary expressions of other signals – their dependencies are dynamic and resolved during computation. This occasionally requires rollbacks and makes signal computation slower.

Scala.React has a single reactive container called reactive sequence [4]. Due to opaque signals, sequence combinators are divided in two – slower variants that can use opaque signals and their more efficient scalar versions. In Reactive Collections, this division is avoided by relying on higher-order containers such as aggregates.

Pinte et al. describe a distributed reactive framework known as ambient clouds [8]. In ambient clouds the reactive collections are considered *volatile* – there are no atomic guarantees when querying and composing collections. Where Reactive Collections can distribute isolates using channels and build distributed collections on top of that, in ambient clouds all reactive collections can be accessed concurrently, trading performance and intuitive semantics for convenience.

For reasons of space, in this paper we steered clear from the topic of time and memory leaks [2]. Within our framework we rely on the approach by Maier [3], in which signals keep *weak references* to their reactors. This allows automatically garbage collecting no longer reachable reactives.

7. Conclusion

Our reactive programming model is based on reactive values and their functional composition. We showed how reactive containers propagate events more efficiently than their counterparts based on just reactive values. Finally, we introduced reactive isolates to tackle concurrency. The abstractions in this work can be expressed in terms of three basic primitives: emitters, subscriptions and isolate creation. A calculus that captures the Reactive Collections programming model in terms and shows its expressive power is important future work.

8. Acknowledgements

We would like to thank Erik Meijer for the discussions we had, his useful advice and feedback on this work.

References

[1] E. Czaplicki and S. Chong. Asynchronous functional reactive programming for GUIs. In *PLDI*, 2013.

[2] C. Elliott and P. Hudak. Functional reactive animation. In *ICFP*, 1997.

[3] I. Maier and M. Odersky. Deprecating the Observer Pattern with Scala.react. Technical report, 2012.

[4] I. Maier and M. Odersky. Higher-order reactive programming with incremental lists. In *ECOOP*, 2013.

[5] E. Meijer. Your mouse is a database. *CACM*, 55(5), 2012.

[6] E. Meijer, M. M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *FPCA*, 1991.

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

[8] K. Pinte, A. Lombide Carreton, E. Gonzalez Boix, and W. Meuter. Ambient clouds: Reactive asynchronous collections for mobile ad hoc network applications. In J. Dowling and F. Taïani, editors, *Distributed Applications and Interoperable Systems*, volume 7891 of *Lecture Notes in Computer Science*, pages 85–98. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-38540-7. . URL `http://dx.doi.org/10.1007/978-3-642-38541-4_7`.

[9] A. Prokopec, P. Bagwell, T. Rompf, and M. Odersky. A generic parallel collection framework. In *EuroPar*, 2011.

[10] A. Prokopec, H. Miller, T. Schlatter, P. Haller, and M. Odersky. FlowPools: A lock-free deterministic concurrent dataflow abstraction. In *LCPC*, 2012.