Functional Reactive Programming with nothing but Promises

Implementing Push/Pull FRP using JavaScript Promises

Alan Jeffrey   Tom Van Cutsem

Alcatel-Lucent Bell Labs

ajeffrey@bell-labs.com, Tom.Van_Cutsem@alcatel-lucent.com

Abstract

Functional Reactive Programming (FRP) is a model of reactive programming defined by having a well-defined semantics given by time-indexed values. Promises are one-shot communication channels which allow asynchronous programs to be written in a synchronous style. In this paper, we show how *timed promise lists*, a timestamped linked list structure using promises rather than pointers, can be used to implement FRP. This idea originated with Elliott's *Push/Pull FRP*, and we show that it can be expressed idiomatically in a strict functional language with promises, JavaScript. We identify a potential space leak with JavaScript's built-in promises and propose an alternative implementation that avoids the leak.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming

1. Introduction

1.1 Overview

Functional Reactive Programming (FRP) [9, 17] is a model of reactive programming distinguished by having a well-defined semantics given by time-indexed values. It allows a developer to write reactive programs without descending into the 'callback hell' of event-driven systems. FRP is agnostic about its implementation, which may be pull-driven functions, or push-driven event handlers.

Promises [2, 10] are also intended to avoid the complexity of event-driven systems, but are one-shot communication channels: once a promise is *resolved* to a value, it stays fixed. Promises were added to JavaScript in ECMAScript 6 (ES6) [7]. They allow asynchronous programs to be written in a synchronous style.

In this paper, we investigate using promises in an implementation of FRP. The implementation uses *timed promise lists*, and is a port of Elliott's *Push/Pull FRP* [8] from Haskell to JavaScript.

1.2 Motivating Example

The 'hello, world' application for reactive programming is an echo chatbot, which just repeats back any messages sent to it, as seen in Figure 1. Our implementation uses the XMPP [22] chat protocol, but this is mostly invisible to the developer.

[Copyright notice will appear here once 'preprint' option is removed.]

Applications such as this typically use configuration files containing constant bindings such as user identifiers, passwords, and natural-language strings for internationalization. An example configuration for the echo chatbot is in Figure 2.

For simple systems, a change of configuration is straightforward: edit the configuration file, then restart the server. This becomes problematic as systems become more complex: the delay and down time of system restart may be unacceptable, and the server may store session state that has to be saved and restored. As a result, systems are often better served by dynamic configuration, which allows configuration change without system restart.

A typical implementation of a chatbot is as an event-driven system, with explicit event handler registration and deregistration. Such systems rapidly become complex, especially in the presence of multiple event sources. For example, the echo chatbot has two event sources: configuration file changes, and chat messages. Most implementations of the echo chatbot would suffer from nondeterministic transient behavior. For example, if the `jid` (the XMPP user id) were changed from `alice@example.com` to `bob@example.com`, it would be very easy for a message received by Alice to be responded to by Bob. Taming such behavior requires concurrency controls such as locks or transactions.

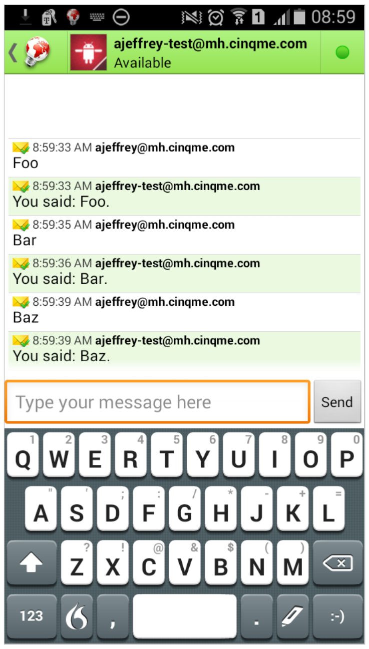
In this paper, we adopt the philosophy of (*discrete-time*) *Functional Reactive Programming* (FRP), in which events are no longer the primary concern. Instead, the API is concerned with *reactive values*, whose semantics is given as time-indexed values. We use TypeScript [20] as a language of types for JavaScript, writing `R<T>` for reactive values of type `T`. These have semantics as time-indexed values of type `T`:

$$[\![\texttt{R<T>}]\!] = \mathsf{Time} \to [\![\texttt{T}]\!]$$

A key property of FRP is that the kind of nondeterministic transient behavior outlined earlier can be prevented by consistently reading the value of multiple reactive values at the same time t. In FRP terminology, such nondeterministic transient behavior is called a *glitch* [4]. It is the task of an FRP library or language to prevent such glitches.

1.3 Promises

Promises [10] also known as futures [2] were introduced as a mechanism that allows a producer thread to calculate a value, and for any consumer threads to block waiting for the value to be produced (this is called *resolving* the promise). The E programming language [15] pioneered an asynchronous interface to promises: rather than blocking waiting for the value to be produced, the consumer can register a callback to be executed when the promise is resolved. The result of registering a callback is itself a promise, so promises can be *chained*, in a style similar to monadic programming. ES6 Promises are based on this asynchronous interface.


08:59

ajeffrey-test@mh.cinqme.com
Available

8:59:33 AM ajeffrey@mh.cinqme.com
Foo

8:59:33 AM ajeffrey-test@mh.cinqme.com
You said: Foo.

8:59:35 AM ajeffrey@mh.cinqme.com
Bar

8:59:36 AM ajeffrey-test@mh.cinqme.com
You said: Bar.

8:59:39 AM ajeffrey@mh.cinqme.com
Baz

8:59:39 AM ajeffrey-test@mh.cinqme.com
You said: Baz.

Figure 1. An echo chatbot screenshot

```
{ "xmpp": {
    "jid": "ajeffrey-test@mh.cinqme.com",
    "password": "abc"
  },
  "strings": {
    "prefix": "You said: ",
    "suffix": "."
} }
```

Figure 2. Echo chatbot configuration

In this paper, for simplicity we are ignoring errors, and the fragment of ES6 promises [7] we are using is given in Figure 3. The type `Promise<T>` is inhabited by promises which resolve to a (possibly `undefined`) value of type `T`. Note that in ES6 promises, `then` and `resolve` are overloaded to operate both on promise and non-promise values. As a result, the API doesn't allow for the construction of a `Promise<Promise<T>>`.

There is also a function `race(ps)`, which is resolved when the first `ps[i]` is resolved. It plays a similar role to McCarthy amb [13], and we use it for the same reason as Elliott [8, §10].

1.4 Contributions

In §2 we present a simple library for reactive programming in JavaScript. The API of this library is fairly straightforward and is not this paper's main contribution. Our main contribution is described in §3 where we show how FRP can be implemented using ES6 promises, using a data structure we are calling a *timed promise list*. This data structure is essentially the same as Elliott's *Push/Pull FRP* implementation of reactives [8]. We show that timed promise lists can be expressed idiomatically in a strict functional language with promises.

In §4 we identify a potential space leak when implementing FRP using JavaScript's built-in Promise API. We provide an alternative Promise implementation using which the leak can be avoided.

```
interface Promise<T> {
  then (fun: (arg?: T) => Promise<U>): Promise<U>;
  then (fun: (arg?: T) => U): Promise<U>;
}
function resolve<T> (val?: Promise<T>): Promise<T>;
function resolve<T> (val?: T): Promise<T>;
function race<T> (ps: Promise<T>[]): Promise<T>;
```

Figure 3. Promises API (ignoring errors)

2. Using Reactive Values

Before diving into the details of how we *represent* reactive values using timed promise lists, we discuss how one can *use* reactive values using a simple JavaScript API. We use the running example of the echo chat bot from the Introduction to illustrate the API.

Recall that our chat bot has two event sources: configuration file changes, and chat messages. Let's first focus on dealing with configuration file changes. Our file system API includes a function `readJSON` that reflects the current contents of a JSON file. In an event-driven system, using `readJSON` requires registering event handlers to respond to each file change. In our FRP system, `readJSON` just returns a reactive value:

```
function readJSON (name: string): R<any>;
```

Since file names are often specified in configuration files, (and TypeScript supports function overloading) we can give a type for `readJSON` that allows the file name to be reactive:

```
function readJSON (name: R<string>): R<any>;
```

Our XMPP API further includes a function `login` which logs into an XMPP server, and returns two values: an `inbox` with the input messages, and a `send(outbox)` function called with the output messages:

```
function login (config: R<any>): {
  inbox: R<any>;
  send (outbox: R<any>);
};
```

In Figure 5 we give an echo chatbot implementation. For readability, the types for each variable are listed separately. To understand the code, one must consider two layers of abstraction: the code makes use of high-level bridging functions for converting reactive objects to and from objects containing reactives. These convenience functions are implemented in terms of a simple core API. We first explain the convenience functions and then the core API.

Bridging Objects and Reactives `filter` is a bridging function to convert from reactives-containing-objects to objects-containing-reactives. The `filter` function on a reactive takes as its argument a *pattern*. A pattern is either:

- `true`, which matches anything,
- `false`, which matches nothing,
- a regular expression `r`, with the usual definition of matching,
- a function `f`, which matches `v` if `f(v)` is true, or
- an object `o`, which matches an object `v` if `o.f` matches `v.f` for every field `f` of `o`.

On reactives, `filter` is `undefined` when the pattern does not match. If `vs` is a reactive, and pattern `p` has field `f`, then so does `vs.filter(p)`. Moreover `vs.filter(p).f` is equivalent to `map(getF, vs.filter(p))` where `getF(v)` is `v.f` when `v` is defined, and `undefined` otherwise. The `map` function is part of our core API, explained below.

```
function constant<T> (
  val: T
): R<T>;
function map<T,U> (
  fun: (arg: T) => U,
  vals: R<T>
): R<U>;
function zip<T1,T2,U> (
  fun: (arg1: T1, arg2: T2 )=> U,
  vals1: R<T1>,
  vals2: R<T2>
): R<U>;
```

$$\begin{aligned}
[\![\texttt{constant(v)}]\!](t) &= [\![\texttt{v}]\!] \\
[\![\texttt{map(f,vs)}]\!](t) &= [\![\texttt{f}]\!]([\![\texttt{vs}]\!](t)) \\
[\![\texttt{zip(f,vs,ws)}]\!](t) &= [\![\texttt{f}]\!]([\![\texttt{vs}]\!](t), [\![\texttt{ws}]\!](t))
\end{aligned}$$

Figure 4. Core FRP API

This feature lets us use the familiar dot-notation to access properties on reactive objects. In the example code, `config` is a reactive value constructed by `filter`. In the call to `XMPP.login`, we can simply write `config.xmpp` to construct a derived reactive value. Given the transformation described above, this is equivalent to:

```
function getXMPP(c) { return c.xmpp; }
var xmpp = XMPP.login(Reactive.map(getXMPP, config))
```

The other way around, `Reactive.build` converts an object-containing-reactives to a reactive-containing-objects. `build` takes as its single argument an object whose properties may contain reactive values (let's call these "sources"). It returns a new reactive value that, at any given time t, has the same shape as its input argument but with all sources replaced by their value at time t. In order to consistently read the value of *multiple* reactives, the implementation of `build` depends on a core API function called `zip`. In Figure 5, `build` is used to construct a reactive value called `output`. It is equivalent to:

```
function mkOut(c,i) { return { message: {
  to: i.message.from,
  body:
    [ c.strings.prefix,
      i.message.body,
      c.strings.suffix ]
} }; }
var output = Reactive.zip(mkOut,config,input);
```

Core API The bridging functions are essentially convenience functions to work with reactive values using an object-oriented API. Under the hood, however, these functions rely on a functional core API, which is given in Figure 4. `constant` lifts any value inside a reactive value. `map` maps a unary function over a single reactive, returning a new reactive. Finally, `zip` maps a binary function over two reactive values. Given `zip`, it is easy to construct a multi-way map that operates on three or more inputs.

As we shall see later, mapping a function over a single reactive is much more straightforward than mapping a function over multiple reactive values. This is because `zip` must pair elements from its input reactive values such that the function passed to `zip` only ever sees values consistent with time. In other words, `zip` must prevent glitches.

A program using the bridging functions of our FRP library can be thought of as a staged computation: in the first stage, the bridging functions internally 'link up' reactive values via calls to `constant`, `map` and `zip`. The 'residual' of this stage is effectively an implicitly constructed dataflow dependency graph. In the next stage, reactive values start changing and their updated values flow through this graph. At this stage, only core API functions are involved.

So far, we have considered values of type `R<T>` as black boxes to be manipulated using the core API. In the next section, we turn our attention to the *representation* of reactive values, and how to implement the core FRP API given a concrete representation.

```
var XMPP = require('frp-xmpp');
var FS = require('frp-fs');
var Reactive = require('frp-reactive');

// Load and validate configuration
var config = FS.readJSON('config.json').filter({
  xmpp: true,
  strings: {
    prefix: true,
    suffix: true
  } });

// Log in to the xmpp server
var xmpp = XMPP.login(config.xmpp);

// Grab the input messages
var input = xmpp.inbox.filter({
  message: {
    from: true,
    body: true
  } });

// Generate the output messages
var output = Reactive.build({
  message: {
    to: input.message.from,
    body:
      [ config.strings.prefix,
        input.message.body,
        config.strings.suffix ]
  } });

// Send the output
xmpp.send(output);

// The types of the above variables
declare var config: R<{
  xmpp: any;
  strings: { prefix: any; suffix: any; };
}>;
declare var xmpp: {
  inbox: R<any>;
  send(outbox: R<any>);
};
declare var input: R<{
  message: { from: any; body: any; };
}>;
declare var output: R<{
 message: { to: any; body: any[]; };
}>;
```

Figure 5. Echo chatbot source code

3. Implementing Reactive Values

Recall that reactive values R<T> have semantics as time-indexed values of type T. We therefore aim to represent a reactive value as a linked list of timestamped values, each entry representing its value at the given time. A reactive value is "updated" simply by appending a new timestamped entry to its list. Memory is reclaimed by the garbage collector in the usual fashion; the implementation takes care not to keep list entries alive longer than necessary.

We make no assumptions on the particular timestamps used other than that they are comparable numbers. In particular, note that the timestamps are only loosely coupled to wall clock time. Due to processing delays, a change with timestamp t may be processed at a time significantly later than t. In an extreme case (such as a laptop or virtual machine being suspended and resumed) there can be arbitrary delay. A timestamp may also represent logical time, implemented as a simple counter.

3.1 Attempt 1: linked lists

Our first attempt at representing reactive values uses a traditional linked list implementation, where a list is either empty (represented by the object { first: undefined }) or a cons cell with a timestamp t, a head v and a tail vs (represented by the object { first: { time: t, head: v, tail: vs } }).

```
interface List<T> {
  first?: Cons<T>;
}
interface Cons<T> {
  time: number;
  head: T;
  tail: List<T>;
}
```

We use this to represent a reactive value by recording the times when the value changes, for example a reactive value which starts out as hello, and at time t becomes world is represented as:

```
var hw = { first: {
  time: -Infinity,
  head: "hello",
  tail: { first: {
    time: t,
    head: "world",
    tail: { first: undefined }
  } } } };
```

In a lazy language, this could form the basis of a streaming I/O library, in the style of Haskell's lazy I/O, but in a strict language such as JavaScript, lists do not allow for future behaviour, where the list's contents are only known after the list is created.

3.2 Attempt 2: promise lists

Promises are designed for exactly this kind of future behaviour, where the value a promise contains is only known after the promise is created. We can adapt the traditional linked list to a *promise list*.

```
interface PList<T> {
  first: Promise<PCons<T>>;
}
interface PCons<T> {
  time: number;
  head: T;
  tail: PList<T>;
}
```

Promise lists are a well-known folklore data structure. They are a common idiom in the E language [15] and have their roots in concurrent logic programming [19] (where an unbound logic variable plays the role of a promise). They are related to lazy linked lists, except that a lazy list is often driven by the consumer of the list, while promise lists can only be driven by the producer.

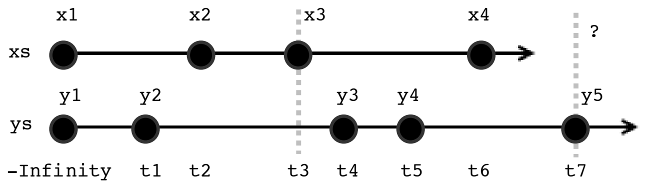
The representation of reactive values is the same as for linked lists, but with promises, for example:

```
var hw = { first: resolve({
  time: -Infinity,
  head: "hello",
  tail: { first: resolve({
    time: t,
    head: "world",
    tail: { first: resolve() }
  }) } }) };
```

We now need to implement the three core API functions, assuming reactive values are represented as promise lists. The functions constant and map are simple to implement:

```
function constant(x) { return {
  first: resolve({
    time: -Infinity,
    head: x,
    tail: resolve()
  }) } }
function map(f,xs) { return {
  first: xs.first.then(function(cons) { return {
    time: cons.time,
    head: f(cons.head),
    tail: map(f,cons.tail)
  } }) } }
```

The main challenge lies in the implementation of zip. On ordinary lists, zip traditionally pairs elements from the input lists based on their *position* in the list. However, given two promise lists, our zip function needs to pair up elements by time, *not* position, which merely indicates event arrival order. The diagram below depicts two reactive values xs and ys. When zipped, x3, which is the third update to xs, must be paired with y2, as that is the value of ys at time t3, even though y2 is only the second update of ys.



Unfortunately, zip cannot be implemented for promise lists. The problem is that in zip(f,xs,ys) we may have that ys.first resolves to { time: t7, head: y5, tail: vs }. In order for zip(f,xs,ys) to resolve to { time: t7, ... } we need to know that xs does not have any changes before time t7. Currently, the implementation has no way to get this negative information: we can find out that a list has an element at time t, but not that it has *no* elements before time t.

3.3 Finally: timed promise lists

The ability to find out if a list has any elements before time t is the missing piece of the implementation. We can add this function, remembering that this is potentially future information, and so must be wrapped in a promise.

```
interface TPList<T> {
  first: Promise<TPCons<T>>;
  before (time : number) : Promise<boolean>;
}
```

```
interface TPCons<T> {
  time: number;
  head: T;
  tail: TPList<T>;
}
```

If xs.before(t) eventually resolves to false, then we know that xs will not generate any elements timestamped before t. In other words, the next element of xs will have a logical timestamp greater than t.

Timed promise lists were introduced (using different terminology) in Push/Pull FRP [8]. The representation of reactive values is the same, but with negative information given by the before functions, for example:

```
  var ptrue = resolve(true);
  var pfalse = resolve(false);
  function p(b) { return (b? ptrue: pfalse); }
  var hw = {
    before: function() { return ptrue; },
    first: resolve({
      time: -Infinity,
      head: "hello",
      tail: {
        before: function(u) { return p(t <= u); }
        first: resolve({
          time: t,
          head: "world",
          tail: {
            before: function() { return pfalse; }
            first: resolve()
          } }) } }) };
```

We maintain some invariants for TPList objects:

1. Any externally visible list xs has xs.first resolve to cons where cons.time is -Infinity.
2. If xs.first resolves to cons1 and cons1.tail.first resolves to cons2 then cons1.time < cons2.time.
3. xs.before(t) is true if and only if xs.first resolves to cons where cons.time <= t.

In implementing zip the important function is mint(xs)(t), which returns the smaller of xs' timestamp or t.

```
  function getTime(cons) {
    return (cons? cons.time: Infinity);
  }
  function mint(xs) { return function(t) {
    return xs.before(t).then(function(b) {
      return (b? xs.first.then(getTime): t);
    }); } }
```

From this, we can write a function first(xs,ys), which returns the smaller timestamp of xs and ys.

```
  function first(xs,ys) {
    var p = xs.first.then(getTime).then(mint(ys));
    var q = ys.first.then(getTime).then(mint(xs));
    return race([p,q]);
  }
```

From first, we can implement zip. We first implement a function ffwd(t)(xs) (where xs is a cons cell) which returns the first suffix ys of xs such that ys.tail.before(t) resolves to false:

```
  function ffwd(t) { return function(xs) {
    return xs.tail.before(t).then(function (b) {
      return (b? xs.tail.first.then(ffwd(t)): xs);
    }); } }
```

We then implement zip on cons cells by finding the timestamp of the first change, fast-forwarding both lists to that time, and building the result (for simplicity, we ignore the case of the empty list, but it is straightforward to handle):

```
  function zipc(f,xs,ys) {
    return first(xs.tail,ys.tail).then(function(t) {
      return ffwd(t)(xs).then(function(xs) {
        return ffwd(t)(ys).then(function(ys) {
          return {
            time: t,
            head: f(xs.head,ys.head),
            tail: {
              before: either(xs.tail,ys.tail),
              first: zipc(f,xs,ys)
            } } }) }) }); }
```

where either lifts disjunction up to acting on before functions:

```
  function either(xs,ys) { return function(t) {
    return xs.before(t).then(function (b) {
      return b || ys.before(t);
    }); } }
```

From this, zip is direct:

```
  function zip(f,xs,ys) {
    return xs.first.then(function(xs) {
      return ys.first.then(function(ys) {
        return {
          before: function(t) { return ptrue; },
          first: zipc(f,xs,ys)
        } }) }); }
```

With zip completed, we have now fully implemented the core reactive API for timed promise lists. Note how zip is much more involved and less efficient than map. This is because zip needs to match up elements from the corresponding timed promise lists based on their timestamp, not their position. It is this correlation of elements based on their time that prevents glitches.

4. External resources and space leaks

4.1 Interfacing to event-driven APIs

Most libraries for reactive programming in JavaScript are event-driven, so we provide a binding from callbacks to reactive values (implemented as timed promise lists). Ignoring errors, the API for doing so is:

```
  function mapGenerator<T,U> (
    generator: (
      value: T,
      init: (arg: U) => void,
      append: (arg: U) => void
    ) => { close : () => void },
    values : R<T>
  ) : R<U>;
```

This calls generator(value,i,a) for each value in the input reactive, where i is a function to initialize the output reactive and a is a function to update the output reactive (i.e. append a new item to its TPList). Each call to generator may append multiple values, all of which are 'flattened' into a single output reactive.

The generator function may allocate resources. These should be cleaned up by the close method returned by the callback, which is called every time the input reactive changes.

As an example, consider creating a reactive value that represents the current contents of a file. Whenever the file is updated, the value should be updated as well. Moreover, the file to read from may itself change over time (e.g. its name may be read from a reactive configuration file). Reading the contents of the file must be done using the platform's built-in event-driven API (e.g. node.js's `fs` module), which uses callbacks, so we need to bridge between the FRP world and the event-driven world using `mapGenerator`:

```
function readText(fnames: R<string>): R<string> {
  function generator(fname,init,append) {
    fs.readFile(fname, function(err,data) {
      init(data);
    });
    var watcher = fs.watch(fname, function() {
      fs.readFile(fname, function(err,data) {
        append(data);
      }) });
    return {close: function() {watcher.close()}};
  }
  return mapGenerator(generator, fnames);
}
```

`mapGenerator` internally timestamps each item. Items added with `init` have the same timestamp as the `value` that generated them, after that the wall clock is used to generate timestamps. By generating the timestamps inside the FRP library, we can maintain the invariants on time described earlier.

An issue with interfacing events to FRP using `mapGenerator` is what to do with events which are generated 'too late'. For example, if `mapGenerator(g,vs)` is used to generate an event stream, and `vs` contains `v` at time `s` and `w` at time `t`, then `g` will be called twice, with value `v` then value `w`. If the first call generates an item at time `u>t`, then what should be done with it? If we emit the item, then it was an event generated from `v` rather than `w`, and so we have a potential violation of consistency. If we don't emit the item, then we are losing events. There is no clear winner here; in our implementation we chose to lose events and keep consistency.

4.2 Space usage

The timed promise list implementation has mostly the same space usage as any linked list implementation does. Care must be taken not to keep lists live longer than necessary, as this can cause an entire event stream to buffer in memory, and not to be garbage collected. There is one such source of space leaks, caused by the space usage of the promise library. To understand the leak, we first need to briefly outline the implementation of promises.

A promise may hold onto two resources: if it is resolved, it stores a reference to its resolved value. If it is unresolved, it holds a queue of observers to be notified with its value, if and when the promise gets resolved. When using `p2 = p.then(f)` to map a function `f` over a promise `p`, an observer is added to `p` which, when notified of the promise's value `v`, in turn resolves `p2` with `f(v)`.

Our implementation of `zip(f,xs,ys)` makes use of a call to `Promise.race([p,q])`, where `p` is a promise waiting on `xs.first` to resolve, and `q` is a promise waiting on `ys.first`. If `xs` is a slow-changing value (for example a configuration file) and `ys` is a fast-changing value (for example a stream of chat messages) then there may be many calls of `race([p,q])` where `q` resolves much faster than `p`. This is fine, as long as `race([p,q])` reclaims any space used (in particular, observers it has registered with all the promises that *lost* the race) once either `p` or `q` has resolved. Unfortunately, the space model for ECMAScript promises does not make this guarantee, and so there is a potential space leak.

The solution to this leak is to reimplement `race` as follows: in a call to `race(ps)`, when `ps[i]` wins the race, `race` must explicitly deregister the observers still registered on each of the other `ps[j]` input promises. Unfortunately, the ECMAScript Promise API does not export a 'deregister' method to be able to remove a previously registered observer from a promise. As such, our only option is to reimplement the entire Promise API. A reimplementation of the core Promise API with a leak-free implementation of `race` is provided in the appendix. The trade-off is thus between using built-in promises (with the potential for significant performance gains and optimization) against non-native promises (with stronger space usage guarantees).

5. Discussion and Related Work

5.1 Synchronous vs non-synchronous FRP

FRP systems embedded in strict languages, and JavaScript-based FRP systems in particular, typically employ an evaluation strategy that involves the construction of a dependency graph between reactive values, combined with an update propagation algorithm that fires updates in strict dependency order (to prevent glitches) [4]. In these systems, updates propagate through the dataflow graph synchronously, i.e. all nodes must be visited before the next event is processed. Changes are not (and need not be) explicitly timestamped. Exemplars include FlapJax [14] and BaconJS [1].

Elm [5] is a strict functional language that compiles to JavaScript. In Elm, multiple updates can propagate in a pipelined fashion through the dataflow graph, each node processing its inputs at its own pace. Each input edge in the dataflow graph acts as a buffer to hold input events as they await processing. Glitches are prevented by a global scheduler that sends out a special `noChange` event to all nodes whose input does not change. As such, there is a single global update rate for all reactive values (called 'Signals'), permitting simple and fast position-based rather than time-based pairing of multiple inputs. The downside of this strategy is that the overall rate of updates through the graph is determined by the fastest-changing signal, even if the majority of input signals change only slowly.

RxJS [18] is a port of the .NET Rx library to JavaScript. However, RxJS's reactive values (called Observables) have no time-indexed semantics and in fact RxJS does not prevent glitches. RxJS exposes a `zip` function that pairs elements from multiple Observables based on position, not based on time.

Our representation of reactive values as timed promise lists – a timestamped linked list structure – is based on Elliott's work on *Push/Pull FRP* [8]. We depart from the synchronous update strategy, instead allowing updates to take place completely asynchronously (i.e. without a global scheduler). Glitches are prevented by 1) explicitly timestamping values and 2) keeping a history of values (as opposed to only the most recent value), thus allowing a program to observe a consistent view across multiple reactive values at any time t by using only values whose timestamp indicates it is valid at time t. Timed promise lists thus act as buffers, keeping history as long as necessary for the slowest reader to catch up.

The strategy of keeping a history of timestamped values bears resemblance to Multiversion Concurrency Control (MVCC) [3], an optimistic concurrency control protocol for databases and software transactional memory. An MVCC system stores multiple versions of each value, allowing a transaction to access the version of the value in place when the transaction started, even if it was modified in the mean time by another transaction. This ensures that the transaction always sees a consistent 'snapshot' of the database at a particular point in time. Space leaks can be prevented by storing only the last n versions. A lagging transaction that requests a 'forgotten' version can simply be rolled back and restarted.

5.2 Applicative vs monadic FRP

An issue brought up by `mapGenerator` is that of resource reclamation. We allow resources to be cleaned up in `mapGenerator` by the `close` method returned by the callback. For example, in `readText` we use resource reclamation to stop watching a file. There is no other support for resource reclamation, and in general there is no way for a user of a timed promise list to indicate that no further items are required, and that the list should reclaim any resources it has allocated.

This is visible from the fact that our FRP library is an applicative functor [12] rather than a monad [16, 21]. In particular there is no function to flatten a reactive-of-reactives to a reactive:

```
function doesntExist<T> (xss: R<R<T>>): R<T>;
```

$$[\![\texttt{doesntExist(xss)}]\!](t) = [\![\texttt{xss}]\!](t)(t)$$

The reason why no such function exists is that it causes problems for resource reclamation. If `xss` includes `ys` at time `s` and `zs` at time `t`, then at time `t`, we should close down `ys`, and reclaim any of its resources. However, no such function exists on timed promise lists, and introducing one would require implementing a form of reference counting garbage collector. Note that Elliott [8] provides a monadic FRP implementation using timed promise lists. This is not a contradiction, as he does not provide bindings to event-driven systems, and so does not need a treatment of resource reclamation.

This is an instance of the general difference between FRP as an applicative functor, and FRP as a monad [5, §3.3]. FRP implementations typically build a data flow graph which is then executed. In applicative FRP (e.g. this work and Elm [5]) the data flow graph is static, and does not change after it has been built. In monadic FRP, the data flow graph is dynamic, and may change after it is first built (as in FlapJax [14]). When the data flow graph is dynamic, nodes in the graph may become unreachable and require garbage collection.

6. Further Work

There are a number of issues left unresolved by the implementation of FRP using timed promise lists.

Errors: In this paper, we have ignored errors, but this should be addressed. A treatment of errors may distinguish between catastrophic errors (such as programmer error) and expected errors (such as missing files or parse errors).

Optimization: There are many opportunities for optimizing FRP programs. In particular, `map` is much more efficient than `zip`, as it does not need to use `first` to find the smallest timestamp of its arguments. In the case where `xs` and `ys` produce changes at the same rate, however, `zip(f,xs,ys)` can be implemented with the same efficiency as `map(f,xs)` (`zip` can then simply pair up elements from the lists by their arrival position).

One way of achieving this optimization is by changing the timed promise list representation such that the timestamps are separated from the actual values. If we call a promise list of timestamps a "clock", and if we link each reactive value to such a clock, then `zip(f,xs,ys)` can simply check whether the clocks of `xs` and `ys` are the same. If they are, `xs` and `ys` will update at the same rate and `zip` can use simple positional pairing of elements. If they are not, `zip` must continue to use time-based pairing.

Distribution: Existing approaches to distributed FRP are based on propagating updates in explicitly constructed dependency graphs [6, 11]. The main issue with distributing our FRP implementation is that timestamps are given as `numbers`, and so there is a global total order on timestamps. This is problematic for distribution, which would more naturally use a partial order such as vector clocks. It is not obvious how to use timed promise lists in the case of partially ordered time.

7. Conclusion

We have shown how Elliott's *Push/Pull FRP* [8] can be idiomatically expressed in a strict language (JavaScript) using promises. The key observation is that reactive values can be represented as *timed promise lists*. A simple reactive API (consisting of `constant`, `map` and `zip` functions) can be implemented on top of timed promise lists. The implementation of `zip` is challenging, as elements must be paired based on time, not position, in order to prevent glitches. Internally, `zip` uses JavaScript's `Promise.race` function, which can lead to space leaks when racing a fast-changing with a slow-changing timed promise list. Avoiding this space leak requires extending the Promise API with the ability to deregister Promise observers.

References

[1] Bacon.js. `https://baconjs.github.io/`.

[2] H. Baker and C. Hewitt. The incremental garbage collection of processes. In *Proc. Symp. Artificial Intelligence Programming Languages*, pages 55–59, 1977.

[3] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, 1981.

[4] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *Proc. ESOP 2006*, pages 294–308, 2006.

[5] E. Czaplicki. Elm: Concurrent FRP for functional guis. Senior thesis, Harvard U., 2012.

[6] J. Drechsler, G. Salvaneschi, R. Mogk, and M. Mezini. Distributed REScala: An update algorithm for distributed reactive programming. In *Proc. OOPSLA 2014*, pages 361–376, 2014.

[7] Draft 6th ed ECMAScript language specification. ECMA-262, 2015.

[8] C. Elliott. Push-pull functional reactive programming. In *Proc. Haskell Symp.*, 2009.

[9] C. Elliott and P. Hudak. Functional reactive animation. In *Proc. Int. Conf. Functional Programming*, pages 263–273, 1997.

[10] D. Friedman and D. Wise. Aspects of applicative programming for parallel processing. In *Proc. Int. Conf. Parallel Processing*, pages 263–272, 1976.

[11] A. Margara and G. Salvaneschi. We have a DREAM: Distributed reactive programming with consistency guarantees. In *Proc. DEBS 2014*, pages 142–153, 2014.

[12] C. McBride and R. Paterson. Applicative programming with effects. *J. Functional Programming*, 18(1):1–13, 2008.

[13] J. McCarthy. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*. North-Holland, 1963.

[14] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: A programming language for ajax applications. In *Proc. OOPSLA 2009*, pages 1–20, 2009.

[15] M. S. Miller, E. D. Tribble, and J. S. Shapiro. Concurrency among strangers: Programming in E as plan coordination. In *Proc. Int. Symp. Trustworthy Global Computing*, pages 195–229, 2005.

[16] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[17] H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *Proc. Workshop on Haskell*, pages 51–64, 2002.

[18] Rxjs: The reactive extensions for javascript. `https://rxjs.codeplex.com/`.

[19] E. Shapiro, editor. *Concurrent Prolog: Collected Papers*. MIT Press, 1987.

[20] Typescript. `http://typescriptlang.org/`.

[21] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.

[22] The XMPP standards foundtion. `http://xmpp.org/`.

A. Leak-free Promises

```
// Based on https://github.com/then/promise/blob/master/lib/core.js
// Adds two new callbacks to promise creation:
//
//   new Promise(init,undo,redo);
//
// acts as new Promise(init) except that when there are no more observers,
// undo() is called, and if new observers then arrive, redo() is called.
// A typical usage is with:
//
//   function init(resolve,reject) { ...; allocate resources; }
//   function undo() { ...; deallocate resources; }
//   function redo() { ...; allocate resources; }
//
// Under the hood, this uses reference counting
var uids = 0;

function doResolve(fn, onFulfilled, onRejected) {
  var done = false;
  try {
    fn(function (value) {
      if (done) { return; }
      done = true;
      onFulfilled(value);
    }, function (reason) {
      if (done) { return; }
      done = true;
      onRejected(reason);
    });
  } catch (ex) {
    if (done) { return; }
    done = true;
    onRejected(ex);
  }
}

function Promise(fn,undo,redo) {
  // ... typechecking on fn, undo, redo omitted
  var state = null;
  var value = null;
  var resolves = {};
  var rejects = {};
  var observers = 0;
  var self = this;

  this.then = function(onFulfilled, onRejected) {
    var resolve3; var reject3;
    function init(resolve, reject) {
      function resolve2(val) {
        var ret;
        try { ret = onFulfilled(val); } catch (e) { return reject(e); }
        resolve(ret);
      }
      function reject2(err) {
        var ret;
        try { ret = onRejected(err); } catch (e) { return reject(e); }
        resolve(ret);
      }
      resolve3 = (typeof onFulfilled === 'function'? resolve2: resolve);
      reject3 = (typeof onRejected === 'function'? reject2: reject);
      self.addObservers(resolve3, reject3);
    }
    function undo() {
      self.rmObservers(resolve3, reject3);
    }
    function redo() {
      self.addObservers(resolve3, reject3);
    }
    return new Promise(init,undo,redo);
  };

  this.catch = function(onRejected) {
    return self.then(null,onRejected);
  };

  this.addObservers = function(resolve,reject) {
    if (state === null) {
      if (typeof resolve === 'function') {
        if (!resolve.uid) { resolve.uid = ++uids; }
        if (!resolves) { resolves = {}; rejects = {}; if (redo) { redo(); }
              }
        if (!resolves[resolve.uid]) {
          resolves[resolve.uid] = resolve; observers++;
        }
      }
      if (typeof reject === 'function') {
        if (!reject.uid) { reject.uid = ++uids; }
        if (!rejects) { resolves = {}; rejects = {}; if (redo) { redo(); }
              }
        if (!rejects[reject.uid]) {
          rejects[reject.uid] = reject; observers++;
        }
      }
    } else if (state) {
      process.nextTick(function() { resolve(value); });
    } else {
      process.nextTick(function() { reject(value); });
    }
  };

  this.rmObservers = function(resolve,reject) {
    if (observers) {
      if (resolve && resolve.uid && resolves[resolve.uid]) {
        delete resolves[resolve.uid]; observers--;
      }
      if (reject && reject.uid && rejects[reject.uid]) {
        delete rejects[reject.uid]; observers--;
      }
      if (!observers) {
        resolves = null; rejects = null; if (undo) { undo(); }
      }
    }
  };

  function resolve(newValue) {
    try {
      if (newValue === self) { throw new TypeError('A promise cannot be
            resolved with itself.'); }
      if (newValue && (typeof newValue === 'object' ||
                       typeof newValue === 'function')) {
        var then = newValue.then;
        if (typeof then === 'function') {
          return doResolve(then.bind(newValue), resolve, reject);
        }
      }
      state = true;
      value = newValue;
      finale();
    } catch (e) { reject(e); }
  }

  function reject(newValue) {
    state = false;
    value = newValue;
    finale();
  }

  function finale() {
    var cbs = (state? resolves: rejects);
    process.nextTick(function() { for (var i in cbs) { cbs[i](value); } });
    observers = 0;
    resolves = null;
    rejects = null;
  }

  doResolve(fn, resolve, reject);
}

Promise.resolve = function(val) {
  if (val instanceof Promise) { return val; }
  return new Promise(function(resolve) {
    resolve(val);
  });
};
Promise.reject = function(val) {
  return new Promise(function(resolve,reject) {
    reject(val);
  });
};
Promise.all = function(ps) {
  var unresolved = ps.length;
  var result = new Array(ps.length);
  var resolves; var reject2;
  var qs = ps.map(Promise.resolve);
  function init(resolve,reject) {
    resolves = ps.map(function(p,i) { return function(val) {
      if (result[i] === undefined) {
        result[i] = val;
        if (--unresolved === 0) { resolve(result); }
      }
    }; });
    reject2 = reject;
    for (var i=0; i<ps.length; i++) {
      qs[i].addObservers(resolves[i],reject);
    }
  }
  function undo() {
    for (var i=0; i<ps.length; i++) {
      qs[i].rmObservers(resolves[i],reject2);
    }
  }
  function redo() {
    for (var i=0; i<ps.length; i++) {
      qs[i].addObservers(resolves[i],reject2);
    }
  }
  return new Promise(init,undo,redo);
};
Promise.race = function(ps) {
  var qs = ps.map(Promise.resolve);
  var resolve2; var reject2;
  function init(resolve,reject) {
    resolve2 = function(val) { resolve(val); undo(); };//fixes space leak!
    reject2 = function(err) { reject(err); undo(); };
    for (var i=0; i<ps.length; i++) {
      qs[i].addObservers(resolve2,reject2);
    }
  }
  function undo() {
    for (var i=0; i<ps.length; i++) {
      qs[i].rmObservers(resolve2,reject2);
    }
  }
  function redo() {
    for (var i=0; i<ps.length; i++) {
      qs[i].addObservers(resolve2,reject2);
    }
  }
  return new Promise(init,undo,redo);
};
```