Functional Reactive Programming with Liveness Guarantees

As relations are to set-valued functions, so event sources are to behaviours

Alan Jeffrey
Bell Labs, Alcatel-Lucent
ajeffrey@bell-labs.com

Abstract

Functional Reactive Programming (FRP) is an approach to the development of reactive systems which provides a pure functional interface, but which may be implemented as an abstraction of an imperative event-driven layer. FRP systems typically provide a model of behaviours (total time-indexed values, implemented as pull systems) and event sources (partial time-indexed values, implemented as push systems). In this paper, we investigate a type system for event-driven FRP programs which provide liveness guarantees, that is every input event is guaranteed to generate an output event. We show that FRP can be implemented on top of a model of sets and relations, and that the isomorphism between event sources and behaviours corresponds to the isomorphism between relations and set-valued functions. We then implement sets and relations using a model of continuations using the usual double-negation CPS transform. The implementation of behaviours as pull systems based on futures, and of event sources as push systems based on the observer pattern, thus arises from first principles. We also discuss a Java implementation of the FRP model.

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

General Terms Design, Theory

Keywords Functional Reactive Programming, Concurrency, Liveness

1. Introduction

Many classes of programs are *reactive*: they run for a long period of time during which they interact with their environment. Examples of reactive programs include control systems, servers, and any program with a graphical user interface.

Many reactive programs are implemented using an event-driven model, in which stateful components send and receive events which update their state, and may cause side-effects such as network traffic or screen updates. The event-driven model forms the basis of Actors [12], and the Model View Controller architecture of Smalltalk [5].

The event-driven model has a number of challenging features, including:

- *Concurrency*: reactive programs often have concurrent features such as dealing with multiple simultaneous events. This either leads to multithreaded languages such as Java, with complex concurrency models [4, Ch. 17], or single-threaded languages such as ECMAScript which do not naturally support multicore execution, and rely on cooperative multitasking.
- *Imperative programming*: components are stateful, and may respond to events by updating their internal state. These hidden side-effects can result in complex implicit component interdependencies.
- *Referential opacity*: since components support mutable state, component identity is important. The semantics for components is not referentially transparent, since creating a component and copying it is not equivalent to creating multiple components.
- *Callbacks*: the idiom for programming in an event-driven model is registering callbacks rather than blocking function calls. For example, in ECMAScript an HTTP request is not a blocking method call, but instead a non-blocking call which registers a callback to handle the result of the HTTP request. This essentially requires the programmer to convert their program to Continuation Passing Style (CPS) [22]. Manual CPS transformation can be error-prone, for example, calling the wrong continuation, or mistakenly calling a continuation twice. In the absence of call/cc, CPS transformation is a whole-program translation, so can require a large codebase to be rewritten.

Functional Reactive Programming allows reactive programs to be written in a pure functional style. Originally developed by Elliot and Hudak [11] as part of the Fran functional animation system, there are now a number of implementations, including Agda FRP [15], Flapjax [20], Frappé [8], Froc [9], FrTime [7], Grapefruit [16], Reactive [10], Reactive-Banana [2], and Yampa [23].

Comparing FRP with the event-driven model, we have:

- *Pure functional model*: there are no implicit interactions caused by shared mutable state, and a simple concurrency model.
- *Referentially transparent*: signals can be copied without altering their semantics.
- *Direct*: FRP programs are given in direct style rather than CPS.

Comparing FRP with synchronous dataflow languages such as Esterel [3], some key distinctions are:

- *Fine-grained time*: FRP often models time as a continuous domain (such as $\mathbb{R}$) or using a much finer unit of time than the sample frequency of a synchronous language (such as 1ms).
- *Higher-order signals*: FRP allows signals of signals, which model dynamically reconfigurable dataflow networks.

- *Embedded DSL*: FRP is typically implemented as an embedded DSL library in a functional host language (often Haskell, but also Agda [13], ECMAScript [20], Java [8], OCaml [9], or Scheme [7]).

In this paper, we consider the problem of *liveness* of an FRP program.

- We develop a type system and model of FRP which provides liveness guarantees.
- We show how to define guarded fixed points, and provide liveness in the presence of recursion.
- We give an implementation of FRP in terms of programming over sets and relations, which are in turn implemented using callbacks.
- We discuss a multithreaded Java implementation of the model.

This is the first work to provide liveness guarantees for event-based programs in FRP. Moreover, in implementing FRP in terms of a model of sets and relations:

- The isomorphism between event sources and behaviours corresponds to the isomorphism between relations and set-valued functions.
- The implementation of sets and relations using the usual double-negation CPS transform allows the implementation of behaviours as pull systems based on futures, and of event sources as push systems based on the observer pattern, to arise from first principles.
- The implementation of event sources as relations allows naturally for concurrent execution, and in particular for out-of-order arrival of events. Most combinators naturally allow for concurrency, and it is only combinators which rely on ordering (such as an accumulation function) which force an order on events.

This is the first such investigation of the relationship between FRP and a CPS implementation of sets and relations, and the first implementation of concurrent FRP using out-of-order events.

2. History-free programs

In this section, we present our core FRP model, with types, combinators, and their semantics. For readability, we present the model in pseudo-Haskell: the actual implementation is in Java, as discussed in Section 6. We start by presenting the combinators for history-free programs: history-sensitive programs are discussed in the next section.

The semantics of FRP is defined in terms of *behaviours* over a total order Time, whose semantics are given as total functions from time to values:

$$[\![\mathsf{Behaviour}\ A]\!] = \mathsf{Time} \rightarrow [\![A]\!]$$

and *event sources* which are partial functions:

$$[\![\mathsf{Event}\ A]\!] = \mathsf{Time} \rightarrow [\![A]\!]_\bot$$

where:

$$X_\bot = \{\bot\} \cup \{\mathsf{lift}\, x \mid x \in X\}$$

We will sometimes refer to *signals* when we want to talk about both behaviours and events.

The timestamps for events are based on the time an event first enters the FRP system, not the current wall-clock time. For example, a function on event sources which processes a event x with timestamp t to an event $f(x)$ will still emit $f(x)$ with timestamp t, irrespective of the running time of f.

Signal types come with a collection of combinators such as:

$$\mathsf{map} : (A \rightarrow B) \rightarrow \mathsf{Event}\ A \rightarrow \mathsf{Event}\ B$$
$$\mathsf{filter} : (A \rightarrow \mathsf{Boolean}) \rightarrow \mathsf{Event}\ A \rightarrow \mathsf{Event}\ A$$

whose semantics are as one might expect:

$$[\![\mathsf{map}]\!]\, f\, \sigma\, t = \begin{cases} \mathsf{lift}\,(f\,x) & \text{if } \sigma\, t = \mathsf{lift}\, x \\ \bot & \text{otherwise} \end{cases}$$

$$[\![\mathsf{filter}]\!]\, f\, \sigma\, t = \begin{cases} \mathsf{lift}\, x & \text{if } \sigma\, t = \mathsf{lift}\, x \text{ and } f\, x \\ \bot & \text{otherwise} \end{cases}$$

For example, a simple web server, which responds "Hello, World!" to any `GET` request might be written:

```
server : Event HTTPRequest → Event HTTPResponse
server reqs = getResps where
  getReqs = filter isGetRequest reqs
  getResps = map hello getReqs
  hello req = httpResponse 200 "Hello, World!"
```

assuming a library for HTTP messages:

```
isGetRequest : HTTPRequest → Boolean
httpResponse : Integer → String → HTTPResponse
```

Unfortunately, this server has a serious problem: it suffers from a lack of *liveness*, in that a non-`GET` request will result in no response at all. To fix this, we introduce explicit error handling:

```
server : Event HTTPRequest → Event HTTPResponse
server reqs = union getResps otherResps where
  (getReqs, otherReqs) = partition isGetRequest reqs
  getResps = map hello getReqs
  otherResps = map error otherReqs
  hello req = httpResponse 200 "Hello, World!"
  error req = httpResponse 405 "Method not supported."
```

This uses combinators for partitioning and recombining events:

$$\mathsf{partition} : (A \rightarrow \mathsf{Boolean}) \rightarrow \mathsf{Event}\ A \rightarrow (\mathsf{Event}\ A \times \mathsf{Event}\ A)$$
$$\mathsf{union} : \mathsf{Event}\ A \rightarrow \mathsf{Event}\ A \rightarrow \mathsf{Event}\ A$$

The semantics of partitioning is simple:

$$[\![\mathsf{partition}]\!]\, f\, \sigma = ([\![\mathsf{filter}]\!]\, f\, \sigma, [\![\mathsf{filter}]\!]\, (\neg \circ f)\, \sigma)$$

The semantics of recombining is not so simple, since we have to account for the possibility that events will arrive simultaneously from both sources. The easiest thing to do is to prioritize one source over the other, for example to prioritize the first source:

$$[\![\mathsf{union}]\!]\, \sigma\, \tau\, t = \begin{cases} \mathsf{lift}\, x & \text{if } \sigma\, t = \mathsf{lift}\, x \\ \tau\, t & \text{otherwise} \end{cases}$$

Unfortunately, this approach is problematic for three reasons:

- This semantics for union xs ys is not commutative, even though in its idiomatic usage xs and ys have disjoint domains.
- The types for partition and union permit the original buggy server, so do not provide liveness guarantees.
- The implementation of union xs ys is difficult, in that an event from ys at time t cannot be emitted until we know that xs will not generate an event at time t. This may cause events from ys to be unnecessarily buffered, and requires the presence of negative information (xs will not generate an event at time t).

For these reasons, we propose an alternative type system for behaviours and events. Rather than event sources being partial func-

$$[\![\mathsf{Event}\,T\,A]\!] = [\![T]\!] \to [\![A]\!]$$
$$[\![\mathsf{Behaviour}\,T\,A]\!] = [\![T]\!] \to [\![A]\!]$$
$$[\![\mathsf{Minus}\,T\,S]\!] = [\![T]\!] \setminus [\![S]\!]$$

$$\mathsf{event} : \mathsf{Behaviour}\,T\,A \to \mathsf{Event}\,T\,A$$
$$[\![\mathsf{event}]\!]\,\sigma = \sigma$$

$$\mathsf{buffer} : \mathsf{Event}\,T\,A \to \mathsf{Behaviour}\,T\,A$$
$$[\![\mathsf{buffer}]\!]\,\sigma = \sigma$$

$$\mathsf{map} : (A \to B) \to \mathsf{Event}\,T\,A \to \mathsf{Event}\,T\,B$$
$$[\![\mathsf{map}]\!]\,f\,\sigma\,t = f\,(\sigma\,t)$$

$$\mathsf{map} : (A \to B) \to \mathsf{Behaviour}\,T\,A \to \mathsf{Behaviour}\,T\,B$$
$$[\![\mathsf{map}]\!]\,f\,\sigma\,t = f\,(\sigma\,t)$$

$$\mathsf{map2} : (A \to B \to C) \to \mathsf{Behaviour}\,T\,A \to \mathsf{Behaviour}\,T\,B \to \mathsf{Behaviour}\,T\,C$$
$$[\![\mathsf{map2}]\!]\,f\,\sigma\,\tau\,t = f\,(\sigma\,t)\,(\tau\,t)$$

$$\mathsf{map2} : (A \to B \to C) \to \mathsf{Event}\,T\,A \to \mathsf{Behaviour}\,T\,B \to \mathsf{Event}\,T\,C$$
$$[\![\mathsf{map2}]\!]\,f\,\sigma\,\tau\,t = f\,(\sigma\,t)\,(\tau\,t)$$

$$\mathsf{partition} : (A \to \mathsf{Boolean}) \to \mathsf{Event}\,T\,A \to \exists(S \le T)(\mathsf{Event}\,S\,A \times \mathsf{Event}\,(\mathsf{Minus}\,T\,S)\,A)$$
$$[\![\mathsf{partition}]\!]\,f\,\sigma = (X, \tau, \rho) \text{ where } X = \{t \mid f\,(\sigma\,t)\} \text{ and } \tau\,t = \sigma\,t \text{ and } \rho\,t = \sigma\,t$$

$$\mathsf{union} : \mathsf{Event}\,S\,A \to \mathsf{Event}\,(\mathsf{Minus}\,T\,S)\,A \to \mathsf{Event}\,T\,A$$
$$[\![\mathsf{union}]\!]\,\sigma\,\tau = \sigma \cup \tau$$

Figure 1. History-free fragment, where $[\![S]\!] \subseteq [\![T]\!] \subseteq \mathsf{Time}$ is locally finite and well-ordered

tions with domain Time, we propose viewing them as total functions whose domain is a subset of Time, that is:

$$[\![\mathsf{Event}\,T\,A]\!] = [\![T]\!] \to [\![A]\!]$$

Since event sources are total, they guarantee *liveness* in the sense that they will produce a value for any time in $[\![T]\!]$. For symmetry, we parameterize behaviours by the times at which they change:

$$[\![\mathsf{Behaviour}\,T\,A]\!] = [\![T]\!] \to [\![A]\!]$$

To ensure the existence of fixed points, as discussed in Section 3, we require that $[\![T]\!] \subseteq \mathsf{Time}$ is locally finite (that is every interval $[s, t]$ from $[\![T]\!]$ is finite) and well-ordered (every non-empty subset of T has a least element). Note that T is being used as a *phantom type* [19]: it is a fiction of the types for signals, and its implementation is not necessarily as a subset of Time.

Semantically, events and behaviours are equivalent, and indeed they form an isomorphism given by:

$$\mathsf{event} : \mathsf{Behaviour}\,T\,A \to \mathsf{Event}\,T\,A$$
$$\mathsf{buffer} : \mathsf{Event}\,T\,A \to \mathsf{Behaviour}\,T\,A$$

As we shall see, their implementations are quite different (behaviours are implemented as pull systems, and events are implemented as push systems). In particular, the event function introduces no overhead, but the buffer function introduces buffering, and so may have a significant memory overhead.

We are careful about when we use Behaviour types and when we use Event types to ensure that the only source of buffering from the primitives is the buffer function. For example, we provide two functions for mapping a binary function over two signals:

$$\mathsf{map2} : (A \to B \to C) \to$$
$$\quad \mathsf{Behaviour}\,T\,A \to \mathsf{Behaviour}\,T\,B \to \mathsf{Behaviour}\,T\,C$$
$$\mathsf{map2} : (A \to B \to C) \to$$
$$\quad \mathsf{Event}\,T\,A \to \mathsf{Behaviour}\,T\,B \to \mathsf{Event}\,T\,C$$

In both cases, one of the signals must be a behaviour, and if we wish to combine two event sources, we must first buffer one of them, for example if xs, ys : Event T Integer then:

$$\mathsf{map2}\,(\cdot + \cdot)\,\mathsf{xs}\,(\mathsf{buffer}\,\mathsf{ys}) : \mathsf{Event}\,T\,\mathsf{Integer}$$

We do *not* provide a combinator:

$$\mathsf{map2} : (A \to B \to C) \to$$
$$\quad \mathsf{Event}\,T\,A \to \mathsf{Event}\,T\,A \to \mathsf{Event}\,T\,A$$

since map2 f xs ys cannot be implemented without buffering: an event with timestamp t may arrive from xs at wall-clock time t_1, with the matching event from ys arriving at wall-clock time t_2; events must be buffered to allow for the differences in wall-clock time.

The type for map is unchanged, and its semantics is simplified, since events are now total:

$$\mathsf{map} : (A \to B) \to \mathsf{Event}\,T\,A \to \mathsf{Event}\,T\,B$$
$$[\![\mathsf{map}]\!]\,f\,\sigma\,t = f\,(\sigma\,t)$$

The type for filter must change: if xs has domain T, then filter f xs will not have domain T, but will instead have some subdomain of T. To model this, we use *bounded existential types* [6], with semantics:

$$[\![\exists(A \le B)(F\,A)]\!] = \{(X, [\![F]\!]X) \mid X \subseteq [\![B]\!]\}$$

The type and semantics of filter are then:

$$\mathsf{filter} : (A \to \mathsf{Boolean}) \to \mathsf{Event}\,T\,A \to \exists(S \le T)(\mathsf{Event}\,S\,A)$$
$$[\![\mathsf{filter}]\!]\,f\,\sigma = (X, \tau) \text{ where}$$
$$\quad X = \{t \mid f\,(\sigma\,t)\} \text{ and } \tau\,t = \sigma\,t$$

For example, if:

$$[\![\mathsf{things}]\!] = \{0 \mapsto \mathsf{apple}, 1 \mapsto \mathsf{cat}, 2 \mapsto \mathsf{banana}\}$$

then:

$$[\![\mathsf{filter\,isFruit\,things}]\!] = (\{0,2\}, \{0 \mapsto \mathsf{apple}, 2 \mapsto \mathsf{banana}\})$$

For partitioning, we could just continue to treat a partition as a pair of filters, which would give the type:

$$\begin{aligned}&\mathsf{partition} : (A \to \mathsf{Boolean}) \to \mathsf{Event}\, T\, A \to \\ &\quad (\exists(S \leq T)(\mathsf{Event}\, S\, A) \times \exists(R \leq T)(\mathsf{Event}\, R\, A))\end{aligned}$$

However, with this type, we have lost the information that T has been partitioned into S and R. Instead, we provide a more refined type, in which we set R to be $\mathsf{Minus}\, T\, S$:

$$\begin{aligned}&\mathsf{partition} : (A \to \mathsf{Boolean}) \to \mathsf{Event}\, T\, A \to \\ &\quad \exists(S \leq T)(\mathsf{Event}\, S\, A \times \mathsf{Event}\,(\mathsf{Minus}\, T\, S)\, A)\end{aligned}$$

where $\mathsf{Minus}\, T\, S$ is interpreted as set subtraction (maintaining an invariant that $[\![S]\!] \subseteq [\![T]\!]$):

$$[\![\mathsf{Minus}\, T\, S]\!] = [\![T]\!] \setminus [\![S]\!]$$

The semantics of partition is:

$$\begin{aligned}&[\![\mathsf{partition}]\!]\, f\, \sigma = (X, \tau, \rho) \text{ where} \\ &\quad X = \{t \mid f\,(\sigma\, t)\} \text{ and } \tau\, t = \sigma\, t \text{ and } \rho\, t = \sigma\, t\end{aligned}$$

For example:

$$\begin{aligned}&[\![\mathsf{partition\,isFruit\,things}]\!] = \\ &\quad (\{0,2\}, \{0 \mapsto \mathsf{apple}, 2 \mapsto \mathsf{banana}\}, \{1 \mapsto \mathsf{cat}\})\end{aligned}$$

The benefit of this refined type for partition is that it allows for a better treatment of union:

$$\begin{aligned}&\mathsf{union} : \mathsf{Event}\, S\, A \to \mathsf{Event}\,(\mathsf{Minus}\, T\, S)\, A \to \mathsf{Event}\, T\, A \\ &[\![\mathsf{union}]\!]\, \sigma\, \tau = \sigma \cup \tau\end{aligned}$$

For example, if we define:

$$\begin{aligned}&\mathsf{mkDog\,xs} = \mathsf{union\,fruit\,dogs\,where} \\ &\quad (\mathsf{fruit}, \mathsf{rest}) = \mathsf{partition\,isFruit\,xs} \\ &\quad \mathsf{dogs} = \mathsf{map}\,(\lambda x\,.\,\mathsf{dog})\,\mathsf{rest}\end{aligned}$$

then we have:

$$[\![\mathsf{mkDog\,things}]\!] = \{0 \mapsto \mathsf{apple}, 1 \mapsto \mathsf{dog}, 2 \mapsto \mathsf{banana}\}$$

This addresses the three issues we highlighted earlier:

- The semantics of union is commutative.
- The fixed version of the HTTP server typechecks with type $\mathsf{server} : \mathsf{Event}\, T\, \mathsf{HTTPRequest} \to \mathsf{Event}\, T\, \mathsf{HTTPResponse}$. The buggy server does not typecheck with this type. In general, the semantics of $\mathsf{Event}\, T\, A$ require liveness for all of T.
- The implementation of union xs ys is simple: when an event from either xs or ys is generated, it can immediately be emitted. No negative information is required.

In Figure 1 we summarize the history-free fragment of our model.

3. History-sensitive programs

So far the FRP combinators only allow history-free programs to be defined, in that output behaviour at time t can only depend on input behaviour at time t, not at any time before t. We now present combinators which support history-sensitive programming.

The simplest form of history-sensitivity is the delay function, which takes a behaviour, and delays it by one unit of time:

$$\begin{aligned}&\mathsf{delay} : \mathsf{Behaviour}\, T\, A \to \mathsf{Event}\,(\mathsf{Tail}\, T)\, A \\ &[\![\mathsf{delay}]\!]\, \sigma\, t_{i+1} = \sigma\, t_i\end{aligned}$$

For example:

$$[\![\mathsf{delay\,things}]\!] = \{1 \mapsto \mathsf{apple}, 2 \mapsto \mathsf{cat}\}$$

Here we are making use of the requirement that T is a locally finite and well-ordered, since this means that:

$$[\![T]\!] = \{t_0 < t_1 < t_2 < \cdots\}$$

In particular we can then define the tail of T to be every element other than the least one:

$$[\![\mathsf{Tail}\, T]\!] = \text{if } [\![T]\!] = \emptyset \text{ then } \emptyset \text{ else } [\![T]\!] \setminus \{t_0\}$$

We can similarly define the tail of a behaviour to be every element but the first:

$$\begin{aligned}&\mathsf{tail} : \mathsf{Behaviour}\, T\, A \to \mathsf{Behaviour}\,(\mathsf{Tail}\, T)\, A \\ &[\![\mathsf{tail}]\!]\, \sigma\, t = \sigma\, t\end{aligned}$$

For example:

$$[\![\mathsf{tail\,things}]\!] = \{1 \mapsto \mathsf{cat}, 2 \mapsto \mathsf{banana}\}$$

We cannot, however, provide a matching head function with type $\mathsf{Behaviour}\, T\, A \to A$, since t_0 may be in the future (and indeed T may be empty, so no such A may exist). Instead, we allow a peek function, which allows the first element to be inspected as long as an event is being produced:

$$\begin{aligned}&\mathsf{peek} : \mathsf{Behaviour}\, T\, A \to (A \to \mathsf{Event}\, T\, B) \to \mathsf{Event}\, T\, B \\ &[\![\mathsf{peek}]\!]\, \sigma\, f = f\,(\sigma\, t_0)\end{aligned}$$

For example:

$$[\![\mathsf{peek\,things}\,(\lambda \mathsf{x}\,.\,\mathsf{if\,isFruit\,x\,then\,ys\,else\,zs})]\!] = [\![\mathsf{ys}]\!]$$

These functions allow signals to be decomposed, we can also compose them, for example:

$$\begin{aligned}&\mathsf{cons} : A \to \mathsf{Event}\,(\mathsf{Tail}\, T)\, A \to \mathsf{Event}\, T\, A \\ &[\![\mathsf{cons}]\!]\, x\, \sigma\, t = \begin{cases} x & \text{if } t = t_0 \\ \sigma\, t & \text{otherwise}\end{cases}\end{aligned}$$

for example:

$$\begin{aligned}&[\![\mathsf{cons\,dog\,(event\,(tail\,things))}]\!] \\ &\quad = \{0 \mapsto \mathsf{dog}, 1 \mapsto \mathsf{cat}, 2 \mapsto \mathsf{banana}\}\end{aligned}$$

Finally, we consider recursively defined behaviours. We will only allow the solution to guarded equations, in the sense that behaviour at time t_{i+1} can only depend on recursive behaviour up to time t_i. We ensure this by introducing an explicit delay, that is we are solving:

$$\sigma = f\,([\![\mathsf{delay}]\!]\, \sigma)$$

Since all definable functions on signals are causal (output at time t only depends on input up to time t) we have that σ's value at time t_{i+1} only depends on $f\,([\![\mathsf{delay}]\!]\, \sigma)$'s value at time t_{i+1}, which only depends on $([\![\mathsf{delay}]\!]\, \sigma)$'s value up to time t_{i+1}, which only depends on σ's value up to time t_i. Thus, σ is well-defined. For a more formal treatment of the well-definedness of guarded equations, see Krishnaswami and Benton's ultrametric space semantics [18], or the author's use of parametricity to ensure causality [14].

Unfortunately, the obvious type for fixed points:

$$\mathsf{fix} : (\mathsf{Event}\,(\mathsf{Tail}\, T)\, A \to \mathsf{Behaviour}\, T\, A) \to \mathsf{Behaviour}\, T\, A$$

has a problem. We are maintaining an invariant that all time domains are locally finite and well-ordered, but if we were to allow fix to exist for all T, then we could use it to inhabit $\mathsf{Behaviour}\, \mathbb{R}^+\, A$, for example:

$$\mathsf{fix}(\lambda \mathsf{xs}\,.\,\mathsf{buffer}(\mathsf{cons}(0, \mathsf{xs})))$$

Since $\mathbb{R}^+$ is locally infinite and non-well-ordered, this would be problematic. To avoid this, we introduce a type of *clocks*, which is only inhabited when $T \subseteq \mathsf{Time}$ is locally finite and well-ordered, in which case its semantics is trivial:

$$[\![\mathsf{Clock}\, T]\!] = \{*\}$$

$$\llbracket \mathsf{Tail}\,T \rrbracket = \text{if } \llbracket T \rrbracket = \emptyset \text{ then } \emptyset \text{ else } \llbracket T \rrbracket \setminus \{t_0\}$$
$$\llbracket \mathsf{Clock}\,T \rrbracket = \{*\}$$

$$\mathsf{peek} : \mathsf{Behaviour}\,T\,A \to (A \to \mathsf{Event}\,T\,B) \to \mathsf{Event}\,T\,B$$
$$\llbracket \mathsf{peek} \rrbracket\,\sigma\,f = f\,(\sigma\,t_0)$$

$$\mathsf{tail} : \mathsf{Behaviour}\,T\,A \to \mathsf{Behaviour}\,(\mathsf{Tail}\,T)\,A$$
$$\llbracket \mathsf{tail} \rrbracket\,\sigma\,t = \sigma\,t$$

$$\mathsf{delay} : \mathsf{Behaviour}\,T\,A \to \mathsf{Event}\,(\mathsf{Tail}\,T)\,A$$
$$\llbracket \mathsf{delay} \rrbracket\,\sigma\,t_{i+1} = \sigma\,t_i$$

$$\mathsf{cons} : A \to \mathsf{Event}\,(\mathsf{Tail}\,T)\,A \to \mathsf{Event}\,T\,A$$
$$\llbracket \mathsf{cons} \rrbracket\,x\,\sigma\,t = \begin{cases} x & \text{if } t = t_0 \\ \sigma\,t & \text{otherwise} \end{cases}$$

$$\mathsf{clock} : \mathsf{Behaviour}\,T\,A \to \mathsf{Clock}\,T$$
$$\llbracket \mathsf{clock} \rrbracket\,\sigma = *$$

$$\mathsf{fix} : \mathsf{Clock}\,T \to (\mathsf{Event}\,(\mathsf{Tail}\,T)\,A \to \mathsf{Behaviour}\,T\,A) \to \mathsf{Behaviour}\,T\,A$$
$$\llbracket \mathsf{fix} \rrbracket\,c\,f = \sigma \text{ where } \sigma = f\,(\llbracket \mathsf{delay} \rrbracket\,\sigma)$$

Figure 2. History-sensitive fragment of our FRP model, where $\llbracket T \rrbracket = \{t_0 < t_1 < t_2 < \cdots\}$

The modified type for fix uses a clock element to ensure that it is safe to inhabit $\mathsf{Behaviour}\,T\,A$:

$$\mathsf{fix} : \mathsf{Clock}\,T \to (\mathsf{Event}\,(\mathsf{Tail}\,T)\,A \to \mathsf{Behaviour}\,T\,A) \to \mathsf{Behaviour}\,T\,A$$
$$\llbracket \mathsf{fix} \rrbracket\,c\,f = \sigma \text{ where } \sigma = f\,(\llbracket \mathsf{delay} \rrbracket\,\sigma)$$

For example, we can use fix to define a constant behaviour:

$$\mathsf{constant} : \mathsf{Clock}\,T \to A \to \mathsf{Behaviour}\,T\,A$$
$$\mathsf{constant}\,\mathsf{c}\,\mathsf{x} = \mathsf{fix}\,\mathsf{c}\,(\lambda \mathsf{xs}\,.\,\mathsf{buffer}(\mathsf{cons}(\mathsf{x},\mathsf{xs})))$$
$$\llbracket \mathsf{constant} \rrbracket\,c\,x\,t = x$$

We can also use fix to define an accumulator function:

$$\mathsf{accum} : (B \to A \to B) \to B \to \mathsf{Behaviour}\,T\,A \to \mathsf{Behaviour}\,T\,B$$
$$\mathsf{accum}\,\mathsf{f}\,\mathsf{y}\,\mathsf{xs} = \mathsf{fix}\,(\mathsf{clock}\,\mathsf{xs})\,(\lambda \mathsf{ys}\,.\,\mathsf{buffer}(\mathsf{map2}\,\mathsf{f}\,\mathsf{cons}(\mathsf{y},\mathsf{ys})\,\mathsf{xs}))$$

making use of a function to extract a clock from a behaviour:

$$\mathsf{clock} : \mathsf{Behaviour}\,T\,A \to \mathsf{Clock}\,T$$
$$\llbracket \mathsf{clock} \rrbracket\,\sigma = *$$

For example if:

$$\sigma = \{t_0 \mapsto x_0, t_1 \mapsto x_1, t_2 \mapsto x_2, \ldots\}$$

then:

$$\llbracket \mathsf{accum} \rrbracket\,(\cdot + \cdot)\,0\,\sigma =$$
$$\{t_0 \mapsto x_0, t_1 \mapsto (x_0 + x_1), t_2 \mapsto (x_0 + x_1 + x_2), \ldots\}$$

In Figure 2 we summarize the history-sensitive fragment of our FRP model.

4. Implementation of behaviours and events

The implementation of signals is built on top of a library of sets and relations. In Figure 3 we give the model of sets and relations, we defer discussing its implementation until the next section. It provides two types, for sets and relations:

$$\llbracket \mathsf{Set}\,A \rrbracket = \mathcal{P}\,\llbracket A \rrbracket$$
$$\mathsf{Relation}\,A\,B = \mathsf{Set}\,(A \times B)$$

together with a collection of combinators for these types. Of particular interest is the isomorphism between relations ($\mathcal{P}\,(X \times Y)$) and set-valued functions ($X \to \mathcal{P}\,Y$) given by the functions:

$$\mathsf{relation} : \mathsf{Set}\,A \to (A \to \mathsf{Set}\,B) \to \mathsf{Relation}\,A\,B$$
$$\mathsf{relation}\,\mathsf{xs}\,\mathsf{f} = \mathsf{pmap}\,(\lambda \mathsf{x}\,.\,\mathsf{map}\,(\mathsf{x},\cdot)\,(\mathsf{f}\,\mathsf{x}))\,\mathsf{xs}$$
$$\llbracket \mathsf{relation} \rrbracket\,X\,f = \{(x,y) \mid x \in X, y \in f\,x\}$$
$$\mathsf{buffer} : \mathsf{Relation}\,A\,B \to A \to \mathsf{Set}\,B$$
$$\llbracket \mathsf{buffer} \rrbracket\,R\,x = \{y \mid (x,y) \in R\}$$

Given a relation R whose domain (that is $\{x \mid \exists y\,.\,(x,y) \in R\}$) is X, we have:

$$\llbracket \mathsf{relation} \rrbracket\,X\,(\llbracket \mathsf{buffer} \rrbracket\,R) = R$$

and given a function f whose domain (that is $\{x \mid \exists y\,.\,y \in f\,x\}$) is X, we have:

$$\llbracket \mathsf{buffer} \rrbracket\,(\llbracket \mathsf{relation} \rrbracket\,X\,f) = f$$

$$\llbracket \mathsf{Set}\,A \rrbracket = \mathcal{P}\,\llbracket A \rrbracket$$
$$\mathsf{Relation}\,A\,B = \mathsf{Set}\,(A \times B)$$

$$\mathsf{buffer} : \mathsf{Relation}\,A\,B \to A \to \mathsf{Set}\,B$$
$$\llbracket \mathsf{buffer} \rrbracket\,R\,x = \{y \mid (x,y) \in R\}$$

$$\mathsf{empty} : \mathsf{Set}\,A$$
$$\llbracket \mathsf{empty} \rrbracket = \emptyset$$

$$\mathsf{singleton} : A \to \mathsf{Set}\,A$$
$$\llbracket \mathsf{singleton} \rrbracket\,x = \{x\}$$

$$\mathsf{union} : \mathsf{Set}\,A \to \mathsf{Set}\,A \to \mathsf{Set}\,A$$
$$\llbracket \mathsf{union} \rrbracket\,X\,Y = X \cup Y$$

$$\mathsf{pmap} : (A \to \mathsf{Set}\,B) \to \mathsf{Set}\,A \to \mathsf{Set}\,B$$
$$\llbracket \mathsf{pmap} \rrbracket\,f\,X = \{y \mid x \in X, y \in f\,x\}$$

$$\mathsf{fix} : ((A \to \mathsf{Set}\,B) \to (A \to \mathsf{Set}\,B)) \to A \to \mathsf{Set}\,B$$
$$\llbracket \mathsf{fix} \rrbracket\,f = \text{the smallest } g \text{ such that } \forall x\,.\,f\,g\,x \subseteq g\,x$$

Figure 3. Model of sets and relations

We shall see later that this isomorphism generates the isomorphism between behaviours and event sources.

The combinators include a fixed point function:

$$\mathsf{fix} : ((A \to \mathsf{Set}\, B) \to (A \to \mathsf{Set}\, B)) \to A \to \mathsf{Set}\, B$$
$$[\![\mathsf{fix}]\!]\, f = \text{the smallest } g \text{ such that } \forall x \,.\, f\, g\, x \subseteq g\, x$$

This function is well-defined because the only implementable functions are monotone with respect to $\subseteq$.

Using the combinators given in Figure 3, we can derive other combinators such as:

$$\mathsf{map} : (A \to B) \to \mathsf{Set}\, A \to \mathsf{Set}\, B$$
$$\mathsf{map\, f} = \mathsf{pmap}\,(\lambda \mathsf{x}\,.\, \mathsf{singleton(f\, x)})$$
$$[\![\mathsf{map}]\!]\, f\, X = \{f\, x \mid x \in X\}$$

$$\mathsf{pmapr} : (B \to \mathsf{Set}\, C) \to \mathsf{Relation}\, A\, B \to \mathsf{Relation}\, A\, C$$
$$\mathsf{pmapr\, f} = \mathsf{pmap}\,(\lambda \mathsf{(x, y)}\,.\, \mathsf{map}\,(x, \cdot)\,(\mathsf{f\, y}))$$
$$[\![\mathsf{pmapr}]\!]\, f\, R = \{(x, z) \mid (x, y) \in R, z \in f\, y\}$$

$$\mathsf{filter} : (A \to \mathsf{Boolean}) \to \mathsf{Set}\, A \to \mathsf{Set}\, A$$
$$\mathsf{filter\, f} = \mathsf{pmap}\,(\lambda \mathsf{x}\,.\, \mathsf{if\, f\, x\, then\, singleton\, x\, else\, empty})$$
$$[\![\mathsf{filter}]\!]\, f\, X = \{x \mid x \in X, f\, x\}$$

$$\mathsf{domain} : \mathsf{Relation}\, A\, B \to \mathsf{Set}\, A$$
$$\mathsf{domain} = \mathsf{map}\,(\lambda(\mathsf{x}, \_)\,.\, \mathsf{x})$$
$$[\![\mathsf{domain}]\!]\, R = \{x \mid \exists y\,.\, (x, y) \in R\}$$

In particular, we can implement the categorical structure of relations, with identity and composition:

$$\mathsf{diagonal} : \mathsf{Set}\, A \to \mathsf{Relation}\, A\, A$$
$$\mathsf{diagonal} = \mathsf{map}\,(\lambda \mathsf{x}\,.\, (\mathsf{x}, \mathsf{x}))$$
$$[\![\mathsf{diagonal}]\!]\, X = \{(x, x) \mid x \in X\}$$

$$\mathsf{comp} : \mathsf{Relation}\, A\, B \to \mathsf{Relation}\, B\, C \to \mathsf{Relation}\, A\, C$$
$$\mathsf{comp\, q\, r} = \mathsf{pmapr}\,(\mathsf{buffer\, r})\, \mathsf{q}$$
$$[\![\mathsf{comp}]\!]\, Q\, R = \{(x, z) \mid (x, y) \in Q, (y, z) \in R\}$$

We shall now consider an implementation of FRP using sets and relations, beginning with the implementation of clocks. A clock is implemented as an abstract datatype whose implementation is:

```
data Clock $T$ = Clock {
    times : Set Time,
    first : Set Time,
    prev : Relation Time Time,
    parent : $T$
}
```

We maintain the following invariants about $\mathsf{c} : \mathsf{Clock}\, T$, where $[\![T]\!] \subseteq \mathsf{Time}$ is locally finite and well-ordered, and so $[\![T]\!]$ is of the form $\{t_0 < t_1 < t_2 < \cdots\}$:

- $[\![\mathsf{times\, c}]\!] = [\![T]\!]$.
- $[\![\mathsf{first\, c}]\!] = \{t_0\}$ (or $\emptyset$ when $[\![T]\!] = \emptyset$).
- $[\![\mathsf{prev\, c}]\!] = \{(t_{i+1}, t_i) \mid t_{i+1} > t_0\}$.

Note that every element of $\mathsf{Clock}\, T$ is equal (and in fact we use memoization to ensure that every T has at most one $\mathsf{Clock}\, T$). Clocks are needed as a way to access at run time the total order on T. We can implement a function:

```
tail : Clock $T$ → Clock (Tail $T$)
tail c = Clock {
    times = domain (prev c)
    first = pmap (λt . domain(filter (λ(_, u) . t = u) (prev c))) (first c)
    prev = pmap (λt . filter (λ(_, u) . t < u) (prev c)) (first c)
    parent = c
}
```

The reason for having a parent element is to implement:

$$\mathsf{tail}^{-1} : \mathsf{Clock}\,(\mathsf{Tail}\, T) \to \mathsf{Clock}\, T$$

The concrete implementation of $\mathsf{Tail}\, T$ is:

$$\mathsf{data\ Tail}\, T = \mathsf{Tail}\, \{\mathsf{clock} : \mathsf{Clock}\, T\}$$

so we can define:

$$\mathsf{tail}^{-1}\, \mathsf{c} = \mathsf{clock\,(parent\, c)}$$

The implementation of $\mathsf{Minus}\, T$ and minus^{-1} are similar. We can now turn our attention from clocks to behaviours and event sources. The concrete implementation of a behaviour is a clock together with a function from time to sets of events:

```
data Behaviour $T$ $A$ = Behaviour {
    clock : Clock $T$,
    function : Time → Set $A$
}
```

and the concrete implementation of an event source is a clock together with a relation between time and events:

```
data Event $T$ $A$ = Event {
    clock : Clock $T$,
    relation : Relation Time $A$
}
```

As we shall see when we look at the implementation of sets and relations, behaviours are implemented as pull systems, and event sources are implemented as push systems. We maintain the following invariants about $\mathsf{xs} : \mathsf{Behaviour}\, T\, A$ and $\mathsf{ys} : \mathsf{Event}\, T\, A$:

- For any $t \in [\![T]\!]$, the set $[\![\mathsf{function\, xs}]\!]\, t$ is a singleton.
- The domain of $[\![\mathsf{relation\, ys}]\!]$ is $[\![T]\!]$.
- For any $(t, x) \in [\![\mathsf{relation\, ys}]\!] \ni (t, y)$, we have $x = y$.

These invariants mean that it is straightforward to translate the concrete semantics of signal to their abstract semantics:

$$[\![\mathsf{xs}]\!]\, t = \text{the unique } x \text{ such that } x \in [\![\mathsf{function\, xs}]\!]\, t$$
$$[\![\mathsf{ys}]\!]\, t = \text{the unique } x \text{ such that } (t, x) \in [\![\mathsf{relation\, ys}]\!]$$

We can then implement the FRP combinators, and verify that they maintain the invariants and have the expected semantics. Most are quite straightforward given the model of sets and relations, for example the isomorphism between behaviours and events is just given by lifting the isomorphism between functions to sets and relations:

```
event : Behaviour $T$ $A$ → Event $T$ $A$
event (Behaviour c f) = Event c (relation (times c) f)

buffer : Event $T$ $A$ → Behaviour $T$ $A$
buffer (Event c r) = Behaviour c (buffer r)
```

$$\llbracket \mathsf{Not}\, A \rrbracket = \llbracket A \rrbracket \to 2$$

$$\mathsf{buffer} : \mathsf{Not}\,(\mathsf{Not}\,(A \times B)) \to A \to \mathsf{Not}\,(\mathsf{Not}\, B)$$
$$\llbracket \mathsf{buffer} \rrbracket\, k\, x\, \ell = k\, (\lambda\, (x', y)\, .\, (x = x') \wedge \ell(y))$$

$$\mathsf{discard} : \mathsf{Not}\, A$$
$$\llbracket \mathsf{discard} \rrbracket\, x = \mathsf{false}$$

$$\mathsf{doubleNot} : A \to \mathsf{Not}\,(\mathsf{Not}\, A)$$
$$\llbracket \mathsf{doubleNot} \rrbracket\, x\, k = k\, x$$

$$\mathsf{andthen} : \mathsf{Not}\, A \to \mathsf{Not}\, A \to \mathsf{Not}\, A$$
$$\llbracket \mathsf{andthen} \rrbracket\, k\, \ell\, x = k\, x \vee \ell\, x$$

$$\mathsf{copmap} : (B \to \mathsf{Not}\,(\mathsf{Not}\, A)) \to \mathsf{Not}\, A \to \mathsf{Not}\, B$$
$$\llbracket \mathsf{copmap} \rrbracket\, f\, k\, y = f\, y\, k$$

$$\mathsf{fix} : ((A \to \mathsf{Not}\, B) \to (A \to \mathsf{Not}\, B)) \to A \to \mathsf{Not}\, B$$
$$\llbracket \mathsf{fix} \rrbracket\, f = \text{the smallest } g \text{ such that } \forall x\, .\, \forall y\, .\, f\, g\, x\, y \Rightarrow g\, x\, y$$

Figure 4. Model of continuations

The delay function is implemented by mapping the behaviour over the previous time relation:

$$\mathsf{delay} : \mathsf{Behaviour}\, T\, A \to \mathsf{Event}\,(\mathsf{Tail}\, T)\, A$$
$$\mathsf{delay}\,(\mathsf{Behaviour}\, \mathsf{c}\, \mathsf{f}) = \mathsf{Event}\,(\mathsf{tail}\, \mathsf{c})\,(\mathsf{pmapr}\, \mathsf{f}\,(\mathsf{prev}\, \mathsf{c}))$$

The cons function is just the union of the head and the tail:

$$\mathsf{cons} : A \to \mathsf{Event}\,(\mathsf{Tail}\, T)\, A \to \mathsf{Event}\, T\, A$$
$$\mathsf{cons}\, \mathsf{x}\,(\mathsf{Event}\, \mathsf{c}\, \mathsf{r}) = \mathsf{Event}\, \mathsf{c}'\,(\mathsf{union}\, \mathsf{r}\, \mathsf{r}') \text{ where}$$
$$\mathsf{c}' = \mathsf{tail}^{-1}\, \mathsf{c}$$
$$\mathsf{r}' = \mathsf{map}\,(\mathsf{first}\, \mathsf{c}')\,(\lambda \mathsf{t}\, .\, (\mathsf{t}, \mathsf{x}))$$

The union function is just union of relations:

$$\mathsf{union} : \mathsf{Event}\, S\, A \to \mathsf{Event}\,(\mathsf{Minus}\, T\, S)\, A \to \mathsf{Event}\, T\, A$$
$$\mathsf{union}\,(\mathsf{Event}\, \_\, \mathsf{q})\,(\mathsf{Event}\, \mathsf{c}\, \mathsf{r}) = \mathsf{Event}\,(\mathsf{minus}^{-1}\, \mathsf{c})\,(\mathsf{union}\, \mathsf{q}\, \mathsf{r})$$

We defer a discussion of the implementation of partition to Section 6. The implementation and verification of the remaining primitives is straightforward.

5. Implementation of sets and relations

The implementation of signals is built on top of a library of continuations. In Figure 4 we give the model of continuations, we defer discussing its implementation until the next section. It provides a type of continuations:

$$\llbracket \mathsf{Not}\, A \rrbracket = \llbracket A \rrbracket \to 2$$

together with a collection of combinators for these types. These combinators correspond directly to the combinators for sets, since we can define:

$$\mathsf{data}\, \mathsf{Set}\, A = \mathsf{Set}\, \{\mathsf{unSet} : \mathsf{Not}\,(\mathsf{Not}\, A))\}$$

and then define:

$$\mathsf{buffer}\,(\mathsf{Set}\, \mathsf{xs}) = \mathsf{Set} \cdot (\mathsf{buffer}\, \mathsf{xs})$$
$$\mathsf{empty} = \mathsf{Set}\, \mathsf{discard}$$
$$\mathsf{singleton}\, \mathsf{x} = \mathsf{Set}\,(\mathsf{doubleNot}\, \mathsf{x})$$
$$\mathsf{union}\,(\mathsf{Set}\, \mathsf{xs})\,(\mathsf{Set}\, \mathsf{ys}) = \mathsf{Set}\,(\mathsf{andthen}\, \mathsf{xs}\, \mathsf{ys})$$
$$\mathsf{pmap}\, \mathsf{f}\,(\mathsf{Set}\, \mathsf{xs}) = \mathsf{Set}\,(\mathsf{comap}\,(\mathsf{copmap}\,(\mathsf{unSet} \cdot \mathsf{f})))$$
$$\mathsf{fix}\, \mathsf{f} = \mathsf{Set} \cdot \mathsf{fix}\,(\lambda \mathsf{g}\, .\, \mathsf{unSet} \cdot \mathsf{f}\,(\mathsf{Set} \cdot \mathsf{g}))$$

```
public class Events<T,A> {

  public final Clock<T> clock;
  public final Relation<Time,A> relation;

  public<B> Events<T,B>
    map(Function<A,B> f) { ... }
  public Behaviour<T,A>
    buffer() { ... }
  public PartitionedEvents<?,T,A>
    partition(Predicate<A> p) { ... }
  ...

}

public interface PartitionedEvents<S extends T,T,A> {

  public Events<S,A> yes();
  public Events<Minus<T,S>,A> no();
  public<Z> Z
    split(Function2<
      Events<S,A>,
      Events<Minus<T,S>,A>,
      Z> f);

}
```

Figure 5. Event sources in Java

where:

$$\mathsf{comap} : (B \to A) \to \mathsf{Not}\, A \to \mathsf{Not}\, B$$
$$\mathsf{comap}\, \mathsf{f} = \mathsf{copmap}\,(\lambda \mathsf{y}\, .\, \mathsf{doubleNot}\,(\mathsf{f}\, \mathsf{y}))$$

We can translate from the concrete semantics of sets to their abstract semantics as:

$$\llbracket \mathsf{xs} \rrbracket = \{x \mid \llbracket \mathsf{unSet}\, \mathsf{xs} \rrbracket\,(\lambda x'\, .\, x = x')\}$$

It is straightforward to verify that the combinators on sets have their expected semantics.

We have now shown that a model of FRP can be translated into a model of sets and relations, and from there to continuations. Composing these translations, we get:

$$\mathsf{Behaviour}\, T\, A \subseteq \mathsf{Clock}\, T \times (\mathsf{Time} \to \mathsf{Not}\,(\mathsf{Not}\, A))$$
$$\mathsf{Event}\, T\, A \subseteq \mathsf{Clock}\, T \times \mathsf{Not}\,(\mathsf{Not}\,(\mathsf{Time} \times A))$$

that is:

- Behaviours are implemented as *pull* systems: a user of a behaviour can call it with a time $t \in \llbracket T \rrbracket$, and receive back a *future* value, of type $\mathsf{Not}\,(\mathsf{Not}\, A)$. A client can then register callbacks of type $\mathsf{Not}\, A$ with the future, which will be executed when the value is known. Since we maintain an invariant that behaviours return singletons, each callback will be executed precisely once.
- Events are implemented as *push* systems: a user of an event can register a callback of type $\mathsf{Not}(\mathsf{Time} \times A)$ with it, which will be executed with pairs (t, x) whenever an event x arrives at time t. We maintain an invariant that each time t has a unique x, and that all times $t \in \llbracket T \rrbracket$ will generate an event.

It is interesting that the implementation of reactive programs as push or pull systems, and the uses of futures and the observer pattern fall out naturally from the translation of signals to sets and relations, and then to continuations. Moreover, the isomorphism between push systems and pull systems is given by the isomorphism between relations and set-valued functions.

```
public<T> Events<T,HttpResponse>
  responses(Events<T,HttpRequest> requests)
{
  return requests.partition(request ->
    request.isGetRequest()
  ).split((getReqs,otherReqs) ->
    getReqs.map(request ->
      new HttpResponse(200,"Hello, World!")
    ).union(otherReqs.map(request ->
      new HttpResponse(405)
    ))
  );
}
```

Figure 6. Example "Hello, World!" server implemented in Java

```
public interface Not<A> {

  public void accept(A x);

  default public<B> Not<B> comap(Function<B,A> f) {
    return y -> { this.accept(f.apply(y)); };
  }
  default public Not<A> andthen(Not<A> other) {
    return x -> { this.accept(x); other.accept(x); };
  }
  ...

}


public interface Not2<A,B> {

  public void accept(A x, B y);
  ...

}
```

Figure 7. Continuation classes in Java

6. Java implementation

The FRP implementation is in Java, and is a direct translation of the model discussed in Sections 2–5. Part of the interface for event sources is given in Figure 5, and is the Java equivalent of the relevant fragment of Figure 1. We make use of Java 8's support for anonymous functions (where $\lambda x \,.\, M$ is written in Java as `x -> M`).

There is a slight difference in the way Java treats bounded existential types, which is by wildcarding (the `?` type in the return type of `partition`). Java does not support direct unpacking of existential types, and instead unpacking must be performed by helper functions. For example in Java one cannot write:

```
  Events<T,Integer> xs = ...;
  PartitionedEvents<S,T,Integer> p =
    xs.partition(x -> 0 <= x);
  Events<S,Integer> pos = p.yes();
  Events<Minus<T,S>,Integer> neg = p.no();
  return pos.union(neg.map(Math.abs));
```

because there is no way to introduce the new bound type variable S. Instead, one writes:

```
  Events<T,Integer> xs = ...;
```

```
public class NotImpl<A> implements Not<A> {

  protected List<Not<A>> conts;
  final WeakReference<NotNotImpl<A>> not;

  public NotImpl(NotNotImpl<A> not) {
    this.not = new WeakReference<>(not);
  }

  public void accept(A x) {
    List<Not<A>> ks;
    NotNotImpl<A> not;
    synchronized(this) {
      ks = this.conts;
      not = this.not.get();
      if (not != null) {
        not.values = new List<>(x,not.values);
      }
    }
    while (ks != null) {
      ks.hd.accept(x); ks = ks.tl;
    }
  }

}

public class NotNotImpl<A> implements Not<Not<A>> {

  protected List<A> values;
  final NotImpl<A> not = new NotImpl<>(this);

  public void accept(Not<A> k) {
    List<A> xs;
    synchronized(this.not) {
      xs = this.values;
      this.not.conts = new List<>(k,this.not.conts);
    }
    while (xs != null) {
      k.accept(xs.hd); xs = xs.tl;
    }
  }

}
```

Figure 8. Mutable implementation of continuations

```
  return xs.partition(x ->
    0 <= x
  ).split((pos,neg) ->
    pos.union(neg.map(Math.abs))
  )
```

Inside the anonymous helper function, an anonymous type variable S is created, `pos` is given type `Events<S,Integer>` and `neg` is given type `Events<Minus<T,S>,Integer>`.

The example "Hello, World!" web application is shown in Figure 6. An implementation of an asynchronous servlet allows it to execute in any servlet container, resulting in the interactions:

```
$ curl -i localhost:8080/hello
HTTP/1.1 200 OK
Content-Length: 13
Server: Jetty(9.0.0.M5)

Hello, World!
```

```
public class FunctionNotNotRef<A,B>
  implements Function<A,Not<Not<B>>>,
    Not<Function<A,Not<Not<B>>>>
{

  protected Function<A,Not<Not<B>>> delegate;
  protected HashMap<A,NotNotImpl<B>> cache =
    new HashMap<>();

  public Not<Not<B>> apply(A x) {
    Function<A,Not<Not<B>>> f;
    NotNotImpl<B> k = null;
    synchronized (this) {
      f = this.delegate;
      if (f == null) {
        k = this.cache.get(x);
        if (k == null) {
          k = new NotNotImpl<B>();
          this.cache.put(x,k);
        }
      }
    }
    if (f == null) {
      return k;
    } else {
      return f.apply(x);
    }
  }

  public void accept(Function<A,Not<Not<B>>> f) {
    HashMap<A,NotNotImpl<B>> cache;
    synchronized (this) {
      cache = this.cache;
      this.cache = null;
      this.delegate = f;
    }
    for(
      Entry<A,NotNotImpl<B>> e : cache.entrySet()
    ) {
      A x = e.getKey();
      Not<B> k = e.getValue().not;
      f.apply(x).accept(k);
    }
  }

}

public<A,B> Function<A,Not<Not<B>>> fix(
  Function<
    Function<A,Not<Not<B>>>,
    Function<A,Not<Not<B>>>
  > f
) {
  FunctionNotNotRef<A,B> g =
    new FunctionNotNotRef<A,B>();
  Function<A,Not<Not<B>>> h =
    f.apply(g);
  g.accept(h);
  return h;
}
```

Figure 9. Implementation of fix

```
public class FunctionNotNotImpl<A,B>
  implements Function<A,Not<Not<B>>>, Not2<A,B>
{

  final WeakHashMap<A,NotNotImpl<B>> cache =
    new WeakHashMap<>();

  public synchronized NotNotImpl<B> apply(A x) {
    NotNotImpl<B> k = this.cache.get(x);
    if (k == null) {
      k = new NotNotImpl<B>();
      this.cache.put(x,k);
    }
    return k;
  }

  public void accept(A x, B y) {
    this.apply(x).not.accept(y);
  }

}

public<A,B> Function<A,Not<Not<B>>>
  buffer(Not<Not2<A,B>> k)
{
  FunctionNotNotImpl<A,B> f =
    new FunctionNotNotImpl<A,B>();
  k.accept(f);
  return f;
}
```

Figure 10. Implementation of buffer

and:

```
curl -i -d "Stuff" localhost:8080/hello
HTTP/1.1 405 Method Not Allowed
Content-Length: 0
Server: Jetty(9.0.0.M5)
```

Most of the work in the FRP implementation is in implementing the continuation classes `Not<A>` which represents $\mathsf{Not}\,A$ and `Not2<A,B>` which represents $\mathsf{Not}\,(A \times B)$. The implementation of `Not<A>` is as an interface with a single abstract method `void accept(A x)`. This method is allowed to be side-effecting (although care is required from any programmer working at this low level to respect the high-level semantics). Figure 7 shows some of the implementation of the classes.

One implementation of continuations is a pair of mutable containers, given in Figure 8. One is a container of $\mathsf{Not}\,A$ continuations, and implements $\mathsf{Not}\,A$; the other is a container of A values, and implements $\mathsf{Not}\,(\mathsf{Not}\,A)$. When a new continuation is added (by calling `accept` on the value container), it is applied to all of the existing values. Similarly when a new value is added (by calling `accept` on the continuation container), all the existing continuations are applied to it. Note that there is only a weak reference from the continuation container to the value container, so the value container can be garbage collected without impacting the continuation container.

To ensure thread-safety of the implementation, appropriate locking is used on critical sections. Note that we never call any callbacks while holding a lock, so there is never any attempt to acquire two locks simultaneously. Thus, the code is deadlock-free, and will not introduce issues with liveness.

```
public class ClockImpl<T> extends Clock<T> {

  final Not<Time> ntimes;
  final Not<Time> nfirst;
  final Not2<Time,Time> nprev;
  ...

}

public class EventsImpl<T,A> extends Events<T,A> {

  final public ClockImpl<T> clock;
  final public Not2<Time,A> nrelation;
  ...

}

public class PartitionedEventsImpl<T,A>
  implements PartitionedEvents<T,T,A>
{

  final Predicate<A> pred;
  final EventsImpl<T,A> yes;
  final EventsImpl<Minus<T,T>,A> no;

  final WeakHashMap<Time,Boolean> flags;
  final HashMap<Time,Time> prevUnknown;
  final HashMap<Time,Time> nextUnknown;
  final HashMap<Time,Time> prevSame;
  final HashMap<Time,Time> nextSame;
  Time first = null;

  public PartitionedEventsImpl(
    Predicate<A> pred, Events<T,A> xs
  ) {
    this.pred = pred;
    this.yes = new EventsImpl<>(...);
    this.no = new EventsImpl<>(...);
    xs.relation.pairs.accept(this::acceptEvent);
    ...
  }

  public void emitTime(Time t, Boolean flag) {
    if (flag) { this.yes.clock.ntimes.accept(t); }
    else { this.no.clock.ntimes.accept(t); }
  }

  public void emitEvent(Time t, A x, Boolean flag) {
    if (flag) { this.yes.nrelation.accept(t,x); }
    else { this.no.nrelation.accept(t,x); }
  }

  public void acceptEvent(Time t, A x) {
  Boolean tflag = this.pred.apply(x);
    synchronized (this) { this.flags.put(t,tflag); }
    this.emitEvent(t,x,tflag);
    this.emitTime(t,tflag);
    ...
  }

  ...

}
```

Figure 11. Implementation of partition

The two methods of $\mathsf{Not}\, A$ which require effort are fix (which finds the fixed point of set-valued functions) and buffer (which implements half of the isomorphism between relations and set-valued functions). These two functions are implemented similarly, in that under the hood they are based on mutable continuations, and use a `HashMap` to buffer values. The implementations are given in Figures 9 and 10.

In the case of fix, we make use of a class `FunctionNotNotRef` which implements $A \to \mathsf{Not}\,(\mathsf{Not}\, B)$ by acting as a delegate to another function of the same type. Crucially, the delegate may be set after the reference has been created, so the function has to respond to `apply` in the absence of its delegate. It does so by creating a new mutable $\mathsf{Not}\,(\mathsf{Not}\, A)$, which it stores in a cache before returning. When the delegate is set, it runs through the cache, initializing all of the cached continuations appropriately. Thus, a suitable mix of buffering and mutable state allows cyclic structures to be built which give the appearance of pure functional fixed points.

The implementation of buffer is similar, but there is one crucial difference. In the case of fix, the buffering is only required after the reference has been created, but its delegate is initialized. In most cases, this will be transitory, and the buffer will not be long-lived. In the case of buffer, the buffer will be long-lived, and we need to take care that it does not cause a space leak. For this reason, we use a weak hash map to store the buffer. When a key in the buffer can be garbage collected, it will be, and the corresponding value will be removed.

In its use in the FRP implementation, `FunctionNotNotImpl` is always used to implement a function of type $\mathsf{Time} \to \mathsf{Not}\,(\mathsf{Not}\, A)$, and so every use of buffering will introduce a `WeakHashMap` whose keys are `Time`. This makes `Time` objects quite expensive: even though each `Time` object is small (just containing a `long` timestamp) each live `Time` object is also keeping alive many entries in a buffer. For this reason, the FRP interface does not expose `Time` objects to user code, since it would be very easy to introduce a space leak by holding on to a time object. In our setting, this is the form that *time leaks* take [21], even though Java is a strict language.

There is one method of signals which we implement directly at a low level rather than going through the abstraction of sets and relations. An outline of the implementation of partition is given in Figure 11. It uses implementations of $\mathsf{Clock}\, T$ and $\mathsf{Event}\, T\, A$ which are backed by mutable sets to create two new event sources `yes` and `no`. When an event `x` arrives at time `t`, it calls `pred` on `x` to determine whether to send the event on to `yes` or to `no`. Most of the effort is in implementing `yes.clock` and `no.clock`, and to do this we maintain the following data structures:

- `flags` is a map from `Time` to `Boolean` such that $(t, b) \in [\![$`flags`$]\!]$ whenever there is some $(t, x) \in [\![$`xs.relation`$]\!]$ and $[\![$`pred`$]\!]\, x = b$,
- `prevUnknown` is a map from `Time` to `Time` such that $(t, s) \in [\![$`prevUnknown`$]\!]$ whenever $s \to t$ and either s or t is not in the domain of $[\![$`flags`$]\!]$,
- `nextUnknown` is the inverse of `prevUnknown`,
- `prevSame` is a map from `Time` to `Time` such that $(t, s) \in [\![$`prevSame`$]\!]$ whenever $s \to^*_{?/c} t$ and there is no $r \to_{?/c} s$,
- `nextSame` is a map from `Time` to `Time` such that $(s, t) \in [\![$`prevSame`$]\!]$ whenever $s \to^*_{b/?} t$ and there is no $t \to_{b/?} u$, and
- `first` $\in [\![$`xs.clock.first`$]\!]$,

where we write:

- $s \to t$ whenever $(t, s) \in [\![$`xs.clock.prev`$]\!]$,

- $s \to_{b/c} t$ whenever $s \to t$ and $(s, b) \in [\![\texttt{flags}]\!] \ni (t, c)$,
- $s \to_{b/?} t$ whenever $s \to_{b/c} t$ for some c, and
- $s \to_{?/c} t$ whenever $s \to_{b/c} t$ for some b.

This data is enough to construct `yes.clock` and `no.clock`, for example $(t, s) \in [\![\texttt{yes.clock.prev}]\!]$ whenever:

$$s \to_{\mathsf{true/true}} t \quad \text{or} \quad s \to_{\mathsf{true/false}} \cdot \to^{*}_{\mathsf{false/false}} \cdot \to_{\mathsf{false/true}} t$$

One unusual feature of our FRP implementation is that it allows out-of-order arrival of events. Due to multithreading, it is possible that an event (t, x) will arrive before an event (u, y) even when $t > u$. For example FrTime [7] makes use of an ordering scheme similar to that of Acar's [1] self-adjusting computation to ensure in-order arrival of events. In our setting, we are prioritizing liveness over execution order, and so we only introduce time dependencies where they are needed (for example in `accum`) and allow most event sources to proceed at their own pace.

7. Comparison with push/pull FRP

The most closely related work is Elliott's [10] *push-pull FRP*. Elliott reconstructs the behaviour and event source types using futures. Elliott's type $\mathsf{Future}\, A$ can be encoded in our system as:

$$\mathsf{Future}\, A = \mathsf{Not}\,(\mathsf{Not}\,(\mathsf{Time} \times A))$$

so Elliott's type for event sources:

$$\mathsf{Event}\, A = \mathsf{Future}\,(A \times \mathsf{Event}\, A)$$

is the equivalent of:

$$\mathsf{Event}\, A = \mathsf{Not}\,(\mathsf{Not}\,(\mathsf{Time} \times A \times \mathsf{Event}\, A))$$

Comparing this to our type:

$$\mathsf{Event}\, T\, A = \mathsf{Not}\,(\mathsf{Not}\,(\mathsf{Time} \times A))$$

we can see some of the important distinctions between the approaches:

- Push-pull FRP is based on the one-shot Future type, rather than the implementation of Set using callbacks.
- In push-pull FRP, events are guaranteed to arrive in order. If $(s, x, \sigma) \in \rho$ and $(t, y, \tau) \in \sigma$ then $s < t$. In our implementation of FRP, events may arrive out-of-order. In particular, push-pull FRP uses McCarthy amb to implement union. Out-of-order events also occur in Jeltsch's push-based FRP [17, §4.2.6].
- Push-pull FRP does not make use of the isomorphism between relations and set-valued functions.

There are also some differences of focus: we have focused on liveness, which is not an issue in push-pull FRP, and we have not focussed on continuous-time behaviours.

8. Conclusions and future work

We have presented a model of FRP which provides liveness guarantees by static typing. The implementation of FRP is based on a model of sets and relations, which in turn is implemented using continuations. This implementation shows how the pull implementation of behaviours using futures, and the push implementation of event sources using the observer pattern, arise naturally.

There are two areas of future work: on stopping space and time leaks and on distribution.

Currently, there is effort made to ensure that signals do not cause space and time leaks, by appropriate use of weak pointers. However, it is possible for users to cause a leak by keeping a reference to a signal live. We are already tracking time domains, so it should be possible to use them as Jeltsch's [16] era parameters, and ensure that signals do not have to buffer all their events.

The Java implementation of the FRP library is already linked against a servlet class, to allow it to be used as the processing engine in an HTTP server. It would be interesting to see if HTTP could be used as the communication protocol between FRP instances, and so build a distributed FRP implementation.

References

[1] U. A. Acar. *Self-Adjusting Computation*. PhD thesis, Carnegie Mellon Univ., 2005.

[2] H. Apfelmus. `http://www.haskell.org/haskellwiki/Reactive-banana`.

[3] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Sci. Computer Programming*, 19(2):87–152, 1992.

[4] G. Bracha, J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Sun Microsystems, third edition, 2005.

[5] S. Burbeck. Applications programming in Smalltalk-80: How to use model-view-controller (MVC), 1987.

[6] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 28(150), 1985.

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

[8] A. Courtney. Frappé: Functional reactive programming in Java. In *Proc. Symp. Pratical Aspects of Declarative Languages*, pages 29–44, 2001.

[9] J. Donham. Functional reactive programming in OCaml. `https://github.com/jaked/froc`.

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

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

[12] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. *Proc. Int. Joint Conf. Artificial Intelligence*, pages 235–245, 1973.

[13] A. S. A. Jeffrey. `https://github.com/agda/agda-frp-js/`.

[14] A. S. A. Jeffrey. Causality for free!: Parametricity implies causality for functional reactive programs. In *Proc. ACM Workshop Programming Languges meets Program Verification*, pages 57–68, 2013.

[15] A. S. A. Jeffrey. Provably correct web applications: FRP in Agda in HTML5. In *Proc. Int. Symp. Practical Aspects of Declarative Languages*, 2013.

[16] W. Jeltsch. Signals, not generators! In *Proc. Symp. Trends in Functional Programming*, pages 283–297, 2009.

[17] W. Jeltsch. *Strongly Typed and Efficient Functional Reactive Programming*. PhD thesis, BTU Cottbus, 2012.

[18] N. Krishnaswami and N. Benton. Ultrametric semantics of reactive programs. In *Proc. IEEE Logic in Computer Science*, pages 257–266, 2011.

[19] D. Leijen and E. Meijer. Domain specific embedded compilers. In *Proc. Conf. Domain-Specific Languages*, pages 109–122, 1999.

[20] 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, ACM Conf. Object-Oriented Programming Systems, Languages and Applications*, pages 1–20, 2009.

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

[22] G. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[23] Yale Haskell Group. Yampa library for programming hybrid systems. `http://www.haskell.org/haskellwiki/Yampa`.