Practical Principled FRP

Forget the Past, Change the Future, *FRPNow*!

Atze van der Ploeg   Koen Claessen

Chalmers University of Technology, Sweden

{atze,koen}@chalmers.se

Abstract

We present a new interface for practical Functional Reactive Programming (FRP) that (1) is close in spirit to the original FRP ideas, (2) does not have the original space-leak problems, without using arrows or advanced types, and (3) provides a simple and expressive way for performing I/O actions from FRP code. We also provide a denotational semantics for this new interface, and a technique (using Kripke logical relations) for reasoning about which FRP functions may "forget their past", i.e. which functions do not have an inherent space-leak. Finally, we show how we have implemented this interface as a Haskell library called *FRPNow*.

Categories and Subject Descriptors   D.3.2 [*Applicative (functional) languages*]

Keywords   Functional Reactive Programming, Space-leak, Purely functional I/O, Kripke logical relations

1. Introduction

Many computer programs are *reactive*: they continuously interact with their environment. Examples of such programs are servers, control software, and programs with a graphical user interface. Such systems are often constructed by using callbacks and/or concurrency, even when using functional programming languages. These methods lead the programmer to rely on mutable state and/or introduce nondeterminism, making reactive programs hard to construct, compose and understand.

Functional Reactive Programming (FRP) was introduced by Elliott and Hudak [8] with their Haskell library *Fran*, an elegant and powerful way of *modeling* reactive animations. Their interface provides a purely functional way of describing *events*, values that are known from some point in time, and *behaviors*, values that change over time.

These abstractions also provide an attractive way of *programming* reactive systems. However, Fran has two problems which limit its applicability to practical programming: (a) Fran easily leads to severe space leaks[9, 11, 15, 16], and (b) Fran does not provide a general way to interact with the outside world from an FRP context.

In this paper, we slightly modify the Fran interface and its denotational semantics so that these two problems are solved,

without compromising the original spirit behind FRP, and present an implementation of this interface in Haskell. Our contribution is thus a principled and practical way of programming reactive systems with FRP, without callbacks, nondeterminism or mutable state.

Let us delve a bit deeper into the two problems mentioned earlier.

Space Leaks   The first problem, the space leak problem, can be analyzed as follows. A program in FRP can lead to space leaks in three ways:

1. The program using the FRP library can have a space leak.

2. The implementation of the FRP library can have a space leak.

3. The *interface* of the FRP library, i.e. the set of functions offered by the library, can be *inherently leaky*.

Each of these implies the previous: if we have an interface which is inherently leaky, then we cannot hope for an implementation without a space leak, and if we have an implementation with a space leak then any program using that implementation is likely to have a space leak as well.

The Fran interface is inherently leaky, in that it *prevents the past to be forgotten*. To see this, consider the following function supported by Fran:

$$snapshot :: Behavior\ a \rightarrow Event\ () \rightarrow Event\ a$$

which samples the behavior when the event occurs, producing an event with that value that occurs at the same time. For example, the expression $snapshot\ mousePos\ easter$ gives the mouse position at Easter. Such an expression can occur inside other events, and hence when evaluating this expression it may be Christmas. In that case, we must have remembered the mouse position at Easter at least until Christmas. Typically, this means that at least all the mouse positions from Easter till Christmas are remembered, leading to a severe space leak.

In some cases, a sufficiently smart runtime might have figured out before or at Easter that the mouse position at Easter was needed at Christmas, and that the other mouse positions do not have to be remembered. However, in general events that occur in the future may be *unknown* and their future evaluation may depend on past values. The runtime cannot possibly predict at Easter, exactly what should be remembered till Christmas.

In this paper, we slightly modify the Fran interface so that it is guaranteed that implementations *can forget all past values of all behaviors*. The key idea behind our solution is that we use behaviors as a *reader monad* in time and modify the interface such this reader monad can only be "run" at the present time or in the future, but *not* in the past.

Our solution differs from other solutions to the space-leak problems of FRP [9, 11, 15, 16], in that behaviors are still first-class values (we don't use arrows for example), the types used in the interface are simple types (we stay within Haskell'98), and our

interface is very close in spirit to the Fran interface. However, our solution does not statically prevent *all* space leaks; space leaks of type 1 above are still possible. Of course, this holds in general for libraries for general purpose languages (such as *Haskell* and *ML*).

The main challenge we faced was to design the new *interface* so that it becomes possible to implement the library without space leaks. To this end, we introduce a method of determining whether or not an FRP function is inherently leaky, which uses an Kripke logical relation called *equality up to time-observation*. Two values are equal up to time-observation from time t, if they cannot be distinguished anymore by observations at time t or later. We then show, using this relation, that our interface indeed allows implementations to forget the past. The implementation we discuss later on in the paper is evidence that it is indeed possible to implement the interface.

I/O *in* FRP The second problem with Fran is that interaction with the outside world is limited to a few built-in primitives: there is no general way to interact with the outside world. Arrowized FRP does allow general interaction with the outside world, by organizing the FRP program as a function of type $Behavior\ Input \rightarrow Behavior\ Output$[1], where $Input$ is a type containing all input values the program is interested in and $Output$ is a type containing all I/O requests the program can do. This function is then passed to a wrapper program, which actually does the I/O, processing requests and feeding input to this function.

This way of doing I/O is reminiscent of the stream based I/O that was used in early versions and precursors to Haskell, before monadic I/O was introduced. It has a number of problems (the first two are taken from Peyton Jones [10] discussing stream based I/O):

- It is hard to extend: new input and output facilities can only be added by changing the $Input$ and $Output$ types, and then changing the wrapper program.

- There is no close connection between a request and its corresponding response. For example, an FRP program may open multiple files simultaneously. To associate the result of opening a file to its the request, we have to resort to using unique identifiers.

- All I/O must flow through the top-level function, meaning the programmer must manually route each input to the place in the program where it is needed, and route each output from the place where the request is done.

Other FRP formulations partially remedy this situation[1, 21], but none overcome all of the above issues. We present a solution that is effectively the FRP counterpart of monadic I/O. We employ a monad, called the Now monad, that allows us to (1) *sample* behaviors at the current time, and (2) plan to execute Now computations in the future and (3) start I/O actions with the function:

$$async :: IO\ a \rightarrow Now\ (Event\ a)$$

which starts the IO action and immediately returns the event associated with the completion of the I/O action. The key idea is that all actions inside the Now monad are *synchronous*[2], i.e. they return immediately, conceptually taking zero time, making it easier to reason about the sampling of behaviors in this monad. Since starting an I/O action takes zero time, its effects do not occur now, and hence $async$ does not change the present, but "changes the future". Like the I/O monad, the Now monad is used to deal with input as well as output, both via $async$. This approach does not have the problems associated with stream-based IO, and is as flexible and modular as regular monadic I/O.

---

[1] In Arrowized FRP, this would be type $SF\ Input\ Output$.

[2] Synchronous in the same sense as *synchronous* dataflow programming: the input is synchronous with the output.

Implementation These two interface changes, namely (a) ensuring that implementations can forget the past, and (b) adding I/O from an FRP context, give a principled basis for practical FRP programming. An implementation of this interface that itself has no space leak and implements the I/O interface such that operations in the Now monad appear to take zero time, is not trivial. Consider for example, a straightforward implementation of behaviors, as used by Elliott [7], as an initial value and initial change event:

data $Behavior\ a = a\ 'Step'\ Event\ (Behavior\ a)$

Although elegant, this implementation of behaviors is problematic: any reference to a behavior, for example $mousePos$, will refer to the initial value of the behavior and the event when it first changes. This event in turn holds a reference to the second value of the behavior and the event that it changes the second time, and so on. In this way, this definition prevents old values to be garbage collected. We present an implementation of our interface in Haskell that (a) does forget the past, and (b) gives the illusion that actions in the Now monad are immediate.

Contributions We start with a background section, introducing a modernized subset of the Fran interface. We then arrive at our contributions:

- We present a simple modification to the subset of the Fran interface of Section 2, which allows for implementations to forget the past (Section 3).

- We present a simple method of distinguishing functions which allow implementations to forget the past from functions which do not, by introducing the notion of *time-observational equality*. (Section 3).

- We introduce a simple, yet flexible, way to let pure FRP code asynchronously interact with the outside world. (Section 4)

- We demonstrate that the restrictions of our interface do not rule out useful programs by showing how the functionality provided by other FRP interfaces can be also be achieved in our interface. (Section 5)

- We present an implementation of our FRP interface in Haskell that indeed forgets the past and that gives the illusion that actions in the Now monad are immediate (Section 6).

In Section 7 we discuss related work and in Section 8 we discuss and conclude.

The implementation in Haskell of the interface described in this paper is available at:

$$\texttt{https://github.com/atzeus/FRPNow/}$$

We plan to shortly release a library based on the ideas in this paper, under the name *FRPNow*.

2. Introducing FRP

In this section we introduce FRP by presenting a modernized subset of the Fran[8] interface, inspired by Elliott's modernized FRP interface[7]. The denotational semantics of the modernized subset of Fran are shown in Figure 1(a).

In this paper we use $\overset{\circ}{=}$ to indicate semantic equality. Hence, these definitions do not give implementations, but denotations (i.e. mathematical meaning). For clarity of notation, the denotations in this paper are also given in Haskell syntax. The denotational semantics assume that all values are total, we leave its strictness properties as future work. In the remainder of this section, we discuss the definitions in the denotational semantics in sequence.

The main concepts are behaviors (B), i.e. values that change over time, and events (E), i.e. values that are known from some point in time on. Examples of behaviors are the position of the mouse,

$$\textbf{type } B\ a \stackrel{\circ}{=} Time \rightarrow a$$
$$\textbf{type } E\ a \stackrel{\circ}{=} (Time^+, a)$$

$$never :: E\ a$$
$$never \stackrel{\circ}{=} (\infty, \bot)$$

$\textbf{instance } Monad\ B\ \textbf{where}$
$$\quad return\ x \quad \stackrel{\circ}{=} \lambda t \rightarrow x$$
$$\quad m \ggg f \quad \stackrel{\circ}{=} \lambda t \rightarrow f\ (m\ t)\ t$$

$\textbf{instance } Monad\ E\ \textbf{where}$
$$\quad return\ x \quad \stackrel{\circ}{=} (-\infty, x)$$
$$\quad (ta, a) \ggg f \stackrel{\circ}{=} \textbf{let } (tb, x) \stackrel{\circ}{=} f\ a$$
$$\qquad\qquad\qquad \textbf{in } (max\ ta\ tb, x)$$

$$switch :: B\ a \rightarrow E\ (B\ a) \rightarrow B\ a$$
$$switch\ b\ (t, s) \stackrel{\circ}{=} \lambda n \rightarrow \textbf{if } n < t \textbf{ then } b\ n \textbf{ else } s\ n$$

(a) Functions taken from Fran.

$$whenJust :: B\ (Maybe\ a) \rightarrow B\ (E\ a)$$
$$whenJust\ b \stackrel{\circ}{=} \lambda t \rightarrow$$
$$\quad \textbf{let } w \stackrel{\circ}{=} minSet\ \{\, t' \mid t' \geqslant t \wedge isJust\ (b\ t') \,\}$$
$$\quad \textbf{in if } w \equiv \infty \textbf{ then } never$$
$$\qquad \textbf{else } (w, fromJust\ (b\ w))$$

(b) Forgetful function to observe changes.

Figure 1. Our FRP interface and its denotational semantics.

of type $B\ Point$, and an animation, of type $B\ Picture$. Examples of events are the next mouse button that will be pressed, of type $E\ Button$, and the final selection of a color from a color picker, of type $E\ Color$.

A behavior is a value that changes over time, and hence its denotation is a function from time to value. In Fran, $Time$ is equal to the real numbers ($\mathbb{R}$), but we only assume that time is totally ordered and that it has a least element ($-\infty$). We do *not* require that time is enumerable, and hence there is no notion of a *next timestep* in our interface. Unlike the original FRP interface, the type $Time$ is *not* part of the interface, but only of the denotational model.

The denotation of an event is a pair of a point in time and a value. To include events that will never occur, the point in time at which an event can occur is $Time^+ = Time \cup \{\infty\}$. This gives us a $never$ occurring event for each type. The use of $\bot$ in $never$ may seem to contradict our assumption that all values are total, but the $\bot$ in never can never be observed[3].

Both behaviors and events are commutative monads[4]. A behavior is semantically a *reader* monad in time, whereas an event is semantically a *writer* monad in time, with the monoid instance $(max, -\infty)$. Since any monad gives rise to an applicative functor [14], both behaviors and events also support the applicative functor interface. The functions on applicative functors used in this paper and their definitions using a monad instance are shown in Figure 2. As an example usage of this interface for behaviors, the expression $(isInside \diamondsuit mousePos \circledast rect)$ gives a behavior that indicates whether the mouse cursor is inside the (potentially moving) rectangle at any point in time. As an example usage of the applicative functor interface for events, the expression

$$pure\ x\ =\ return\ x$$
$$(\circledast) :: Monad\ m \Rightarrow m\ (a \rightarrow b) \rightarrow m\ a \rightarrow m\ b$$
$$f \circledast x = \textbf{do } fv \leftarrow f; xv \leftarrow x; return\ (fv\ xv)$$
$$(\diamondsuit) :: Monad\ m \Rightarrow (a \rightarrow b) \rightarrow m\ a \rightarrow m\ b$$
$$f \diamondsuit x = pure\ f \circledast x$$
$$(\lhd) :: Monad\ m \Rightarrow a \rightarrow m\ b \rightarrow m\ a$$
$$x \lhd y = const\ x \diamondsuit y$$

Figure 2. Applicative functor functions from monads.

$(mixColor \diamondsuit colorSelection1 \circledast colorSelection2)$ gives an event carrying the composite of two colors, as soon as the user has selected both colors. We will give examples utilizing the full the monad interface for behaviors and events later.

To *introduce* a change over time, the $switch$ function can be used, which when given a behavior b and an event e containing another behavior, returns a behavior that acts as b initially, and switches to the behavior inside e as soon as it occurs. As an example, the expression $(animColor\ `switch`\ (pure \diamondsuit pickColor))$ gives a behavior that acts as $animColor$ initially, which animates between green and red, until the user has picked a color, after which it will be that color.

In our interface, $switch$ is the only way to introduce a change to a behavior, and hence all behaviors only change discretely, i.e. they are piecewise-constant functions, whereas in Fran behaviors can change continuously. This does not make any difference for programming with behaviors, as there is no way to distinguish, through observation, a continuously changing behavior from a discretely changing behavior.

Finally, to *observe* (*eliminate*) a change over time, the following function can be used[5]:

$$whenJust^\dagger :: B\ (Maybe\ a) \rightarrow E\ () \rightarrow E\ a$$
$$whenJust^\dagger\ b\ (t, ()) \stackrel{\circ}{=}$$
$$\quad \textbf{let } w \stackrel{\circ}{=} minSet\ \{\, t' \mid t' \geqslant t \wedge isJust\ (b\ t') \,\}$$
$$\quad \textbf{in if } w \equiv \infty \textbf{ then } never$$
$$\qquad \textbf{else } (w, fromJust\ (b\ w))$$

The notation $minSet\ x$ indicates the minimum element of the set x, which is not valid Haskell, but is a valid denotation. The function $whenJust^\dagger$ is the only function that is problematic, it can for example be used to define the $snapshot$ function from the introduction. In this paper, a superscript $\dagger$ indicates an inherently leaky definition.

As an example usage of $whenJust^\dagger$, suppose $localCoordinates$ is a behavior of type $B\ (Maybe\ Point)$ that gives $Just$ the local mouse coordinates inside a rectangle when the mouse is inside that rectangle, and $Nothing$ otherwise. The expression $whenJust\ localCoordinates\ e$ then gives the earliest time that the mouse is inside the rectangle after or during event e, along with the local mouse coordinates at that time. If the mouse is never again inside the rectangle after the event e, then $minSet$ gives the minimum element of the empty set, which is ∞, and hence the result will be a never occurring event.

The Fran interface has several other functions (namely $time$, $(.|.)$ and $timeTransformation$) which are not part of the core FRP interface presented in this section. In section 5, we will discuss how the unproblematic parts of the functionality that these functions provide can also be provided by the (modified version of) this interface.

---

[3] Equivalently, the denotation of events could be chosen to be $Maybe\ (Time, a)$ where $Nothing$ indicates $never$, eliminating the need for $\bot$. We have opted not to do this since it obfuscates that E is a writer monad.

[4] The original FRP interface only supported what now can be considered an applicative functor interface for behaviors.

[5] This functionality provided by this function was provided by $pred$ and $snapshot$ in Fran, which can be expressed in terms of $whenJust$ and vice versa. We show how this is done in a document accompanying the online code.

3. Forget the Past!

The FRP interface presented in the previous section, like the original Fran interface, is inherently leaky. In this section, we first informally present our solution to this problem, and afterwards introduce the notion of *time-observational equality*, which gives a method to formally distinguish functions which allow implementations to forget the past from functions which do not. We then prove that $whenJust^\dagger$ does not allow implementations to forget the past and that our new version, $whenJust$, does.

Semantically, there is no notion of "now": the semantics simply state how values relate to each other. Of course, when running an FRP program there *is* a notion of "now": some events have already occurred and some have not, and a behavior consists of past, present and future values. The interface from the previous section allows access to past values of a behavior, leading to space leak problems.

More specifically, the only problematic function in the interface is $whenJust^\dagger$. Consider the expression $whenJust^\dagger\ b\ e$: the event e may lie in the past when evaluation of the expression occurs, which makes it necessary to keep remember all old values of the behavior b since e.

3.1 Solution

To fix this problem, we slightly change the function $whenJust^\dagger$ to its forgetful version in Figure 1(b). Instead of taking an argument of the type $E\ ()$, we produce a value in the behavior monad. Semantically, this does not make much difference, a value of type $E\ ()$ is the same as a point in time (ignoring the case when the event never occurs) and hence the type of $whenJust^\dagger$ can be thought of as $B\ (Maybe\ a) \to Time \to E\ a$, which is equivalent to the type of $whenJust$, namely $B\ (Maybe\ a) \to B\ (E\ a)$.

This change has an important effect which allows implementations to forget the past: the result of $whenJust$ is in the behavior monad. As we will see later when discussing our I/O interface, the only way to observe a behavior from the outside world is to request its value now. This makes it impossible to request past values of a behavior, thus ensuring that implementations can forget the past.

Whereas with $whenJust^\dagger$ the sample time is given as an event, with $whenJust$ the sample time is provided by the monadic context. As an example, the following function gives the next time the input behavior changes, at any point in time.

$$change :: Eq\ a \Rightarrow B\ a \to B\ (E\ ())$$
$$change\ b = \textbf{do}\ cur \leftarrow b$$
$$when\ ((cur \not\equiv) \lozenge b)$$
$$when :: B\ Bool \to B\ (E\ ())$$
$$when\ b = whenJust\ (boolToMaybe \lozenge b)$$

Where $boolToMaybe$ converts $True$ to $Just\ ()$ and $False$ to $Nothing$. Here we use a behavior as a monad where we can sample other behaviors. We first sample b to obtain its current value, and then use that value to sample $when\ ((cur \not\equiv) \lozenge b)$.

While $whenJust$ no longer allow us to sample *past* values of a behavior, it does allow us to to sample in the *future*, as shown by the non-leaky version of the $snapshot$ function:

$$snapshot :: B\ a \to E\ () \to B\ (E\ a)$$
$$snapshot\ b\ e = \textbf{let}\ e' = (Just \lozenge b) \lessdot e$$
$$\textbf{in}\ whenJust\ (pure\ Nothing\ `switch`\ e')$$

The resulting behavior changes depending on whether the argument event lies in the future or in the past. If the argument event lies in the future (or present), then the value of resulting event is the value of the behavior at the time of the argument event. If the argument event lies in the past, then we will not sample the behavior in the past, instead giving the present value of the behavior. The denotation of $snapshot$ shows this behavior more clearly :

$$snapshot\ b\ (t,()) \doteq \lambda n \to \textbf{let}\ t' \doteq max\ t\ n\ \textbf{in}\ (t', b\ t')$$

3.2 Making Forgetfulness Precise

Our solution raises the question of which FRP functions allow implementations to forget the past, and which functions are inherently leaky. To answer this question, we define the notion of *equality up to time-observation*, a coarser notion of equality than regular equality. Informally speaking, two values are equal up to time-observation from a time t, if they cannot be distinguished through observations from time t onwards.

To simplify our presentation, we consider only the set of types generated by the following grammar:

$$\Theta ::= o \mid B\ \Theta \mid E\ \Theta \mid \Theta \to \Theta$$

Where o is some base type, for example $Integer$. Although this does not include all possible Haskell types, we argue that results for this set of types are transferable to all Haskell types. For instance, algebraic data types are isomorphic to their encodings as functions (for instance, for lists we can use the Church encoding).

When two values are equal up to time observation depends on their types. Informally speaking, the cases are as follows:

- Two values of a base type are equal up to time observation from any time if they are equal.

- Two behaviors are equal up to time observation from now if their values at any point in the present or future are equal up time-observation from that point.

- Two events are equal up to time observation from now on if one of the following cases holds:

  ▪ Both events have already occurred and their values are equal up to time observation from now on.

  ▪ Both events occur at the same point in the future, and their values are are equal up to time observation from that point in time.

  ▪ Both events never occur.

- Two functions are equal up to time observation from time t if they cannot be used to distinguish two values that are equal up to time observation from any time $\geqslant t$.

This is formally stated as follows:

Definition 1. *Equality up to time-observation*, is a family of binary relations $\overset{t}{\rightrightarrows}_\theta$, where $\theta \in \Theta$ and $t \in Time$, such that $a\ \overset{t}{\rightrightarrows}_\theta\ b$ if and only if one of the following cases holds:

- $\theta = o \wedge a = b$
- $\theta = B\ x \wedge \forall t' \geqslant t.\ a\ t'\ \overset{t'}{\rightrightarrows}_x\ b\ t'$
- $\theta = E\ x \wedge a \doteq (t_a, v_a) \wedge b \doteq (t_b, v_b)$
  $\wedge \max t_a\ t = \max t_b\ t \wedge (t_a = \infty \vee v_a\ \overset{\max\ t_a\ t}{\rightrightarrows}_x\ v_b)$
- $\theta = x \to y \wedge \forall p\ q\ (t' \geqslant t).\ p\ \overset{t'}{\rightrightarrows}_x\ q \to a\ p\ \overset{t'}{\rightrightarrows}_y\ b\ q$

We write $\overset{t}{\rightrightarrows}$ for equality up to time observation from time t between two values of any type, defined as $\bigcup_{\theta \in \Theta} \overset{t}{\rightrightarrows}_\theta$.

Equality up to time-observation is a binary Kripke logical relation, which means that it is a binary relation parametrized over a base type (o) and a pre-order (the total order $Time$ in our case), where the last of the above cases holds, and $t \leqslant t' \to \overset{t}{\rightrightarrows} \subseteq \overset{t'}{\rightrightarrows}$.

If, for all t, $\overset{t}{\rightrightarrows}$ is an equality relation (meaning reflexive, symmetric and transitive) on the set of all values that can be created by using an FRP interface, then it is safe for an implementation of that interface to forget old values of a behavior. In particular, this means that if $e \doteq (t, s)$, then $b\ `switch`\ e \overset{t}{\rightrightarrows} s$, and hence there is no way to distinguish $b\ `switch`\ e$ from s after time t, making it safe

```
type Now a
instance Monad Now
async    :: IO a         → Now (E a)
sample   :: B a          → Now a
plan_Now :: E (Now a) → Now (E a)
runNow :: Now (E a) → IO a
```

Figure 3. I/O interface.

to forget that it was ever equal to b 'switch' e, only remembering that it is equal to s from now on.

The symmetric and transitive requirements follow from the definition, but the reflexivity requirement does not. Hence to show that an FRP interface allows implementations to forget the past, it suffices to show that for each function f in the interface, $f \cong^t f$, for all t. Which brings us to the following definition:

Definition 2. A function f is *forgetful* if and only if $\forall t.f \cong^t f$. When a function is not forgetful, we call it *inherently leaky*.

Lemma 1. $whenJust^\dagger$ is inherently leaky.

Proof. By counterexample: Let $es \overset{\circ}{=} (1, pure\ Nothing), el \overset{\circ}{=} (0, ()), er \overset{\circ}{=} (2, ())$ and $b \overset{\circ}{=} pure\ (Just\ ())$ 'switch' es. We know that $b \cong^2 b$ and $el \cong^2 er$, but $whenJust^\dagger\ b\ el \overset{\circ}{=} el$ and $whenJust^\dagger\ b\ er = never$ and hence $whenJust^\dagger\ b\ el \ncong^2 whenJust^\dagger\ b\ er$ □

Lemma 2. $whenJust$ is forgetful.

Proof. To prove: $\forall b_1\ b_2\ t.b_1 \cong^t b_2 \rightarrow whenJust\ b_1 \cong^t whenJust\ b_2$
By $b_1 \cong^t b_2$ we know that: $minSet\ \{\ t'\ |\ t' \geqslant t \wedge isJust\ (b_1\ t')\ \}$ is equal to $minSet\ \{\ t'\ |\ t' \geqslant t \wedge isJust\ (b_2\ t')\ \}$. Let $w \geqslant t$ be the outcome of $minSet$. If $w = \infty$ then we are done by $never \cong^t never$, otherwise we know that $fromJust\ (b_1\ w) \cong^t fromJust\ (b_2\ w)$ by $b_1 \cong^t b_2$. □

The other functions in our FRP interface are all forgetful. The proofs are straightforward and are hence not presented here, but they can be found in the online git repository accompanying this paper.

4. Putting the Act Back in Functional Reactive Programming

The interface given in the previous sections allows the programmer to express pure computations involving time, but provides no way to interact with the outside world at all. In fact, using this interface we can only create events at $-\infty$ and ∞, and hence it only allows us to express constant behaviors. To do anything interesting we need events from the outside world.

Our I/O interface, show in Figure 3, is centered around a commutative monad called the Now monad, which allow us to start I/O actions, sample behaviors and plan to execute Now computations in the future. Starting an I/O action is done using the $async$ function, which immediately returns an event that will occur when the I/O action is done, carrying the result of the I/O action. Unlike running I/O actions in the IO monad, $async$ does *not* block until the I/O action is completed.

To schedule I/O actions in the future, we provide the $plan_{Now}$ function. This function takes an event carrying a Now computation, and makes sure that computation is executed as soon as the event occurs. The function $plan_{Now}$ also immediately returns an event, carrying the result of the future Now computation. Like the $snapshot$ function, $plan_{Now}$ does not run Now computations in the past, instead running the Now computation immediately if the event already occurred.

We can use this interface for both input and output. As an example of input, suppose we have a function $nextMousePos$, of type $IO\ Point$, which blocks until the mouse is moved, and then returns its new coordinates. We can use this function to implement a behavior which always gives the current mouse position:

```
getMousePos :: Now (B Point)
getMousePos = loop (0,0) where
  loop p = do e ← async nextMousePos
              e' ← plan_Now (loop <≫ e)
              return (pure p 'switch' e')
```

Here we first initialize the mouse position at point $(0, 0)$ and asynchronously start an $nextMousePos$ action. As soon as the mouse position moves, we start another $nextMousePos$ action and *switch* to the new mouse position.

As an example of output, suppose that we have a behavior giving the picture that should be drawn on screen at any point in time, and a function $drawPict :: Picture \rightarrow IO\ ()$ that performs the side-effect of actually drawing this picture to the screen. We can then keep the screen up to date as follows:

```
drawAll :: B Picture → Now ()
drawAll b = loop where
  loop = do p ← sample b
            d ← async (drawPict p)
            e ← sample (change b)
            plan_Now (loop <≪ (d ≫ e))
            return ()
```

Here we first sample the current value of the picture behavior. We then start the action of drawing it and obtain the event that the drawing is done as d. The event that the picture is different than it is now is obtained as e. We then plan to do the whole thing again, when the picture has changed *and* we are done drawing ($d \gg e$).

We can also use $async$ to run an expensive *pure* computation asynchronously. As an example consider a chess program: the next move of the computer is an expensive computation and we do not want to block the rest of the program while it is computed. We can simply run this computation asynchronously by doing :

$$async\ (evaluate\ nextMove)$$

Which starts a separate thread for the computation and returns the event that occurs when the computation is done.

We can use this interface to do anything one could do in the IO monad. For example, we can dynamically open files or dynamically create new widgets, we do not have to set up the connection to files or widgets from outside the FRP context.

On the top-level, the evaluation of a Now monad is started using the $runNow$ function, which executes the initial Now computation and the Now computations that it plans for the future, until the event given by the initial Now computation occurs, after which the corresponding value of the event will be returned. All Now computation that were still planned are then canceled.

None of the actions in the Now monad conceptually take any time to execute, which is essential for our programming model. Hence it is guaranteed that:

$$\mathbf{do}\ x \leftarrow sample\ b; y \leftarrow sample\ b; return\ (x, y)$$
$$= \mathbf{do}\ x \leftarrow sample\ b; return\ (x, x)$$

Furthermore, since all functions in the Now monad are instantaneous, the events returned by $async$ will always lie in the future.

It is however possible to create an event or behavior from one $runNow$ context and then to use that event or behavior in another $runNow$ context. In that case, the resulting behavior of the program is undefined and our implementation throws an error. Another possible approach is to rule out the mixing of contexts *statically*

using techniques know from the ST monad [12]. This would add an extra parameter to the types E, B and Now, and would have changed the type of the $runNow$ function to the following:

$$runNow :: (\forall s.\ Now\ s\ (E\ s\ a)) \rightarrow IO\ a$$

We have opted *not* to apply this technique, because the extra type parameter pervades client code and can be somewhat inconvenient for users[13].

5. Programming with FRPNow!

In the previous two sections, we first restricted the FRP interface by making sure that the past can be forgotten, and then generalized the interface by providing arbitrary interaction with the outside world. In this section we aim to convince the reader that this new interface does not rule out useful programs, by means of examples.

5.1 State over Time

Carrying state over time becomes a bit different with $whenJust$ from with $whenJust^{\dagger}$. As an example, in the FRP formulation with $whenJust^{\dagger}$, we could define an inherently leaky function with the following type:

$$countChanges^{\dagger} :: Eq\ a \Rightarrow B\ a \rightarrow B\ Int$$

Which gives a behavior that indicates how often the input behavior has changed *since the start of the program*. Since we can apply functions like $countChanges^{\dagger}$ to any behavior at any time, this implies that any behavior would need to retain references to all its past values. With our interface, we can create a function that achieves the same effect, but *is* forgetful, with the following type:

$$countChanges :: Eq\ a \Rightarrow B\ a \rightarrow B\ (B\ Int)$$

The result of this function is now $B\ (B\ Int)$ instead of $B\ Int$, because the number of changes to the input behavior depends on *when the counting started*. When $countChanges\ b$ is sampled at time t, it will return a behavior, of type $B\ Int$, that counts the changes to the original behavior since t. This construction enables us to carry state over time, while still being able to forget the past.

The implementation of $countChanges$ is as follows:

```
countChanges b = loop 0 where
  loop :: Int → B (B Int)
  loop i = do e  ← change b
              e' ← snapshot (loop (i + 1)) e
              return (pure i 'switch' e')
```

We first obtain the current value of $change\ b$, which gives us the event that b changes. As soon as this event occurs, we would like to run the loop again. Since $loop\ (i + 1)$ has type $B\ (B\ Int)$, we can $snapshot$ this behavior when the event e occurs, to obtain the result of the next iteration of the loop, of type $E\ (B\ Int)$. We then produce a behavior that is initially i, and switches to the behavior given by the next iteration of the loop as soon the argument behavior changes.

We can generalize the construction of $countChanges$ to a *left fold* over the values of a behavior (provided we can distinguish different values by using an Eq instance):

```
foldB :: Eq a ⇒ (b → a → b) → b → B a → B (B b)
foldB f i b = loop i where
  loop :: b → B (B b)
  loop i = do c  ← b
              let i' = f i c
              e  ← change b
              e' ← snapshot (loop i') e
              return (pure i' 'switch' e')
```

The function $countChanges$ can then be more concisely expressed:

$$countChanges = foldB\ (\lambda x\ \_ \rightarrow x + 1)\ (-1)$$

Since this initial value of the input behavior does not constitute a change, the initial value passed to $foldB$ is -1.

5.2 Remembering the Past

While our interface ensures that implementations can forget all past values of all behaviors, this does *not* mean that it is impossible to remember the past. The difference is that with our interface, the past must be remembered *explicitly*, whereas with $whenJust^{\dagger}$ the past is remembered *implicitly*.

For instance, we can define a function that gives the *previous* value of a behavior. This is similar to, but not the same as, the *delay* function known from synchronous dataflow programming. The difference is that the $delay$ function delays an input until the next time step, whereas in our interface there is no notion of a next timestep, and hence $prev$ "delays" a behavior until it changes, which can be any interval of time later, or never. Remembering the past is a form of carrying state over time, and can hence be expressed using $foldB$.

$$prev :: Eq\ a \Rightarrow a \rightarrow B\ a \rightarrow B\ (B\ a)$$
$$prev\ i\ b = (fst \lll) \lll foldB\ (\lambda(\_, p)\ c \rightarrow (p, c))\ (\bot, i)\ b$$

The function given to $foldB$ takes as the first argument a tuple containing the value before the previous value and the previous value, as the second argument the current value, and gives a tuple containing the previous and current value. The behavior that $foldB$ returns then always gives a tuple of the previous and current value, of which we select the former.

As another example, consider the following function which gives the last n values of an input behavior in reverse chronological order.

$$buffer :: Eq\ a \Rightarrow Int \rightarrow B\ a \rightarrow B\ (B\ [a])$$
$$buffer\ n\ b = foldB\ (\lambda l\ e \rightarrow take\ n\ (e : l))\ [\ ]\ b$$

The argument function is immediately called with the current value of the argument b, and hence the lists in the resulting behavior are never empty (provided that $n > 0$).

5.3 Event Streams

Next to events and behaviors, another often useful abstraction is *event streams*, such as the stream of mouse-click events or the stream of incoming network messages. Naively, one might try to implement such event streams as follows:

newtype $Stream^{\dagger}\ a = S\ (E\ (a, Stream^{\dagger}\ a))$

However, this implementation gives rise to space leaks: a reference to an event stream will always point to the first element of the event stream, preventing the past to be forgotten (i.e. garbage collected).

With our interface, we can create a value that represents the present and future values in an event stream, forgetting the past values. For this, we employ the following insight: an event stream of type a is denotationally a value of type[7] $[(Time^{+}, a)]$, such that the points in time of successive elements are strictly increasing[6]. Such values can also be represented by the following type:

$$Time \rightarrow (Time^{+}, a)$$

such that, when given a time t, the function gives the time, t_e, of the first event in the list with $t < t_e$, i.e. the time of the *next* event. Using this insight, we define an event stream as follows:

---

[6] Alternatively, we could choose to make the points in time of successive elements non-decreasing instead of strictly increasing, which would allow multiple events simultaneous events in the stream. In this subsection we choose the points in time to be strictly increasing for simplicity of presentation.

newtype $Stream\ a = S\ \{\ next :: B\ (E\ a)\ \}$

The idea here is that the behavior always points to the *next* event in the event stream. As soon as the next event in the stream occurs, the behavior switches to the first event in the stream that has not occurred yet. If there is no next event in the stream, then the behavior gives $never$. To ensure that all behaviors that are an argument to the S constructor behave this way, we do not export the S constructor from the event stream module.

We can construct event streams with the following function:

$$repeatIO :: IO\ a \to Now\ (Stream\ a)$$
$$repeatIO\ m = S \lessdot\!\!\gtrdot loop\ \textbf{where}$$
$$\quad loop = \textbf{do}\ h \leftarrow async\ m$$
$$\qquad\qquad t \leftarrow plan_{Now}\ (loop \lessdot\!\!\$\ h)$$
$$\qquad\qquad return\ (pure\ h\ \text{`}switch\text{`}\ t)$$

Which executes the given I/O action repeatedly, and gives the event stream of the results. As an example, suppose have a blocking I/O function that gives the next mouse click called $nextClick$. The event stream of clicks is then given by $clicks \leftarrow repeatIO\ nextClick$.

We can sample a behavior each time an event in an event stream occurs:

$$snapshots :: B\ a \to Stream\ () \to Stream\ a$$
$$snapshots\ b\ (S\ s) = S\ \$\ \textbf{do}\ e \leftarrow s$$
$$\qquad\qquad\qquad snapshot\ b\ e$$

For example, suppose that, as before, $localCoordinates$ is a behavior that gives $Just$ the local mouse coordinates inside a rectangle when the mouse is inside that rectangle. We can sample the $Maybe$ the mouse position inside the rectangle, each time the mouse button is clicked:

$$maybeLocalClicks = snapshots\ localCoordinates\ clicks$$

We would now like to filter out the $Just$ values of this event stream. To define this function, we employ the following helper function:

$$plan :: E\ (B\ a) \to B\ (E\ a)$$
$$plan\ e = whenJust$$
$$\qquad (pure\ Nothing\ \text{`}switch\text{`}\ ((Just \lessdot\!\!\gtrdot) \lessdot\!\!\gtrdot e))$$

This is the the behavior version of $plan_{Now}$: it is similar to $snapshot$ but the behavior that is sampled is carried inside the given event, instead of as an separate argument. Armed with $plan$, we define filtering out the $Just$ values of an event stream as follows:

$$catMaybesStream :: Stream\ (Maybe\ a) \to Stream\ a$$
$$catMaybesStream\ (S\ s) = S\ loop\ \textbf{where}$$
$$\quad loop :: B\ (E\ a)$$
$$\quad loop = \textbf{do}\ e \leftarrow s$$
$$\qquad\qquad join \lessdot\!\!\gtrdot plan\ (nxt \lessdot\!\!\gtrdot e)$$
$$\quad nxt :: Maybe\ a \to B\ (E\ a)$$
$$\quad nxt\ (Just\ a) = return\ (return\ a)$$
$$\quad nxt\ Nothing = loop$$

We first obtain the next event from s, which we then plan to process with nxt. In nxt we see if the given value is $Just$. If so, we return this value in an event occurring now. Otherwise, the result should be the next event in the *rest* of the steam, which we obtain with $loop$. Since s switches as soon as an event occurs, it is already pointing to the next element. Because $nxt \lessdot\!\!\gtrdot e$ is of type $E\ (B\ (E\ a))$, which $plan$ turns to $B\ (E\ (E\ a))$, we do a $join \lessdot\!\!\gtrdot$ after $plan$ to join the resulting event.

As an example, we can obtain an event stream that tells us when the user clicks inside the rectangle by:

$$localClicks = catMaybesStream\ maybeLocalClicks$$

The expression $next\ localClicks$ then gives the next click inside the rectangle, carrying the local mouse position at that time. In the

code online, we show the definition of various other functions on event streams, such as a function that merges two event streams.

5.4 Observing Time Itself

A primitive behavior in Fran is $time$, which is defined as follows:

$$time :: B\ Time$$
$$time \stackrel{\circ}{=} \lambda t \to t$$

This introduces $Time$ as an interface-level type, for example as an alias for $Double$, whereas in our interface it is only a set which is used in the denotational semantics.

Such a function could also be added to our interface, but the functionality that $time$ provides can also be created using our I/O interface. By using I/O actions that wait for time to pass, we can create an behavior that changes often of type:

$$time :: Now\ (B\ Time)$$

The resulting behavior can then be used to do animation and approximate integration much like with the original $time$.

When $time$ is not a primitive, the clock which is used to drive animations is always explicit, i.e. instead of: $animation :: B\ Picture$, we use $animation :: B\ Time \to B\ Picture$. The timing of the animation can then be adjusted by adjusting the input behavior of $animation$. For instance, if the clock is c, then we can speed up the animation with a factor 2 by passing $(2*) \lessdot\!\!\gtrdot c$, instead of c, to $animation$. Fran supported an inherently leaky combinator called $timeTransformation$ that can be used for this purpose when the clock is implicit.

5.5 The Earliest of Two Events

Fran supported an choice operator, $(.|.)$, which gives the earliest of two events:

$$(ta, a)\ .|.\ (tb, b) \stackrel{\circ}{=} \textbf{if}\ ta \leqslant tb\ \textbf{then}\ (ta, a)\ \textbf{else}\ (tb, b)$$

We do not support this operator because it is not forgetful: we can observe the ordering on past events. Remembering the order in which past events occurred would not necessarily lead to space-leaks, but does require the implementations to remember that ordering (for example by associating time-stamps with events), which makes implementations a bit less efficient.

Often we do not want to know the earliest of two *past* events, but the earliest of two *future* events. This can be implemented using our interface, using the following helper function which converts an event to a behavior which holds $Just$ if the event occurred, and $Nothing$ otherwise:

$$occ :: E\ a \to B\ (Maybe\ a)$$
$$occ\ e = pure\ Nothing\ \text{`}switch\text{`}\ ((pure \circ Just) \lessdot\!\!\gtrdot e)$$

We can then implement a function which almost does the same as $(.|.)$:

$$first :: E\ a \to E\ a \to B\ (E\ a)$$
$$first\ l\ r = whenJust\ (occ\ r\ \text{`}switch\text{`}\ ((pure \circ Just) \lessdot\!\!\gtrdot l))$$

If only one of the events lies in the past, or both events lie in the future then the result of binding $first\ l\ r$ would be the same as $l\ .|.\ r$. However, if both events lie in the past, then the result will always be l. In the case that the ordering on two past events is required, this can be achieved by binding $first\ l\ r$ before both l and r have occurred and then manually remembering which of the two was earlier.

6. Implementation

In this section we discuss an implementation of our interface. We first discuss the implementation of events and then discuss an

optimization that makes events more efficient. We then do the same for behaviors: we first discuss their basic implementation and afterwards discuss an optimization that makes behaviors more efficient, in particular allowing us to forget the past. We then relate our implementation to the denotational semantics, which shows that these optimizations do not change the meaning of any program. Afterwards, we show how we can give the illusion of that computation takes no time. Finally, we discuss how this all comes together in the implementation of the *runNow* function and the main FRP loop.

6.1 Making Events Happen

To implement events and behaviors, we use a monad called M, which gives the runtime environment. We will elaborate on this monad later and will introduce functions for this monad as needed.

Using this M monad, events are defined by the following datatype, of which the constructor is not exported:

data $E\ a = E\ \{\ runE :: M\ (Either\ (E\ a)\ a)\}$

An event is represented by an M computation that gives *Left* a new version of the event, if the event did not occur yet, or *Right* the associated value of the event, if the event did occur. In this subsection, we assume that there is some way to convert a *primitive event*, i.e. an event that is the result of *async*, to this event type. We explain how this is done in section 6.6. Since the M computation will give the result or a new version of the event at any time, values of the E datatype tell us at any point in time whether the event has already occurred or not.

The definitions of *never* and the monad instance for events are as follows:

$never = E\ (return\ (Left\ never))$

instance *Monad* E **where**
  $return\ x = E\ (return\ (Right\ x))$
  $m \ggg' f = E\ \$$
    $runE\ m \ggg \lambda r \to$ **case** r **of**
      $Right\ x \to runE\ (f\ x)$
      $Left\ e'\ \to return\ (Left\ (e' \ggg' f))$

In this section, a prime, such as the prime in $\ggg'$, indicates that the given definition is not the final definition, but will be adopted later. We will adopt $\ggg'$ in the next subsection to introduce sharing on events.

When implementing behaviors, we also need a function that when given two events, gives an event that occurs when either of them occurs:

$minTime :: E\ x \to E\ y \to E\ ()$
$minTime\ l\ r = E\ (merge \lessgtr runE\ l \gtrless runE\ r)$ **where**
  $merge\ (Right\ \_)\ \_\ \ \ \ \ \ \ = Right\ ()$
  $merge\ \_\ \ \ \ \ \ \ \ (Right\ \_) = Right\ ()$
  $merge\ (Left\ l')\ \ (Left\ r'\ ) = Left\ (minTime\ l'\ r')$

This function is implementable *without* tagging each event with a timestamp, whereas the similar function(.|.) which we discussed in Section 5.5 is not. The reason for this is that (.|.) tell us *which* of the events occurred first, whereas $minTime$ only tells us if either of the events already occurred.

6.2 Making Efficient Events Happen

The implementation of events as presented in the previous subsection has a problem: computations on events are not *shared*. As an example, suppose we have two events a and b, carrying the value 2 and 3 respectively. Suppose furthermore that there is an expression

$e = $ **do** $x \leftarrow a$ -- step 1
          $y \leftarrow b$ -- step 2
          $return\ (x * y)$ -- step 3

The problem is that *each* invocation of $runE\ e$ will perform all 3 steps. We would like to share the outcome of each step between invocations of $runE\ e$: once step i has executed successfully, no invocation of $runE\ e$ should perform step i again.

To achieve such sharing we note that we can create a more efficient version of an event by applying the following function to the outcome of $runE$ on that event:

$unrunE :: Either\ (E\ a)\ a \to E\ a$
$unrunE\ (Left\ e)\ \ = e$
$unrunE\ (Right\ a) = pure\ a$

The resulting event is equal up to time-observation to the original event, but may be less expensive to compute. For instance, the result of $unrunE \lessgtr runE\ e$ after a has occurred, but before b has occurred, is equivalent to **do** $y \leftarrow b; return\ (2 * y)$, omitting step 1. The result of $unrunE \lessgtr runE\ e$ after both a and b have occurred is equivalent to $return\ 6$, omitting all three steps.

Using the IO monad, we can transform an event to an equivalent event that always uses the latests, simplest version of the event, by creating a mutable cell and using that to store the latest version of an event:

$memoEIO :: E\ a \to IO\ (E\ a)$
$memoEIO\ einit =$
  do $r \leftarrow newIORef\ einit$
    $return\ (usePrevE\ r)$
$usePrevE :: IORef\ (E\ a) \to E\ a$
$usePrevE\ r = E\ \$$
  do $e \leftarrow liftIO\ (readIORef\ r)$
    $res \leftarrow runE\ e$
    $liftIO\ (writeIORef\ r\ (unrunE\ res))$
    $return\ res$

Where $liftIO :: IO\ a \to M\ a$ lifts an IO action to an M action. The mutable cell always contains the latest version of the event: each time we run the event computation the mutable cell is updated to the newest version.

However, we do not want the user of the FRP library to have to manually apply $memoEIO$ using the IO monad for all events. Even though the M monad allows us to perform IO actions via $liftIO$, we cannot simply create the mutable variable in the M computation that is contained in the event: each invocation of $runE$ on the event would then create a *separate* mutable variable. Instead, we want a single mutable cell which is shared between invocations of $runE\ e$. Hence, we want to achieve the same effect as $memoEIO$, but without the need for the enclosing IO context. We achieve this by committing a heinous crime:

$memoE :: E\ a \to E\ a$
$memoE\ e = unsafePerformIO\ \$\ memoEIO\ e$

If we have a variable $x = memoE\ e$, then evaluating the value x will lead to the creation the reference. Because of regular sharing of values, x will only be evaluated one, and all invocations of $runE\ x$ will share the mutable cell. The only function on events which is available to the user of the FRP library and benefits from sharing is $\ggg$. Hence, we introduce sharing on events by redefining $\ggg$:

$m \ggg f = memoE\ (m \ggg' f)$

In this way, we obtain sharing for each event expression of the form $m \ggg f$ that the user of the FRP library writes.

The sharing of computations on events now follows regular sharing. As an example, consider two functions:

$$z\ a\ b = \mathbf{let}\ x = (*) \lessdot\!\!\gg a \circledast b\ \mathbf{in}\ (x,x)$$
$$z'\ a\ b = ((*) \lessdot\!\!\gg a \circledast b, (*) \lessdot\!\!\gg a \circledast b)$$

The denotation of these function is the same, but the z will share the result of computations between both elements of the tuple, whereas z' will not (if the compiler does not perform common subexpression elimination).

6.3 Behaving as Behaviors

A straightforward implementation of behaviors, as given in the introduction, is as an *initial* value and an *initial* switching event:

$$\mathbf{data}\ B^{\dagger}\ a = a\ \text{`Step`}\ E\ (B^{\dagger}\ a)$$

This definition does not allow us to forget the past: any reference to a behavior will prevent garbage collection of the entire history of the behavior. Instead, we use the following definition, which gives the *current* value and the *next* switching event:

$$\mathbf{data}\ B\ a = B\ \{runB :: M\ (a, E\ (B\ a))\}$$

Without the optimization we present in the next subsection, this definition also does *not* forget the past: an expression $b\ \text{`switch`}\ e$ will be represented in the same way whether e has occurred or not, effectively remembering all past values of a behavior in same way as the $Step$ construction.

With this definition of behaviors, the definition of $switch$ is as follows:

$$switch' :: B\ a \to E\ (B\ a) \to B\ a$$
$$switch'\ b\ e = B\ \$$$
$$\quad runE\ e \ggg \lambda r \to \mathbf{case}\ r\ \mathbf{of}$$
$$\qquad Right\ x \to runB\ x$$
$$\qquad Left\ e' \to \mathbf{do}\ (h,t) \leftarrow runB\ b$$
$$\qquad\qquad\qquad\qquad return\ (h, switchE\ t\ e')$$

The functions $switch'$ uses a helper function, $switchE$, which gives the next switching event and has the following type:

$$switchE :: E\ (B\ a) \to E\ (B\ a) \to E\ (B\ a)$$

When given two events l and r carrying behaviors, this function gives the earliest of the events $(\text{`switch`}r) \lessdot\!\!\gg l$ and r. We implement this function using $minTime$:

$$switchE\ l\ r = ((pure\ \bot\ \text{`switch'`}\ l)\ \text{`switch'`}\ r) \lessdot$$
$$\qquad\qquad minTime\ l\ r$$

When this event occurs, either l or r has occurred, and hence we will never encounter the undefined value: it will immediately be switched out. If l was first, the event is equal up to observation to $(\text{`switch'`}r) \lessdot\!\!\gg l$. If r occurred first, then $(pure\ \bot\ \text{`switch'`}\ l)$ will be immediately switched out, and the result is equal up to observation to r.

The monad instance for behaviors is probably easiest to understand via its $join$:

$$joinB' :: B\ (B\ a) \to B\ a$$
$$joinB'\ m = B\ \$$$
$$\quad \mathbf{do}\ (h,t) \leftarrow runB\ m$$
$$\qquad runB\ \$\ h\ \text{`switch'`}\ (joinB' \lessdot\!\!\gg t)$$

This function works as follows: we first sample the outer behavior to obtain the inner behavior and the next switching event of the outer behavior. We then act as the inner behavior until the outer behavior switches, in which case we $switch$ to the new joined behavior. We can then implement $\ggg$ using the standard construction $m \ggg f = join\ (fmap\ f\ m)$, where the functor instance for behaviors is defined straightforwardly.

To implement $whenJust$, we need some support from the environment: we need to be able to plan to execute an M computation in the future. For this we provide the following function, of which we discuss the implementation in Section 6.7:

$$plan_M :: E\ (M\ a) \to M\ (E\ a)$$

Using this function, $whenJust$ is defined as follows:

$$whenJust' :: B\ (Maybe\ a) \to B\ (E\ a)$$
$$whenJust'\ b = B\ \$$$
$$\quad \mathbf{do}\ (h,t) \leftarrow runB\ b$$
$$\qquad \mathbf{case}\ h\ \mathbf{of}$$
$$\qquad\quad Just\ x \to return\ (return\ x, whenJust' \lessdot\!\!\gg t)$$
$$\qquad\quad Nothing \to$$
$$\qquad\qquad \mathbf{do}\ en \leftarrow plan_M\ (runB \circ whenJust' \lessdot\!\!\gg t)$$
$$\qquad\qquad\quad return\ (en \ggg fst, en \ggg snd)$$

If the value of b is currently $Just$, we return an event, occurring now, containing the value from the $Just$ and state that the resulting behavior will switch when the input behavior switches. If the current value is $Nothing$, we plan to re-run $whenJust$ when the input behavior switches, on event t. This gives us an event en, of type $E\ (E\ a, E\ (B\ (E\ a)))$, which we convert to the desired type $(E\ a, E\ (B\ (E\ a)))$ by using the event monad.

6.4 Making Behaviors Behave

The implementation of behaviors developed in the previous subsection does not yet forget the past. To forget the past, i.e to not hold on to references that are no longer needed, we need to mutate the representation of $b\ \text{`switch`}\ e$ after e has happened, such that we no longer reference b. This is achieved in a similar manner as introducing sharing on events.

We can create a more efficient version of an behavior, by applying the following function to the outcome of $runB$ on that behavior:

$$unrunB :: (a, E\ (B\ a)) \to B\ a$$
$$unrunB\ (h,t) = B\ \$$$
$$\quad runE\ t \ggg \lambda x \to \mathbf{case}\ x\ \mathbf{of}$$
$$\qquad Right\ b \to runB\ b$$
$$\qquad Left\ t' \to return\ (h,t')$$

In particular, executing $unrunB \lessdot\!\!\gg (b\ \text{`switch`}\ (pure\ 1 \lessdot e))$, after e has occurred will give the behavior $pure\ 1$, which does not contain a reference to the no longer needed value b.

Hence, to forget the past, we only need to ensure that we reuse previously computed, more efficient versions of behaviors. We achieve this by defining $memoB$ in much the same way as $memoE$:

$$memoBIO :: B\ a \to IO\ (B\ a)$$
$$memoBIO\ einit =$$
$$\quad \mathbf{do}\ r \leftarrow newIORef\ einit$$
$$\qquad return\ (usePrevB\ r)$$
$$usePrevB :: IORef\ (B\ a) \to B\ a$$
$$usePrevB\ r = B\ \$$$
$$\quad \mathbf{do}\ b \leftarrow liftIO\ (readIORef\ r)$$
$$\qquad res \leftarrow runB\ b$$
$$\qquad liftIO\ (writeIORef\ r\ (unrunB\ res))$$
$$\qquad return\ res$$
$$memoB :: B\ a \to B\ a$$
$$memoB\ b = unsafePerformIO\ \$\ memoBIO\ b$$

We then apply $memoB$ to the functions which befenefit from sharing:

$$switch\ b\ e\ = memoB\ (switch'\ b\ e)$$
$$fmap\ f\ m\ = memoB\ (fmap'\ f\ m)$$
$$joinB\ b\quad = memoB\ (joinB'\ b)$$
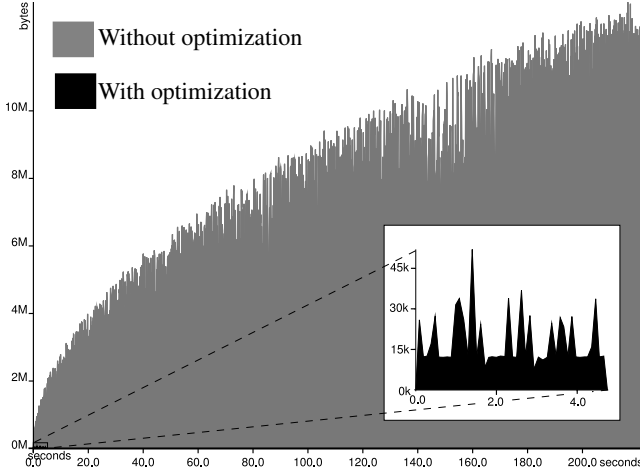$$whenJust\ b = memoB\ (whenJust'\ b)$$

Figure 4. Heap profiles with and without forgetting the past.

As a demonstration of the value of this optimization, consider a simple program using our interface:

$$test :: Int \rightarrow Now\ (E\ ())$$
$$test\ n = \mathbf{do}\ b \leftarrow count$$
$$sample\ (when\ ((n \equiv) \lessdot\!\!> b))$$

$$count :: Now\ (B\ Int)$$
$$count = loop\ 0\ \mathbf{where}$$
$$loop\ i = \mathbf{do}\ e \leftarrow async\ (return\ ())$$
$$e' \leftarrow plan_{Now}\ (loop\ (i+1) \lessdot\!\!\!s\ e)$$
$$return\ (pure\ i\ `switch`\ e')$$

Which creates a behavior that increases over time, and then observes when that behavior is equal to n. Note that executing *IO* computations with *async always* takes time, and that hence *async* (*return* ()) is *not* equal to *return* ().

Two runs of $runNow$ ($test$ 11000) are shown in Figure 4, one with the optimization enabled, and one without. The run with the optimization enabled takes a maximum of about 50k of memory, whereas the run without the optimization grows to fill about 14Mb. The run without this optimization enabled also takes much more *time*: this is because after n switches, sampling the behavior means *traversing n switches*, making the program run in $O(n^2)$, whereas the program with the optimization runs in $O(n)$. This is the reason that memory in the run without the optimization does not grow linearly in time: the amount of time each round takes also grows linearly. This optimization is essential for any practical program, in the code online we show a simple drawing program that comes to a grinding halt after a minute or so without it.

6.5 Relation to Denotational Semantics

In this subsection, we state the relation of our implementation to the denotational semantics, a proof that this relation indeed does hold is beyond the scope of this paper. To establish this relation, we need a denotation for the monad M. Like behaviors, M is a reader monad in time, but this monad does not offer *switch* or *whenJust*. Using the denotation of M as a reader monad in time, we can define functions that convert the implementations of events and behaviors to their denotations:

$$denotE\ (E\ m) \doteq$$
$$\mathbf{let}\ t \doteq minSet\ \{\ t \mid isRight\ (runReader\ m\ t)\ \}$$
$$\mathbf{in\ if}\ t \equiv \infty\ \mathbf{then}\ (\infty, \bot)$$
$$\mathbf{else}\ (t, fromRight\ (runReader\ m\ t))$$

```
data Round
data Clock
data PrimE
newClock      :: IO  Clock
spawn         :: IO a → Clock → IO (PrimE a)
curRound      :: Clock → IO Round
waitNextRound :: Clock → IO ()
observeAt     :: PrimE a → Round → Maybe a
```

Figure 5. Rounds and primitive events interface.

$$denotB\ (B\ m) \doteq \lambda t \rightarrow fst\ (runReader\ m\ t)$$

The monad M offers $plan_M$, with the following denotation:

$$plan_M\ e \doteq \lambda n \rightarrow \mathbf{let}\ (t, x) \doteq denotE\ e$$
$$t' \doteq max\ n\ t$$
$$\mathbf{in}\ (t', runReader\ x\ t')$$

The implementations of any function f from the interface presented in Figure 1 should be should be such that the denotation of a result of the implementation of f, is the equal up to time observation from any time to taking the denotations of the arguments and applying the denotational version of f. For example, for *switch* it should hold that:

$$\forall t.\ (denotB\ b)\ `switch`\ (denotE\ e) \equiv^t$$
$$denotB\ (b\ `switch`\ e)$$

Where the left hand side *switch* is the denotational version and the right hand side *switch* is the implementation version.

Furthermore, the new versions of events and behaviors given by applying $unrunE$ and $unrunB$ should be equal up to time observation to the original events and behaviors. More precisely, for events it should hold that :

$$\forall t.\quad denotE\ (unrunE\ (runReader\ (runE\ x)\ t))$$
$$\equiv^t denotE\ x$$

And for behaviors it should hold that:

$$\forall t.\quad denotB\ (unrunB\ (runReader\ (runB\ x)\ t))$$
$$\equiv^t denotB\ x$$

This also states that using $memoE = id$ and $memoB = id$ gives the same result, up to time observation, as using the efficient version of $memoE$ and $memoB$. Hence the relation of our implementation to the denotational semantics tell us that we got away with our heinous crime: our optimization does not change the results of programs.

6.6 Primitive Events

In this subsection, we discuss a mechanism to create *primitive events*, i.e. events that are the result of an I/O action. The interface we use to create and observe primitive events is shown in Figure 5. To create the illusion that actions in the *Now* monad take zero time, we divide time into periods called *rounds*. If a primitive event occurs at a time t, then it occurs in the first round i, with start time s_i, such that $t > s_i$. How time is divided into round is controlled by a *Clock*. The action *spawn* runs the given I/O action on a separate thread, and when it completes, it looks at the given clock and makes the resulting primitive event ($PrimE$) occur in the *next* round.

The function $observeAt$ gives *Just* the value of the event if the it occurred before or in the given round, and *Nothing* otherwise. It is safe to implement this as a function, instead of as an action in the *IO* monad, for two reasons:

311

- A primitive event always occurs in the *next* round. This guarantees that for any round i, the set of events that occurs before or in round i does not change after the start of round i.

- The only way to create a value of type *Round* is via the *curRound* function, and hence if we have a *Round* value for round i then it is guaranteed that round i already started.

It *is* however possible to create two clocks and to use a round obtained from one clock to observe a primitive event created using the other clock. Our implementation will then simply throw an error, observing the difference between clocks (they each have a unique identifier). It is also possible to use ST monad-like techniques to statically prevent this situation.

When all new I/O actions for the current round have been started, we can start the next round with *waitNextRound*. This function *blocks* until at least one I/O action, spawned from this clock, has occurred in the next round. The reason for blocking is that any change in an event or behavior must come from a primitive event, and hence blocking until there is at least one new primitive event saves the effort of re-sampling behaviors and events while there can be no change.

This interface is implemented by using an *MVar* containing the current round, contained in the *Clock* datatype, which is observed when an when a *spawn*ed I/O action completes. Afterwards, we set a concurrent flag, which the *waitNextRound* function blocks on before it increases the round. A bit of care is then taken to prevent race conditions.

6.7 Execute Plans, Make Plans, Rinse, Repeat

Globally, the *runNow* function first executes the initial *Now* computation given to it, which leads to starting some I/O actions and some *plans*: events of type $E\ (M\ a)$ where the $M\ a$ computation they carry should be executed as the event has occured. The main FRP loop then consists of first waiting for some new I/O action to complete, signaling the next round, after which we try to execute all plans, which can lead to starting new I/O actions and making new plans. After we have executed all plans which could be executed in this round, the loop repeats.

Before we can more precisely define the main FRP loop, we define need to define the environment, M, that we have been using in the previous subsections. The environment is defined as a reader in the clock, so that we can obtain the current round and spawn new I/O actions, and a writer in plans, so that we can implement the $plan_M$ and $plan_{Now}$ functions.

type $M = WriterT\ Plans\ (ReaderT\ Clock\ IO)$

The definition of the *Now* monad is a **newtype** wrapper around M:

newtype $Now\ a = Now\ \{\ getNow :: M\ a\ \}$
$\qquad\qquad\qquad$ **deriving** *Monad*

Armed with these definitions, we can implement *async* as follows:

$async :: IO\ a \to Now\ (E\ a)$
$async\ m = Now\ \$ \ \mathbf{do}\ c \leftarrow ask$
$\qquad\qquad\qquad\qquad toE \mathbin{<\!\!\gg\!\!>} liftIO\ (spawn\ c\ m)$

Where *toE* converts a primitive event to a regular event:

$toE :: PrimE\ a \to E\ a$
$toE\ p = E\ (toEither \circ (p`observeAt`) \mathbin{<\!\!\gg\!\!>} getRound)$
$\quad \mathbf{where}\ toEither\ Nothing = Left\ (toE\ p)$
$\qquad\qquad\ \ toEither\ (Just\ x) = Right\ x$
$\quad getRound = ask \ggg liftIO \circ curRound$

data *Ref a*
$makeStrongRef :: a \to IO\ (Ref\ a)$
$makeWeakRef\ \ :: a \to IO\ (Ref\ a)$
$deRef\ \qquad\quad :: Ref\ a \to IO\ (Maybe\ a)$

Figure 6. Unified interface to weak and strong references.

To implement $plan_M$ and $plan_{Now}$, we define a plan as an event carrying an M computation and an *IORef* telling us whether we already executed the plan, and if so, its outcome:

data $Plan\ a = Plan\ (E\ (M\ a))\ (IORef\ (Maybe\ a))$

From such a *Plan*, we can construct an event for its outcome, which upon inspection checks whether the plan has already been executed, and if not, tries to execute it now and store the results:

$planToEv :: Plan\ a \to E\ a$
$planToEv\ p@(Plan\ ev\ ref) = E\ \$$
$\quad liftIO\ (readIORef\ ref) \ggg \lambda pstate \to$
$\quad \mathbf{case}\ pstate\ \mathbf{of}$
$\qquad Just\ x\ \ \to return\ (Right\ x)$
$\qquad Nothing \to runE\ ev \ggg \lambda estate \to$
$\qquad\quad \mathbf{case}\ estate\ \mathbf{of}$
$\qquad\qquad Left\ \_\ \ \to return\ \$\ Left\ (planToEv\ p)$
$\qquad\qquad Right\ m \to \mathbf{do}\ v \leftarrow m$
$\qquad\qquad\qquad\qquad\qquad liftIO\ \$\ writeIORef\ ref\ (Just\ v)$
$\qquad\qquad\qquad\qquad\qquad return\ \$\ Right\ v$

To ensure that we execute each plan as soon as the event that it depends on occurs, the main FRP loop keeps track of all plans which have not yet executed and tries to execute them each round. However, plans made by $plan_M$ only have to be executed if some other part of the program is interested in (i.e. has a reference to) the result of the plan. The reason for this is that $plan_M$ is only used for the implementation of *whenJust*, and hence it is guaranteed that plans made by $plan_M$ do not produce any side-effects which are observable by the user of the FRP library. In contrast, plans made with $plan_{Now}$ can lead to arbitrary side-effects, such as sounding an alarm, and hence must be executed even if no other part of the program is interested in the result.

By forgetting plans that have no observable side-effects and that no part of the program is interested in, we can save time, for executing the plan, and save space, for storing the plan. As executing plans may lead to new plans, these savings can be quite significant. To save space and time, we store a *weak* reference[18] to plans made with $plan_M$, whereas we use an ordinary, strong, reference for plans made with $plan_{Now}$.

To deal with both types of plans in an uniform matter, we use the interface for references shown in Figure 6, which unifies weak and strong references . The function *deRef* returns *Nothing* if the reference was a weak reference and the value it references was garbage-collected, and *Just* the value otherwise. The sequence of plans that we still need to be executed is then represented as follows:

data $SomePlan = \forall a.\ SomePlan\ (Ref\ (Plan\ a))$
type $Plans = Seq\ SomePlan$

Where is *Seq* is a sequence datatype. The implementations of $plan_M$ and $plan_{Now}$ then use weak and strong references respectively:

$plan_M :: E\ (M\ a) \to M\ (E\ a)$
$plan_M\ e = plan\ makeWeakRef\ e$

$plan_{Now} :: E\ (Now\ a) \to Now\ (E\ a)$
$plan_{Now}\ e = Now\ \$\ plan\ makeStrongRef\ \$\ getNow \mathbin{<\!\!\gg\!\!>} e$

$plan :: (\forall x.\ x \to IO\ (Ref\ x)) \to E\ (M\ a) \to M\ (E\ a)$

```
plan makeRef e =
   do p ← Plan e ⟨⟩ liftIO (newIORef Nothing)
      pr ← liftIO (makeRef p)
      addPlan pr
      return (planToEv p)

addPlan :: Ref (Plan a) → M ()
addPlan = tell ∘ singleton ∘ SomePlan
```

We now finally arrive at the definition of *runNow*:

```
runNow :: Now (E a) → IO a
runNow (Now m) =
   do c ← newClock
      runReaderT (runWriterT m ≫ mainLoop) c
```

The *runNow* function first creates a new clock and executes the initial *Now* computation. From this it obtains a tuple containing the *ending event*, i.e. the event that breaks the FRP loop, and a sequence of plans, which we both pass to the main loop.

```
mainLoop :: (E a, Plans) → ReaderT Clock IO a
mainLoop (ev, pl) = loop pl where
   loop pli =
      do (er, ple) ← runWriterT (runE ev)
         let pl = pli ⋈ ple
         case er of
           Right x → return x
           Left _  → do endRound
                        pl' ← tryPlans pl
                        loop pl'

endRound :: ReaderT Clock IO ()
endRound = ask ≫ liftIO ∘ waitNextRound
```

Each iteration of the main loop first checks if the ending event occurred. This may lead to some new plans, *ple*, which we add to the other plans using sequence concatenation (⋈). If the ending event occurred, we break out of the loop and return the value inside the event. Otherwise, we wait for a new I/O action to complete using *waitNextRound*. We then try to run the plans, executing them if their corresponding event occurred. Executing plans gives a new sequence of plans *pl'*. We pass these plans to the next iteration of the main loop.

The *tryPlans* function tries to execute each plan which we did not execute yet to obtain the plans that should be executed in the next round. If a plan is still needed (it has not been garbage collected) and the *M* computation of the plan has not occurred yet, we add it to the sequence of plans which should be tried in the next round:

```
tryPlans :: Plans → ReaderT Clock IO Plans
tryPlans pl = snd ⟨⟩ runWriterT (mapM_ tryPlan pl)

tryPlan (SomePlan pr) =
   do ps ← liftIO (deRef pr)
      case ps of
        Just p → do eres ← runE (planToEv p)
                    case eres of
                      Right x → return ()
                      Left _ → addPlan pr
        Nothing → return ()
```

It might seem from this definition that the order of plans in the sequence matters, but this is not the case. If a plan, *a*, depends on another plan, *b*, which occurs later in the sequence then *a*, then trying plan *a* will observe the outcome event of plan *b*, which leads to trying plan *b* via *planToEv* described above. When we arrive at the position of *b* in the sequence, we will (redundantly) try *b* again.

7. Related Work

Elliott presents a modernized version of the FRP interface, that is not forgetful[7], that was the inspiration for the modernized FRP interface in Section 2. Elliott develops a push-based implementation for his interface, whereas we use a pull-based implementation which prevents needless re-computation through sharing.

In a previous paper[20], the first author presented a forgetful FRP interface without first class behaviors called *Monadic* FRP. The actions in the monads in this interface, in contrast to the monads presented in this paper, *take time*, leading to a style which can be more natural when behaviors consist of multiple phases, similar to the *task* abstraction[17].

Synchronous dataflow programming languages, such as Lustre[3], Esterel[2], and Lucid Synchrone[19], also provide a way to program reactive systems without callbacks, non-determinism and mutable state. In addition, these languages provide very strong guarantees on resource and time usage, making them an ideal candidate for programming embedded systems. As these languages focus on strong resource and time guarantees, they have never had an issue with inherently leaky abstractions, even for higher-order dataflow[4] (where dataflow networks can be sent over dataflow networks). To make these guarantees, they are more restrictive than the FRP interface presented in this paper.

In the interface presented in this paper, as well as in Fran and in Elliott's modernized FRP interface[7], there is no notion of a next time step (time does not have to enumerable). In synchronous dataflow programming, there is a notion of a next time step by the *delay* operator.

Solutions to the Space-leak Problem of Fran *Arrowized* FRP[5, 15] solves the space leak problem of Fran by disallowing behaviors (called signals in Arrowized FRP) as first class values. Instead, *signal functions*, functions from a signal to a signal, are the basic form of abstraction. These signal functions can be composed using the *Arrow* type class interface, augmented with a switching function, leading to a very different programming style than when behaviors are first class. The arrow type class, use of signal functions instead of first class behaviors, and the inclusion of *delay* signal function make Arrowized FRP instead very similar to higher-order data-flow programming.

Krishnaswami[11] presents a programming language with first class behaviors that bears many similarities to our approach. The operational semantics of his programming language erases all past values on each tick of the clock. A specialized type system ensures that no past values can be accessed, and a proof of the soundness of this type system is given, also employing Kripke logical relations. In contrast, in our approach types play no role in ensuring that old values are not accessed again. Instead, our approach ensures this by the functions which are available, which would even provide this guarantee in an untyped setting (provided that we can keep the implementation of behaviors and events abstract). Another difference is that Krishnaswami's programming language, like dataflow programming, features a delay modality. Krishnaswami's programming language allows for the definition of arbitrary temporal recursive types, i.e. types that are recursive through time, whereas we only provide behaviors and events. His programming language also ensures that all loop structures are well-founded and causal, ensuring that all programs do not get stuck in a non-productive loop. Our FRP interface ensures that behaviors are causal in the sense that their value cannot depend on the future, but does not exclude behaviors that are undefined at points in time. We do not exclude non-productive loops, as these are not excluded by our host language Haskell.

---

[7] In particular, his functions *join* on event streams and *accumE* and *accumR* are not forgetful.

Patai[16] gives a forgetful interface for higher order *stream* programming in Haskell. His interface makes a type level distinction between streams, and stream generators, i.e. streams of streams. Like our approach, his interface also ensures that old values of a stream cannot be accessed again by employing monads, but with a very different explanation, involving shifting the diagonal of a matrix.

Jeltsch[9] presents an FRP interface for Haskell that employs *era* parameters to signal types, which give a static approximation of the time when the signal is active. Rank-2 types are then used to ensure that signals which are not from the same era cannot be combined directly, instead they should be "aged" first, i.e. converted to signals which have forgotten their past. In contrast to our interface, his interface makes extensive use of advanced type system features.

I/O *in* FRP As we state in the introduction, in Arrowized FRP the I/O is organized in a manner similar to stream based I/O, which leads to several problems. Winograd-Cort, Lui and Hudak present an alternative mechanism for I/O in Arrowized FRP, which allows resources, such as files, screens and MIDI devices, to be represented as signal functions. To ensure that resources are not used in undefined ways, for example by sending two picture streams to the same screen resource, they employ a specialized version of the *Arrow* type class where each arrow type is augmented with a phantom type indicating the set of resources the arrow uses. They also discuss a special kind of resource called a *wormhole*, which allows the communication from one arrow to another without explicit routing. These techniques solve the issues with modularity and routing that were present in the standard way of doing I/O in Arrowized FRP. A limitation of their approach is that for each resource a separate phantom type must be declared statically and hence the number of resources in the program cannot change dynamically. Hence, in contrast to our approach, dynamic resources, for example for dynamically opened files or dynamically created widgets, are problematic as each resource needs a separate type and resources cannot be created from an FRP context.

The Haskell library *Reactive banana*[1] also partially solves the problems associated with stream-like I/O. In this interface, the connection to the outside world is is setup by installing *handlers* in the *IO* monad, which give input event streams and perform output, when the FRP program is initiated. Like the approach of Winograd-Cort et. al. this solves the modularity and routing issues, but because all handler must be installed *before* starting the FRP program, dealing with dynamically created resources is problematic.

Czaplicki and Chong[6] present a forgetful first order (no behaviors of behaviors) FRP language, called *Elm*, that also supports asynchronous I/O in a modular and flexible way. The main difference with their approach is that we make a clear separation between pure behaviors and the *Now* monad. In contrast, in their approach no such separation is made, each behavior may start I/O.

8. Conclusion

We have presented a new interface for FRP which resembles the original Fran interface, but whose functions are *forgetful*, which means that it is possible to implement them without a space leak. We have also introduced a new feature to FRP, namely *internalized IO*, through means of the *Now* monad.

We have also shown that the restriction to forgetful functions does not mean exclusion of interesting programs. Together with our implementation, which, as experimentally shown, exhibits the expected absence of space leaks, this provides a principled basis for practical programming of reactive systems, without callbacks, non-determinism or mutable state.

There are many interesting directions for future research and experimentation. A few things we are experimenting with are:

- A *MonadFix* instance for behaviors, with the usual denotation for reader monads, such that more recursive behaviors can be expressed.

- A *pure* interface for creating events, so that pure parts of reactive programs can be (automatically) tested.

- A cancel-able version of *async* where exceptions to the *IO* action can be thrown, for example to cancel a chess computation or to cancel reading in a large file.

- An implementation of this interface where plans are not tried every round.

References

[1] Heinrich Apfelmus. *Reactive banana*. Available at: `hackage.haskell.org/package/reactive-banana` .

[2] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of computer programming*, 19(2):87–152, 1992.

[3] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: A declarative language for real-time programming. In *Principles of Programming Languages (POPL)*, pages 178–188, 1987.

[4] Jean-Louis Colaço, Alain Girault, Grégoire Hamon, and Marc Pouzet. Towards a Higher-order Synchronous Data-flow Language. In *Conference on Embedded Software*, 2004.

[5] Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Haskell Workshop*, 2001.

[6] Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for GUIs. In *PLDI*, pages 411–422, 2013.

[7] Conal Elliott. Push-pull functional reactive programming. In *Haskell Symposium*, pages 25–36, 2009.

[8] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming (ICFP)*, 1997.

[9] Wolfgang Jeltsch. Signals, not generators! *Trends in Functional Programming*, pages 145–160, 2009.

[10] Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction*, 2001.

[11] Neelakantan R Krishnaswami. Higher-order functional reactive programming without spacetime leaks. In *International Conference on Functional Programming (ICFP)*, 2013.

[12] John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. In *Programming Language Design and Implementation (PLDI)*, pages 24–35, 1994.

[13] Simon Marlow, Ryan Newton, and Simon Peyton Jones. A monad for deterministic parallelism. In *Haskell Symposium*, pages 71–82, 2011.

[14] Conor Mcbride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, January 2008.

[15] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Haskell Workshop*, 2002.

[16] Gergely Patai. Efficient and compositional higher-order streams. In *Functional and Constraint Logic Programming (WFLP)*. 2011.

[17] John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with haskell. In *Practical Aspects of' Declarative Languages (PADL)*, 1999.

[18] Simon L. Peyton Jones, Simon Marlow, and Conal Elliott. Stretching the storage manager: Weak pointers and stable names in haskell. In *Implementation of Functional Languages (IFL)*, pages 37–58, 2000.

[19] Marc Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI, April 2006. Available at `www.lri.fr/~pouzet/lucid-synchrone`.

[20] Atze van der Ploeg. Monadic functional reactive programming. In *Haskell Symposium*, pages 117–128, 2013.

[21] Daniel Winograd-Cort, Hai Liu, and Paul Hudak. Virtualizing real-world objects in FRP. In *Practical Aspects of Declarative Languages (PADL)*, pages 227–241. 2012.