Functional Reactive Programming from First Principles

Zhanyong Wan
Yale University
Department of Computer Science
P.O. Box 208285
New Haven, Connecticut 06520
zhanyong.wan@yale.edu

ABSTRACT

Functional Reactive Programming, or *FRP*, is a general framework for programming hybrid systems in a high-level, declarative manner. The key ideas in FRP are its notions of *behaviors* and *events*. Behaviors are time-varying, reactive values, while events are time-ordered sequences of discrete-time event occurrences. FRP is the essence of *Fran*, a domain-specific language embedded in Haskell for programming reactive animations, but FRP is now also being used in vision, robotics and other control systems applications.

In this paper we explore the formal semantics of FRP and how it relates to an implementation based on *streams* that represent (and therefore only approximate) continuous behaviors. We show that, in the limit as the sampling interval goes to zero, the implementation is faithful to the formal, continuous semantics, but only when certain constraints on behaviors are observed. We explore the nature of these constraints, which vary amongst the FRP primitives. Our results show both the power and limitations of this approach to language design and implementation. As an example of a limitation, we show that streams are incapable of representing instantaneous predicate events over behaviors.

1. INTRODUCTION

How does one show that a language implementation is correct? In the programming language research community, we normally do this by showing that the implementation is faithful, in some formal sense, to an abstract denotational or operational semantics of the language. Indeed, if all goes well, we can formally *derive* the implementation from the semantics. Such is the nature of "provably correct compilation."

However, in the case of Functional Reactive Progamming (FRP), a novel language involving continuous time-varying values as well as discrete events, the situation is not as clear, and several questions arise:

PLDI 2000, Vancouver, British Columbia, Canada.

Paul Hudak
Yale University
Department of Computer Science
P.O. Box 208285
New Haven, Connecticut 06520
paul.hudak@yale.edu

1. How does one express the formal semantics of FRP? We partially answered this question in a previous paper [7], and we refine that strategy here.

2. How does one implement continuous time-varying behaviors? In this paper we explore perhaps the most obvious technique, one based on *streams* that represent sampled behaviors (in a signal processing sense). However, this representation is only an approximation to the continuous values, which leads to the next question.

3. In what sense is an approximating stream-based implementation *correct* with respect to the formal se-

mantics, and what are its limitations (for example, are there values that cannot be represented)? The interaction of these issues with the reactive component of FRP makes this especially interesting.

In this paper we provide answers to all of these questions. Specifically, we give a denotational semantics to FRP, and show that in the limit as the sampling interval goes to zero, a stream-based implementation corresponds precisely to the formal semantics, but only with suitable constraints on the nature of behaviors. The good news here is that most of the common things that we express with FRP programs are well behaved, and laws that we expect to hold in mathematics are justified, in the limit, when reasoning about FRP programs. For example, we can safely apply most FRP primitives, such as integration, to behaviors that are *discontinuous* (in a certain way to be described later), which is critically important given that the reactive component of FRP creates discontinuities quite often.

The bad news is that, since FRP is mathematically very rich, many ill behaved values can be expressed. As a result, an FRP term does not always have a meaningful semantics. In such cases we say that the term denotes $\perp$ (and is thus denotationally equivalent to non-termination or error). Of course, this is not very informative. Even worse, it is possible to write egregious behaviors for which either the implementation does not converge when we increase the sampling rate, or it converges to something other than its semantics. However, we are able to identify a set of sufficient conditions which guarantees the fidelity of the implementation. These conditions are neither complete nor decidable in general, which means the burden of "good behavior" is on the programmer. However, it is perhaps not surprising given such a rich mathematical language.

2. AN INTRODUCTION TO FRP

In this section we give a very brief introduction to FRP; see [5, 7] for more details. FRP is an example of an *embedded domain-specific language* [9]. In our case the "host" is Haskell [11], a higher-order, typed, polymorphic, lazy and purely functional language, and thus all of our examples (as well as our implementation) are in Haskell syntax.

There are two key polymorphic data types in FRP: the `Behavior` and the `Event`. A value of type `Behavior a` is a value of type `a` that varies over continuous time. Constant behaviors include numbers (such as `1 :: Behavior Real`), colors (such as `red :: Behavior Color`), and others. The most basic time-varying behavior is time itself: `time :: Behavior Time`, where `Time` is a synonym for `Real`. More interesting time-varying behaviors include animations of type `Behavior Picture` (which is the key idea behind Fran [5, 7], a language for functional reactive animations), sonar readings of type `Behavior Sonar`, velocity vectors of type `Behavior (Real,Real)`, and so on (the latter two examples are used in Frob [13, 14], an FRP-based language for controlling robots). (Note: In our implementation the type `Real` is approximated by `Float`.)

A value of type `Event a` is a time-ordered sequence of event occurrences, each carrying a value of type `a`. Basic events include left button presses and keyboard presses, represented by the values `lbp :: Event ()` and `key :: Event Char`, respectively. The declarative reading of `lbp` (and `key`) is that it is an event *sequence* containing all of the left button presses (and key presses), not just one.

Behaviors and events are both first-class values in FRP, and there is a rich set of operators (combinators) that the user can use to compose new behaviors and events from existing ones. An FRP *program* is just a set of mutually-recursive behaviors and events, each of them built up from static (non-time-varying) values and/or other behaviors and events.

Suppose that we wish to generate a color behavior which starts out as red, and changes to blue when the left mouse button is pressed. In FRP we would write:

```
> color :: Behavior Color
> color = red `until` (lbp -=> blue)
```

This can be read "behave as red until the left button is pressed, then change to blue." We can then use `color` to color an animation, as follows:

```
> ball :: Behavior Picture
> ball = paint color circ
>
> circ :: Behavior Region
> circ = translate (cos time, sin time) (circle 1)
```

Here `circle 1` creates a circle with radius 1, and the translation causes it to revolve about the center of the screen with period 2π seconds. Thus `ball` is a revolving circle that changes from red to blue when the left mouse button is pressed.

Sometimes it is desirable to choose between two different behaviors based on user input. For example, this version of `color`:

```
> color2 = red `until`
>       (lbp -=> blue) .|. (key -=> yellow)
```

will start off as red and change to blue if the left mouse button is pressed, or to yellow if a key is pressed. The `.|.` operator can be read as the "or" of its event arguments.

The function `when` transforms a Boolean behavior into an event that occurs exactly "when" the Boolean behavior becomes `True`; this is called a *predicate event.* For example:

```
> color3 = red `until`
>       (when (time >* 5) -=> blue)
```

defines a color that starts off as red and becomes blue after time is greater than 5.

Sometimes it is desirable to "lift" an ordinary value or function to an analogous behavior. The family of functions

```
> lift0 :: a -> Behavior a
> lift1 :: (a -> b) -> (Behavior a -> Behavior b)
```

and so on, perform such coercions in FRP. Sometimes Haskell overloading permits us to use the same name for lifted and unlifted functions, such as most of the arithmetic operators. When this is not possible, we use the convention of placing a "*" after the unlifted function name. For example, `>*` in the `color3` example is the lifted version of `>`.

Finally, one of the most useful operations in FRP is *integration* of numeric behaviors over time. For example, the physical equations that describe the position of a mass under the influence of an accelerating force `f` can be written as:

```
> s,v :: Behavior Real
> s = s0 + integral v
> v = v0 + integral f
```

where `s0` and `v0` are the initial position and velocity, respectively. Note the similarity of these equations to the mathematical equations describing the same physical system:

$$s(t) = s_0 + \int_0^t v(\tau)\, \mathrm{d}\tau$$
$$v(t) = v_0 + \int_0^t f(\tau)\, \mathrm{d}\tau$$

This example demonstrates well the declarative nature of FRP. A major design goal for FRP is to free the programmer from "presentation" details by providing the ability to think in terms of "modeling." It is common that an FRP program is concise enough to also serve as a specification for the problem it solves.

There are many other useful operations in FRP, but we introduce them only as needed in the remainder of the paper.

3. THE SEMANTIC FRAMEWORK

In this section we present the semantic framework for behaviors and events. The semantics of each FRP construct will be given individually in Section 6.

FRP's notion of continuous time is denoted by the domain $Time$, which is a synonym for the set of real numbers $\mathbb{R}$. Let $\langle \text{Behavior}_\alpha \rangle$ and $\langle \text{Event}_\alpha \rangle$ denote the set of all FRP terms of type `Behavior` α and `Event` α respectively, where α is any Haskell data type. The meaning of behaviors and events is given by the following semantic functions:

$$\begin{aligned} \textbf{at} &: \langle \text{Behavior}_\alpha \rangle \to Time \to Time \to \alpha \\ \textbf{occ} &: \langle \text{Event}_\alpha \rangle \to Time \to Time \to [Time \times \alpha] \end{aligned}$$

where $[-]$ is the list type constructor.

Intuitively, the meaning of a behavior, as given by **at**, is a function mapping a start time and a time of interest to the value of the behavior at the given time of interest. Start times relate to the reactive nature of FRP. For example, in (b `` `until` `` e), if an event occurrence (t, b') of e causes the overall behavior to switch to b', we say that b' starts at time t. A behavior is unaware of any event occurrences that happened before its start time.

The meaning of an event, given by **occ**, is a function that takes also a start time T and a time of interest t, and returns a finite list of time-ascending occurrences of the event in the interval $(T, t]$. The start time of an event is analogous to the start time of a behavior. Note that the lower end of the interval is open, which means an occurrence precisely at the start time is not detected.

Note that, for simplicity, we have omitted real-world events such as user input and general I/O from this semantic framework. However, predicate events such as described in the last section are still present, which are sufficient to demonstrate all interesting aspects of the semantics and implementation. Nevertheless, for completeness, we describe how to add user input in Section 8.

4. A STREAM IMPLEMENTATION OF FRP

Our stream-based implementation of FRP is interesting in its own right, but because of space limitations we omit a detailed discussion of it here; the basic idea is outlined in [6] and elaborated in [10].

The core data types in FRP, `Behavior` and `Event`, are given by:

```
> type Behavior a = [Time] -> [a]
> type Event a    = [Time] -> [Maybe a]
```

Here `Maybe a` is a data type whose values are either `Nothing` or `Just` x, where x is some `a`.

Intuitively, a behavior is a stream transformer: a function that takes an infinite stream of sample times, and yields an infinite stream of values representing its behavior. Similarly, an event is also a stream transformer, and can be thought of as a behavior where, at each time t, the event either occurs (indicated as `Just` x for some x), or does not occur (indicated as `Nothing`). Note that using this implementation strategy for events means that we must ensure that the time associated with each event occurrence actually appears in the time stream, but this is easily done.

The implementation itself can be divided into two parts: (1) definitions of FRP's primitive behaviors, events, and combinators as stream transformers, and (2) a "run-time system" that interprets the behaviors and events by building an infinite stream of sample times and applying the behavior/event to the stream. The latter task is explained in the remainder of this section; we return to the former in Section 6.

To simplify the presentation of the run-time system, we omit the interface to the operating system that extracts events, grabs the clock time, etc. The resulting abstract implementation is captured by the following pair of "interpreters," one for behaviors, the other for events:

$$\widetilde{at} \; : \; \langle \text{Behavior}_\alpha \rangle \to [Time] \to \alpha$$

$$\widetilde{at}[\![b]\!] \; ts \stackrel{\text{def}}{=} last \; (\lfloor b \rfloor \;\; ts)$$

$$\widetilde{occ} \; : \; \langle \text{Event}_\alpha \rangle \to [Time] \to [Time \times \alpha]$$

$$\widetilde{occ}[\![e]\!]\ ts \stackrel{\text{def}}{=} justValues\ ts\ (\lfloor e \rfloor\ \ ts)$$

where (1) we write $\lfloor - \rfloor$ for the value (which could be a function) denoted by the Haskell term $-$, (2) *last* returns the last element of a list, and (3) the auxiliary function:

$$justValues\ :\ [Time] \rightarrow [Maybe_\alpha] \rightarrow [Time \times \alpha]$$

time-stamps a stream of "`Maybe`" values while dropping the "`Nothing`'s." For example,

$$justValues\ [0.0, 0.1, 0.2, 0.3, 0.4]$$
$$[\texttt{Nothing}, \texttt{Just False}, \texttt{Nothing}, \texttt{Nothing}, \texttt{Just True}]$$

returns $[(0.1, \texttt{False}), (0.4, \texttt{True})]$.

Intuitively, $\widetilde{at}$ takes a behavior and an ordered finite list of sample $Time$'s, the first in the list being the start time of the behavior and the last being the time of interest. It returns as result the value of the behavior at the time of interest. Similar is $\widetilde{occ}$, which returns all the occurrences detected up until the time of interest. In essence, $\widetilde{at}$ and $\widetilde{occ}$ define an operational semantics for FRP. (Note that b is a Haskell term of type `[Time] -> [`α`]`, so $\lfloor b \rfloor$ is a function of type $[Time]$ `->` $[\alpha]$, and therefore $\lfloor b \rfloor\ \ ts$ is a value of type $[\alpha]$.)

In this paper we are most interested in the limit of the operational semantics as the sampling interval goes to zero. Thus we define:

$$\widetilde{at}^* : \langle \text{Behavior}_\alpha \rangle \to Time \to Time \to \alpha$$

$$\widetilde{at}^* [\![b]\!]\ T\ t \stackrel{\text{def}}{=} \begin{cases} \lim_{|P_T^t| \to 0} \widetilde{at}[\![b]\!]\ P_T^t & \text{such limit exists} \\ \perp & \text{otherwise} \end{cases}$$

$$\widetilde{occ}^* : \langle \text{Event}_\alpha \rangle \to Time \to Time \to [Time \times \alpha]$$

$$\widetilde{occ}^* [\![e]\!]\ T\ t \stackrel{\text{def}}{=} \begin{cases} \lim_{|P_T^t| \to 0} \widetilde{occ}[\![e]\!]\ P_T^t & \text{such limit exists} \\ \perp & \text{otherwise} \end{cases}$$

where P_T^t is a partition of $[T, t]$.

Definition 1. (Partition and norm of partition) A *partition* P of a closed interval $[a, b]$ is a non-empty finite list $[x_0, x_1, \ldots, x_n]$, such that $a = x_0 < x_1 < \cdots < x_n = b$, where $n \geq 0$. Such a partition is often written as P_a^b. The *norm* of P, written as $|P|$, is defined as the maximum of the set $\{x_i - x_{i-1} \mid 1 \leq i \leq n\}$ when $n \geq 1$, or 0 when $n = 0$.

(Note that we overload the notation $[a, b]$ for both a closed interval and a list of two elements, and similarly (a, b) for both an open interval and a tuple. However, the meaning is always clear from context.)

5. FAITHFUL IMPLEMENTATIONS AND UNIFORM CONVERGENCE

In the next section we will give both the denotational semantics and stream-based implementation of each FRP construct in turn, and show in each case that the implementation is *faithful* to the semantics, though possibly only under certain constraints, in the following formal sense:

$$\begin{aligned} \widetilde{at}^* [\![b]\!]\ T\ t &= \mathbf{at}[\![b]\!]\ T\ t \qquad (1) \\ \widetilde{occ}^* [\![e]\!]\ T\ t &= \mathbf{occ}[\![e]\!]\ T\ t \end{aligned}$$

In addition, we will identify the cases where the implementation *converges uniformly*, a property (defined below) that is necessary to ensure that the integral of a numeric behavior is well defined. Analogous to the concept of uniform convergence for real number function series [1, page 393], we define *uniform convergence* for functions defined on partitions of real intervals:

Definition 2. (Uniform Convergence) Given a set S, we say that a function F defined on $\mathcal{P}_T$, which is the set of all partitions whose left (i.e. smaller) end is T, *converges uniformly* to f on S if, for every $\epsilon > 0$, there exists a $\delta > 0$ (depending only on ϵ) such that for every $t \in S$ and P_T^t satisfying $|P_T^t| < \delta$,

$$|F(P_T^t) - f(t)| < \epsilon$$

We denote this symbolically by writing

$$F(P_T^t) \rightarrowtail f(t) \text{ uniformly on } S$$

So, when possible, we will indicate when the following condition holds:

$$\widetilde{at}[\![b]\!]\ P_T^t \rightarrowtail \mathbf{at}[\![b]\!]\ T\ t \text{ uniformly} \qquad (2)$$

Note that (2) implies (1), and for conciseness we will not write out (1) if we have already established (2).

Because FRP is an "embedded" DSL, it is difficult to draw a clear line between FRP and Haskell. Thus a full treatment of the FRP "language" inevitably requires a treatment of all of Haskell. As this would obscure our main interest, we choose to discuss only the constructs specific to FRP.

6. CORRESPONDANCE BETWEEN SEMANTICS AND IMPLEMENTATION

The proof of the important theorems in this section can be found in appendix A.

Time

The primitive behavior `time` is implemented as:

```
> time :: Behavior Time
> time = \ts -> ts
```

and its semantics is given by:

$$\mathbf{at}[\![\mathtt{time}]\!]\ T\ t = t$$

We can show that the implementation of `time` is faithful to its semantics, and that its convergence is uniform:

THEOREM 1. $\widetilde{at}[\![\texttt{time}]\!]\ P_T^t \rightarrowtail t$ *uniformly on* $(-\infty, \infty)$.

Lifting

Here we show the correctness of the three most useful lifting operators: `lift0`, `lift1` and `lift2`. The result easily extends to any arity of lifting. The lifting operators are implemented as:

```
> ($*) :: Behavior (a -> b)
>          -> Behavior a -> Behavior b
> ff $* fb =
>   \ts -> zipWith ($) (ff ts) (fb ts)
>
> lift0 :: a -> Behavior a
> lift0 x = map (const x)
>
> lift1 :: (a -> b) -> (Behavior a -> Behavior b)
> lift1 f b1 = lift0 f $* b1
>
> lift2 :: (a -> b -> c)
>           -> (Behavior a
>                  -> Behavior b -> Behavior c)
> lift2 f b1 b2 = lift1 f b1 $* b2
```

The semantics of `lift0` is given by:

$$\mathbf{at}[\![\texttt{lift0}\ \ c]\!]\ T\ t = \lfloor c \rfloor$$

Unsurprisingly, the implementation converges to the semantics uniformly:

THEOREM 2. *$\widetilde{at}[\![\texttt{lift0}\ \ c]\!]\ \ P_T^t \ \ \rightarrowtail \ \ \lfloor c \rfloor$ uniformly on $(-\infty, \infty)$.*

The semantics of `lift1` is given by:

$$\mathbf{at}[\![\texttt{lift1}\ \ f\ b]\!]\ T\ t = \lfloor f \rfloor\ \ (\mathbf{at}[\![b]\!]\ T\ t)$$

The implementation of `lift1` is faithful to its semantics, but only when the lifted function is continuous:

THEOREM 3. *If $\widetilde{at}^*[\![b]\!]\ T\ t = b_t$, and $\lfloor f \rfloor$ is continuous at b_t, then $\widetilde{at}^*[\![\texttt{lift1}\ \ f\ b]\!]\ T\ t = \lfloor f \rfloor\ \ b_t$.*

It is worth noting that we only require $\lfloor f \rfloor$ to be continuous at b_t, not necessarily continuous everywhere. Since most functions we deal with in FRP are either globally continuous or piecewise continuous, the theorem applies in most cases.

To see whether the convergence of `lift1` is uniform, we need a concept called *uniform continuity*[1, page 74], as found in most treatments of calculus:

Definition 3. (Uniform Continuity) A function f is said to be *uniformly continuous* on a set S if for every $\epsilon > 0$, there exists a $\delta > 0$ (depending only on ϵ) such that if $x, y \in S$ and $|x - y| < \delta$, then $|f(x) - f(y)| < \epsilon$.

THEOREM 4. *If* $\widetilde{at}[\![b]\!]\ P_T^t \rightarrowtail fb(t)$ *uniformly on* S, *and* $\lfloor f \rfloor$ *is uniformly continuous, then*

$$\widetilde{at}[\![\texttt{lift1}\ \ f\ b]\!]\ P_T^t \rightarrowtail \lfloor f \rfloor\ \ (fb(t))\ \ \textit{uniformly on } S$$

For example, the lifted sin function is defined as:[1]

```
> instance Floating a => Floating (Behavior a) where
> sin = lift1 sin
```

Note that the `sin` on the right hand side of the definition is the static version as in the standard Haskell library:

```
> sin :: Floating a => a -> a
```

As an example, we can use theorem 4 to prove that the expression "`sin time`" in FRP actually denotes the mathematical notion of $\sin(t)$, where t is the current time.

COROLLARY 1. *$\widetilde{at}[\![\texttt{sin time}]\!]\ P_T^t \rightarrowtail \sin t$ uniformly on $(-\infty, \infty)$.*

The semantics of `lift2` is given by:

$$\mathbf{at}[\![\texttt{lift2}\ \ f\ b\ d]\!]\ T\ t = \lfloor f \rfloor\ \ (\mathbf{at}[\![b]\!]\ T\ t)\ (\mathbf{at}[\![d]\!]\ T\ t)$$

Similar to `lift1`, we can show:

THEOREM 5. *If $\widetilde{at}^*[\![b]\!]\ T\ t = b_t$, $\widetilde{at}^*[\![d]\!]\ T\ t = d_t$, and (uncurry $\lfloor f \rfloor$) is continuous at (b_t, d_t), then*

$$\widetilde{at}^*[\![\texttt{lift2}\ \ f\ b\ d]\!]\ T\ t = \lfloor f \rfloor\ \ b_t\ d_t.$$

as well as the following for uniform convergence:

THEOREM 6. *If $\widetilde{at}[\![b]\!]\ P_T^t \rightarrowtail fb(t)$ uniformly on S, $\widetilde{at}[\![d]\!]\ P_T^t \rightarrowtail fd(t)$ uniformly on S, and (uncurry $\lfloor f \rfloor$) is uniformly continuous, then*

$$\widetilde{at}[\![\texttt{lift2}\ \ f\ b\ d]\!]\ P_T^t \rightarrowtail \lfloor f \rfloor\ \ (fb(t))\ (fd(t))$$

uniformly on S.

[1]The instance declaration shown here is how, using Haskell's type class system, functions are overloaded. In this case, `sin` is a method in the class `Floating`, and the instance declaration says that `sin` may now be used for values of type `Behavior a`, for any type `a` that is already an instance of the class `Floating`.

For example, we can use this theorem to verify the semantics of the lifted binary operator `+`:

```
> instance Num a => Num (Behavior a) where
>     (+) = lift2 (+)
```

COROLLARY 2. $\widetilde{at}^{*}[\![b \texttt{ + } d]\!]\ T\ t = \widetilde{at}^{*}[\![b]\!]\ T\ t + \widetilde{at}^{*}[\![d]\!]\ T\ t$

Integration

We use a very simply numerical algorithm to calculate the Riemann integration of numeric behaviors:

```
> integral :: Behavior Real -> Behavior Real
> integral fb =
>   \ts@(t:ts') -> 0 : loop t 0 ts' (fb ts)
>     where loop t0 acc (t1:ts) (a:as)
>              = let acc' = acc + (t1-t0)*a
>                  in acc' : loop t1 acc' evs ts as
```

The formal semantics of `integral` is given by:

$$\mathbf{at}[\![\texttt{integral}\ \ f]\!]\ T\ t = \int_T^t (\mathbf{at}[\![f]\!]\ T\ \tau)\,\mathrm{d}\tau$$

As mentioned earlier, this stream-based integrator is only sound mathematically if the behavior to be integrated converges uniformly:

THEOREM 7. *If* $\widetilde{at}[\![b]\!]\ P_T^\tau \rightarrowtail fb(\tau)$ *uniformly on* $[T,t]$, *then*

$$\widetilde{at}[\![\texttt{integral}\ \ b]\!]\ P_T^\tau \rightarrowtail \int_T^\tau fb(\eta)\,\mathrm{d}\eta$$

uniformly on $[T,t]$.

If $\widetilde{at}[\![b]\!]\ P_T^\tau \rightarrowtail fb(\tau)$ *non-uniformly*, we can say nothing about $\widetilde{at}^*[\![\texttt{integral}\ \ b]\!]\ T\ t$. As an instance, consider the behavior `bizarre` (inspired by [1, page 401]), which is non-uniformly convergent on $[0,1]$, and is defined as:

```
> bizarre :: Behavior Real
> bizarre = 0 `until` e ==> b
>   where e = when (time >* 0) `snapshot` time
>         b = \(_,t1) -> c*c*time*(1 - time)**c
>                                   where c = 1/t1
```

On time interval $[0, 1]$, the above is equivalent to:

```
> bizarre = \(t0:t1:ts) ->
>   0 : 0 : map (let c = 1/(t1 - t0)
>                  in \t -> c*c*t*(1 - t)**c) ts
```

When $t \in [0, 1]$, we have

$$
\begin{aligned}
& \widetilde{at}^{*}[\![\texttt{bizarre}]\!]\ 0\ t \\
= \ & \lim_{|P_0^t| \to 0} \mathit{last}\ \left(\lfloor \texttt{bizarre} \rfloor\ P_0^t\right) \\
= \ & \lim_{|P_0^t| \to 0} c^2 t(1-t)^c, \text{ where} \\
& \qquad 1/c = \text{the length of the first sub-interval of } P_0^t \\
= \ & 0
\end{aligned}
$$

Hence $\int_0^1 \left(\widetilde{at}^{*}[\![\texttt{bizarre}]\!]\ 0\ t\right)\ \mathrm{d}t = 0$. However, we have

$$
\begin{aligned}
& \widetilde{at}^{*}[\![\texttt{integral bizarre}]\!]\ 0\ 1 \\
= \ & \lim_{|P_0^1| \to 0} \sum_{i=1}^{n} \left(\lfloor \texttt{bizarre} \rfloor\ P_0^1\right)_{\langle i \rangle} \cdot \Delta t_i \\
& \qquad \text{where } P_0^1 = [t_0 = 0, t_1, \ldots, t_n = 1], \text{ and} \\
& \qquad \text{subscript }_{\langle i \rangle} \text{ means the } i\text{-th element of a list} \\
= \ & \lim_{|P_0^1| \to 0} \sum_{i=2}^{n} c^2 t_{i-1}(1 - t_{i-1})^c \cdot \Delta t_i, \\
& \qquad \text{where } c = 1/\Delta t_1
\end{aligned}
$$

and

$$\lim_{c\to\infty}\int_0^1 c^2 t(1-t)^c \,\mathrm{d}t \;=\; \lim_{c\to\infty}\frac{c^2}{(c+1)(c+2)} \;=\; 1$$

Therefore

$$\begin{aligned} & \widetilde{at}^{*}[\![\texttt{integral bizarre}]\!]\ 0\ 1 \\ \neq\ & \int_0^1 \left(\widetilde{at}^{*}[\![\texttt{bizarre}]\!]\ 0\ t\right)\,\mathrm{d}t \end{aligned}$$

In other words, the limit of the integral doesn't agree with the integral of the limit for `bizzare`.

It turns out that global uniform convergence is usually too strong a condition to achieve in practice, part of the reason being that the interplay of behaviors and events often results in $\perp$ at the point of behavior switching. The following theorem relaxes the requirement considerably:

THEOREM 8. *If* $\widetilde{at}[\![b]\!]\ P_T^{\tau} \rightarrowtail fb(\tau)$ *uniformly on* $[T,t]$, *except at* n *(finite) points* $\tau_1,\ \tau_2,\ \ldots,\ \tau_n$*, and* $\widetilde{at}[\![b]\!]\ P_T^{\tau_i}$ *is bounded as* $|P_T^{\tau_i}| \to 0$ *for every* $1 \leq i \leq n$*, then*

$$\widetilde{at}[\![\texttt{integral}\ b]\!]\ P_T^{\tau} \rightarrowtail \int_T^{\tau} F(\eta)\,\mathrm{d}\eta$$

uniformly on $[T,t]$*, where*

$$F(\eta) = \begin{cases} fb(\eta) & \eta \neq \tau_i for\ every 1 \leq i \leq n \end{cases}$$

$$F(\eta) = \begin{cases} \\ \textit{any finite value} & \textit{otherwise} \end{cases}$$

Since most behaviors we encounter in practice are bounded on every finite interval, the above condition is not hard to satisfy.

As shown in theorem 7, integration preserves the uniform convergence property. This allows us to safely calculate the second integral, the third, and so on:

COROLLARY 3. *If* $\widetilde{at}[\![b]\!]\ P_T^\tau \rightarrowtail fb(\tau)$ *uniformly on* $[T,t]$, *then*

$$\widetilde{at}[\![\overbrace{\texttt{integral (integral ...(integral}}^{n}\ b)\ldots)]\!]\ P_T^\tau$$
$$\rightarrowtail \int_T^\tau \int_T^{\tau_1} \ldots \int_T^{\tau_{n-1}} fb(\tau_n)\,\mathrm{d}\tau_n\,\mathrm{d}\tau_{n-1}\ \ldots\ \mathrm{d}\tau_1$$

uniformly on $[T,t]$.

Event Mapping

The `==>` operator essentially maps a function over the event stream, and is implemented as:

```
> (==>) :: Event a -> (a->b) -> Event b
> fe ==> f = map (map f) . fe
```

Its semantics is given by:

$$\mathbf{occ}[\![e \texttt{ ==> } f]\!]\ T\ t = \\ [(t_1, \lfloor f \rfloor\ \ v_1), (t_2, \lfloor f \rfloor\ \ v_2), \ldots, (t_n, \lfloor f \rfloor\ \ v_n)], \text{ where} \\ \mathbf{occ}[\![e]\!]\ T\ t = [(t_1, v_1), (t_2, v_2), \ldots, (t_n, v_n)]$$

THEOREM 9. *If* $\widetilde{occ}^*[\![e]\!]\ T\ t = [(t_1, v_1), (t_2, v_2), \ldots, (t_n, v_n)]$, *and* $\lfloor f \rfloor$ *is continuous at* v_i *for every* $1 \leq i \leq n$, *then*

$$\widetilde{occ}^*[\![e \texttt{ ==> } f]\!]\ T\ t = \\ [(t_1, \lfloor f \rfloor\ \ v_1), (t_2, \lfloor f \rfloor\ \ v_2), \ldots, (t_n, \lfloor f \rfloor\ \ v_n)]$$

The `-=>` operator we used previously is just syntactic sugar:

$$e \texttt{ -=> } b \stackrel{\text{def}}{=} e \texttt{ ==> \textbackslash\_ -> } b$$

Choice

`.|.` can be used to merge two events of the same type; it is implemented as:

```
> (.|.) :: Event a -> Event a -> Event a
> fe1 .|. fe2 =
>   \ts -> zipWith aux (fe1 ts) (fe2 ts)
>     where aux Nothing  Nothing  = Nothing
>           aux (Just x) _        = Just x
>           aux _        (Just x) = Just x
```

The semantics of `.|.` operator is given by:

If $\mathbf{occ}[\![e_1]\!]\ T\ t = [(t_1, v_1), (t_2, v_2), \ldots, (t_n, v_n)]$, $\mathbf{occ}[\![e_2]\!]\ T\ t = [(t'_1, v'_1), (t'_2, v'_2), \ldots, (t'_m, v'_m)]$, and $t_1, t_2, \ldots, t_n, t'_1, t'_2, \ldots, t'_m$ are distinct, then

$$\mathbf{occ}[\![e_1\ \ \texttt{.|.}\ \ e_2]\!]\ T\ t = [(\tau_1, w_1), (\tau_2, w_2), \ldots, (\tau_{n+m}, w_{n+m})],$$

where $ts = \{t_1, t_2, \ldots, t_n, t'_1, t'_2, \ldots, t'_m\}$, and

$$(\tau_i, w_i) = \begin{cases} (t_j, v_j) & t_j \text{ is the } i\text{-th smallest of } ts \\ (t'_k, v'_k) & t'_k \text{ is the } i\text{-th smallest of } ts \end{cases}$$

Note that we require the occurrence times are distinct, because the result of merging two simultaneous occurrences is nondeterministic.

We can show that the implementation of `(.|.)` converges to its semantics. To save space, we don't give the formal statement here.

Behavior Switching

The `until` operator is implemented as:

```
> until :: Behavior a -> Event (Behavior a)
>                          -> Behavior a
> fb `until` fe =
>   \ts -> loop ts (fe ts) (fb ts)
>     where loop ts@(_:ts') ~(e:es) (b:bs) =
```

```
>               b : case e of
>                     Nothing  -> loop ts' es bs
>                     Just fb' -> tail (fb' ts)
```

and its semantics is given by:

If $\mathbf{occ}[\![e]\!]\ T\ t = [(t_1, \lfloor b_1 \rfloor), \ldots, (t_n, \lfloor b_n \rfloor)]$, then for any $\tau \in [T, t]$,

$$\mathbf{at}[\![b \text{ `}\mathtt{until}\text{` } e]\!]\ T\ \tau = \begin{cases} \mathbf{at}[\![b]\!]\ T\ \tau & n = 0 \text{ or } \tau \leq t_1 \\ \mathbf{at}[\![b_1]\!]\ t_1\ \tau & \text{otherwise} \end{cases}$$

This operator is precisely where behaviors interact with events, and thus its good behavior is critical to the goodness of FRP. We can show:

THEOREM 10. *If $\widetilde{occ}^*[\![e]\!]\ T\ t = [(t_1, \lfloor b_1 \rfloor), \ldots, (t_n, \lfloor b_n \rfloor)]$, then for any $\tau \in [T, t]$ where $\tau \neq t_1$,*

$$\widetilde{at}^*[\![b \text{ `}\mathtt{until}\text{` } e]\!]\ T\ \tau = \begin{cases} \widetilde{at}^*[\![b]\!]\ T\ \tau & n = 0 \ \mathit{or}\ \tau < t_1 \\ \lim_{\eta \to t_1} \widetilde{at}^*[\![b_1]\!]\ \eta\ \tau & \mathit{otherwise} \end{cases}$$

Most behaviors are continuous with respect to their start times (i.e. a small change in the start time only results in a small change in the result value); for example this is true of `integral`. Some behaviors are even independent of the start time, as with `time`. In such cases, the limit operator in the above theorem can be dropped, and the implementation becomes consistent with the semantics.

Snapshot

`snapshot` samples a behavior at the exact moments an event occurs.

```
> snapshot :: Event a -> Behavior b
>                        -> Event (a,b)
> snapshot fe fb
>   = \ts -> zipWith aux (fe ts) (fb ts)
>       where aux (Just x) y = Just (x,y)
>             aux Nothing  _ = Nothing
```

The semantics:

If $\mathbf{occ}[\![e]\!]\ T\ t = [(t_1, a_1), (t_2, a_2), \ldots, (t_n, a_n)]$ and $\mathbf{at}[\![b]\!]\ T\ t_i = b_i$, then

$$\mathbf{occ}[\![e\ \text{`}\mathtt{snapshot}\text{`}\ b]\!]\ T\ t = \\ \big[\big(t_1, (a_1, b_1)\big), \big(t_2, (a_2, b_2)\big), \ldots, \big(t_n, (a_n, b_n)\big)\big]$$

The implementation is faithful to the semantics unconditionally:

THEOREM 11. *If* $\widetilde{occ}^*[\![e]\!]\ \ T\ \ t\ \ =\ \ [(t_1, a_1), (t_2, a_2), \ldots,$

$(t_n, a_n)]$ *and* $\widetilde{at}^* [\![b]\!]\ T\ t_i = b_i$, *then*

$$\widetilde{occ}^* [\![e\ \texttt{`snapshot`}\ b]\!]\ T\ t = \\ [(t_1, (a_1, b_1)), (t_2, (a_2, b_2)), \ldots, (t_n, (a_n, b_n))]$$

Predicate Events

We can turn a Boolean behavior into an event that occurs every time the behavior changes from `False` to `True`. To do so we use the `when` combinator defined as:

```
> when :: Behavior Bool -> Event ()
> when fb =
>   \ts -> zipWith up (True : bs) bs
>     where bs = fb ts
>           up False True = Just ()
>           up _     _    = Nothing
```

We define the semantics of `when` as follows: Given $T, t \in Time$, let $fb(\tau) = \textbf{at}[\![b]\!]\ T\ \tau$. If there are $c_1, c_2, \ldots, c_n \in Bool$ and a partition $[t_0, t_1, \ldots, t_n]$ of $[T, t]$, such that:

1. For all $1 \leq i \leq n - 1$, $c_i = \neg c_{i+1}$;
2. For all $1 \leq i \leq n - 1$, $\tau \in (t_{i-1}, t_i)$ implies $fb(\tau) = c_i$;
3. $fb(T) \neq \bot$ or $c_1 =$ `False`, and
4. $fb(t) \neq \bot$ or $c_n =$ `True`.

then $\mathbf{occ}[\![\texttt{when}\ \ b]\!]\ T\ t = occs \neq \bot$, where $occs$ is the shortest time-ascending list satisfying:

1. If $c_1 = \texttt{True}$ and $fb(T) = \texttt{False}$, then $(T, ()) \in occs$;
2. For $1 \leq i \leq n-1$, $(t_i, ()) \in occs$ if $c_i = \texttt{False}$, and
3. If $c_n = \texttt{False}$ and $fb(t) = \texttt{True}$, then $(t, ()) \in occs$.

Otherwise $\mathbf{occ}[\![\texttt{when}\ \ b]\!]\ T\ t = \bot$.

This may seem complicated, but it basically says that $\texttt{when}\ b$ has an occurrence at time τ iff $\mathbf{at}[\![b]\!]\ T\ \eta$ (viewed as a function of η) jumps from $\texttt{False}$ to $\texttt{True}$ as η crosses point τ from the left (i.e. the negative side).

According to this rule, if $fb(\tau)$ toggles its value back and forth instantaneously at some $\tau_0 \in (T, t)$, then $\mathbf{occ}[\![\texttt{when}\ \ b]\!]\ T\ t = \bot$. To see why this is true, suppose $\mathbf{occ}[\![\texttt{when}\ \ b]\!]\ T\ t \neq \bot$, then there must be $c_1, \ldots, c_n$ and $t_0, \ldots, t_n$ satisfying the above constraints. In addition, τ_0 must be equal to t_k for some $1 \leq k \leq n-1$, because $fb(\tau)$ remains constant in each of (t_{i-1}, t_i) where $1 \leq i \leq n$. However, this means $c_k = c_{k+1}$, which violates the constraint $c_i = \neg c_{i+1}$.

This rule implies that on any finite time interval, a predicate event can only occur a finite number of times (for the number n above is finite). Therefore, if the Boolean behavior ever oscillates at an infinite frequency (we will see such an example later), the semantics of the event is $\perp$.

The implementation of `when` b is faithful to the semantics if the implementation of b converges uniformly:

THEOREM 12. *Given a time interval* $[T, t]$, *if* $\widetilde{at}[\![b]\!]\ P_T^\tau \rightarrowtail fb(\tau)$ *uniformly on* $[T, t]$, *then*

$$\widetilde{occ}^*[\![\texttt{when}\ \ b]\!]\ T\ t = \textbf{\textit{occ}}[\![\texttt{when}\ \ b]\!]\ T\ t.$$

We require $\widetilde{at}[\![b]\!]\ P_T^\tau \rightarrowtail fb(\tau)$ *uniformly* on $[T, t]$, because the meer existence of $\widetilde{at}^*[\![b]\!]\ T\ \tau$ is not sufficient here. For example, given the behavior `bizarre` we discussed in Section 6, $\widetilde{at}^*[\![\texttt{bizarre >* 1}]\!]\ 0\ \tau = \texttt{False}$ for every $\tau \in [0, 1]$, and thus $\textbf{occ}[\![\texttt{when (bizarre >* 1)}]\!]\ 0\ 1 = []$, but

$$\widetilde{occ}^*[\![\texttt{when (bizarre >* 1)}]\!]\ 0\ 1 = [(0, ())] \neq []$$

7. EGREGIOUS BEHAVIORS AND EVENTS

As mentioned earlier, it is possible to define certain egregious behaviors and events in FRP, and a good understanding of them is helpful in understanding the semantic rules and theorems that we have introduced. Consider first this event:

```
> sharp :: Event ()
> sharp = when (time ==* 1)
```

This looks innocent enough, but the predicate is true only instantaneously at time = 1. To sample `sharp`, let's consider a series of partitions $\{P_n \mid n \in \mathbb{N}\}$ of $[0, 2]$, where $P_n = [0, \frac{1}{2n+1}, \frac{2}{2n+1}, \ldots, \frac{4n+1}{2n+1}, 2]$. Obviously $\lim_{n\to\infty} |P_n| = 0$, however none of the partitions divides $[0, 2]$ at point 1. Hence our sampling based implementation could fail to find the event occurrence at time 1. This explains why our semantic rule for `when` gives $\perp$ as the denotation of `sharp`.

It is worth pointing out that one can write a well-behaved definition for `sharp`:

```
> sharp2 = when (time >=* 1)
```

Consider next this encoding of *Zeno's Paradox*:

```
> zeno :: Event ()
> zeno = when (lift1 f time) where
>     f t = t < 2 && even (floor (log2 (2 - t)))
```

This example demonstrates an infinitely dense sequence of events at times $t_0 = 1$, $t_1 = 1.5$, $t_2 = 1.75$, and in general $t_{n+1} = t_n + 2^{-(n+1)}$. This creates obvious problems with an implementation. But even if we could implement such event sequences, there is a more fundamental semantic problem. Suppose there is an electric light in a room, initially off. A daemon comes in and turns the light on at time t_0, off at time t_1, and so on. A simple calculation shows that $\lim_{n\to\infty} t_n = 2$, so the whole process comes to an end at time $t = 2$. Now the question is: is the light on or off after the daemon stops? The answer might be surprising: it could be either on or off; i.e., this is a natural expression of nondeterminism. Our semantic rules give that $\mathbf{occ}[\![\texttt{zeno}]\!]\ T\ t = \bot$ for any $T < 2 \leq t$, which provides no information about what the implementation will give us, and therefore is compatible with the nondeterministic semantics one might expect.

Finally, consider this unpredictable behavior:

```
> unpredictable :: Behavior Real
> unpredictable =
>   0 `until` (when (time >* 1)
>                `snapshot` sin (1/(time - 1))
>                ==> (\(_,x) -> lift0 x))
```

In a stream-based implementation, we don't know what value `unpredictable` will yield at run time, since it depends on the sampling frequency and phase. For this example, the semantic rules give a value in terms of $\sin\frac{1}{t-1}$ where $t = 1$, which is undefined due to the discontinuity of the function.

`sharp`, `zeno` and `unpredictable` are all examples where the semantics is $\perp$. There are other egregious values where the semantics is not $\perp$, but the implementation does not agree with it. `integal bizarre` is one of them.

8. INTERFACE WITH REAL WORLD

As was mentioned earlier, behaviors and events can also react to user actions, which we can capture formally through the notion of an *environment*, which can be viewed as a finite set of primitive behaviors and events. Thus, strictly speaking, the semantic functions should have type:

$$\begin{array}{lcl} \textbf{at} & : & \langle \text{Behavior}_\alpha \rangle \rightarrow \mathit{Env} \rightarrow \mathit{Time} \rightarrow \mathit{Time} \rightarrow \alpha \\ \textbf{occ} & : & \langle \text{Event}_\alpha \rangle \rightarrow \mathit{Env} \rightarrow \mathit{Time} \rightarrow \mathit{Time} \rightarrow [\mathit{Time} \times \alpha] \end{array}$$

where Env is the abstract data type for all the input to the FRP system.

For example, in Fran, the environment includes mouse movements, mouse button presses, and keyboard presses. Thus we can define $\mathit{Env} = \mathit{PBeh}_{\texttt{Int}\times\texttt{Int}} \times \mathit{PEvt}_{()} \times \mathit{PEvt}_{()} \times \mathit{PEvt}_{\texttt{Char}}$, where the four components of the tuple correspond to mouse position, left button press, right button press and keyboard press, respectively. Type of the form PBeh_α and PEvt_α are defined as:

$$\mathit{PBeh}_\alpha \quad = \quad \mathit{Time} \rightarrow \alpha$$

$$PEvt_\alpha \quad = \quad Time \rightarrow [Time \times \alpha]$$

This idea can be justified by noting that primitive behaviors and events are in fact observations of physical signals outside of the FRP system. Their values at a particular time do not depend on when we start the observation. Therefore a value of type $PBeh_\alpha$ is just a mapping from the current time to the value, and a value of type $PEvt_\alpha$ is a mapping from current time to all the occurrences since the initialization of the system.

The treatment of environment is almost orthogonal to the treatment of the individual FRP operators. This allows us to address the issue separately. To extend the stripped-down version of FRP to incorporate environments, we need only to:

1. Define the semantics for the FRP constructs that represent user actions.

   For `mouse :: Behavior (Int,Int)`, we have:

   $$\mathbf{at}[\![\texttt{mouse}]\!]\ (mouse, lbp, rbp, key)\ T\ t = mouse\ t.$$

   For `lbp :: Event ()`, we have:

   $$\mathbf{occ}[\![\texttt{lbp}]\!]\ (mouse, lbp, rbp, key)\ T\ t = after\ T\ (lbp\ t),$$

   where $after\ T\ list$ drops all elements in $list$ whose time-stamp is less than or equal to T.

2. Pass the environment parameter around in the semantic equations for composite behaviors/events. For instance, the meaning for `lift2` is now given by:

$$\mathbf{at}[\![\texttt{lift2}\ f\ b_1\ b_2]\!]\ env\ T\ t = \lfloor f \rfloor\ (\mathbf{at}[\![b_1]\!]\ env\ T\ t)\ (\mathbf{at}[\![b_2]\!]\ env\ T\ t)$$

9. CONCLUSIONS AND RELATED WORK

Although the signal processing literature is full of foundational work on the validity and accuracy of sampling techniques, we are not aware of any work attempting to define the semantics of a reactive programming language such as FRP. Also, most of the signal processing work shies away from discontinuous signals, whereas we have shown that under the right conditions values are still well behaved.

In [7] we described a denotational semantics for Fran. The semantics given in this paper is different in that it parameterizes the start time for behaviors and events, and contains a more precise characterization of events. Various implementation techniques for Fran are discussed in [6], including the basic ideas behind a stream-based implementation; in [10] the particular implementation used in this paper is described in detail.

It is worth noting that we concentrated here on just one implementation technique for FRP; it may well be that other techniques either have more or fewer constraints than those discovered for streams. In particular, it is worth pointing out that *interval analysis* can be used to safely capture instantaneous predicate events [6, 7].

Finally, we point out that all of our results depend on sufficient accuracy of the underlying number system implementation. In the limit, of course, that requires an implementation of exact real arithmetic. Numerical analysis techniques are ultimately needed to ensure the stability of any system based on floating-point numbers.

CML (Concurrent ML) formalized synchronous operations as first-class, purely functional, values called "events" [15]. Our event combinators "`.|.`" and "`==>`" correspond to CML's `choose` and `wrap` functions. There are substantial differences, however, between the meaning given to "events" in these two approaches. In CML, events are ultimately used to perform an *action*, such as reading input from or writing output to a file or another process. In contrast, our events are used purely for the values they generate. These values often turn out to be behaviors, although they can also be new events, tuples, functions, etc.

Concurrent Haskell [12] contains a small set of primitives for explicit concurrency, designed around Haskell's monadic support for I/O. While this system is purely functional in the technical sense, its semantics has a strongly imperative feel. That is, expressions are evaluated without side-effects to yield concurrent, imperative computations, which are executed to perform the implied side-effects. In contrast, modeling entire behaviors as implicitly concurrent functions of continuous time yields what we consider a more declarative feel.

Several languages have been proposed around the *synchronous data-flow* notion of computation. The general-purpose functional language Lucid [16] is an example of this style of language, but more importantly are the languages Signal [8], Lustre [4], and Esterel [2, 3] which were specifically designed for control of real-time systems. In Signal, the most fundamental idea is that of a *signal*, a time-ordered sequence of values. Unlike FRP, however, time is not a value, but rather is implicit in the ordering of values in a signal. By its very nature time is thus discrete rather than continuous, with emphasis on the relative ordering of values in a data-flow-like framework. The designers of Signal have also developed a clock calculus with which one can reason about Signal programs. Lustre is a language similar to Signal, rooted again in the notion of a sequence, and owing much of its nature to Lucid.

Esterel is perhaps the most ambitious language in this class, for which compilers are available that translate Esterel programs into finite state machines or digital circuits for embedded applications. More importantly in relation to our current work, a large effort has been made to develop a formal semantics for Esterel, including a constructive behavioral semantics, a constructive operational semantics, and an electrical semantics (in the form of digital circuits). These semantics are shown to correspond in a certain way, constrained only by a notion of stability.

10. ACKNOWLEDGEMENTS

We would like to thank the PLDI referees for their insightful comments and instructive feedback. Also thanks to our funding agencies, NSF through grants CCR-9706747 and CCR-9900957, and DARPA through grant F33615-99-C-3013 administered by the AFOSR.

11. REFERENCES

[1] Tom M. Apostol. *Mathematical Analysis — A Modern Approach to Advanced Calculus*. Addison-Wesley, 1957.

[2] Gerard Berry. *The Foundations of Esterel*. MIT Press, 1998.

[3] Gerard Berry. The constructive semantics of pure esteral (draft version 3). Draft Version 3, Ecole des Mines de Paris and INRIA, July 1999.

[4] P. Caspi, N. Halbwachs, D. Pilaud, and J.A. Plaice. Lustre: A declarative language for programming synchronous systems. In *14th ACM Symp. on Principles of Programming Languages*, January 1987.

[5] Conal Elliott. Modeling interactive 3D and multimedia animation with an embedded language. In *Proceedings of the first conference on Domain-Specific Languages*, pages 285–296. USENIX, October 1997.

[6] Conal Elliott. Functional implementations of continuous modeled animation. In *Proceedings of PLILP/ALP '98*. Springer-Verlag, 1998.

[7] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 163–173, June 1997.

[8] Thierry Gautier, Paul Le Guernic, and Loic Besnard. Signal: A declarative language for synchronous programming of real-time systems. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lect Notes in Computer Science, edited by G. Goos and J. Hartmanis*, pages 257–277. Springer-Verlag, 1987.

[9] Paul Hudak. Modular domain specific languages and tools. In *Proceedings of Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society, June 1998.

[10] Paul Hudak. *The Haskell School of Expression – Learning Functional Programming through Multimedia.* Cambridge University Press, New York, 2000.

[11] Paul Hudak, Simon Peyton Jones, and Philip Wadler (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.

[12] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *ACM Symposium on the Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1996.

[13] John Peterson, Gregory Hager, and Paul Hudak. A language for declarative robotic programming. In *International Conference on Robotics and Automation*, 1999.

[14] John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with Haskell. In *First International Workshop on Practical Aspects of Declarative Languages*. SIGPLAN, Jan 1999.

[15] John H. Reppy. CML: A higher-order concurrent language. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 293–305, 1991.

[16] W.W. Wadge and E.A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press U.K., 1985.

APPENDIX

A. PROOF OF THEOREMS

We first point out the following useful property that *every* behavior and event observes:

Definition 4. (n-equality) Given an integer n and two lists l_1 and l_2, if

$$\begin{array}{l} \mathit{length}\ l_1 \geq n,\ \mathit{length}\ l_2 \geq n, \text{ and} \\ \mathit{take}\ n\ l_1 = \mathit{take}\ n\ l_2, \end{array}$$

then we say that l_1 and l_2 are *n-equal*, which we write as $l_1 \stackrel{n}{=} l_2$.

LEMMA 1. *For any $b \in \langle Behavior_\alpha \rangle$, $e \in \langle Event_\alpha \rangle$, $ts, ts' \in [Time]$, and $n \in \mathbb{N}$, $ts \stackrel{n}{=} ts'$ implies that:*

$$\begin{array}{l} \lfloor b \rfloor\ ts \stackrel{n}{=} \lfloor b \rfloor\ ts', \textit{ and} \\ \lfloor e \rfloor\ ts \stackrel{n}{=} \lfloor e \rfloor\ ts'. \end{array}$$

This Lemma essentially says that the current value of a behavior or event does not depend on the future. The proof of this lemma is an induction on the syntactic structure of an FRP expression.

LEMMA 2. $\lfloor \texttt{lift1} \rfloor\ f\ b\ ts = \mathit{map}\ f\ (b\ ts)$.

PROOF.

$$
\begin{aligned}
& \lfloor \texttt{lift1} \rfloor\ f\ b\ ts \\
= \ & (\lfloor \texttt{lift0} \rfloor\ f\ \lfloor \texttt{\$*} \rfloor\ b)\ ts \qquad \text{(definition of } \texttt{lift1}\text{)} \\
= \ & (\lambda ts.zipWith\ \lfloor \texttt{(\$)} \rfloor (\lfloor \texttt{lift0} \rfloor\ f\ ts)\ (b\ ts))\ ts \\
& \qquad\qquad \text{(definition of } \texttt{\$*}\text{)} \\
= \ & zipWith\ \lfloor \texttt{(\$)} \rfloor\ (\lfloor \texttt{lift0} \rfloor\ f\ ts)\ (b\ ts) \\
= \ & zipWith\ \lfloor \texttt{(\$)} \rfloor\ (map\ (const\ f)\ ts)\ (b\ ts) \\
& \qquad\qquad \text{(definition of } \texttt{lift0}\text{)} \\
= \ & map\ f\ (b\ ts)
\end{aligned}
$$

$\square$

LEMMA 3. $\lfloor \texttt{lift2} \rfloor\ f\ b_1\ b_2\ ts = zipWith\ f\ (b_1\ ts)\ (b_2\ ts)$.

The proof is not hard and omitted.

Theorem 1

PROOF. For any $\epsilon > 0$, let $\delta = 1$, for any P_T^t such that $|P_T^t| < \delta$, we have

$$
\begin{aligned}
& \widetilde{at}[\![\texttt{time}]\!]\ P_T^t \\
= \ & last\ \left(\lfloor \texttt{time} \rfloor\ P_T^t\right) \qquad \text{(definition of } \widetilde{at}\text{)} \\
= \ & last\ P_T^t \qquad \text{(definition of } \texttt{time}\text{)} \\
= \ & t
\end{aligned}
$$

Hence $\left|\widetilde{at}[\![\texttt{time}]\!]\ P_T^t - t\right| = 0 < \epsilon$. $\square$

Theorem 2

PROOF. For any $\epsilon > 0$, let $\delta = 1$, for any P_T^t such that $|P_T^t| < \delta$, we have

$$
\begin{array}{ll}
& \widetilde{at}[\![\texttt{lift0}\ \ c]\!]\ P_T^t \\
= & \mathit{last}\ \left(\lfloor\texttt{lift0}\ \ c\rfloor\ \ P_T^t\right) \\
= & \mathit{last}\ \left(\lfloor\texttt{lift0}\rfloor\ \ \lfloor c\rfloor\ \ P_T^t\right) \\
= & \mathit{last}\ \ \left(\mathit{map}\ (\mathit{const}\ \ \lfloor c\rfloor)\ P_T^t\right) \qquad \text{(definition of } \texttt{lift0}\text{)} \\
= & \lfloor c\rfloor
\end{array}
$$

Hence $\left|\widetilde{at}[\![\texttt{lift0}\ \ c]\!]\ P_T^t - \lfloor c\rfloor\right| = 0 < \epsilon$. $\square$

Theorem 3

PROOF.

$$
b_t \quad = \quad \lim_{\left|P_T^t\right| \to 0} \widetilde{at}[\![b]\!]\ P_T^t
$$

Thus

$$
\begin{array}{ll}
& \lfloor f\rfloor\ \ b_t \\
= & \lfloor f\rfloor\ \ \left(\displaystyle\lim_{\left|P_T^t\right| \to 0} \widetilde{at}[\![b]\!]\ P_T^t\right) \\
= & \displaystyle\lim_{\left|P_T^t\right| \to 0} \lfloor f\rfloor\ \ \left(\widetilde{at}[\![b]\!]\ P_T^t\right)
\end{array}
$$

$$
\begin{aligned}
& \qquad (\lfloor f \rfloor \text{ is continuous at } \lim_{|P_T^t| \to 0} \widetilde{at}[\![b]\!]\ P_T^t) \\
= & \lim_{|P_T^t| \to 0} \lfloor f \rfloor\ \big(last\ (\lfloor b \rfloor\ P_T^t)\big) \\
= & \lim_{|P_T^t| \to 0} last\ \big(map\ \lfloor f \rfloor\ (\lfloor b \rfloor\ P_T^t)\big) \\
= & \lim_{|P_T^t| \to 0} last\ \big(\lfloor \texttt{lift1} \rfloor\ \lfloor f \rfloor\ \lfloor b \rfloor\ P_T^t\big) \\
& \qquad \text{(lemma 2)} \\
= & \widetilde{at}^{*}[\![\texttt{lift1}\ f\ b]\!]\ T\ t
\end{aligned}
$$

$\square$

Theorem 4

PROOF. For any $\epsilon > 0$, since $\lfloor f \rfloor$ is uniformly continuous, there is an $\eta > 0$, such that for any v, v', $|v - v'| < \eta$ implies $|\lfloor f \rfloor\ v - \lfloor f \rfloor\ v'| < \epsilon$.

Since $\widetilde{at}[\![b]\!]\ P_T^t \rightarrowtail fb(t)$ uniformly on S, for the above η, there is a $\delta > 0$, such that for every $t \in S$, $|P_T^t| < \delta$ implies $\left|\widetilde{at}[\![b]\!]\ P_T^t - fb(t)\right| < \eta$.

Hence for every $t \in S$, $|P_T^t| < \delta$ implies

$$
\begin{aligned}
& \left|\widetilde{at}[\![\texttt{lift1}\ f\ b]\!]\ P_T^t - \lfloor f \rfloor\ \big(fb(t)\big)\right| \\
= & \left|\lfloor f \rfloor\ \big(\widetilde{at}[\![b]\!]\ P_T^t\big) - \lfloor f \rfloor\ \big(fb(t)\big)\right| \\
< & \ \epsilon
\end{aligned}
$$

$\square$

Corollary 1

PROOF. Since

$$\widetilde{at}[\![\texttt{time}]\!]\ P_T^t \rightarrowtail t \text{ uniformly on } (-\infty, \infty)$$

(by theorem 1), and $\lfloor \texttt{sin} \rfloor = \sin$ is uniformly continuous, we have

$$\widetilde{at}[\![\texttt{lift1 sin time}]\!]\ P_T^t \rightarrowtail \sin t$$

uniformly on $(-\infty, \infty)$ (by theorem 4).

According to the definition of lifted `sin`:

```
> instance Floating a => Floating (Behavior a) where
> sin = lift1 sin
```

we have

$$\widetilde{at}[\![\texttt{sin time}]\!]\ P_T^t \rightarrowtail \sin t \text{ uniformly on } (-\infty, \infty).$$

$\square$

Theorem 7

PROOF. For any $\tau \in [T, t]$,

$$\lim_{|P_T^\tau| \to 0} \widetilde{at}[\![\texttt{integral f}]\!]\ P_T^\tau, \quad \text{where } P_T^\tau = [t_0, t_1, \ldots, t_n].$$

$$= \lim_{|P_T^\tau| \to 0} \mathit{last}\ (\lfloor \texttt{integral}\ f \rfloor\ P_T^\tau)$$

$$= \lim_{|P_T^\tau| \to 0} \sum_{i=1}^{n} \left(\lfloor f \rfloor\ P_T^\tau\right)_{\langle i \rangle} \cdot \Delta t_i, \qquad \text{where } \Delta t_i = t_i - t_{i-1}.$$

(definition of `integral`)

$$= \lim_{|P_T^\tau| \to 0} \sum_{i=1}^{n} \left(\mathit{last}\ \left(\lfloor f \rfloor\, \widetilde{P}_i\right)\right) \cdot \Delta t_i,$$

where $\widetilde{P}_i = \mathit{take}\ i\ P_T^\tau$. (lemma 1)

$$= \lim_{|P_T^\tau| \to 0} \sum_{i=1}^{n} \left(\widetilde{at}[\![f]\!]\ \widetilde{P}_i\right) \cdot \Delta t_i$$

Furthermore,

$$\left| \lim_{|P_T^\tau| \to 0} \sum_{i=1}^{n} \left(\widetilde{at}[\![f]\!]\ \widetilde{P}_i\right) \cdot \Delta t_i - \int_T^\tau \left(\widetilde{at}^*[\![f]\!]\ T\ \eta\right)\, \mathrm{d}\eta \right|$$

$$= \left| \lim_{|P_T^\tau| \to 0} \sum_{i=1}^{n} \left(\widetilde{at}[\![f]\!]\ \widetilde{P}_i\right) \cdot \Delta t_i - \lim_{|P_T^\tau| \to 0} \sum_{i=1}^{n} \left(\widetilde{at}^*[\![f]\!]\ T\ t_i\right) \cdot \Delta t_i \right|$$

$$= \left| \lim_{|P_T^\tau| \to 0} \sum_{i=1}^{n} \left(\widetilde{at}[\![f]\!]\ \widetilde{P}_i - \widetilde{at}^*[\![f]\!]\ T\ t_i\right) \cdot \Delta t_i \right|$$

$$\leq \lim_{|P_T^\tau| \to 0} \sum_{i=1}^{n} e_i \cdot \Delta t_i,$$

$$\text{where } e_i = \left| \widetilde{at}[\![f]\!]\ \widetilde{P}_i - \widetilde{at}^*[\![f]\!]\ T\ t_i \right|.$$

Since $\widetilde{at}[\![f]\!]\ P_T^\tau \rightarrowtail \widetilde{at}^*[\![f]\!]\ T\ \tau$ uniformly converges on $[T, t]$, for every given $\epsilon > 0$, there is a $\delta > 0$, such that as long as $|P_T^\tau| < \delta$,

$$e_i < \epsilon, \text{ for every } 1 \leq i < \mathit{length}\ P_T^\tau.$$

Thus when $|P_T^\tau| < \delta$, for every $\tau \in [T, t]$ we have

$$\sum_{i=1}^{n} e_i \cdot \Delta t_i \ \leq \ \sum_{i=1}^{n} \epsilon \cdot \Delta t_i \ = \ \epsilon \cdot (\tau - T) \ \leq \ \epsilon \cdot (t - T)$$

Since ϵ is arbitrary, $\widetilde{at}[\![\texttt{integral}\ f]\!]\ P_T^\tau$ uniformly converges to $\int_T^\tau (\widetilde{at}^*[\![f]\!]\ T\ \eta)\, \mathrm{d}\eta$. $\square$

Theorem 10

PROOF. If $\widetilde{occ}^*[\![e]\!]\ T\ t = []$, then there is a $\delta > 0$ such that for every partition P_T^t of $[T, t]$ where $|P_T^t| < \delta$, $\mathit{justValues}\ P_T^t\ (\lfloor e \rfloor\ P_T^t) = []$. Therefore

$$\lim_{|P_T^t| \to 0} \mathit{last}\ \left(\lfloor b\ \text{`}\texttt{until}\text{`}\ e \rfloor\ P_T^t\right)$$

$$= \lim_{|P_T^t| \to 0} \mathit{last}\ \left(\lfloor b \rfloor\ P_T^t\right)$$

(definition of `until`)

$$= \quad \widetilde{at}^* [\![b]\!] \; T \; t$$

If $\widetilde{occ}^* [\![e]\!] \; T \; t = [(t_1, \lfloor b_1 \rfloor), \ldots, (t_m, \lfloor b_m \rfloor)]$, where $m > 0$, then for a partition $P = [\eta_0, \eta_1, \ldots, \eta_n]$ of $[T, t]$, when $|P|$ is small enough, *justValues* P $(\lfloor e \rfloor \; P)$ will have m elements. Let the first of the m elements be $(\eta_k, \lfloor b' \rfloor)$, and $\widetilde{P^k} = [\eta_k, \eta_{k+1}, \ldots, \eta_n]$, then

$$\begin{aligned}
& \lim_{|P| \to 0} \mathit{last} \; (\lfloor b \text{ `}\mathtt{until}\text{` } e \rfloor \; P) \\
= \quad & \lim_{|P| \to 0} \mathit{last} \; \left(\lfloor b' \rfloor \; \widetilde{P^k} \right) \\
& \qquad \text{(definition of } \mathtt{until}\text{)} \\
= \quad & \lim_{\eta \to t_1} \widetilde{at}^* [\![b_1]\!] \; \eta \; t
\end{aligned}$$

□