Real-Time FRP

Zhanyong Wan Walid Taha Paul Hudak
Yale University
Department of Computer Science
{wan-zhanyong,taha-walid,hudak-paul}@cs.yale.edu

ABSTRACT

Functional reactive programming (FRP) is a declarative programming language that combines a notion of continuous, time-varying behaviors with a notion of discrete, event-based reactivity. In previous work we investigated key theoretical properties of FRP but did not formally address the issue of cost. But one of our goals is to use FRP in real-time applications. In these applications, it is crucial that the cost of running programs be known before hand. Ideally, this cost should also be fixed.

To address these concerns, we present *Real-Time FRP* (RT-FRP), a statically typed language where the cost of each execution step is bounded by a constant, and the space needed for running a program is also constant. These two properties guarantee that RT-FRP can be executed in real-time and in constant space, making it a suitable model for programming real-time applications and embedded systems. The syntax of RT-FRP is developed as restricted subset of that of FRP. The exact restrictions are described using two novel concepts inspired by the notion of tail recursion. Most but not all of our existing programs fall in this restricted subset of FRP. We deal with the balance through a mechanism to integrate RT-FRP with an external language.

1. INTRODUCTION

Functional reactive programming (FRP) has proven to be a useful programming model for a diverse set of application domains, including graphics and animation [6, 8], robotics [16, 17], computer vision [20], and graphical user interfaces [22, 4]. FRP can be implemented effectively as an embedded DSL in Haskell.

In addition to our applications research, we have been studying the theoretical properties of FRP from a programming languages perspective. In previous work we defined a denotational semantics for FRP that is based on continuous real time [8]. We also defined an abstract stream-based implementation that simulates continuous time by sampling [12, 7]. In addition, we have shown that, in the limit as the sampling interval goes to zero, the stream-based implementation is faithful to the formal denotational semantics [24].

While these are encouraging theoretical results, most of the applications that we are interested in have significant performance demands, including real-time constraints. But none of our previous work addresses these concerns. For example, real-time animations must run fast enough to yield smooth motion, robots must be able to plan quickly and react swiftly to environmental events, and graphical user interfaces must respond promptly to mouse clicks and key presses. For the most part, we have found FRP to be "fast enough" for many important applications. But if it is to be used successfully in the most demanding applications, it will have to run faster and, perhaps more importantly, it will have to satisfy certain real-time constraints.

To address these shortcomings, we have designed a core variant of FRP, called *Real-Time FRP* (RT-FRP), whose operational semantics is such that each step can be carried out in constant time. Furthermore, RT-FRP has the property that term size is also bounded: it is not possible for terms to grow arbitrarily. These two properties guarantee that RT-FRP programs can be executed in real time, and in constant space: there are no space leaks (a notorious source of inefficiency in functional programs). RT-FRP is also type-preserving: run-time type errors for well-typed programs are not possible.

An interesting aspect of RT-FRP is its treatment of *recursion*, which is an important source of its expressiveness. We identify two different "styles" of recursion: one for pure signals, and one for reactivity. In order to prevent terms from growing in size, these recursions need to be constrained. We do so for pure signals by insisting that they be *well-formed* in a certain sense, and is analogous to defining recursive, but constant-space, data structures in a functional language. And we do so for reactivity by insisting that the recursions be in a *tail-recursive context*, much like tail recursion in Scheme [19].

Although RT-FRP is an interesting language design in its own right, our main interest is in compiling a higher-level version of FRP (and ultimately all of FRP) into RT-FRP. Thus we also describe how the most commonly used FRP constructs can be compiled into RT-FRP. In addition, RT-FRP can either be interpreted or further compiled into a lower-level object language such as C. An important application of this idea is *embedded systems*, where static and running code sizes, as well as run-time performance, are extremely important.

2. A BRIEF INTRODUCTION TO FRP

In this section we give a very brief introduction to FRP; see [6, 8] for more details. FRP is an example of an *embedded domain-specific language* [14, 11]. In our case the "host language" is Haskell [13], a higher-order, typed, polymorphic, lazy and purely functional language, and thus all of our examples 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 Int`), colors (such as `red :: Behavior Color`), and others. The most basic time-varying behavior is time itself: `time :: Behavior Real`. More interesting time-varying behaviors include animations of type `Behavior Picture` (the key idea behind Fran [6, 8], a language for graphics and 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 [16, 17], an FRP-based language for controlling robots).

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 mouse button presses and keyboard events, 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.

For example, suppose 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.

Furthermore, it is often useful to switch between behaviors repetitively as successive events occur. A variation of `until` called `switch` can be used for this purpose. For example, the code:

```
> color3 = red `switch`
>       (lbp -=> blue) .|. (rbp -=> yellow)
```

behaves just as `color2` for the first button press, but in addition, subsequent button presses will cause it to flip between blue and yellow, depending on which button was pressed.

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:

```
> color4 = red `until`
>       (when (time >* 5) -=> blue)
```

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

It is often 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. Haskell overloading often 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 the *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. 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. REAL-TIME FRP

FRP is a rich language. It is higher-order, thus having all of the power of the lambda calculus. In addition, one can define behaviors of behaviors, recursive behaviors, function-defining behaviors, and behavior-defining functions. Inherent in this expressiveness is the possibility of writing programs that do not terminate, that perform intractible amounts of computation for each time instant, or that have subtle and sometimes large space leaks. Of course, this is no different from using a functional (or other high-level) language in programming any application. However, the beauty of FRP is that it deals explicitly with *time*, and is thus best suited for applications that care about time. Unfortunately, such applications also often care that too much time may be taken to achieve some goal: good performance and real-time guarantees are often paramount.

For this reason we have designed a small language that we call *Real-Time FRP*, or RT-FRP, that is provably efficient in both time and space. Although expressiveness is naturally sacrificed in such an endeavor, RT-FRP sill captures the intrinsic nature of FRP, and allows the expression of the most common kinds of behaviors we have encountered in our applications work. For convenience, RT-FRP also has a mechanism to "escape" to the level of the lambda calculus, and our theorems are thus qualified with respect to the lambda terms: if they are time- and space-efficient, then the overall RT-FRP terms are also guaranteed to be time- and space-efficient.

3.1 The Syntax of RT-FRP

In what follows, x and c are the syntactic categories for variables and real numbers, respectively. To avoid confusion between FRP and RT-FRP terms, we used typewriter font in Section 2 for FRP code, and we will use a mathematical font for RT-FRP.

In FRP we distinguish between behaviors and events. In reality, the following isomorphism holds:

```
> Event a = Behavior (Maybe a)
```

where `Maybe` is Haskell's option data type, with values `Nothing` and `Just x`. Thus it is convenient in RT-FRP to combine behaviors and events into a common type that we call a *signal*. The `Maybe` type is replaced by an explicit lifting, where $\bot$ replaces `Nothing`, and $x_\bot$ replaces `Just x`.

The full syntax of RT-FRP is given first by a set of *functional terms*:

$$e \quad ::= \quad x \,|\, c \,|\, () \,|\, (e_1, e_2) \,|\, e_\bot \,|\, \bot \,|\, \lambda x.e \,|\, e_1\ e_2 \,|\, \mathsf{fix}\ x.e$$

Except for the lifting explained above, this is the conventional lambda calculus with real numbers, pairs, and a unit value. For clarity, we occasionally take the liberty of using some common syntax not provided here, such as "where" clauses and "if ... then ... else" in functional terms.

A functional term may evaluate to a *value*, defined by:

$$v \quad ::= \quad c \,|\, () \,|\, (v_1, v_2) \,|\, v_\bot \,|\, \bot \,|\, \lambda x.e$$

$$\frac{}{\Gamma, x : g \vdash_\lambda x : g}\ (\text{tp-var}) \qquad \frac{}{\Gamma \vdash_\lambda c : \mathsf{real}}\ (\text{tp-real}) \qquad \frac{}{\Gamma \vdash_\lambda () : \mathsf{unit}}\ (\text{tp-unit})$$

$$\frac{\Gamma \vdash_\lambda e_1 : g_1 \quad \Gamma \vdash_\lambda e_2 : g_2}{\Gamma \vdash_\lambda (e_1, e_2) : g_1 \times g_2}\ (\text{tp-prod}) \qquad \frac{\Gamma \vdash_\lambda e : g}{\Gamma \vdash_\lambda e_\perp : g_\perp}\ (\text{tp-lift}) \qquad \frac{}{\Gamma \vdash_\lambda \perp : g_\perp}\ (\text{tp-bot})$$

$$\frac{\Gamma, x : g_1 \vdash_\lambda e : g_2}{\Gamma \vdash_\lambda \lambda x.e : g_1 \to g_2}\ (\text{tp-fun}) \qquad \frac{\Gamma \vdash_\lambda e_1 : g_1 \to g_2 \quad \Gamma \vdash_\lambda e_2 : g_1}{\Gamma \vdash_\lambda e_1\ e_2 : g_2}\ (\text{tp-app}) \qquad \frac{\Gamma, x : g \vdash_\lambda e : g}{\Gamma \vdash_\lambda \mathsf{fix}\ x.e : g}\ (\text{tp-fix})$$

Figure 1: Typing Rules for Functional Terms

$$\frac{}{\Gamma; \Delta \vdash_S \mathsf{input} : \mathsf{input}}\ (\text{tp-input}) \qquad \frac{}{\Gamma; \Delta \vdash_S \mathsf{time} : \mathsf{real}}\ (\text{tp-time}) \qquad \frac{\Gamma \vdash_\lambda e : g}{\Gamma; \Delta \vdash_S \mathsf{ext}\ e : g}\ (\text{tp-ext})$$

$$\frac{\Gamma \vdash_\lambda v : g \quad \Gamma; \Delta \vdash_S s : g}{\Gamma; \Delta \vdash_S \mathsf{delay}\ v\ s : g}\ (\text{tp-delay}) \qquad \frac{\Gamma; \Delta \vdash_S s_1 : g_1 \quad \Gamma, x : g_1; \Delta \vdash_S s_2 : g_2}{\Gamma; \Delta \vdash_S \mathsf{let\ signal}\ x = s_1\ \mathsf{in}\ s_2 : g_2}\ (\text{tp-signal})$$

$$\frac{\Gamma; \Delta \vdash_S s_1 : g_1 \quad \Gamma; \Delta \vdash_S ev : g_{2\perp} \quad \Gamma, x : g_2; \Delta \vdash_S s_2 : g_1}{\Gamma; \Delta \vdash_S s_1\ \mathsf{switch\ on}\ x = ev\ \mathsf{in}\ s_2 : g_1}\ (\text{tp-switch})$$

Figure 2: Typing Rules for Signals

Finally, the syntax for *signals* is given by:

$$\begin{array}{rcl} s, ev & ::= & \mathsf{input} \mid \mathsf{time} \mid \mathsf{ext}\ e \mid \mathsf{delay}\ v\ s \mid \\ & & \mathsf{let\ signal}\ x = s_1\ \mathsf{in}\ s_2 \mid \\ & & s_1\ \mathsf{switch\ on}\ x = ev\ \mathsf{in}\ s_2 \end{array}$$

Here input is the generic signal for system input; time gives the current time; ext (external call) invokes the λ-calculus to evaluate a functional term; delay delays a signal by one tick; let signal binds the current value of a signal to a variable, which in turn can be used in an ext term; s_1 switch on $x = ev$ in s_2 is a signal that starts off as s_1, and switches to s_2 whenever the event ev occurs.

For simplicity, note that we only allow one branch in the switch construct. However, it is easy to extend the syntax[1] to s switch on $\{\, x_j = ev_j\ \mathsf{in}\ s_j \,\}^{j \in \{1..n\}}$, where n events are simultaneously tested, and the one that occurs first triggers the switching. It is also straightforward to extend the typing rules and operational semantics to handle this more general case.

Although RT-FRP's switching construct is less expressive than the corresponding construct in FRP, RT-FRP still has the general flavor of FRP. We will show in a later section how many FRP primitives can be defined in terms of RT-FRP; for example, `integral` can be defined in terms of delay.

[1]We use set comprehension notation $\{f_j\}^{j \in \{1..n\}}$ as shorthand for $\{f_1, f_2, \cdots, f_n\}$, where f_j is some formula parameterized over j. We omit the superscript "$j \in \{1..n\}$" when it is obvious from the context.

3.2 Static Semantics

The types used to describe RT-FRP terms have the following form:

$$g \quad ::= \quad \mathsf{input} \mid \mathsf{real} \mid \mathsf{unit} \mid g \times g \mid g_\perp \mid g \to g$$

A *variable context* Γ gives types to variables. Γ can be viewed as a set, so the order of bindings does not make a difference. For simplicity, we require all variable names in a program to be distinct. A *context* consists of a variable context Γ and a *continuation context* Δ. For now we do not actually make use of Δ, whose purpose will become clear when we extend our language to allow tail-recursive signals in section 4.2.

$$\begin{array}{lll} \text{Variable contexts} & \Gamma & ::= \ [\,] \mid \Gamma, x : g \\ \text{Contexts} & \Gamma; \Delta & \end{array}$$

We define two typing judgements: $\Gamma \vdash_\lambda e : g$, which can be read "$e$ is a functional term of type g;" and $\Gamma; \Delta \vdash_S s : g$, which can be read "$s$ is a signal that carries values of type g" (in Haskell/FRP, these would be written `e :: g` and `s :: Behavior g`, respectively). The typing rules for functional terms are essentially those of the simply-typed λ-calculus, and are shown in figure 1. The typing rules for signals are shown in figure 2, and are relatively straightforward.

3.3 Operational Semantics

In this section we give a structured operational semantics for RT-FRP [18]. Execution of an RT-FRP program is carried out in a *variable environment* $\mathcal{E}$ and a *continuation environment* $\mathcal{K}$.

Variable environments	$\mathcal{E}$	$::=$	$[\,] \mid \mathcal{E},\, x \mapsto v$
Continuation environments	$\mathcal{K}$	$::=$	$[\,] \mid \mathcal{K},\, k \mapsto (x, s)$

For now we can ignore the continuation environments, which are not needed until in section 4.2.

An RT-FRP program is executed in discrete steps, driven by a stream of time-stamped input events. Given the current time t and input i, a term s evaluates to a value v, and transitions to another term s'. We write this as $\mathcal{E} \vdash s \overset{t,i}{\hookrightarrow} v$ and $\mathcal{E}; \mathcal{K} \vdash s \overset{t,i}{\longrightarrow} s'$. For brevity, we sometimes combine these two rules into the more compact notation $\mathcal{E}; \mathcal{K} \vdash s \overset{t,i}{\hookrightarrow\!\!\!\rightarrow} s', v$. The resulting structural operational semantics for RT-FRP is shown in figures 3 and 4.

Most of these rules are straightforward. For example:

1. The instantaneous value of delay v s is just v. But it transitions to a new term delay v' s', where v' is the previous instantaneous value of s, and s' is the new term resulting from execution of s.

2. Evaluation of a let signal expression is expressed as two rules, depending on whether the bound variable is free in the body of the expression, but is otherwise straightforward. More interesting are the transition rules: one of them takes advantage of the fact that the bound variable is not free in the body, and thus abandons the binding, effectively performing a garbage collection.

3. The most complex rules are those for switch. The key point to note is that, in the case that there is no event, the transition rules do not "age" the signal that is to be ultimately switched to. In other words, that signal begins execution at the moment just after the event occurs.

4. RECURSION

As it stands now, our language has a severe limitation: although one can use fix to introduce recursion in λ-terms, there is no way to define recursive signals, because the typing rules prohibit recursive bindings. This limitation prohibits writing many interesting programs, such as having a signal react to itself. For example:

$$v = \int a\, \mathrm{d}t \text{ until } v >= 50 \text{ then } 50$$

expresses the idea that a body accelerates until the speed is 50 meter/second, and then maintains that speed. It is also common to write recursive integrations, such as:

$$v = \int (f - kv)/m\, \mathrm{d}t$$

which describes the velocity of a mass m under accelerating force f and friction kv. These are common idioms in FRP, and we need a way to express them in RT-FRP.

4.1 Recursive Signals

To solve this problem, we allow let signal bindings to be recursive. No changes in syntax are needed. Rather, we must modify the semantics by replacing the rules tp-signal and tr-signal with tp-signal' and tr-signal', respectively, which are shown in figure 5. In section 4.3 we define a well-formedness condition that must be satisfied to guarantee that the resulting recursions can still be executed in real time.

Now the two examples given above can be encoded as follows (integral is defined below, and until and when are defined in section 6):

```
let signal v = integral a until
  (when ext (v ≥ 50)) then 50
in ···
```

and

```
let signal v = integral (ext (f - k*v)/m)
in ···
```

Stateful Signals

When writing programs in RT-FRP, we often want to define signals that carry internal state. Integration is a good example of this, which is essentially keeping a running sum of the instantaneous values of a signal. Getting a handle on previous instantaneous values is the purpose of delay, and when combined with the ability to define recursive signals using let signal, we have all the tools we need to define stateful signals. The basic idea is that in:

$$\text{let signal } x = \text{delay } v\ s_1[x] \text{ in } s_2[x],$$

we use x as a state that initially has value v and is updated by $s_1[x]$. For example, the running maximum of a signal s can be expressed as:

```
let signal cur = s in
  let signal rmax = delay -∞ (ext max{rmax, cur}) in
    ext rmax
```

rmax is $-\infty$ at tick 0. At tick $n+1$, it is updated to be the larger one of its previous value and the value of s at tick n. Therefore rmax records the maximum value of s up to the previous tick.

$$\frac{}{\mathcal{E} \vdash \mathsf{input} \overset{t,i}{\hookrightarrow} i}\ (\text{ev-input}) \qquad \frac{}{\mathcal{E} \vdash \mathsf{time} \overset{t,i}{\hookrightarrow} t}\ (\text{ev-time}) \qquad \frac{\mathcal{E} \vdash e \hookrightarrow v}{\mathcal{E} \vdash \mathsf{ext}\ e \overset{t,i}{\hookrightarrow} v}\ (\text{ev-ext})$$

$$\frac{}{\mathcal{E} \vdash \mathsf{delay}\ v\ s \overset{t,i}{\hookrightarrow} v}\ (\text{ev-delay}) \qquad \frac{x \notin FV(s_2) \quad \mathcal{E} \vdash s_2 \overset{t,i}{\hookrightarrow} v_2}{\mathcal{E} \vdash \mathsf{let\ signal}\ x = s_1\ \mathsf{in}\ s_2 \overset{t,i}{\hookrightarrow} v_2}\ (\text{ev-signal-gc})$$

$$\frac{x \in FV(s_2) \quad \mathcal{E} \vdash s_1 \overset{t,i}{\hookrightarrow} v_1 \quad \mathcal{E}, x \mapsto v_1 \vdash s_2 \overset{t,i}{\hookrightarrow} v_2}{\mathcal{E} \vdash \mathsf{let\ signal}\ x = s_1\ \mathsf{in}\ s_2 \overset{t,i}{\hookrightarrow} v_2}\ (\text{ev-signal}) \qquad \frac{\mathcal{E} \vdash s_1 \overset{t,i}{\hookrightarrow} v}{\mathcal{E} \vdash s_1\ \mathsf{switch\ on}\ x = ev\ \mathsf{in}\ s_2 \overset{t,i}{\hookrightarrow} v}\ (\text{ev-switch})$$

Figure 3: Operational Semantics for Evaluation

$$\frac{}{\mathcal{E}; \mathcal{K} \vdash \mathsf{input} \overset{t,i}{\longrightarrow} \mathsf{input}}\ (\text{tr-input}) \qquad \frac{}{\mathcal{E}; \mathcal{K} \vdash \mathsf{time} \overset{t,i}{\longrightarrow} \mathsf{time}}\ (\text{tr-time}) \qquad \frac{}{\mathcal{E}; \mathcal{K} \vdash \mathsf{ext}\ e \overset{t,i}{\longrightarrow} \mathsf{ext}\ e}\ (\text{tr-ext})$$

$$\frac{\mathcal{E}; \mathcal{K} \vdash s \overset{t,i}{\hookrightarrow\!\!\!\rightarrow} s', v'}{\mathcal{E}; \mathcal{K} \vdash \mathsf{delay}\ v\ s \overset{t,i}{\longrightarrow} \mathsf{delay}\ v'\ s'}\ (\text{tr-delay}) \qquad \frac{x \notin FV(s_2) \quad \mathcal{E}; \mathcal{K} \vdash s_2 \overset{t,i}{\longrightarrow} s_2'}{\mathcal{E}; \mathcal{K} \vdash \mathsf{let\ signal}\ x = s_1\ \mathsf{in}\ s_2 \overset{t,i}{\longrightarrow} s_2'}\ (\text{tr-signal-gc})$$

$$\frac{x \in FV(s_2) \quad \mathcal{E} \vdash s_1 \overset{t,i}{\hookrightarrow} v_1 \quad \mathcal{E}; \mathcal{K} \vdash s_1 \overset{t,i}{\longrightarrow} s_1' \quad \mathcal{E}, x \mapsto v_1; \mathcal{K} \vdash s_2 \overset{t,i}{\longrightarrow} s_2'}{\mathcal{E}; \mathcal{K} \vdash \mathsf{let\ signal}\ x = s_1\ \mathsf{in}\ s_2 \overset{t,i}{\longrightarrow} \mathsf{let\ signal}\ x = s_1'\ \mathsf{in}\ s_2'}\ (\text{tr-signal})$$

$$\frac{\mathcal{E}; \mathcal{K} \vdash s_1 \overset{t,i}{\longrightarrow} s_1' \quad \mathcal{E}; \mathcal{K} \vdash ev \overset{t,i}{\hookrightarrow\!\!\!\rightarrow} ev', \bot}{\mathcal{E}; \mathcal{K} \vdash s_1\ \mathsf{switch\ on}\ x = ev\ \mathsf{in}\ s_2 \overset{t,i}{\longrightarrow} s_1'\ \mathsf{switch\ on}\ x = ev'\ \mathsf{in}\ s_2}\ (\text{tr-switch-noc})$$

$$\frac{\mathcal{E}; \mathcal{K} \vdash ev \overset{t,i}{\hookrightarrow\!\!\!\rightarrow} ev', v_\bot}{\mathcal{E}; \mathcal{K} \vdash s_1\ \mathsf{switch\ on}\ x = ev\ \mathsf{in}\ s_2 \overset{t,i}{\longrightarrow} s_2[v/x]\ \mathsf{switch\ on}\ x = ev\ \mathsf{in}\ s_2}\ (\text{tr-switch-occ})$$

Figure 4: Operational Semantics for Transitions

$$\frac{\Gamma, x : g_1; \Delta \vdash_{\mathsf{S}} s_1 : g_1 \quad \Gamma, x : g_1; \Delta \vdash_{\mathsf{S}} s_2 : g_2}{\Gamma; \Delta \vdash_{\mathsf{S}} \mathsf{let\ signal}\ x = s_1\ \mathsf{in}\ s_2 : g_2}\ (\text{tp-signal'})$$

$$\frac{x \in FV(s_2) \quad \mathcal{E} \vdash s_1 \overset{t,i}{\hookrightarrow} v_1 \quad \mathcal{E}, x \mapsto v_1 \vdash s_1 \overset{t,i}{\longrightarrow} s_1' \quad \mathcal{E}, x \mapsto v_1 \vdash s_2 \overset{t,i}{\longrightarrow} s_2'}{\mathcal{E}; \mathcal{K} \vdash \mathsf{let\ signal}\ x = s_1\ \mathsf{in}\ s_2 \overset{t,i}{\longrightarrow} \mathsf{let\ signal}\ x = s_1'\ \mathsf{in}\ s_2'}\ (\text{tr-signal'})$$

Figure 5: Typing and Semantic Rules for Recursive let signal

By the same principle, numerical integration using the forward Euler method can be defined as follows:

$$
\begin{aligned}
& \mathsf{integral}\ s \\
\equiv\ & \mathsf{let\ signal}\ t = \mathsf{time\ in} \\
& \quad \mathsf{let\ signal}\ v = s\ \mathsf{in} \\
& \quad\quad \mathsf{let\ signal}\ st = \mathsf{delay}\ (0,(0,0)) \\
& \quad\quad\quad (\mathsf{ext}\ ((i,(v,t))\ \text{where} \\
& \quad\quad\quad\quad (i_0,(v_0,t_0)) = st \\
& \quad\quad\quad\quad i = i_0 + v_0(t - t_0)) \\
& \quad\quad \mathsf{in\ ext}\ (\text{fst}\ st)
\end{aligned}
$$

4.2 Tail Signals

Another very important kind of recursion arises implicitly in switch expressions, which can be used to express repeating signals. For example:

integral (ext 1) switch on x = ev in integral (ext 1)

is a signal that increases at a constant rate, and is reset everytime event ev occurs.

In fact this simple repetition is a special case of *hybrid automata* [10, 15]. A hybrid automaton is a commonly used formal model for a hybrid system, and consists of a finite number of *control modes*. Discrete events trigger the system to jump from one mode to another. Within one mode, the system state changes continuously.

A repeating signal can be viewed as a hybrid automaton with only one mode, where a jump leads back to the same mode. A more interesting example is a thermostat, which has two modes. In the *On* mode, the heater is on, and the temperature rises according to some flow condition. When the jump condition "temperature $\geq T_{high}$" is met, it switches to the *Off* mode, and the temperature gradually drops. The thermostat jumps back to the *On* mode again when "temperature $\leq T_{low}$" becomes true.

Although switch is adequate for expressing repetition, it is not suitable for encoding more general hybrid automata, especially when the number of modes is not small and the jump conditions are non-trivial. The difficulty mainly lies in that the switching event is fixed in a switch-expression while an automaton often requires different jump conditions for different modes.

Indeed, it has been noted previously that another kind of recursive signal is needed to solve this problem, which is different from let signal [5]. We can achieve this in FRP simply by defining mutually recursive signals that switch into each other, which gives us enough expressive power to encode any hybrid automaton. Unfortunately, this solution is *too* general for RT-FRP, since resource bounds cannot be guaranteed for programs written in this style. To see why, let's look at this snippet of code:

```
let b = b0 `until` ev ==> f b
in  b
```

The behavior b is initially b0, and after ev has occurred n times, it becomes $\mathtt{f}^n$ b0. Hence with each occurrence of ev, the amount of computation in one tick builds up and eventually exceeds any fixed bound.

Our solution is to add two constructs to RT-FRP for writing mutually switchable signals, and impose some syntactic constraints such that resource bounds can still be guaranteed. We call signals in this style *tail signals*, because the syntax for such signals is analogous to that for tail recursion. With tail signals, any hybrid automaton is easily defined.

The two constructs we add are let continue and until:

$$
\begin{aligned}
s, ev \quad ::= \quad & \cdots \mid \\
& \mathsf{let\ continue}\ \{\, k_j\ x_j \ = \ s_j \,\}\ \mathsf{in}\ s \mid \\
& s\ \mathsf{until}\ \{\, ev_j \ \Rightarrow\ k_j \,\}
\end{aligned}
$$

where k is the syntactic category for *continuations*, and is drawn from a special set of identifiers.

let continue defines a group of mutually recursive continuations. A continuation is simply a name of a signal parameterized by a variable. If we think of continuations as control modes, and events in until as jump conditions, then let continue defines a hybrid automaton.

The syntactic form of until determines that a continuation can invoke only one continuation (in other words, the expression following a "$\Rightarrow$" can only be a continuation), and therefore the computation will not grow unbounded.

Recall the definition for continuation contexts and continuation environments:

Continuation contexts	Δ	$::=$	$[] \mid \Delta,\, k : g$
Continuation environments	$\mathcal{K}$	$::=$	$[] \mid \mathcal{K},\, k \mapsto (x, s)$

Note that in Δ, the type g we assign to a continuation k is the type of the argument the continuation is expecting, not the type of the signal the continuation is producing. The environment $\mathcal{K}$ maps a continuation to its definition, including the formal argument and the signal parameterized by that argument.

The typing rules for these new constructs can be found in figure 6, and their operational semantics is given in figure 7.

In addition to tail signals, let continue also gives us a way to write signals parameterized by a variable. This allows the same expression to be reused with different values of the parameter.

$$\frac{\Delta' = \{k_j : g_j\} \quad \{\,\Gamma, x_j : g_j;\, \Delta' \vdash_S s_j : g\,\} \quad \Gamma;\, \Delta' \vdash_S s : g}{\Gamma;\, \Delta \vdash_S \text{let continue } \{\, k_j\ x_j \;=\; s_j \,\} \text{ in } s : g} \;(\text{tp-continue})$$

$$\frac{\Gamma;\, [] \vdash_S s : g \quad \{\,\Gamma;\, [] \vdash_S ev_j : g_{j\perp} \quad \Delta(k_j) = g_j\,\}}{\Gamma;\, \Delta \vdash_S s \text{ until } \{\, ev_j \;\Rightarrow\; k_j \,\} : g} \;(\text{tp-until})$$

Figure 6: Typing Rules for let continue and until

$$\frac{\mathcal{E} \vdash s \overset{t,i}{\hookrightarrow} v}{\mathcal{E} \vdash \text{let continue } \{\, k_j\ x_j \;=\; s_j \,\} \text{ in } s \overset{t,i}{\hookrightarrow} v} \;(\text{ev-continue}) \qquad \frac{\mathcal{E} \vdash s \overset{t,i}{\hookrightarrow} v}{\mathcal{E} \vdash s \text{ until } \{\, ev_j \;\Rightarrow\; k_j \,\} \overset{t,i}{\hookrightarrow} v} \;(\text{ev-until})$$

$$\frac{\mathcal{K}' = \{k_j \mapsto (x_j, s_j)\} \quad \mathcal{E};\, \mathcal{K}' \vdash s \overset{t,i}{\longrightarrow} s'}{\mathcal{E};\, \mathcal{K} \vdash \text{let continue } \{\, k_j\ x_j \;=\; s_j \,\} \text{ in } s \overset{t,i}{\longrightarrow} \text{let continue } \{\, k_j\ x_j \;=\; s_j \,\} \text{ in } s'} \;(\text{tr-continue})$$

$$\frac{\mathcal{E};\, \mathcal{K} \vdash s \overset{t,i}{\longrightarrow} s' \quad \{\, \mathcal{E};\, \mathcal{K} \vdash ev_j \overset{t,i}{\hookrightarrow\!\!\rightarrow} ev_j', \perp \,\}}{\mathcal{E};\, \mathcal{K} \vdash s \text{ until } \{\, ev_j \;\Rightarrow\; k_j \,\} \overset{t,i}{\longrightarrow} s' \text{ until } \{\, ev_j' \;\Rightarrow\; k_j \,\}} \;(\text{tr-until-noc})$$

$$\frac{\{\, \mathcal{E} \vdash ev_j \overset{t,i}{\hookrightarrow} \perp \,\}^{j \in \{1..m-1\}} \quad \mathcal{E} \vdash ev_m \overset{t,i}{\hookrightarrow} v_\perp \quad \{\, \mathcal{E};\, \mathcal{K} \vdash ev_j \overset{t,i}{\longrightarrow} ev_j' \,\} \quad \mathcal{K}(k_m) = (x, s')}{\mathcal{E};\, \mathcal{K} \vdash s \text{ until } \{\, ev_j \;\Rightarrow\; k_j \,\} \overset{t,i}{\longrightarrow} s'[v/x]} \;(\text{tr-until-occ})$$

Figure 7: Operational Semantics for let continue and until

4.3 Well-Formed Recursions

Not every well-typed program in RT-FRP is meaningful. For example,

let signal x = ext x in ext x

does not uniquely define a signal. The execution of this program immediately gets stuck.

Since the goal of RT-FRP is to guarantee resource bounds, we would like to be able to detect such ill-founded recursions before execution. We do this by capturing well-formed recursions in a grammar.

Given an RT-FRP program, we require it to be well-typed. In addition, for each expression of the form:

let signal $x = s_1$ in s_2

we require that $s_1 \in W_{\{x\}}$, where W_X (X is a set of variables and continuations) is defined by the grammar (assuming all variables and continuations are distinct):

$$\begin{aligned} W_X \quad ::= \quad & \text{input} \,|\, \text{time} \,|\, \text{ext } e, \text{ where } X \cap FV(e) = \emptyset \,| \\ & \text{delay } v\ s \,|\, \text{let signal } x = W_X \text{ in } W_X \,| \\ & W_X \text{ switch on } x = ev \text{ in } W_X \,| \\ & \text{let continue } \{\, k_j\ x_j \;=\; W_{X\,j} \,\} \text{ in } W_X \,| \\ & \text{let continue } \{\, k_j\ x_j \;=\; s_j \,\} \text{ in } W_{X \cup \{k_j\}} \,| \\ & W_X \text{ until } \{ev_j \Rightarrow k_j\}, \text{ where } X \cap \{k_j\} = \emptyset \end{aligned}$$

This constraint ensures that each recursive let signal in the program is *well-formed*, and therefore the execution will not become stuck.

Given any let signal expression, its well-formedness can be checked statically. This information is only used to reject ill-formed recursive definitions, and is not needed at run-time. Hence the evaluation and transition rules are not aware of W_X, and there is no run-time overhead.

5. PROPERTIES OF RT-FRP

The main goals in designing RT-FRP are that it should run fast, have effective real-time guarantees, and be sufficiently expressive. We have given some evidence of its expressiveness (writing programs for recursive signals, stateful signals, and hybrid automata), and will give more evidence in section 6. In this section we concentrate on proving certain real-time properties of RT-FRP; in particular, that there is no space or time leak, and that the computation in each execution step is bounded.

A variable environment $\mathcal{E}$ assigns values to variables, and a variable context Γ assigns types to variables. If the values given to variables have the right types as specified by the context, we say that $\mathcal{E}$ is *compatible* with Γ, and write

$$\Gamma \vdash \mathcal{E}.$$

Similarly, we write

$$\Gamma; \Delta \vdash \mathcal{K}$$

to indicate that $\mathcal{K}$, which maps continuations to their definitions, is compatible with Γ and Δ, which give types to variables and continuations.

In the following theorems, we assume that $\Gamma, \Delta, \mathcal{E}$ and $\mathcal{K}$ are given, $\Gamma \vdash \mathcal{E}$, $\Gamma; \Delta \vdash \mathcal{K}$, and the external computation is type preserving (i.e. if $\Gamma \vdash_\lambda e : g$ and $\mathcal{E} \vdash e \hookrightarrow v$, then $\Gamma \vdash_\lambda v : g$.).

The first two theorems formally capture a notion of type preservation for terms and values:

THEOREM 1. (*Term Type-preservation*) If $\Gamma; \Delta \vdash_{\mathsf{S}} s : g$, then for all t and i, $\mathcal{E}; \mathcal{K} \vdash s \xrightarrow{t,i} s'$ implies $\Gamma; \Delta \vdash_{\mathsf{S}} s' : g$.

PROOF. By induction on the derivation for $\mathcal{E}; \mathcal{K} \vdash s \xrightarrow{t,i} s'$. $\square$

THEOREM 2. (*Value Type-preservation*) If $\Gamma; \Delta \vdash_{\mathsf{S}} s : g$, then for all t and i, $\mathcal{E} \vdash s \overset{t,i}{\hookrightarrow} v$ implies $\Gamma \vdash_\lambda v : g$.

PROOF. By induction on the derivation for $\mathcal{E} \vdash s \overset{t,i}{\hookrightarrow} v$. $\square$

The next theorem captures a progress property: if the external computations always termiates, then RT-FRP evaluation will never "get stuck":

THEOREM 3. (*Progress*) If $\Gamma; \Delta \vdash_{\mathsf{S}} s : g$, s does not use ill-formed recursions, and all the external expression evaluations used in the ev-ext rule terminate, then there are s' and v such that $\mathcal{E}; \mathcal{K} \vdash s \overset{t,i}{\hookrightarrow\!\!\!\rightarrow} s', v$.

Note that many RT-FRP programs will, by design, not terminate, but the above theorem guarantees that they will still yield instantaneous values at any given time.

To address the issue of bounded term size, we formally define the *size* of a term s, written $|\,s\,|$, to be the number of constructors, external expressions, and continuations in the term:

$$\begin{array}{rcl}
|\,\mathsf{input}\,| & = & 1 \\
|\,\mathsf{time}\,| & = & 1 \\
|\,\mathsf{ext}\ e\,| & = & 2 \\
|\,\mathsf{delay}\ v\ s\,| & = & 2 + |\,s\,| \\
|\,\mathsf{let\ signal}\ x = s_1\ \mathsf{in}\ s_2\,| & = & 2 + |\,s_1\,| + |\,s_2\,|
\end{array}$$

$$\begin{array}{rl}
& |\,s_1\ \mathsf{switch\ on}\ x = ev\ \mathsf{in}\ s_2\,| \\
= & 2 + |\,s_1\,| + |\,ev\,| + |\,s_2\,| \\
& |\,\mathsf{let\ continue}\ \{\,k_j\ x_j\ =\ s_j\,\}\ \mathsf{in}\ s\,| \\
= & 1 + 2n + \Sigma_{j=1}^{n}|\,s_j\,| + |\,s\,| \\
& |\,s\ \mathsf{until}\ \{\,ev_j\ \Rightarrow\ k_j\,\}\,| \\
= & 1 + n + |\,s\,| + \Sigma_{j=1}^{n}|\,ev_j\,|
\end{array}$$

As we will show later, in a given continuation environment $\mathcal{K}$, the size of a term s during execution is bounded. This *term-size bound*, written $\|\,s\,\|_{\mathcal{K}}$, can be formally defined as:

$$\begin{array}{rcl}
\|\,\mathsf{input}\,\|_{\mathcal{K}} & = & 1 \\
\|\,\mathsf{time}\,\|_{\mathcal{K}} & = & 1 \\
\|\,\mathsf{ext}\ e\,\|_{\mathcal{K}} & = & 2 \\
\|\,\mathsf{delay}\ v\ s\,\|_{\mathcal{K}} & = & 2 + \|\,s\,\|_{\mathcal{K}} \\
\|\,\mathsf{let\ signal}\ x = s_1\ \mathsf{in}\ s_2\,\|_{\mathcal{K}} & = & 2 + \|\,s_1\,\|_{\mathcal{K}} + \|\,s_2\,\|_{\mathcal{K}}
\end{array}$$

$$\begin{array}{rl}
& \|\,s_1\ \mathsf{switch\ on}\ x = ev\ \mathsf{in}\ s_2\,\|_{\mathcal{K}} \\
= & 2 + \max\{\|\,s_1\,\|_{\mathcal{K}}, \|\,s_2\,\|_{\mathcal{K}}\} + \|\,ev\,\|_{\mathcal{K}} + \|\,s_2\,\|_{\mathcal{K}} \\
& \|\,\mathsf{let\ continue}\ \{\,k_j\ x_j\ =\ s_j\,\}\ \mathsf{in}\ s\,\|_{\mathcal{K}} \\
= & 1 + 2n + \Sigma_{j=1}^{n}|\,s_j\,| + \|\,s\,\|_{\mathcal{K}'}, \\
& \quad \text{where } \mathcal{K}' = \{k_j \mapsto (x_j, s_j)\} \\
& \|\,s\ \mathsf{until}\ \{\,ev_j\ \Rightarrow\ k_j\,\}\,\|_{\mathcal{K}} \\
= & 1 + n + \|\,s\,\|_{[]} + \Sigma_{j=1}^{n}\|\,ev_j\,\|_{[]} + \max_{j=1}^{m}\{\|\,s_j\,\|_{[]}\}, \\
& \quad \text{where } \{k_j \mapsto (x_j, s_j)\}^{j \in \{1..m\}} = \mathcal{K}
\end{array}$$

By induction on the structure of s, it is easy to prove the following lemma:

LEMMA 1. For all well-typed term s and continuation environment $\mathcal{K}$, $|\,s\,| \le \|\,s\,\|_{\mathcal{K}}$.

THEOREM 4. (*Monotonic Transition*) If $\Gamma; \Delta \vdash_{\mathsf{S}} s : g$ then

1. $\|\,s\,\|_{\mathcal{K}}$ is defined;
2. for all t and i, if $\mathcal{E}; \mathcal{K} \vdash s \xrightarrow{t,i} s'$ then $\|\,s'\,\|_{\mathcal{K}} \le \|\,s\,\|_{\mathcal{K}}$.

PROOF. Statement 1 is proved by induction on the structure of s, and statement 2 is proved by induction on the derivation for $\mathcal{E}; \mathcal{K} \vdash s \xrightarrow{t,i} s'$. $\square$

From Lemma 1 and Theorem 4 we have corollary 1, which shows that $\| s \|_{\mathcal{K}}$ is indeed an upper bound for $| s_n |$, where s can evolve to s_n in $\mathcal{K}$ in zero or many steps:

COROLLARY 1. (*Term Size Bound*) Given a term s that satisfies $\Gamma; \Delta \vdash_{\mathsf{S}} s : g$, a sequence of times $t_1, t_2, \cdots, t_n$, and a sequence of input $i_1, i_2, \cdots, i_n$, if $\mathcal{E}; \mathcal{K} \vdash s \xrightarrow{t_1, i_1} s_1$, $\mathcal{E}; \mathcal{K} \vdash s_1 \xrightarrow{t_2, i_2} s_2$, $\cdots$, and $\mathcal{E}; \mathcal{K} \vdash s_{n-1} \xrightarrow{t_n, i_n} s_n$, then $| s_n | \leq \| s \|_{\mathcal{K}}$.

Define the size of a variable environment $\mathcal{E}$ (written $| \mathcal{E} |$) to be the number of elements in $\mathcal{E}$.

THEOREM 5. (*Variable Environment Size Bound*) Let $N = | \mathcal{E} | + \| s \|_{\mathcal{K}}$. If $\Gamma; \Delta \vdash_{\mathsf{S}} s : g$ and $\mathcal{D}$ is a derivation for $\mathcal{E}; \mathcal{K} \vdash s \overset{t,i}{\hookrightarrow\!\!\!\!\rightarrow} s', v$, then

1. for any $\mathcal{E}'; \mathcal{K}' \vdash s_1 \overset{t,i}{\hookrightarrow\!\!\!\!\rightarrow} s_2, v'$ in $\mathcal{D}$, $| \mathcal{E}' | + \| s_1 \|_{\mathcal{K}'} \leq N$;
2. for any $\mathcal{E}' \vdash e \hookrightarrow v'$ in $\mathcal{D}$, $| \mathcal{E}' | + 1 \leq N$.

PROOF. By inspecting the evaluation and transition rules. □

Define the size of a continuation environment $\mathcal{K}$ (written $| \mathcal{K} |$) to be the number of elements in $\mathcal{K}$.

THEOREM 6. (*Continuation Environment Size Bound*) Let

$$\begin{aligned} N &= \max\{n \mid \mathsf{let\ continue}\ \{ k_j\ x_j = s_j \}^{j \in \{1..n\}}\ \mathsf{in}\ s \\ &\qquad \text{is a sub-expression of } s.\} \\ M &= \max\{N, | \mathcal{K} |\}. \end{aligned}$$

If $\Gamma; \Delta \vdash_{\mathsf{S}} s : g$ and $\mathcal{D}$ is a derivation for $\mathcal{E}; \mathcal{K} \vdash s \overset{t,i}{\hookrightarrow\!\!\!\!\rightarrow} s', v$, then for any $\mathcal{E}'; \mathcal{K}' \vdash s_1 \xrightarrow{t,i} s_2$ in $\mathcal{D}$, $| \mathcal{K}' | \leq M$.

PROOF. By inspecting the evaluation and transition rules. □

THEOREM 7. (*Deterministic Derivation*) For any s, t, and i, there is exactly one derivation that leads to a transition of the form $\mathcal{E}; \mathcal{K} \vdash s \overset{t,i}{\hookrightarrow\!\!\!\!\rightarrow} s', v$.

PROOF. By inspecting the evaluation and transition rules. □

Define $| \Delta |$, the size of a derivation Δ, to be the number of times we use a transition or evaluation rule in Δ.

THEOREM 8. (*Derivation Size Bound*) For any s, t, and i, the size of the derivation for $\mathcal{E}; \mathcal{K} \vdash s \overset{t,i}{\hookrightarrow\!\!\!\!\rightarrow} s', v$ is no greater than $2| s |$.

PROOF. By induction on the structure of s. □

Corollary 1 and Theorem 5, 6 and 8 show that RT-FRP does not have space leak, given that there is no space leak in the external computation.

Theorem 7 and 8 ensure RT-FRP can be implemented efficiently, i.e. there is no time leak given the external compuation does not have time leak.

6. COMPILING FRP INTO RT-FRP

Despite its small size, RT-FRP is fairly expressive. In this section we describe how some of FRP's most common and useful operators can be compiled into RT-FRP.

FRP's family of "lifting" operators are easily compiled as follows:

$$\begin{aligned} &\mathsf{lift0}\ e \equiv \mathsf{ext}\ e \\ &\mathsf{lift1}\ e\ s \equiv \mathsf{let\ signal}\ x = s\ \mathsf{in\ ext}\ (e\ x) \\ &\mathsf{lift2}\ e\ s_1\ s_2 \\ \equiv\ &\mathsf{let\ signal}\ x_1 = s_1\ \mathsf{in} \\ &\quad \mathsf{let\ signal}\ x_2 = s_2\ \mathsf{in\ ext}\ (e\ x_1\ x_2) \end{aligned}$$

In FRP the snapshot operator allows one to grab the value of a behavior just when an event occurs, and to use it in subsequent computations. This operator is compiled as follows:

$$\begin{aligned} &ev\ \mathsf{snapshot}\ s \\ \equiv\ &\mathsf{lift2}\ f\ ev\ s, \text{ where} \\ &\quad f\ \bot\ v_2 = \bot \\ &\quad f\ v_{1\bot}\ v_2 = (v_1, v_2)_\bot \end{aligned}$$

FRP's never and once operators generate event values that never occur and occur exactly once, respectively. They are compiled as:

$$\begin{aligned} &\mathsf{never} \equiv \mathsf{lift0}\ \bot \\ &\mathsf{once}\ ev \\ \equiv\ &\mathsf{let\ signal}\ x_2 = ev\ \mathsf{in} \\ &\quad \mathsf{let\ signal}\ x_1 = \mathsf{delay}\ \bot \\ &\qquad (\mathsf{ext}\ (\text{if } x_1 = \bot \text{ then } x_2 \text{ else } x_1))\ \mathsf{in} \\ &\qquad \mathsf{ext}\ (\text{if } x_1 = \bot \text{ then } x_2 \text{ else } \bot) \end{aligned}$$

Finally, FRP's until and when operators, used in examples given in section 2, can be compiled as follows:

$$\begin{array}{ll} & s_1 \text{ until } ev \text{ then } s_2 \\ \equiv & s_1 \text{ switch on } x = \text{once } ev \text{ in } s_2 \end{array}$$

$$\begin{array}{ll} & \text{when } s \\ \equiv & \text{lift2 } (\lambda x_1.\lambda x_2.\text{ if } \neg x_1 \wedge x_2 \text{ then } ()_\perp \text{ else } \perp) \\ & \quad (\text{delay false } s)\ s \end{array}$$

7. RELATED WORK

Several languages have been proposed around the *synchronous data-flow* notion of computation. The general-purpose functional language Lucid [23] is an example of this style of language, but more significant are the languages Signal [9], Lustre [3], and Esterel [1, 2], 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. This is analogous to the sequence of values generated in the execution of a RT-FRP program. 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.

Various implementation techniques for Fran are discussed in [7], including the basic ideas behind the stream-based implementation developed in [12].

CML (Concurrent ML) formalized synchronous operations as first-class, purely functional, values called "events" [21]. 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.

8. REFERENCES

[1] G. Berry and L. Cosserat. The esterel synchronous programming language and its mathematical semantics. In A.W. Roscoe S.D. Brookes and editors G. Winskel, editors, *Seminar on Concurrency*, volume 197 of *Lect. Notes in Computer Science*, pages 389–448. Springer Verlag, 1985.

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

[3] 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.

[4] Antony Courtney. Frappé: Functional reactive programming in Java. In *Proceedings of Symposium on Practical Aspects of Declarative Languages*. ACM, 2001.

[5] Anthony C. Daniels. *A Semantics for Functions and Behaviours*. PhD thesis, The University of Nottingham, December 1999.

[6] 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.

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

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

[9] 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.

[10] Thomas A. Henzinger. The theory of hybrid automata. Technical report, University of California, Berkeley, 1996.

[11] P. Hudak. Building domain specific embedded languages. *ACM Computing Surveys*, 28A:(electronic), December 1996.

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

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

[14] 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.

[15] O. (Oded) Maler, editor. *Hybrid and real-time systems: international workshop, HART '97, Grenoble, France, March 26–28, 1997: proceedings*, volume 1201 of *Lecture Notes in Computer Science*, New York, NY, USA, 1997. Springer-Verlag.

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

[17] 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.

[18] Gordon Plotkin. A structural approach to operational semantics. Technical report, Computer Science Department, Aarhus University, 1981.

[19] J. Rees and W. Clinger (eds.). The revised^3 report on the algorithmic language Scheme. *SIGPLAN Notices*, 21(12):37–79, December 1986.

[20] Alastair Reid, John Peterson, Greg Hager, and Paul Hudak. Prototyping real-time vision systems: An experiment in DSL design. In *Proc. Int'l Conference on Software Engineering*, May 1999.

[21] 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.

[22] Meurig Sage. FranTk – a declarative GUI language for Haskell. In *Proceedings of Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 106–118, Montreal, Canada, September 2000. ACM.

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

[24] Z. Wan and P. Hudak. Functional reactive programming from first principles. In *Proceedings of Symposium on Programming Language Design and Implementation*. ACM, 2000.