Real-Time FRP*

Zhanyong Wan Walid Taha† Paul Hudak
Yale University
Department of Computer Science
New Haven, CT 06520, USA
{wan-zhanyong,taha-walid,hudak-paul}@cs.yale.edu

ABSTRACT

Functional reactive programming (FRP) is a declarative programming paradigm where the basic notions are continuous, time-varying behaviors and discrete, event-based reactivity. FRP has been used successfully in many reactive programming domains such as animation, robotics, and graphical user interfaces. The success of FRP in these domains encourages us to consider its use in real-time applications, where it is crucial that the cost of running a program be bounded and known before run-time. But previous work on the semantics and implementation of FRP was not explicitly concerned about the issues of cost. In fact, the resource consumption of FRP programs in the current implementation is often hard to predict.

As a first step towards addressing these concerns, this paper presents Real-Time FRP (RT-FRP), a statically-typed language where the time and space cost of each execution step for a given program is statically bounded. To take advantage of existing work on languages with bounded resources, we split RT-FRP into two parts: a reactive part that captures the essential ingredients of FRP programs, and a base language part that can be instantiated to any generic programming language that has been shown to be terminating and resource-bounded. This allows us to focus on the issues specific to RT-FRP, namely, two forms of recursion. After presenting the operational explanation of what can go wrong due to the presence of recursion, we show how the typed version of the language is terminating and resource-bounded.

Most of our FRP programs are expressible directly in RT-FRP. The rest are expressible via a simple mechanism that integrates RT-FRP with the base language.

*Funded by DARPA F33615-99-C-3013 and NSF CCR-9900957.
†Funded also by subcontract #8911-48186 from Johns Hopkins University under NSF agreement Grant # EIA-9996430.

ICFP'01, September 3-5, 2001, Florence, Italy.

1. INTRODUCTION

Many real-world software systems are required to respond to external stimuli in a bounded amount of time. In addition, some need to execute using a fixed amount of memory. Today, many such *real-time* and *embedded* systems are being designed, implemented, and maintained. As this trend continues, the reliability and safety of programming languages for such systems becomes more of a concern, and real-time systems become a natural domain for a high-level programming language.

Functional Reactive Programming (FRP) [16, 39] is a paradigm that has been used for building a host of interesting reactive systems in domains such as animation [8, 9, 33], graphical user interface design [5], and robotics [27, 28, 31]. As such, FRP is a good candidate for a high-level language for real-time programming. The central semantic notions in FRP are *behaviors* and *events*. Each is provided to the user in the form of a parametric type, namely `Behavior a` and `Event a`, respectively. In the original denotational semantics for FRP [9, 39], a behavior is simply a function of continuous time, corresponding to the intuition that a behavior has a value at any given instant. Time itself is modeled by the real numbers. An event, in contrast, is a time-ordered sequence of event occurrences. Together, these two notions provide a natural foundation for describing systems of recursive equations over time-parametric, hybrid (that is, continuous and discrete) operators.

1.1 Problem

Although FRP has proven to be fast enough for most of the applications we have considered, it is not easy to establish strong guarantees about its time and space behavior. Our goal is to provide a practical framework where clear guarantees about the cost of an FRP computation can be made. There are three distinct problems that need to be addressed: First, the reference semantics for FRP is denotational [9, 39]. While a denotational model helps in understanding the meaning of an FRP program, it does not explain how a program can be effectively executed on a digital computer with finite resources, nor does it provide a natural notion of cost. Second, because FRP was initially implemented as an embedded language in Haskell [13, 14, 15], making assertions about the cost of a computation was nontrivial. Third, embedding a language into a higher-order language introduces all of the power of the lambda calculus. Inherent in this expressiveness is the possibility of writing programs that perform unreasonable or unbounded amounts of computation at each time step, or that have subtle (hard

to find) space leaks.

1.2 Our Approach

The first step towards addressing these problems is to specify an operational model of the execution of an FRP program. By their very nature, FRP programs do not terminate: they continuously emit values and interact with the environment. Thus it is appropriate to model FRP program execution as an infinite sequence of steps. In each step, the current time and current stimuli are read, and the result is an output value and an updated program state. Our goal is to guarantee that every step executes in bounded time, and that overall program execution occurs in bounded space. However, we cannot make such guarantees for arbitrary FRP programs, and thus we define a subset of FRP called *Real-Time (RT-)FRP* for which we can make such guarantees. With this approach, we solve each of the above three problems as follows:

1. RT-FRP is given an operational semantics [29] that provides a well-defined notion of cost. That is, the size of the derivation for the judgment(s) defining a step of execution provides a measure of the amount of time and space needed to execute that step on a digital computer.

2. RT-FRP is a *closed language* [21] in the sense that it is not embedded into a larger language such as Haskell. This makes it possible to give a *direct* operational semantics to the language, and therefore to provide a *tractable* notion of cost.

   In addition, having an *explicit* notion of state is a feature of our model that was not present in previous work. Making state explicit allows us to specify an efficient destructive-update semantics, which is hard to enforce with the embedded approach.

3. A key aspect of our approach is to split RT-FRP into two naturally distinguishable parts: a *reactive* part and a *base language* part. Roughly speaking, the reactive part is comparable to a synchronous system [4], and the base language part can be any language that we wish to extend to a reactive setting. As we will show, the reactive part has bounded cost in terms of both time and space, independent of the base language. Thus we can reuse our approach with a new base language without having to re-establish these results. Real-time behavior of the base language can be carried out independently, and such techniques already exist even for a functional base language [12, 18, 19, 24].

One important question that our work needs to address is the treatment of recursion, as it is a source of both expressiveness and computational cost. A key contribution of our work is restricting the reactive part of the language so as to limit both the time and space needed to execute such computations. We achieve this by first distinguishing two different kinds of recursion in FRP: one for pure signals, and one for reactivity. Without constraint, the first form of recursion can lead to programs getting stuck, and the second form can cause terms to grow in size. We address these problems using carefully chosen syntax and a carefully designed type system. Our restrictions on one of the forms of recursion is inspired by tail-recursion [30].

1.3 Organization of the Paper

Section 2 introduces the syntax and basic concepts of RT-FRP. Section 3 defines and explains the type system and operational semantics for RT-FRP. In this section we also explain how two different forms of recursion in RT-FRP can cause the semantics to either get stuck or use unbounded space. Section 4 presents our main technical results, namely, termination and type preservation, and resource-boundedness. All of these properties are qualified by explicit assumptions about the base language. Section 5 discusses related work.

The operational semantics of the example base language that we use is presented in Appendix A.

2. A BRIEF INTRODUCTION TO RT-FRP

In this section we introduce the syntax and basic concepts of RT-FRP.

2.1 Behaviors and Events as Signals

A key difference between discrete models (such as our operational semantics) and continuous models (such as a denotational semantics) is that in the discrete case behaviors and events only need to have values at a countable set of points. This means that there is an interesting type isomorphism [7] relating behaviors and events, namely [16]:

$$\texttt{Event a} \approx \texttt{Behavior (Maybe a)}$$

where `Maybe a` is a data type with data constructors `Nothing` and `Just a`. This isomorphism makes it possible in our work to combine behaviors and events into one common type that we call a *signal*, and thus treat both concepts uniformly.

2.2 A Concrete Base Language Syntax

To simplify the presentation, we will work with a concrete base language, which has the syntax:

$$\begin{array}{rcl} e & ::= & x \mid c \mid () \mid (e_1, e_2) \mid e_\perp \mid \perp \mid \lambda x.e \mid e_1\ e_2 \\ v & ::= & c \mid () \mid (v_1, v_2) \mid v_\perp \mid \perp \mid \lambda x.e \end{array}$$

where x and c are the syntactic categories of variables and real numbers, respectively. The only unusual feature is that the terms $e_\perp$ and $\perp$ are used as a more concise way of writing `Just` e and `Nothing` introduced earlier. For clarity, we also occasionally take the liberty of using some common syntax not provided here, such as where clauses and if-then-else. *Values* in the base language are represented by the terms v.

2.3 Reactive Language Syntax

The reactive part of RT-FRP is given by:

$$\begin{array}{rcl} s, ev & ::= & \mathsf{input} \mid \mathsf{time} \mid \mathsf{ext}\ e \mid \mathsf{delay}\ v\ s \mid \\ & & \mathsf{let\ snapshot}\ x \leftarrow s_1\ \mathsf{in}\ s_2 \mid \\ & & s_1\ \mathsf{switch\ on}\ x \leftarrow ev\ \mathsf{in}\ s_2 \mid \\ & & \mathsf{let\ continue}\ \{ k_j\ x_j = u_j \}\ \mathsf{in}\ s \mid \\ & & u \\ u & ::= & s\ \mathsf{until}\ \langle ev_j\ \Rightarrow\ k_j \rangle \end{array}$$

where k is the syntactic category of continuation variables. Note that the base language terms can occur inside signal terms, but not the other way around. Furthermore, a variable bound by let-snapshot can only be used in the base language.

In the rest of this section we explain each of the reactive constructs of RT-FRP in more detail.

2.3.1 Primitive Signals

The two primitive signals in RT-FRP are the current stimulus input and the current time time in seconds. In this paper we only make use of time, as it is sufficient for illustrating all the operations that one might want to define on external stimuli. In practice, however, input may be instantiated to much more interesting types – such as mouse clicks, keyboard presses, network messages, and so on – since there are few interesting systems that react only to time.

2.3.2 Interfacing with Base Language

The reactive part s of RT-FRP does not provide primitive operations such as addition and subtraction on the values of signals. Instead, this is relegated to the base language e. To interface with the base language, the reactive part of RT-FRP has a mechanism for exporting snapshots of signal values to the base language, and a mechanism for importing base language values back into the signal world. Specifically, to export a signal, we snapshot its current value using the let-snapshot construct, and to invoke an external computation in the base language we use the ext e construct.

To illustrate, suppose we wish to define a signal representing the current time in minutes. We can do this by:

$$\text{let snapshot } x \leftarrow \text{time in ext } (x/60)$$

To compute externally with more than one signal, we have to snapshot each one separately. For example, the term:

$$\begin{array}{l}\text{let snapshot } x \leftarrow s_1 \text{ in} \\ \quad \text{let snapshot } y \leftarrow s_2 \text{ in ext } (x + y)\end{array}$$

is a signal that is the point-wise sum of the signals s_1 and s_2. Those familiar with FRP will recognize this idea as one of the *lifting* primitive operations into the signal world. We can define lifting operators such as:

$$\begin{array}{rcl}[\![\text{lift0 } e]\!] & \equiv & \text{ext } e \\ [\![\text{lift1 } e\ s]\!] & \equiv & \text{let snapshot } x \leftarrow s \text{ in ext } (e\ x) \\ [\![\text{lift2 } e\ s_1\ s_2]\!] & \equiv & \text{let snapshot } x_1 \leftarrow s_1 \text{ in} \\ & & \quad \text{let snapshot } x_2 \leftarrow s_2 \text{ in ext } (e\ x_1\ x_2)\end{array}$$

in which case the above two examples could be written more simply as:

$$\begin{array}{l}\text{lift2 } (/) \text{ time } (\text{lift0 } 60) \\ \text{lift2 } (+)\ s_1\ s_2\end{array}$$

2.3.3 Stateful Constructs

There are two stateful constructs in RT-FRP: delay and switch. The signal delay v s is a delayed version of s, whose initial value is v. To illustrate the use of delay, the following term computes the difference between the current time and the time at the previous program execution step:

$$\begin{array}{l}\text{let snapshot } t_0 \leftarrow \text{time in} \\ \quad \text{let snapshot } t_1 \leftarrow \text{delay 0 time in ext } (t_0 - t_1)\end{array}$$

As another example, the when operator in FRP turns a Boolean signal s into an event that occurs whenever s transitions from False to True. This useful operator can be defined using delay:

$$\begin{array}{rcl}[\![\text{when } s]\!] & \equiv & \text{let snapshot } x_1 \leftarrow s \text{ in} \\ & & \quad \text{let snapshot } x_2 \leftarrow \text{delay False } s \text{ in} \\ & & \quad\quad \text{ext } (\text{if } \neg x_2 \wedge x_1 \text{ then } ()_\perp \text{ else } \perp).\end{array}$$

The second stateful construct, switch, is used to define signals that react to other signals. For example, a sample-and-hold register that remembers the most recent event value it received can be defined as:

$$(\text{ext } 0) \text{ switch on } x \leftarrow ev \text{ in } (\text{ext } x).$$

This signal starts out as 0. Whenever the event ev occurs, its current value is substituted for x in the body ext x, and that value becomes the current value of the overall switch construct.

2.3.4 Recursion

In addition to its role in exporting signal values to the base language, the let-snapshot construct can also be used to define recursive signals. By combining delay with recursive signals, we can add internal state to signals. For example, we can define the running maximum of a signal s as follows:

$$\begin{array}{l}\text{let snapshot } cur \leftarrow s \text{ in} \\ \quad \text{let snapshot } rmax \leftarrow \text{delay } (-\infty)\ (\text{ext } \max\{rmax, cur\}) \\ \quad\quad \text{in ext } rmax\end{array}$$

$rmax$ is $-\infty$ in the initial step. At the $(n+1)$-th step, it is updated to be the larger one of its previous value and the value of s at step n. Therefore $rmax$ records the maximum value of s up to the previous step.

A particularly useful stateful operation that we can express in RT-FRP is *integration* over time, defined below using the forward-Euler method:

$$\begin{array}{rcl}[\![\text{integral } s]\!] & \equiv & \text{let snapshot } t \leftarrow \text{time in} \\ & & \quad \text{let snapshot } v \leftarrow s \text{ in} \\ & & \quad\quad \text{let snapshot } st \leftarrow \text{delay } (0, (0, 0)) \\ & & \quad\quad\quad (\text{ext } ((i, (v, t)) \text{ where} \\ & & \quad\quad\quad\quad (i_0, (v_0, t_0)) = st \\ & & \quad\quad\quad\quad i = i_0 + v_0(t - t_0)) \\ & & \quad\quad \text{in ext } (\text{fst } st).\end{array}$$

Note that the internal state of integral s is a tuple $(i, (v, t))$, where i is the running integral, v is the previous value of s, and t is the previous sample time.

Integration is extremely useful in the definition of control systems. For example, the velocity of a mass m under force f and friction kv can be described by the recursive integral equation:

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

The RT-FRP encoding for this signal is simply:

$$\text{let snapshot } v \leftarrow \text{integral } (\text{ext } (f - k * v)/m).$$

2.3.5 Modes and Continuations

RT-FRP provides two additional constructs, namely let-continue and until, that allow the definition of *multi-modal* signals, that is, signals that shift from one operating mode to another depending on the occurrence of events.[1] For example, here is a system that switches between two signals

[1] It is technically possible to use let-snapshot and switch to do this, but the result is much more awkward, and requires extensive escape to the base language.

s_1 and s_2 depending on the occurrence of an event ev:

$$\begin{array}{l}\mathsf{let\ snapshot}\ y_1 \leftarrow s_1\ \mathsf{in}\\ \mathsf{let\ snapshot}\ y_2 \leftarrow s_2\ \mathsf{in}\\ \mathsf{let\ snapshot}\ x \leftarrow ev\ \mathsf{in}\\ \mathsf{let\ continue}\ \{\ k_1\ y = \mathsf{ext}\ y_1\ \mathsf{until}\ \langle \mathsf{ext}\ x \Rightarrow k_2\rangle,\\ \qquad\qquad k_2\ y = \mathsf{ext}\ y_2\ \mathsf{until}\ \langle \mathsf{ext}\ x \Rightarrow k_1\rangle\}\ \mathsf{in}\\ \mathsf{ext}\ y_1\ \mathsf{until}\ \langle \mathsf{ext}\ x \Rightarrow k_2\rangle.\end{array}$$

The let-continue declaration defines a set of mutually recursive continuations. A *continuation* is essentially a signal parameterized by a variable, and corresponds naturally to a *mode* in control systems design. The until construct is used to jump between continuations at event occurrences.

As a more concrete example, consider the task of implementing a thermostat in RT-FRP. A thermostat has two modes. In the *On* mode, the heater is on, and the temperature rises according to some flow condition. When the condition "temperature $\geq T_{high}$" occurs, it switches to the *Off* mode, and the temperature gradually drops. The thermostat jumps back to the *On* mode when "temperature $\leq T_{low}$" becomes true. The following program defines such a system:

$$\begin{array}{l}\mathsf{let\ snapshot}\ t \leftarrow \mathit{temperature}\ \mathsf{in}\\ \mathsf{let\ continue}\\ \quad \{\ \mathsf{on}\ x = (\mathsf{ext}\ 1)\ \mathsf{until}\ \langle \mathsf{when}\ (\mathsf{ext}\ (t \geq T_{high})) \Rightarrow \mathsf{off}\rangle,\\ \quad\ \ \mathsf{off}\ x = (\mathsf{ext}\ 0)\ \mathsf{until}\ \langle \mathsf{when}\ (\mathsf{ext}\ (t \leq T_{low})) \Rightarrow \mathsf{on}\rangle\ \}\\ \mathsf{in}\ (\mathsf{ext}\ 1)\ \mathsf{until}\ \langle \mathsf{when}\ (\mathsf{ext}\ (t \geq T_{high})) \Rightarrow \mathsf{off}\rangle\end{array}$$

2.3.6 Switch in Terms of Continuations

It is possible to express the switch construct in terms of continuations (where y and k are fresh):

$$\begin{array}{rl} & [\![s_1\ \mathsf{switch\ on}\ x \leftarrow ev\ \mathsf{in}\ s_2]\!]\\ \equiv & \mathsf{let\ snapshot}\ y \leftarrow ev\ \mathsf{in}\\ & \quad \mathsf{let\ continue}\ \{k\ x = s_2\ \mathsf{until}\ \langle \mathsf{ext}\ y \Rightarrow k\rangle\}\\ & \quad\ \ \mathsf{in}\ (s_1\ \mathsf{until}\ \langle \mathsf{ext}\ y \Rightarrow k\rangle).\end{array}$$

For clarity, we will define the typing rule and operational semantics for switch directly. When we present the semantics, it should be easy to verify that the translation is preserved by execution. For this reason we will not consider switch when proving properties about RT-FRP.

3. SEMANTICS OF RT-FRP

In this section we present and explain the type system and operational semantics of RT-FRP.

3.1 Notation

We use the notation $\langle f_j \rangle^{j \in \{1..n\}}$ as shorthand for a finite sequence $\langle f_1, f_2, \ldots, f_n \rangle$. We omit the superscript $j \in \{1..n\}$ when it is obvious from the context. Similarly, we write $\{f_j\}^{j \in \{1..n\}}$ or $\{f_j\}$ for a finite set $\{f_1, f_2, \cdots, f_n\}$.

3.2 Type System

For simplicity of presentation, we do not explicitly distinguish between the reactive types for RT-FRP and the types provided by the base language.

3.2.1 Types

The syntax for RT-FRP types is defined as follows:

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

The meaning of these types will vary depending on whether they are assigned to base language terms or RT-FRP signals. In the first case, g will have its usual interpretation. In the second case, it will mean "a signal carrying a value of type g". In FRP, these would be written e `::` g and s `::` `Behavior` g, respectively. Using two different interpretations makes it possible to describe the latter case as "a signal of type g".

The type input is a placeholder for the signature of the external stimuli visible to the system. The real is for real numbers, and unit is a singleton type. The type $g_\perp$ is a "maybe g" value. Having the latter type in the base language allows us to treat behaviors and events uniformly as signals (in the manner explained at the beginning of the previous section). The type $g_1 \to g_2$ is for functions that take an argument of type g_1 and return a value of type g_2.

It will be convenient in the definition of the type system to identify the following set of *base types*, that is, types that do not have a functional part:

$$b \quad ::= \quad \mathsf{input} \mid \mathsf{real} \mid \mathsf{unit} \mid b \times b \mid b_\perp$$

3.2.2 Contexts

A *variable context* Γ is a function from variables to annotated types. An *annotated type* could be either *exportable* (written $\uparrow g$) or *local* (written g). These annotations are need to ensure that the phase distinction between the evaluation of a term and updating a term is reflected in enough detail so as to allow us to guarantee type safety, even in the presence of recursion.

We will also treat contexts functions as sets (their graph). We require all variable names in a program to be distinct.

$$\text{Variable contexts} \quad \Gamma \quad ::= \quad \{\, x_j : ?g_j \,\}$$

where $?g$ is an annotated type. The annotations allow us to define the following two functions on variable contexts, called *expose* and *export*, respectively:

$$\begin{array}{rcl} \uparrow\{\, x_j : ?g_j \,\} & = & \{\, x_j : \uparrow g_j \,\} \\ \downarrow(\{\, x_j : \uparrow g_j \,\} \cup \{\, x_k : g_k \,\}) & = & \{\, x_j : g_j \,\}. \end{array}$$

A *continuation context* Δ is a function from continuations to types.

$$\text{Continuation contexts} \quad \Delta \quad ::= \quad \{\, k_j : g_j \to g'_j \,\}.$$

A binding $k : g \to g'$ in Δ means that k is a continuation that takes a value of type g and returns a signal of type g'.

3.2.3 Typing Judgment

Figure 1 defines the RT-FRP type system using a judgment $\Gamma; \Delta \vdash_S s : g$, read "$s$ is a signal of type g". The signal input has type input. The signal time has type real. The term ext e is a signal of type g when e is a base language term of type g. In this typing rule, e is typed without using Δ. Intuitively, this means that continuations cannot be exported to the base language.

The type of delay v s is the type of s, given that the base language term v has the same type. Note, however, that the term s must have a *base type* b. This technical restriction is necessary to ensure type preservation.

A term let snapshot $x \leftarrow s_1$ in s_2 has the same type as s_2, assuming that s_1 is well-typed. The typing reflects the fact that this is a recursive binding construct, and that x can occur anywhere. In s_1 switch on $x \leftarrow ev$ in s_2, ev must have

$$\frac{}{\Gamma;\ \Delta \vdash_{\mathsf{S}} \mathsf{input} : \mathsf{input}}\ (\mathsf{t1}) \qquad \frac{}{\Gamma;\ \Delta \vdash_{\mathsf{S}} \mathsf{time} : \mathsf{real}}\ (\mathsf{t2}) \qquad \frac{\Gamma \cup \{x : g_1\};\ \Delta \vdash_{\mathsf{S}}\ s_1 : g_1 \quad \Gamma \cup \{x :\uparrow g_1\};\ \Delta \vdash_{\mathsf{S}}\ s_2 : g_2}{\Gamma;\ \Delta \vdash_{\mathsf{S}} \mathsf{let\ snapshot}\ x\ \leftarrow\ s_1\ \mathsf{in}\ s_2 : g_2}\ (\mathsf{t3})$$

$$\frac{\downarrow\Gamma \vdash_{\lambda}\ e : g}{\Gamma;\ \Delta \vdash_{\mathsf{S}}\ \mathsf{ext}\ e : g}\ (\mathsf{t4}) \qquad \frac{\emptyset \vdash_{\lambda}\ v : b \quad \uparrow\Gamma;\ \Delta \vdash_{\mathsf{S}}\ s : b}{\Gamma;\ \Delta \vdash_{\mathsf{S}}\ \mathsf{delay}\ v\ s : b}\ (\mathsf{t5}) \qquad \frac{\Gamma;\ \Delta \vdash_{\mathsf{S}}\ s_1 : g_1 \quad \uparrow\Gamma;\ \Delta \vdash_{\mathsf{S}}\ ev : g_{2\perp} \quad \Gamma \cup \{x :\uparrow g_2\};\ \Delta \vdash_{\mathsf{S}}\ s_2 : g_1}{\Gamma;\ \Delta \vdash_{\mathsf{S}}\ s_1\ \mathsf{switch\ on}\ x \leftarrow ev\ \mathsf{in}\ s_2 : g_1}\ (\mathsf{t6})$$

$$\frac{\left\{\Gamma \cup \{x_i :\uparrow g_i\};\ \{k_j : g_j \to g'_j\}^{j\in J} \vdash_{\mathsf{S}}\ u_i : g'_i\right\}^{i\in J} \quad \Gamma;\ \Delta \cup \{k_j : g_j \to g'_j\}^{j\in J} \vdash_{\mathsf{S}}\ s : g}{\Gamma;\ \Delta \vdash_{\mathsf{S}}\ \mathsf{let\ continue}\ \{k_j\ x_j\ =\ u_j\}^{j\in J}\ \mathsf{in}\ s : g}\ (\mathsf{t7}) \qquad \frac{\Gamma;\emptyset \vdash_{\mathsf{S}}\ s : g \qquad \{\uparrow\Gamma;\emptyset \vdash_{\mathsf{S}}\ ev_j : g_{j\perp}\}}{\Gamma;\Delta \cup \{k_j : g_j \to g\} \vdash_{\mathsf{S}}\ s\ \mathsf{until}\ \langle\, ev_j\ \Rightarrow\ k_j\,\rangle : g}\ (\mathsf{t8})$$

Figure 1: Type System for Reactive part of RT-FRP

$$\frac{}{\Gamma \cup \{x : g\} \vdash_{\lambda}\ x : g}\ (\mathsf{s1}) \qquad \frac{}{\Gamma \vdash_{\lambda}\ c : \mathsf{real}}\ (\mathsf{s2}) \qquad \frac{}{\Gamma \vdash_{\lambda}\ () : \mathsf{unit}}\ (\mathsf{s3}) \qquad \frac{\Gamma \vdash_{\lambda}\ e : g}{\Gamma \vdash_{\lambda}\ e_{\perp} : g_{\perp}}\ (\mathsf{s4}) \qquad \frac{}{\Gamma \vdash_{\lambda}\ \perp : g_{\perp}}\ (\mathsf{s5})$$

$$\frac{\Gamma \cup \{x : g_1\} \vdash_{\lambda}\ e : g_2}{\Gamma \vdash_{\lambda}\ \lambda x.e : g_1 \to g_2}\ (\mathsf{s6}) \qquad \frac{\Gamma \vdash_{\lambda}\ e_1 : g_1 \to g_2 \quad \Gamma \vdash_{\lambda}\ e_2 : g_1}{\Gamma \vdash_{\lambda}\ e_1\ e_2 : g_2}\ (\mathsf{s7}) \qquad \frac{\Gamma \vdash_{\lambda}\ e_1 : g_1 \quad \Gamma \vdash_{\lambda}\ e_2 : g_2}{\Gamma \vdash_{\lambda}\ (e_1, e_2) : g_1 \times g_2}\ (\mathsf{s8})$$

Figure 2: Type System for an Example Base Language for RT-FRP

an event type, s_1 and s_2 should be of the same type, and the scope of x is in s_2.

The type of a continuation k has the form $g \to g'$, meaning that when fed a value of type g, k becomes a signal of type g'. A group of mutually recursive continuations are defined by let-continue. A continuation definition has the form $k\ x = u$, where x is the formal parameter for k, and u (an until term) is the definition body. Note that u can only contain those continuations being defined in the same declaration. These constraints resemble tail recursion, and so we call such terms *tail signals*. Intuitively, the constraints establish a set of simple scoping rules for continuations:

1. The definitions $\{u_j\}$ in a new continuation declaration cannot refer to surrounding continuation declarations Δ.

2. In s until $\langle\, ev_j \Rightarrow k_j\,\rangle$, none of the sub-terms can contain free continuations, as shown by the rule t8.

The analogy with tail recursion lies in that just as tail calls have to be the last calls made, an invocation of a tail signal has to be the last invocation made.

In a term s until $\langle ev_j \Rightarrow k_j \rangle$, the type of an event ev_j must match the parameter type of the continuation k_j, and the result type of all k_j must be the same as the type of s.

To use a concrete example for the base language, Figure 2 defines a fairly standard type system using a judgment $\Gamma \vdash_{\lambda}\ e : g$, read "$e$ is a base language term of type g."

3.3 Operational Semantics

We now present the full details of the operational semantics of the reactive part of RT-FRP.

3.3.1 Environments

Program execution takes place in the context of a *variable environment* $\mathcal{E}$ and a *continuation environment* $\mathcal{K}$:

Variable environments	$\mathcal{E}$	::=	$\{x_j \mapsto v_j\}$
Continuation environments	$\mathcal{K}$	::=	$\{k_j \mapsto \lambda x_j.u_j\}$.

A variable environment is used to store the current values of signals, and hence maps signal variables to values. The environment $\mathcal{K}$ maps a continuation to its definition. The lambda abstraction makes explicit the formal argument and the signal parameterized by that argument.

3.3.2 Two Judgment Forms

Figure 3 defines the single-step semantics by means of two judgments: $\mathcal{E} \vdash s \overset{t,i}{\hookrightarrow} v$, read "$s$ evaluates to v", and $\mathcal{E};\mathcal{K} \vdash s \overset{t,i}{\longrightarrow} s'$, read "$s$ is updated to become s'". Note that the semantics for each step is parameterized by the current time t and the current input i. Sometimes we combine the two judgments and write $\mathcal{E};\mathcal{K} \vdash s \overset{t,i}{\hookrightarrow\!\!\!\!\rightarrow} v, s'$. When the environments are empty, we write $s \overset{t,i}{\hookrightarrow\!\!\!\!\rightarrow} v, s'$.

The role of evaluation is to compute the output of a term. The role of updating is to modify the state of the program. We will explain the rules of each of these judgments shortly.

The overall execution, or "run", of an RT-FRP program is modeled as an infinite sequence of interactions with the environment, and in that sense does not terminate. Formally, for a given sequence of time stamps and external stimuli $\langle (t_0, i_0), (t_1, i_1), ... \rangle$, a *run* of an RT-FRP program s_0 produces a sequence of values $\langle v_0, v_1, ... \rangle$, where $s_k \overset{t_k,i_k}{\hookrightarrow\!\!\!\!\rightarrow} v_k, s_{k+1}$ for some sequence $\langle s_{k+1} \rangle$. Thus, a run can be visualized as an infinite chain of the form:

$$s_0 \overset{t_0,i_0}{\hookrightarrow\!\!\!\!\rightarrow} v_0, s_1 \overset{t_1,i_1}{\hookrightarrow\!\!\!\!\rightarrow} v_1, s_2 \ldots s_n \overset{t_n,i_n}{\hookrightarrow\!\!\!\!\rightarrow} v_n, s_{n+1} \ldots$$

Evaluation Rules

$$\frac{}{\mathcal{E} \vdash \mathsf{input} \overset{t,i}{\hookrightarrow} i} \text{(e1)}$$

$$\frac{}{\mathcal{E} \vdash \mathsf{time} \overset{t,i}{\hookrightarrow} t} \text{(e2)}$$

$$\frac{\mathcal{E} \vdash e \hookrightarrow v}{\mathcal{E} \vdash \mathsf{ext}\ e \overset{t,i}{\hookrightarrow} v} \text{(e3)}$$

$$\frac{}{\mathcal{E} \vdash \mathsf{delay}\ v\ s \overset{t,i}{\hookrightarrow} v} \text{(e4)}$$

$$\frac{\mathcal{E} \vdash s_1 \overset{t,i}{\hookrightarrow} v_1 \quad \mathcal{E} \cup \{x \mapsto v_1\} \vdash s_2 \overset{t,i}{\hookrightarrow} v_2}{\mathcal{E} \vdash \mathsf{let\ snapshot}\ x \leftarrow s_1\ \mathsf{in}\ s_2 \overset{t,i}{\hookrightarrow} v_2} \text{(e5)}$$

$$\frac{\mathcal{E} \vdash s_1 \overset{t,i}{\hookrightarrow} v}{\mathcal{E} \vdash s_1\ \mathsf{switch\ on}\ x \leftarrow ev\ \mathsf{in}\ s_2 \overset{t,i}{\hookrightarrow} v} \text{(e6)}$$

$$\frac{\mathcal{E} \vdash s \overset{t,i}{\hookrightarrow} v}{\mathcal{E} \vdash \mathsf{let\ continue}\ \{\, k_j\ x_j\ =\ u_j \,\}\ \mathsf{in}\ s \overset{t,i}{\hookrightarrow} v} \text{(e7)}$$

$$\frac{\mathcal{E} \vdash s \overset{t,i}{\hookrightarrow} v}{\mathcal{E} \vdash s\ \mathsf{until}\ \langle\, ev_j\ \Rightarrow\ k_j \,\rangle \overset{t,i}{\hookrightarrow} v} \text{(e8)}$$

Updating Rules

$$\frac{}{\mathcal{E}; \mathcal{K} \vdash \mathsf{input} \overset{t,i}{\longrightarrow} \mathsf{input}} \text{(u1)}$$

$$\frac{}{\mathcal{E}; \mathcal{K} \vdash \mathsf{time} \overset{t,i}{\longrightarrow} \mathsf{time}} \text{(u2)}$$

$$\frac{}{\mathcal{E}; \mathcal{K} \vdash \mathsf{ext}\ e \overset{t,i}{\longrightarrow} \mathsf{ext}\ e} \text{(u3)}$$

$$\frac{\mathcal{E}; \mathcal{K} \vdash s \overset{t,i}{\hookrightarrow\!\!\!\!\rightarrow} v', s'}{\mathcal{E}; \mathcal{K} \vdash \mathsf{delay}\ v\ s \overset{t,i}{\longrightarrow} \mathsf{delay}\ v'\ s'} \text{(u4)}$$

$$\frac{\begin{array}{c}\mathcal{E} \vdash s_1 \overset{t,i}{\hookrightarrow} v_1 \\ \mathcal{E} \cup \{x \mapsto v_1\}; \mathcal{K} \vdash s_1 \overset{t,i}{\longrightarrow} s_1' \\ \mathcal{E} \cup \{x \mapsto v_1\}; \mathcal{K} \vdash s_2 \overset{t,i}{\longrightarrow} s_2'\end{array}}{\begin{array}{l}\mathcal{E}; \mathcal{K} \vdash \mathsf{let\ snapshot}\ x \leftarrow s_1\ \mathsf{in}\ s_2 \overset{t,i}{\longrightarrow} \\ \qquad \mathsf{let\ snapshot}\ x \leftarrow s_1'\ \mathsf{in}\ s_2'\end{array}} \text{(u5)}$$

$$\frac{\mathcal{E}; \mathcal{K} \vdash s_1 \overset{t,i}{\longrightarrow} s_1' \qquad \mathcal{E}; \mathcal{K} \vdash ev \overset{t,i}{\hookrightarrow\!\!\!\!\rightarrow} \bot, ev'}{\begin{array}{l}\mathcal{E}; \mathcal{K} \vdash s_1\ \mathsf{switch\ on}\ x \leftarrow ev\ \mathsf{in}\ s_2 \overset{t,i}{\longrightarrow} \\ \qquad s_1'\ \mathsf{switch\ on}\ x \leftarrow ev'\ \mathsf{in}\ s_2\end{array}} \text{(u6)}$$

$$\frac{\mathcal{E}; \mathcal{K} \vdash ev \overset{t,i}{\hookrightarrow\!\!\!\!\rightarrow} v_\bot, ev'}{\begin{array}{l}\mathcal{E}; \mathcal{K} \vdash s_1\ \mathsf{switch\ on}\ x \leftarrow ev\ \mathsf{in}\ s_2 \overset{t,i}{\longrightarrow} \\ \qquad s_2[x := v]\ \mathsf{switch\ on}\ x \leftarrow ev'\ \mathsf{in}\ s_2\end{array}} \text{(u7)}$$

$$\frac{\mathcal{E}; \mathcal{K} \cup \{k_j \mapsto \lambda x_j.u_j\} \vdash s \overset{t,i}{\longrightarrow} s'}{\begin{array}{l}\mathcal{E}; \mathcal{K} \vdash \mathsf{let\ continue}\ \{\, k_j\ x_j\ =\ u_j \,\}\ \mathsf{in}\ s \overset{t,i}{\longrightarrow} \\ \qquad \mathsf{let\ continue}\ \{\, k_j\ x_j\ =\ u_j \,\}\ \mathsf{in}\ s'\end{array}} \text{(u8)}$$

$$\frac{\mathcal{E}; \emptyset \vdash s \overset{t,i}{\longrightarrow} s' \qquad \{\, \mathcal{E}; \emptyset \vdash ev_j \overset{t,i}{\hookrightarrow\!\!\!\!\rightarrow} \bot, ev_j' \,\}}{\begin{array}{l}\mathcal{E}; \mathcal{K} \vdash s\ \mathsf{until}\ \langle\, ev_j\ \Rightarrow\ k_j \,\rangle \overset{t,i}{\longrightarrow} \\ \qquad s'\ \mathsf{until}\ \langle\, ev_j'\ \Rightarrow\ k_j \,\rangle\end{array}} \text{(u9)}$$

$$\frac{\{\, \mathcal{E} \vdash ev_j \overset{t,i}{\hookrightarrow} \bot \,\}^{j \in \{1..m-1\}} \qquad \mathcal{E} \vdash ev_m \overset{t,i}{\hookrightarrow} v_\bot}{\begin{array}{l}\mathcal{E}; \mathcal{K} \cup \{k_m \mapsto \lambda x.u\} \vdash \\ \qquad s\ \mathsf{until}\ \langle\, ev_j\ \Rightarrow\ k_j \,\rangle \overset{t,i}{\longrightarrow} u[x := v]\end{array}} \text{(u10)}$$

Figure 3: Operational Semantics for RT-FRP

3.3.3 *The Mechanics of Evaluating and Updating*

The evaluation rule for calls to the base language (rule e3) uses the judgment $\mathcal{E} \vdash e \hookrightarrow v$ to evaluate the λ-term e. An example definition for this judgment is given in the appendix. Lambda terms are left unchanged during updating.

Evaluating the input and time constructs simply returns the current value for the input and the time, respectively. Updating these two constructs leaves them unchanged. Note that, operationally, time is just one of the inputs.

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

Evaluation of let snapshot $x \leftarrow s_1$ in s_2 consists of two stages: first, s_1 is evaluated to get v; second, s_2 is evaluated with x bound to v, yielding the result. Updating the term is done by updating both s_1 and s_2.

The rules for s_1 switch on $x \leftarrow ev$ in s_2 are a bit involved. If the event does not occur, the default signal s_1 is evaluated as usual. Note that the updating rules do *not* update the signal s_2 that is to be (eventually) switched into. This means that signal begins execution only after the event occurs. Moving to the next rule, if the event does occur, we make a new default value consisting of s_2 where x has been

replaced by the current value of the event.

The evaluation of let-continue and until is straightforward with one exception: the value returned by s until $\langle ev_j \Rightarrow k_j \rangle$ does not depend on any of the events $\{ev_j\}$. This behavior is similar to that of switch. The motivation for this design is to allow the user to define signals that react to themselves, such as

$$\text{let snapshot } x \leftarrow s_1 \text{ switch on } y \leftarrow \text{when } (\text{ext } (x > 10)) \text{ in } s_2,$$

which switches to the signal s_2 when the value of itself exceeds 10.

The updating rules for let-continue and until are more involved. For the construct

$$\text{let continue } \{k_j\ x_j = u_j\} \text{ in } s$$

the continuation definitions $\{k_j\ x_j = u_j\}$ are unchanged, and s is executed with the continuation environment extended with the new definitions. For the construct

$$s \text{ until } \langle ev_j \Rightarrow k_j \rangle$$

the events $\langle ev_j \rangle$ are tested one after another until the first occurrence, then the signal evolves to the definition of the corresponding continuation, with its formal parameter replaced by the value of the event. If no event occurs, then we get back the original term with the sub-terms updated.

3.4 What Can Go Wrong?

RT-FRP supports two forms of recursion [6]: *recursive pure signals* as defined by let-snapshot, and *recursive switching* as defined by let-continue and until. This section discusses these two forms and the concrete run-time problems they can cause.

3.4.1 Getting Stuck

To see how untyped programs that use let-snapshot can get stuck, consider this example: evaluating

$$\text{let snapshot } x \leftarrow \text{ext } x \text{ in ext } x$$

requires evaluating ext x in an environment where x is not bound (rule e5). The essence of this problem is that any occurrence of x in the body of the let-snapshot should not be "needed" during evaluation, although it could be used during updating. In fact, the distinction between evaluation and updating exists primarily so that evaluation can be used to "bootstrap" the recursion in an early phase, so that we can give a sensible notion of updating to such expressions.

3.4.2 Needing More Space

The second rule for updating until requires special attention, as it replaces a reactive term by a possibly larger reactive term from the environment, and so could lead to unbounded program size. As an example, the program

$$\text{let continue } \{\, k\ x = s_1 \text{ until } \langle ev \Rightarrow k \rangle \,\} \text{ in } s_2 \text{ until } \langle ev \Rightarrow k \rangle$$

becomes larger when ev occurs, if the size of s_1 is larger than that of s_2.

4. PROPERTIES OF RT-FRP

In this section we prove the basic resource-boundedness properties of RT-FRP, namely, that the time and space in each execution step is bounded:

THEOREM 1 (MAIN). *For any closed and well-typed program, we know that*

1. *its single-step execution terminates and preserves type, and*
2. *there is a bound for the time and space it needed during its execution.*

PROOF. The proof of the first part is a special instance of Lemma 4. The proof of the second part requires formalizing notions of cost and showing that they are bounded during the execution of any program. □

In the rest of this section, we present the technical details required to establish this result.

4.1 Type Preservation and Termination

4.1.1 Compatibility

In order to express the type safety property concisely, we must have a concise way for expressing the necessary constraints on the environments involved in the statement of the properties. In general, we will need to assume that a given value environment is consistent with a given type context. We say an environment $\mathcal{E} \equiv \{x_j \mapsto v_j\}^{j \in J}$ is *compatible* with a context $\Gamma \equiv \{x_j : g_j\}^{j \in J}$, and define $\Gamma \vdash \mathcal{E}$ as follows:

$$\frac{\left\{\{x_j : g_j\}^{j \in J} \vdash_\lambda v_i : g_i\right\}^{i \in J}}{\{x_j : g_j\}^{j \in J} \vdash \{x_j \mapsto v_j\}^{j \in J}}.$$

Similarly, we say $\mathcal{K} \equiv \{k_j \mapsto \lambda x_j.u_j\}^{j \in J}$ is compatible with Γ and $\Delta \equiv \{k_j : g_j \to g'_j\}^{j \in J}$, and define $\Gamma; \Delta \vdash \mathcal{K}$ as follows:

$$\frac{\left\{\Gamma \cup \{x_i : \uparrow g_i\};\ \{k_j : g_j \to g'_j\}^{j \in J} \vdash_{\mathsf{S}} u_i : g'_i\right\}^{i \in J}}{\Gamma;\ \{k_j : g_j \to g'_j\}^{j \in J} \vdash \{k_j \mapsto \lambda x_j.u_j\}^{j \in J}}.$$

It is easy to show the above definitions enjoy the following forms of weakening:

LEMMA 2 (BASIC PROPERTIES).

$$\frac{}{\Gamma; \emptyset \vdash \emptyset} \qquad \frac{\Gamma \vdash \mathcal{E} \quad \Gamma \vdash_\lambda v : g}{\Gamma \cup \{x : g\} \vdash \mathcal{E} \cup \{x \mapsto v\}}$$

$$\frac{\Gamma; \Delta \vdash \mathcal{K} \quad \Gamma; \Delta' \vdash \mathcal{K}'}{\Gamma; \Delta \cup \Delta' \vdash \mathcal{K} \cup \mathcal{K}'} \qquad \frac{\Gamma \cup \{x : g_1\}; \Delta \vdash s : g_2}{\uparrow\Gamma \cup \{x : g_1\}; \Delta \vdash s : g_2}$$

4.1.2 Assumptions about Base Language

In order to prove the key properties of RT-FRP, we must be explicit about our assumptions about the base language. We assume three key properties of the base language: First, that evaluation terminates and preserves typing. Second, that values of lifted type be of the two obvious forms distinguished at the head. Third, that the type system enjoys substitutivity. These requirements are formalized as follows:

$$\frac{\downarrow\Gamma \vdash \mathcal{E} \quad \downarrow\Gamma \vdash_\lambda e : g}{\exists v.\, (\mathcal{E} \vdash e \hookrightarrow v \quad \downarrow\Gamma \vdash_\lambda v : g)}$$

$$\frac{\downarrow\Gamma \vdash_\lambda v : g_\perp}{v \equiv \perp \text{ or } \exists v'.\, (v \equiv v'_\perp \quad \downarrow\Gamma \vdash_\lambda v' : g)}$$

$$\frac{\downarrow\Gamma \vdash_\lambda e : g \quad \downarrow\Gamma \cup \{x : g\} \vdash_\lambda e' : g'}{\downarrow\Gamma \vdash_\lambda e'[x := e] : g'}$$

4.1.3 Substitutivity

Using the substitutivity assumption about the base language, it is now possible to establish the following lemma for RT-FRP:

LEMMA 3 (SUBSTITUTION). *Whenever* $\downarrow\Gamma \vdash_\lambda v : g$ *and* $\Gamma \cup \{x : \uparrow g\}; \Delta \vdash_S s : g'$, *we have* $\Gamma;\ \Delta \vdash_S s[x := v] : g'$.

4.1.4 Main Lemma

LEMMA 4 (TYPE PRESERVATION AND TERMINATION). *For all* Γ, Δ, $\mathcal{E}$, $\mathcal{K}$, s, *and* g,

1. *if* $\downarrow\Gamma \vdash \mathcal{E}$ *and* $\Gamma;\ \Delta \vdash_S s : g$, *then there exists a value* v *such that* $\mathcal{E} \vdash s \overset{t,i}{\hookrightarrow} v$ *and* $\downarrow\Gamma \vdash_\lambda v : g$, *and*
2. *if* $\Updownarrow\Gamma \vdash \mathcal{E}$, *and* $\Gamma;\Delta \vdash \mathcal{K}$ *and* $\Gamma;\ \Delta \vdash_S s : g$, *then there exists a term* s' *such that* $\mathcal{E};\mathcal{K} \vdash s \xrightarrow{t,i} s'$ *and* $\Gamma;\ \Delta \vdash_S s' : g$.

PROOF. The proof of both parts is by induction over the height of the typing derivation. The proof of the first part uses the assumptions about the base language to establish the soundness of the ext construct. The proof of the second part uses the substitutivity property, and the first part of the lemma. □

After this property is established, it is easy to see that not only is evaluation always terminating, but it is also deterministic whenever the base language is also deterministic.

4.2 Resource Boundedness

Having established that the language is terminating, we now come to the proof of the second part of our main theorem, namely, time- and space-boundedness. As a measure of the time and space needed for executing a program, we will use term size at run-time (modulo the size of base language terms). This measure is reasonable because of two observations: First, the size of a derivation tree is bounded by the term size. Second, the derivation is syntax directed, and so the time needed to propagate the information around a rule is bounded by the size of the term (assuming a naive implementation, where each term is copied in full). Thus, our focus will be on showing that there exists a measure on programs that does not increase during run-time.

We formally define the *size of a term* s, written $|s|$, to be the number of constructors, base language expressions, and continuations in the term:

$$\begin{aligned} |\mathsf{input}| &= 1 \\ |\mathsf{time}| &= 1 \\ |\mathsf{ext}\ e| &= 2 \\ |\mathsf{delay}\ v\ s| &= 2 + |s| \\ |\mathsf{let\ snapshot}\ x \leftarrow s_1\ \mathsf{in}\ s_2| &= 2 + |s_1| + |s_2| \end{aligned}$$

$$\begin{aligned} & |\mathsf{let\ continue}\ \{\, k_j\ x_j = u_j \,\}^{j\in\{1..n\}}\ \mathsf{in}\ s| \\ = \ & 1 + 2n + \Sigma_{j=1}^{n}|u_j| + |s| \end{aligned}$$

$$\begin{aligned} & |s\ \mathsf{until}\ \langle\, ev_j \Rightarrow k_j \,\rangle^{j\in\{1..n\}}| \\ = \ & 1 + n + |s| + \Sigma_{j=1}^{n}|ev_j|. \end{aligned}$$

As mentioned earlier, the size of a term can grow at run-time. However, we can still show that there exists a bound for the size of the term at run-time. The following function (on continuation environments and terms) will be used to define such a bound:

$$\|\{k_j \mapsto \lambda x_j.u_j\}\| = \max\left(\{0\} \cup \left\{\|u_j\|_0\right\}\right)$$

$$\begin{aligned} \|\mathsf{input}\|_m &= 1 \\ \|\mathsf{time}\|_m &= 1 \\ \|\mathsf{ext}\ e\|_m &= 2 \\ \|\mathsf{delay}\ v\ s\|_m &= 2 + \|s\|_m \\ \|\mathsf{let\ snapshot}\ x \leftarrow s_1\ \mathsf{in}\ s_2\|_m &= 2 + \|s_1\|_m + \|s_2\|_m \end{aligned}$$

$$\begin{aligned} & \|\mathsf{let\ continue}\ \{\, k_j\ x_j = u_j \,\}^{j\in\{1..n\}}\ \mathsf{in}\ s\|_m \\ = \ & 1 + 2n + \Sigma_{j=1}^{n}|u_j| + \|s\|_{\max\{\|\{k_j\mapsto\lambda x_j.u_j\}^{j\in\{1..n\}}\|,\,m\}} \end{aligned}$$

$$\begin{aligned} & \|s\ \mathsf{until}\ \langle\, ev_j \Rightarrow k_j \,\rangle^{j\in\{1..n\}}\|_m \\ = \ & \max\left\{1 + n + \|s\|_0 + \Sigma_{j=1}^{n}\|ev_j\|_0, m\right\}. \end{aligned}$$

where m in $\|s\|_m$ is the size bound for the free continuations in s.

In order to establish the bound, we must always consider a term in the context of a particular continuation environment. Thus we define the *term size bound* for a term s under continuation environment $\mathcal{K}$ to be $\|s\|_{\|\mathcal{K}\|}$. First, it is useful to know that this measure is an upper bound for term size $|s|$:

LEMMA 5. For all m and s, $|s| \le \|s\|_m$.

PROOF. By induction on s. □

Now we can show that even though term size can grow during execution, its size bound as defined by $\|-\|_{\|\mathcal{K}\|}$ does not:

LEMMA 6. (*Bound Preservation*) $\mathcal{E};\mathcal{K} \vdash s \xrightarrow{t,i} s'$ implies $\|s'\|_{\|\mathcal{K}\|} \le \|s\|_{\|\mathcal{K}\|}$.

PROOF. The proof is by induction on the derivation for $\mathcal{E};\mathcal{K} \vdash s \xrightarrow{t,i} s'$. In what follows, we let $m = \|\mathcal{K}\|$.

1. The last rule used in the derivation is u1, u2, or u3. In this case $s \equiv s'$. Hence $\|s'\|_m \le \|s\|_m$.
2. The last rule in the derivation is u4. Then $s \equiv \mathsf{delay}\ v\ s_1$, $s' \equiv \mathsf{delay}\ v'\ s_1'$, and $\mathcal{E};\mathcal{K} \vdash s_1 \xrightarrow{t,i} s_1'$. By induction hypothesis, $\|s'\|_m = 2 + \|s_1'\|_m \le 2 + \|s_1\|_m = \|s\|_m$.
3. The last rule is u5, or u9. The proof is by induction hypothesis.
4. The last rule is u8. We know that
$$\begin{aligned} s &\equiv \mathsf{let\ continue}\ \{\, k_j\ x_j = u_j \,\}\ \mathsf{in}\ s_1 \\ s' &\equiv \mathsf{let\ continue}\ \{\, k_j\ x_j = u_j \,\}\ \mathsf{in}\ s_1' \\ \text{and } &\mathcal{E};\mathcal{K} \cup \{k_j \mapsto \lambda x_j.u_j\} \vdash s_1 \xrightarrow{t,i} s_1' \end{aligned}$$
Note that
$$\begin{aligned} & \|\mathcal{K} \cup \{k_j \mapsto \lambda x_j.u_j\}\| \\ = \ & \max\{\max\{\|u_j\|_0\},\, m\} \\ = \ & \max\{\|\{k_j \mapsto \lambda x_j.u_j\}\|,\, m\} \end{aligned}$$
Hence by induction hypothesis,
$$\begin{aligned} & \|s'\|_m \\ = \ & 1 + 2n + \Sigma_{j=1}^{n}|u_j| + \|s_1'\|_{\max\{\|\{k_j\mapsto\lambda x_j.u_j\}\|,\,m\}} \\ \le \ & 1 + 2n + \Sigma_{j=1}^{n}|u_j| + \|s_1\|_{\max\{\|\{k_j\mapsto\lambda x_j.u_j\}\|,\,m\}} \\ = \ & \|s\|_m \end{aligned}$$

5. The last rule is u10. In this case,

$$s \equiv s_1 \text{ until } \langle\, ev_j \Rightarrow k_j \,\rangle^{j\in\{1..n\}},$$

and $s' \equiv u[x := v]$, where $(k \mapsto \lambda x.u) \in \mathcal{K}$ for some k and x.

$$\begin{array}{rcl} & \|s'\|_m & \\ = & \|u\|_m & \\ = & \max\{\, 1 + n + \|s_2\|_0 + \Sigma_j \|ev_j\|_0, m \,\} & \\ & \quad \text{where } u \equiv s_2 \text{ until } \langle ev_j \Rightarrow k_j \rangle^{j\in\{1..n\}} & \\ = & \max\{\, 1 + n + \|s_2\|_0 + \Sigma_j \|ev_j\|_0, 0, m \,\} & \\ = & \max\{\|u\|_0, m\} & \\ = & m \qquad (\text{since } (k \mapsto \lambda x.u) \in \mathcal{K}) & \\ \leq & \|s\|_m & \end{array}$$

And we are done. □

5. DISCUSSION AND RELATED WORK

Several languages have been proposed around the *synchronous data-flow* notion of computation. The general-purpose functional language Lucid [38] is an example of this style of language. More relevant to the present paper are the languages Signal [10], Lustre [3], and Esterel [1, 2], which were specifically designed for control of real-time systems. In Signal, the central notion a *signal*, a time-ordered sequence of values. This is analogous to the sequence of values generated in the execution of an 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. Compilers are available that translate Esterel programs into finite state machines or digital circuits for embedded applications. 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.

None of these synchronous data flow languages seem to have considered recursion. Synchronous Kahn networks [4] extended these proposals with recursion and higher-order programming, yielding a large increase in expressive power. The downside of such extension is that resource-boundedness is no longer guaranteed. In RT-FRP we have shown that, using some syntactic restrictions and a type system, it is possible to achieve such a bound.

Mycroft and Sharp [25] develop a statically allocated parallel functional language for the specification of hardware. Their language allows recursion, and restricts recursive calls to tail calls. However, they use an explicit notion of a syntactic context to specify what is a tail call, and restrict recursive functions. In our work, we have integrated this restriction into the type system. It will be interesting to see an integration is also possible in their setting.

A hybrid automaton [11, 22] 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. Although we have not formally established such a result, we expect that, in the limit as the maximum sampling period goes to zero, an RT-FRP program can implement a hybrid automaton.

CML (Concurrent ML) formalizes synchronous operations as first-class, purely functional, values called *events* [32]. FRP's event combinators ".|." and "==>" correspond to CML's `choose` and `wrap` functions. There is a basic difference, 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.

Previous work on FRP defined an implementation semantics that uses streams (potentially infinite sequences). This semantics is used as the basis for most current FRP implementations. The fundamental difference in our work is that we define sequences of *whole program execution* and not just sequences of values. As such our work presents an explicit model of the mechanics of executing a subset of FRP programs, and shows how this model can be used to establish guarantees that are relevant to embedded systems applications.

It may be surprising to a reader familiar with FRP that some of its basic constructs are missing in RT-FRP. Many of these constructs, however, are definable in RT-FRP. Indeed,we have already defined when, several lift operators, and integral. In addition, FRP's never and once operators generate event values that never occur and occur exactly once, respectively. They can be expressed in RT-FRP as follows:

$$\begin{array}{rcl} [\![\text{never}]\!] & \equiv & \text{ext } \bot \\ [\![\text{once } ev]\!] & \equiv & \text{let snapshot } x_2 \leftarrow ev \text{ in} \\ & & \quad \text{let snapshot } x_1 \leftarrow \text{delay } \bot \\ & & \quad\quad (\text{ext } (\text{if } x_1 = \bot \text{ then } x_2 \text{ else } x_1)) \text{ in} \\ & & \quad\quad\quad \text{ext } (\text{if } x_1 = \bot \text{ then } x_2 \text{ else } \bot) \end{array}$$

FRP's till operator, similar to the until construct in RT-FRP, can be translated as follows:

$$[\![s_1 \text{ till } ev \text{ then } s_2]\!] \equiv s_1 \text{ switch on } x \leftarrow \text{once } ev \text{ in } s_2$$

Frappé [5] is an efficient implementation of a subset of FRP in Java. The exact subset has not yet been formally described or characterized. With suitable extensions, RT-FRP could serve as a model for Frappé. In addition, a number of interesting evaluation strategies (called push, pull, and hybrid) have been explored in the context of Frappé[2] and we are interested in formalizing these models and studying their properties.

This paper presented a model of machines that work in an environment that provides a type of stimulus. Although we have discussed how a base language can be invoked from within FRP, we have not proposed a method for combining or integrating the kind of machines that we have presented here. Ongoing work at Yale by Antony Courtney, Henrik Nilsson and John Peterson suggests that Hughes' arrows [17] provide a natural mechanism for modeling signals that are explicitly parameterized by an input type[3]. We expect that this approach can be used as a bases for a language for combining RT-FRP machines. We are particularly interested in

[2]Antony Courtney, personal communication, December 2000.

[3]Personal communication, May 2001.

seeing if this approach can be used to model asynchronous systems of synchronous processes.

In this paper, we have chosen to focus on the issue of bounded resources in the presence of recursion. Ultimately, however, we are interested in more sophisticated models for real-time systems, where resources are allocated according to their priority (see Kieburtz [20] for a nice account from the Haskell point of view). Signal, Lustre, and synchronous Kahn networks [4] account for this via a clock calculus. While this technique may apply directly to RT-FRP, this still remains to be established.

There are many connections between the semantics here and the semantics of multi-stage languages. Evaluating recursive let-snapshot declarations requires evaluation under a binder. This problem arises constantly in the context of multi-level and multi-stage languages [35]. Our approach the treatment of this problem, namely, using the exportability annotations in the type system, is inspired by the work on multi-stage languages [23, 37]. Our evaluation and updating functions are also analogous to the evaluation and rebuilding functions of multi-stage language [35]. It will be interesting to see if this analogy continues to hold as we continue the development of RT-FRP.

Finally, we are interested in the study of cost-preserving notions of equivalence in RT-FRP along the lines developed by Sands [34].

Acknowledgements: We would like to thank the members of the FLINT, FRP, and PacSoft groups, Arvind Krishnamurthy, Simon Peyton-Jones and the anonymous reviewers for many comments on drafts of this paper that improved the final version. We would especially like to thank Antony Courtney, Henrik Nilsson, and John Peterson for valuable discussions and for sharing with us their work and ideas in relation to FRP.

APPENDIX

A. BASE LANGUAGE SEMANTICS

Figure 4 gives the operational semantics of the base language used in this paper.

B. 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] Paul Caspi and Marc Pouzet. Synchronous Kahn networks. *ACM SIGPLAN Notices*, 31(6):226–238, 1996.

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

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

[7] Roberto Di Cosmo. *Isomorphisms of Types: from λ-calculus to information retrieval and language design*. Progress in Theoretical Computer Science. Birkhäuser, 1995.

[8] 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.

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

[10] 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.

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

[12] Martin Hofmann. A type system for bounded space and functional in-place update. In *European Symposium on Programming (ESOP)*, Lecture Notes in Computer Science. Springer-Verlag, 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. Building domain specific embedded languages. *ACM Computing Surveys*, 28A:(electronic), December 1996.

[15] 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.

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

[17] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.

[18] John Hughes and Lars Pareto. Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP-99)*, volume 34.9 of *ACM Sigplan Notices*, pages 70–81, N.Y., September 27–29 1999. ACM Press.

[19] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In Guy L. Steele Jr, editor, *In proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, volume 23, St Petersburg, Florida, 1996. ACM Press.

[20] Richard Kieburtz. Real-time reactive programming for embedded controllers. Available from author's home page, March 2001.

[21] Richard B. Kieburtz. Implementing closed domain-specific languages. In *[36]*, pages 1–2, 2000.

$$\frac{}{\mathcal{E} \cup \{x \mapsto v\} \vdash x \hookrightarrow v} \qquad \frac{}{\mathcal{E} \vdash c \hookrightarrow c} \qquad \frac{}{\mathcal{E} \vdash () \hookrightarrow ()} \qquad \frac{\mathcal{E} \vdash e_1 \hookrightarrow v_1 \qquad \mathcal{E} \vdash e_2 \hookrightarrow v_2}{\mathcal{E} \vdash (e_1, e_2) \hookrightarrow (v_1, v_2)} \qquad \frac{\mathcal{E} \vdash e \hookrightarrow v}{\mathcal{E} \vdash e_\perp \hookrightarrow v_\perp}$$

$$\frac{}{\mathcal{E} \vdash \perp \hookrightarrow \perp} \qquad \frac{}{\mathcal{E} \vdash \lambda x.e \hookrightarrow \lambda x.e} \qquad \frac{\mathcal{E} \vdash e_1 \hookrightarrow \lambda x.e \qquad \mathcal{E} \vdash e[x := e_2] \hookrightarrow v}{\mathcal{E} \vdash e_1\ e_2 \hookrightarrow v}$$

Figure 4: Operational Semantics of a Functional Base Language

[22] 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.

[23] Eugenio Moggi, Walid Taha, Zine El-Abidine Benaissa, and Tim Sheard. An idealized MetaML: Simpler, and more expressive. In *European Symposium on Programming (ESOP)*, volume 1576 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, 1999.

[24] A. S. Murawski and C.-H. L. Ong. Can safe recursion be interpreted in light logic? In *Second International Workshop on Implicit Computational Complexity*, Santa Barbara, June 200.

[25] Alan Mycroft and Richard Sharp. A statically allocated parallel functional language. In *Automata, Languages and Programming*, pages 37–48, 2000.

[26] Oregon Graduate Institute Technical Reports. P.O. Box 91000, Portland, OR 97291 -1000,USA. Available online from ftp://cse.ogi.edu/pub/tech-reports/README.html. Last viewed August 1999.

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

[28] 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.

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

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

[31] 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.

[32] 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.

[33] 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.

[34] David Sands. A naïve time analysis and its theory of cost equivalence. *Journal of Logic and Computation*, 5(4):495–541, 1995.

[35] Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Available from [26].

[36] Walid Taha, editor. *Semantics, Applications, and Implementation of Program Generation*, volume 1924 of *Lecture Notes in Computer Science*, Montréal, 2000. Springer-Verlag.

[37] Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. Multi-stage programming: Axiomatization and type-safety. In *25th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 1443 of *Lecture Notes in Computer Science*, pages 918–929, Aalborg, 1998.

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

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