Functional Reactive Animation

Conal Elliott
Microsoft Research
Graphics Group
`conal@microsoft.com`

Paul Hudak
Yale University
Dept. of Computer Science
`paul.hudak@yale.edu`

Abstract

Fran (Functional Reactive Animation) is a collection of data types and functions for composing richly interactive, multimedia animations. The key ideas in Fran are its notions of *behaviors* and *events*. Behaviors are time-varying, reactive values, while events are sets of arbitrarily complex conditions, carrying possibly rich information. Most traditional values can be treated as behaviors, and when images are thus treated, they become animations. Although these notions are captured as data types rather than a programming language, we provide them with a denotational semantics, including a proper treatment of real time, to guide reasoning and implementation. A method to effectively and efficiently perform *event detection* using *interval analysis* is also described, which relies on the partial information structure on the domain of event times. Fran has been implemented in Hugs, yielding surprisingly good performance for an interpreter-based system. Several examples are given, including the ability to describe physical phenomena involving gravity, springs, velocity, acceleration, etc. using ordinary differential equations.

1 Introduction

The construction of richly interactive multimedia animations (involving audio, pictures, video, 2D and 3D graphics) has long been a complex and tedious job. Much of the difficulty, we believe, stems from the lack of sufficiently high-level abstractions, and in particular from the failure to clearly distinguish between *modeling* and *presentation*, or in other words, between *what* an animation is and *how* it should be presented. Consequently, the resulting programs must explicitly manage common implementation chores that have nothing to do with the content of an animation, but rather its presentation through low-level display libraries running on a sequential digital computer. These implementation chores include:

- stepping forward discretely in time for simulation and for frame generation, even though animation is conceptually continuous;
- capturing and handling sequences of motion input events, even though motion input is conceptually continuous;
- time slicing to update each time-varying animation parameter, even though these parameters conceptually vary in parallel; and

By allowing programmers to express the "what" of an interactive animation, one can hope to then automate the "how" of its presentation. With this point of view, it should not be surprising that a set of richly expressive recursive data types, combined with a declarative programming language, serves comfortably for modeling animations, in contrast with the common practice of using imperative languages to program in the conventional hybrid modeling/presentation style. Moreover, we have found that non-strict semantics, higher-order functions, strong polymorphic typing, and systematic overloading are valuable language properties for supporting modeled animations. For these reasons, Fran provides these data types in the programming language Haskell [9].

Advantages of Modeling over Presentation

The benefits of a modeling approach to animation are similar to those in favor of a functional (or other declarative) programming paradigm, and include clarity, ease of construction, composability, and clean semantics. But in addition there are application-specific advantages that are in some ways more compelling, painting the picture from a software engineering and end-user perspective. These advantages include the following:

- *Authoring.* Content creation systems naturally construct models, because the end users of such systems think in terms of models and typically have neither the expertise nor interest in programming presentation details.
- *Optimizability.* Model-based systems contain a presentation sub-system able to render any model that can be constructed within the system. Because higher-level information is available to the presentation sub-system than with presentation programs, there are many more opportunities for optimization.
- *Regulation.* The presentation sub-system can also more easily determine level-of-detail management, as well as sampling rates required for interactive animations, based on scene complexity, machine speed and load, etc.

To appear the International Conference on Functional Programming, June 1997, Amsterdam.

- *Mobility and safety.* The platform independence of the modeling approach facilitates the construction of mobile applications that are provably safe in World Wide Web applications.

The Essence of Modeling Our goal in this paper is to convey the essence of a modeling approach to reactive animations as captured in Fran, as summarized in the following four concepts:

1. **Temporal modeling.** Values, called *behaviors*, that vary over continuous time are the chief values of interest. Behaviors are first-class values, and are built up compositionally; concurrency (parallel composition) is expressed naturally and implicitly. As an example, the following expression evaluates to an animation (i.e., an image behavior) containing a circle over a square. At time t, the circle has size $\sin t$, and the square has size $\cos t$.

```
bigger (sin time) circle  `over`
bigger (cos time) square
```

2. **Event modeling.** Like behaviors, *events* are first-class values. Events may refer to happenings in the real world (e.g. mouse button presses), but also to predicates based on animation parameters (e.g. proximity or collision). Moreover, such events may be combined with others, to an arbitrary degree of complexity, thus factoring complex animation logic into semantically rich, modular building blocks. For example, the event describing the first left-button press after time `t0` is simply `lbp t0`; one describing time squared being equal to 5 is just:

```
predicate (time^2 == 5) t0
```

and their logical disjunction is just:

```
lbp t0 .|. predicate (time^2 == 5) t0
```

3. **Declarative reactivity.** Many behaviors are naturally expressed in terms of reactions to events. But even these "reactive behaviors" have declarative semantics in terms of temporal composition, rather than an imperative semantics in terms of the state changes often employed in event-based formalisms. For example, a color-valued behavior that changes cyclically from red to green with each button press can be described by the following simple recurrence:

```
colorCycle t0 =
  red   `untilB` lbp t0 *=> \t1 ->
  green `untilB` lbp t1 *=> \t2 ->
  colorCycle t2
```

(In Haskell, identifiers are made into infix operators by backquotes, as in `` b `untilB` e ``. Also, infix operators can be made into identifiers by enclosing them in parentheses, as in `(+) x y`. Lambda abstractions are written as "`\` *vars* `->` *exp*".)

4. **Polymorphic media.** The variety of time-varying media (images, video, sound, 3D geometry) and parameters of those types (spatial transformations, colors, points, vectors, numbers) have their own type-specific operations (e.g. image rotation, sound mixing, and numerical addition), but fit into a common framework of behaviors and reactivity. For instance, the "`untilB`" operation used above is polymorphic, applying to all types of time-varying values.

Our Contributions We have captured the four features above as a collection of recursive data types, functions, and primitive graphics routines in a system that we call Fran, for *Functional Reactive Animation.* Although these data types and functions do not form a programming language in the usual sense, we provide them with a formal denotational semantics, including a proper treatment of real time, to allow precise, implementation-independent reasoning. This semantics includes a CPO of real time, whose approximate elements allow us to reason about events before they occur. As would be true of a new programming language, the denotational semantics has been extremely useful in designing Fran. All of our design decisions begin with an understanding of the formal semantics, followed by reflecting the semantics in the implementation. (The semantics is given in Section 2.)

Perhaps the most novel aspect of Fran is its *implicit treatment of time.* This provides a great deal of expressiveness to the multimedia programmer, but also presents interesting challenges with respect to both formal semantics and implementation. In particular, events may be specified in terms of boolean functions of continuous time. These functions may become true for arbitrarily brief periods of time, even instantaneously, and so it is challenging for an implementation to *detect* these events. We solve this problem with a robust and efficient method for *event detection* based on *interval analysis.* (Implementation issues are discussed in Section 4.)

Specifically, the nature of an event can be exploited to eliminate search over intervals of time in which the event provably does not occur, and focus intead on time intervals in which the event may occur. In some cases, such as a collection of bouncing balls, exact event times may be determined analytically. In general and quite frequently, however, analytic techniques fail to apply. We describe intead an algorithm for event detection based on *interval analysis* and relate it to the partial information structure on the CPO of event times.

2 The Formal Semantics of Fran

The two most fundamental notions in Fran are *behaviors* and *events.* We treat them as a pair of mutually recursive polymorphic data types, and specify operations on them via a denotational semantics. (The "media types" we often use with events and behaviors will be treated formally in a later paper; but see also [7].)

2.1 Semantic Domains

The abstract domain of time is called *Time.* The abstract domains of polymorphic behaviors (α-behaviors) and polymorphic events (α-events) are denoted $Behavior_\alpha$ and $Event_\alpha$, respectively.

Most of our domains (integers, booleans, etc.) are standard, and require no explanation. The *Time* domain, however, requires special treatment, since we wish values of time to include *partial elements*. In particular, we would like to know that a time is "at least" some value, even if we don't yet know exactly what the final value will be. To make this notion precise, we define a domain (pointed CPO) of time as follows:

Denote the set of real numbers as $\Re$, and include in that set the elements ∞ and $-\infty$. This set comes equipped with the standard arithmetic ordering $\leq$, including the fact that $-\infty \leq x \leq \infty$ for all $x \in \Re$.

Now define $Time = \Re + \Re$, where elements in the second "copy" of $\Re$ are distinguished by prefixing them with $\geq$, as in ${}_{\geq}42$, which should be read: "at least 42." Then define $\bot_{Time} = {}_{\geq}(-\infty)$, and the domain (i.e. information) ordering on *Time* by:

$$\begin{array}{rcll} x & \sqsubseteq & x, & \forall x \in \Re \\ {}_{\geq}x & \sqsubseteq & y \quad \textit{if } x \leq y, & \forall x, y \in \Re \\ {}_{\geq}x & \sqsubseteq & {}_{\geq}y \quad \textit{if } x \leq y, & \forall x, y \in \Re \end{array}$$

It is easy to see that $\bot_{Time}$ is indeed the bottom element. Also note that a limit point y is just the LUB of the set of partial elements ("pre-times") that approximate it:

$$y = \bigsqcup \{ {}_{\geq}x \mid x \leq y \}$$

Since the ordering on the domain *Time* is chain-like, and every such chain has a LUB (recall that $\Re$ has a top element ∞), the domain *Time* is a pointed CPO. This fact is necessary to ensure that recursive definitions are well defined.

Elements of *Time* are most useful for approximating the time at which an event occurs. That is, an event whose time is approximately ${}_{\geq}t$ is one whose actual time of occurrence is greater than t. Note that the time of an event that *never* occurs is just ∞, the LUB of $\Re$.

Finally, we extend the definition of arithmetic $\leq$ to all of *Time* by defining its behavior across the subdomains as follows:

$$x \;\leq\; {}_{\geq}y \quad \textit{if} \quad x \leq y$$

This can be read: "The time x is less than or equal to a time that is at least y, if $x \leq y$." (${}_{\geq}x \leq y$ and ${}_{\geq}x \leq {}_{\geq}y$ are undefined.) We can easily show that this extended function of type $Time \to Time \to Bool$ is continuous with respect to $\sqsubseteq$. It is used in various places in the semantics that follows.

Semantic Functions We define an interpretation of α-behaviors as a function from time to α-values, producing the value of a behavior b *at* a time t.

$$\mathbf{at} \;:\; Behavior_\alpha \to Time \to \alpha$$

Next, we define an interpretation on α-events as simply non-strict $Time \times \alpha$ pairs, describing the time and information associated with an *occ*urrence of the event.

$$\mathbf{occ} \;:\; Event_\alpha \to Time \times \alpha$$

Now that we know the semantic domains we are working with, we present the various behavior and event combinators with their formal interpretations.

2.2 Semantics of Behaviors

Behaviors are built up from other behaviors, static (non-time-varying) values, and events, via a collection of constructors (combinators).

Time. The simplest primitive behavior is *time*, whose semantics is given by:

$$\begin{array}{l} time : Behavior_{Time} \\ \mathbf{at}[\![time]\!]t = t \end{array}$$

Thus $\mathbf{at}[\![time]\!]$ is just the identity function on *Time*.

Lifting. We would like to have a general way of "lifting" functions defined on static values to analogous functions defined on behaviors. This lifting is accomplished by a (conceptually infinite) family of operators, one for each arity of functions.

$$\begin{array}{l} lift_n : (\alpha_1 \to \ldots \to \alpha_n \to \beta) \to \\ \qquad Behavior_{\alpha_1} \to \ldots \to Behavior_{\alpha_n} \to Behavior_\beta \\ \mathbf{at}[\![lift_n\ f\ b_1 \ldots b_n]\!]t = f\ (\mathbf{at}[\![b_1]\!]t) \ldots (\mathbf{at}[\![b_n]\!]t) \end{array}$$

Note that constant value lifting is just $lift_0$.

Notational aside: In practice, lifting is needed quite frequently, so it would be inconvenient to make lifting explicit everywhere. It is more desirable to use familiar names like "sin", "cos", "+", "*", and even literals like "3" and "*blue*", to refer to *lifted* versions of their standard interpretations. For instance, a literal such as 42 should behave as the constant behavior "$lift_0$ 42," and a summation on behaviors such as "$b_1 + b_2$" should behave as "$lift_2\ (+)\ b_1\ b_2$", where "(+)" is curried addition. In our implementation of Fran in Haskell, type classes help considerably here, since the `Num` class provides a convenient implicit mechanism for lifting numerical values. In particular, with a suitable instance declaration, we achieve exactly the interpretations above, even for literal constants.

Time transformation. A *time transform* allows the user to transform local time-frames. It thus supports what we call *temporal modularity* for behaviors of all types. (Similarly, 2D and 3D transforms support *spatial modularity* in image and geometry behaviors.)

$$\begin{array}{l} timeTransform : Behavior_\alpha \to Behavior_{Time} \to Behavior_\alpha \\ \mathbf{at}[\![timeTransform\ b\ tb]\!] = \mathbf{at}[\![b]\!] \circ \mathbf{at}[\![tb]\!] \end{array}$$

Thus note that *time* is an identity for *timeTransform*:

$$timeTransform\ b\ time = b$$

As examples of the use of time transformation in Fran, the expression:

$$timeTransform\ b\ (time/2)$$

slows down the animation b by a factor of 2, whereas:

$$timeTransform\ b\ (time - 2)$$

delays it by 2 seconds.

Integration. Integration applies to real-valued as well as 2D and 3D vector-valued behaviors, or more generally, to vector-spaces (with limits). Borrowing from Haskell's type class notation to classify vector-space types:

$$\begin{array}{l} integral : VectorSpace\ \alpha \Rightarrow Behavior_\alpha \rightarrow Time \rightarrow Behavior_\alpha \\ \mathbf{at}[\![integral\, b\ t_0]\!]t = \int_{t_0}^{t} \mathbf{at}[\![b]\!] \end{array}$$

Integration allows the specification of velocity behaviors, and, if used twice, acceleration behaviors. For example, if the velocity of a moving ball is given by behavior b (perhaps a constant velocity, perhaps not), then its position relative to starting time t_0 is given by $integral\, b\ t_0$. This provides a natural means to express physics-based animations, examples of which are given in Section 3.

Reactivity. The key interplay in Fran is that between behaviors and events, and is what makes behaviors reactive. Specifically, the behavior b *untilB* e exhibits b's behavior until e occurs, and then switches to the behavior associated with e. More formally:

$$\begin{array}{l} untilB : Behavior_\alpha \rightarrow Event_{Behavior_\alpha} \rightarrow Behavior_\alpha \\ \mathbf{at}[\![b\ untilB\ e]\!]t = \mathbf{if}\ t \leq t_e\ \mathbf{then}\ \mathbf{at}[\![b]\!]t\ \mathbf{else}\ \mathbf{at}[\![b']\!]t \\ \quad \mathbf{where}\ (t_e, b') = \mathbf{occ}[\![e]\!] \end{array}$$

Note that the inequality used here, $t \leq t_e$, is the one defined in Section 2.1. In the next section examples of reactivity are given for each of the various kinds of events.

2.3 Semantics of Events

Event handling. In order to give examples using specific kinds of events, we first describe the notion of *event handlers*, which are applied to the time and data associated with an event using the following operator:

$$\begin{array}{l} (+\!\!\Rightarrow) : Event_\alpha \rightarrow (Time \rightarrow \alpha \rightarrow \beta) \rightarrow Event_\beta \\ \mathbf{occ}[\![e +\!\!\Rightarrow f]\!] = (t_e,\ f\ t_e\ x) \\ \quad \mathbf{where}\ (t_e, x) = \mathbf{occ}[\![e]\!] \end{array}$$

For convenience, we will also make use of the following derived operations, which ignore the time or the data or both:

$$\begin{array}{l} (\Longrightarrow) : Event_\alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow Event_\beta \\ (*\!\!\Rightarrow) : Event_\alpha \rightarrow (Time \rightarrow \beta) \rightarrow Event_\beta \\ (-\!\!\Rightarrow) : Event_\alpha \rightarrow \beta \rightarrow Event_\beta \\ \begin{array}{lcl} ev \Longrightarrow g & = & ev +\!\!\Rightarrow \lambda t\ x.\ g\ x \\ ev *\!\!\Rightarrow h & = & ev +\!\!\Rightarrow \lambda t\ x.\ h\ t \\ ev -\!\!\Rightarrow x' & = & ev +\!\!\Rightarrow \lambda t\ x.\ x' \end{array} \end{array}$$

These different operator symbols are somewhat neumonic: $(+\!\!\Rightarrow)$ receives *all* of the parameters, $(-\!\!\Rightarrow)$ receives *none* of the parameters, $(*\!\!\Rightarrow)$ receives only the *time*, and $(\Longrightarrow)$ receives only the *data*.

Constant events. The simplest kind of event is one specified directly by its time and value.

$$\begin{array}{l} constEv : Time \rightarrow \alpha \rightarrow Event_\alpha \\ \mathbf{occ}[\![constEv\ t_e\ x]\!] = (t_e, x) \end{array}$$

Thus, for example, the behavior:

$$b_1\ untilB\ (constEv\ 10\ x) -\!\!\Rightarrow b_2$$

exhibits behavior b_1 until time 10, at which point it begins exhibiting behavior b_2 (x is ignored in this example, but of course needn't be).

External events. For this paper we only consider one kind of external event—mouse button presses—which can be from either the left or right button. The value associated with a button press event is the corresponding button *release* event, which in turn yields a unit value (() is the unit type):

$$lbp,\ rbp : Time \rightarrow Event_{Event_{()}}$$

The meaning of an event $lbp\ t_0$, for example, is the pair (t_e, e), such that t_e is the time of the first left button press after t_0, and e is the event corresponding to the first left button release after t_e. Thus the behavior:

$$\begin{array}{l} b_1\ untilB\ (lbp\ t_0) \Longrightarrow \lambda e. \\ b_2\ untilB\ e -\!\!\Rightarrow \\ b_3 \end{array}$$

exhibits behavior b_1 until the left button is pressed, at which point it becomes b_2 until the left button is released, at which point it becomes b_3.

Predicates. It is natural to want to specify certain events as the first time that a boolean behavior becomes true after a given time.

$$\begin{array}{l} predicate : Behavior_{Bool} \rightarrow Time \rightarrow Event_{()} \\ \mathbf{occ}[\![predicate\ b\ t_0]\!] = (\inf\ \{t > t_0 \mid \mathbf{at}[\![b]\!]t\}, ()) \end{array}$$

That is, the time of a predicate event is the infimum of the set of times greater than t_0 at which the behavior is true. Note that this time could be equal to t_0.

The behavior:

$$b_1\ untilB\ (predicate\ (\sin\ time = 0.5)\ t_0) -\!\!\Rightarrow b_2$$

thus exhibits b_1 until the first time t after t_0 that $\sin\ t$ is 0.5, after which it exhibits b_2.

If the boolean behavior argument to *predicate* were an arbitrarily complex computable function, then *predicate* would not be computable. To cope with this problem, we restrict behaviors somewhat, to make *predicate* not only computable, but also efficient. We will return to this issue in Section 4.2.

Choice. We can choose the earlier of two events with the .|. operator:

$$\begin{array}{l} (.|.) : Event_\alpha \rightarrow Event_\alpha \rightarrow Event_\alpha \\ \begin{array}{rcll} \mathbf{occ}[\![e\ .|.\ e']\!] & = & (t_e, x), & \mathbf{if}\ t_e \leq t'_e \\ & = & (t'_e, x'), & \text{otherwise} \end{array} \\ \quad \mathbf{where}\ \begin{array}{rcl} (t_e, x) & = & \mathbf{occ}[\![e]\!] \\ (t'_e, x') & = & \mathbf{occ}[\![e']\!] \end{array} \end{array}$$

For example, this behavior:

$$b_1\ untilB\ (lbp\ t_0\ .|.\ predicate\ (time > 5)\ t_0) -\!\!\Rightarrow b_2$$

waits for either a left button press or a timeout of 5 seconds before switching from behavior b_1 to behavior b_2. As an alternative, the following example switches to a different behavior, b_3, upon timeout.

$$b_1\ untilB\ (lbp\ t_0 -\!\!\Rightarrow b_2\ .|.\ predicate\ (time > 5)\ t_0 -\!\!\Rightarrow b_3)$$

Snapshot. At the moment an event occurs it is often convenient to take a "snapshot" of a behavior's value at that point in time.

$$\begin{array}{l} \mathit{snapshot} : \mathit{Event}_\alpha \to \mathit{Behavior}_\beta \to \mathit{Event}_{\alpha\times\beta} \\ \mathbf{occ}[\![e\ \mathit{snapshot}\ b]\!] = (t_e,\ (x,\ \mathbf{at}[\![b]\!]t_e)) \\ \quad \mathbf{where}\ (t_e, x) = \mathbf{occ}[\![e]\!] \end{array}$$

For example, the behavior:

$$b_1\ \mathit{untilB}\ (\mathit{lbp}\ t_0\ \mathit{snapshot}\ (\sin\ \mathit{time})) \Longrightarrow \lambda(e, y).\ b_2$$

grabs the sine of the time at which the left button is pressed, binds it to y, and continues with behavior b_2 which presumably depends on y. Although this example could also be achieved by grabbing the time of the left button press event and computing its sine, in general the behavior being snapshot can be arbitrarily complex, and may in fact be dependent on external events.

Event sequencing. It is sometimes useful to use one event to generate another. The event $\mathit{joinEv}\ e$ is the event that occurs when e' occurs, where e' is the value part of e.

$$\begin{array}{l} \mathit{joinEv} : \mathit{Event}_{\mathit{Event}_\alpha} \to \mathit{Event}_\alpha \\ \mathbf{occ}[\![\mathit{joinEv}\ e]\!] = \mathbf{occ}[\![\mathit{snd}\ (\mathbf{occ}[\![e]\!])]\!] \end{array}$$

(This function is so named because it is the "join" operator for the *Event* monad [22].)

For example, the event

$$\mathit{joinEv}\ (\mathit{lbp}\ t_0 {*}{\Rightarrow}\ \mathit{predicate}\ (b = 0))$$

occurs the first time that the behavior b has the value zero after the first left button press after time t_0.

3 Some Larger Examples

The previous section presented the primitive combinators for behaviors and events, along with their formal semantics. The following examples illustrate the use of some of these combinators. The examples are given as Haskell code, whose correspondence to the formal semantics should be obvious. (All values in these examples are behaviors, though we do not explicitly say so.)

To begin, let's define a couple of simple utility behaviors. The first varies smoothly and cyclically between -1 and +1.

```
wiggle  = sin (pi * time)
```

Using `wiggle` we can define a function that smoothly varies between its two argument values.

```
wiggleRange lo hi =
   lo + (hi-lo) * (wiggle+1)/2
```

Now let's create a very simple animation: a red, pulsating ball.

```
pBall = withColor red
          (bigger (wiggleRange 0.5 1) circle)
```

The function `bigger` scales its second argument by the amount specified by its first argument; since the first argument is a behavior, the result is also a behavior, in this case a ball whose size varies from full size to half its full size.

A key attribute of Fran is that behaviors are *composable*. For example, `pBall` can be further manipulated, as in:

```
rBall = move (vectorPolar 2.0 time)
             (bigger 0.1 pBall)
```

which yields a ball moving in a circular motion with radius 2.0 at a rate proportional to time. The ball itself is the same as `pBall` (red and pulsating), but 1/10 the original size.

Certain external phenomena can be treated as behaviors, too. For example, the position of the mouse can naturally be thought of as a vector behavior. Thus to cause an image to track exactly the position of a mouse, all we need to do is:

```
followMouse im t0 = move (mouse t0) im
```

(The function `move` shifts an image by an offset vector.)

Another natural way to define an animation is in terms of *rates*. For example, we can expand on the mouse-follower idea by having the image follow the mouse at a rate that is dependent on how far the image is from the current mouse position.

```
followMouseRate im t0 = move offset im
 where offset = integral rate t0
       rate   = mouse t0 .-. pos
       pos    = origin2 .+^ offset
```

Note the mutually recursive specification of `offset`, `rate`, and `pos`: The offset starts out as the zero vector, and grows at a rate called `rate`. The rate is defined to be the difference between the mouse's location (`mouse` is a primitive behavior that represents mouse position) and our animation's position `pos`. `pos`, in turn, is defined in terms of the offset relative to the origin. As a result, the given image always pursues the mouse, but moves faster when the distance is greater. (The operation `.+^` adds a point and a vector, yielding a point, and `.-.` subtracts two points, yielding a vector.)

As a variation, we can virtually attach the image to the mouse cursor using a spring. The definition is very similar, with position defined by a starting point and a growing offset. This time, however, the rate is itself changing at a rate we call `accel`. This acceleration is defined in part by the difference between the mouse position and the image's position, but we also add some drag that tends to slow down the image by adding an acceleration in the direction opposite to its movement. (Increasing or decreasing the "drag factor" of 0.5 below creates more or less drag.)

```
followMouseSpring im t0 = move offset im
 where offset = integral rate t0
       rate   = integral accel t0
       accel  = (mouse t0 .-. pos) - 0.5 *^ rate
       pos    = origin2 .+^ offset
```

(The operator `*^` multiplies a real number by a vector, yielding a vector.)

As an example of event handling, the following behavior describes a color that changes between red and blue each time the left button is pressed. We accomplish this change with the help of a function `cycle` that takes two colors, `c1` and `c2`, and gives an animated color that starts out as `c1`. When the button is pressed, it swaps `c1` and `c2` and repeats (using recursion).

```
anim12 t0 = withColor (cycle red blue t0) circle
 where  cycle c1 c2 t0 =
          c1 `untilB` lbp t0 *=> cycle c2 c1
```

```
bounce minVal maxVal y0 v0 g t0 = path
  where path = start t0 (y0,v0)

        start t0 (y0,v0) = y `untilB` doBounce +=> start

        where y = lift0 y0 + integral v t0
              v = lift0 v0 + integral g t0
              reciprocity = 0.8

              doBounce :: Event (RealVal, RealVal)      -- returns new y and v
              doBounce = (collide `snapshot` pairB y v) ==> snd ==> \ (yHit,vHit) ->
                         (yHit, - reciprocity * vHit)
              collide  = predicate (y <=* lift0 minVal &&* v<=*0 ||*
                                    y >=* lift0 maxVal &&* v>=*0) t0
```

Figure 1: One-Dimensional Bounce

Note that the *Time* argument in the recursive call to `cycle` is supplied automatically by `*=>`.

The next example is a number-valued behavior that starts out as zero, and becomes -1 while the left button is pressed or 1 while the right button is pressed.

```
bSign t0 =
  0 `untilB` lbp t0 ==> nonZero (-1)  .|.
             rbp t0 ==> nonZero   1
 where  nonZero r stop =
          r `untilB` stop *=> bSign
```

We can use the function `bSign` above to control the rate of growth of an image. Pressing the left (or right) button causes the image to shrink (or grow) until released. Put another way, the rate of growth is 0, -1, or 1, according to `bSign`.

```
grow im t0 = bigger size im
 where  size = 1 + integral rate t0
        rate = bSign t0
```

A very simple modification to the `grow` function above causes the image to grow or shrink at the rate of its own size (i.e. exponentially).

```
grow' im t0 = bigger size im
 where  size = 1 + integral rate t0
        rate = bSign t0 * size
```

Here's an example that demonstrates that even *colors* can be animated. Using the function `rgb`, a color behavior is created by fixing the blue component, but allowing the red and green components to vary with time.

```
withColor (rgb (abs (cos time))
               (abs (sin (2*time)))
               0.5)
          circle
```

As a final example, let's develop a modular program to describe "bouncing balls." First note that the physical equations describing the position y and velocity v at time t of an object being accelerated by gravity g are:

$$\begin{aligned} y &= y_0 + \int_{t_0}^{t} v \; dt \\ v &= v_0 + \int_{t_0}^{t} g \; dt \end{aligned}$$

where y_0 and v_0 are the initial position and velocity, respectively of the object at time t_0. In Fran these equations are simply:

```
y = lift0 y0 + integral v t0
v = lift0 v0 + integral g t0
```

Next we define a function `bounce` that, in addition to computing the position of an object based on the above equations, also determines when the ball has hit either the floor or the ceiling, and if so reverses the direction of the ball while reducing its velocity by a certain *reciprocity*, to account for loss of energy during the collision. The code for `bounce` is shown in Figure 1. Note that collision is defined as the moment when either the position has exceeded the `minVal` and the velocity is negative, or the position has exceeded the `maxVal` and the velocity is positive. When such a collision is detected, the current position and velocity are snapshot, and the cycle repeats with the velocity negated and scaled by the reciprocity factor. (The various operators with `*` after them are lifted versions of the underlying operators.)

Now that `bounce` is defined, we can also use it to describe *horizontal* movement, using 0 for acceleration. Thus to simulate a bouncing ball in a box, we can simply write:

```
moveXY x y
 (withColor green circle)
where
  x = bounce xMin xMax x0 vx0 0 t0
  y = bounce yMin yMax y0 vy0 g t0
```

where `xMin`, `xMax`, `yMin`, and `yMax` are the dimensions of the box.

4 Implementation

The formal semantics given in Section 2 could almost serve as an implementation, but not quite. In this section, we describe the non-obvious implementation techniques used in Fran. One relatively minor item is integration. While symbolic integration could certainly be used for simple behaviors, we have instead adapted standard textbook numerical techniques. (We chiefly use fourth order Runge Kutta [17].)

4.1 Representing Behaviors

An early implementation of Fran represented behaviors as implied in the formal semantics:

```
data Behavior a  =  Behavior (Time -> a)
```

This representation, however, leads to a serious inefficiency. To see why, consider a simple sequentially recursive reactive behavior like the following.

```
b = toggle True 0
 where toggle val t0 =
        lift0 val `untilB` lbp t0  *=>
        toggle (not val)
```

This behavior toggles between true and false whenever the left button is pressed. Suppose `b` is sampled at a time t_1 after the first button press, and we then need to sample `b` at a time $t_2 > t_1$. Then `b` needs to notice that t_2 is after the first button press, and then see whether it is also beyond the second button press. After n such events, sampling must verify that their given times are indeed past n events, so the running time and the (lazily expanded) representation would be $O(n)$. One could try to eliminate this "space-time leak" by switching to a stateful implementation, but doing so would interfere with a behavior's ability to support multiple simultaneously time-transformed versions of itself.

We solve this problem by having behavior sampling generate not only a value, but also a new, possibly simpler, behavior.

```
data Behavior a =
  Behavior (Time -> (a, Behavior a))
```

(In fact, we use a slightly more complex representation, as explained in Section 4.2 below.) Once an event is detected to be (t_e, b'), the new behavior is sampled and the resulting value and possibly an even further simplified version are returned. In most cases (ones not involving time transform), the original *untilB* behavior is then no longer accessible, and so gets garbage collected. Note that this optimization implies some loss of generality: sampling must be done with monotonically non-decreasing times.

These same efficiency issues apply as well to integration, eliminating the need to re-start integration for each sampling. (In fact, our formulation of numerical integration is as sequentially recursive reactive behaviors.)

4.2 Implementing Events

There are really two key challenges with event detection: (a) how to avoid trying too soon to catch events, and (b) how to catch events efficiently and robustly when we need to. We use a form of laziness for the former challenge, and a technique called *interval analysis* for the latter.

Representing events lazily. Recall the semantics of reactivity:

$$\mathit{untilB} : \mathit{Behavior}_\alpha \to \mathit{Event}_{\mathit{Behavior}_\alpha} \to \mathit{Behavior}_\alpha$$
$$\mathbf{at}[\![b\ \mathit{untilB}\ e]\!]t = \mathbf{if}\ t \le t_e\ \mathbf{then}\ \mathbf{at}[\![b]\!]t\ \mathbf{else}\ \mathbf{at}[\![b']\!]t$$
$$\quad \mathbf{where}\ (t_e, b') = \mathbf{occ}[\![e]\!]$$

Note that values of an *untilB*-based behavior at $t \le t_e$ do not depend on the precise value of t_e, just the *partial information* about t_e that it is at least t. This observation is crucial, because it may be quite expensive or, in the case of user input, even impossible to know the value of t_e before the time t_e arrives. Instead, we represent the time t_e by a chain of lower-bound time values increasing monotonically with respect to the information ordering defined in Section 2.1. Because these chains are evaluated *lazily*, detection is done progressively on demand.

Detecting predicate events. The second implementation challenge raised by events is how to determine when *predicate* events occur. For instance, consider the event that occurs when $te^{4t} = 10$:

```
predicate (time * exp (4 * time) ==* 10) 0
```

Any technique based solely on sampling of behaviors must fail to detect events like this whose boolean behaviors are true only instantaneously. An alternative technique is symbolic equation solving. Unfortunately, except for very simple examples, equations cannot be solved symbolically.

The technique we use to detect predicate events is *interval analysis* (IA) [20]. It uses more information from a behavior than can be extracted purely through sampling, but it does not require symbolic equation solving. Instead, every behavior is able not only to tell how a sample time maps to a sample value, but also to produce a conservative interval bound on the values taken on by a behavior over a given interval I. More precisely, the operation **during**, mapping time intervals to α intervals, has the property that $\mathbf{at}[\![b]\!]t \in \mathbf{during}[\![b]\!]I$ for any α-valued behavior b, time interval I, and time $t \in I$.

An interval is represented simply as a pair of values:

```
data Ivl a  =  a `Upto` a
```

For instance, "``3 `Upto` 10``" represents the interval *[3,10]*, i.e., the set of x such that $3 \le x \le 10$. The implementation of a behavior then contains both the time-sampling and interval-sampling functions:

```
data Behavior a =
  Behavior (Time     -> (a,     Behavior a))
           (Ivl Time -> (Ivl a, Behavior a))
```

As an example, the behavior `time` maps times and time intervals to themselves, and returns an unchanged behavior.

```
time :: Behavior Time
time = Behavior (\ t  -> (t,  time))
                (\ iv -> (iv, time))
```

"Lifting" of functions to the level of behaviors works similarly to the description in Section 2, but additionally maps domain intervals to range intervals, and re-applies the lifted functions to possibly altered behavior arguments. For instance, lift_2 is implemented as follows.

```
lift2 f fi b1 b2 = Behavior sample isample
 where sample  t  = (f x1 x2, lift2 f fi b1' b2')
        where (x1, b1') = b1 `at` t
              (x2, b2') = b2 `at` t
       isample iv = (fi xi1 xi2, lift2 f fi b1' b2')
        where (xi1, b1') = b1 `during` iv
              (xi2, b2') = b2 `during` iv
```

The restriction on behaviors referred to in Section 2.3 that makes event detection possible, is that behaviors are composed of functions `f` for which a corresponding `fi` is known in the $lift_n$ functions. (These `fi` are called "inclusion functions.")

Defining functions' behaviors over intervals is well-understood [20], and we omit the details here, other than to point out that Haskell's type classes once again provide a convenient notation for interval versions of the standard arithmetic operators. For example, evaluating

```
(2 `Upto` 4) + (10 `Upto` 30)
```

yields the interval *[12,34]*. Also, a useful IA technique is to exploit intervals of monotonicity. For instance, the exp function is monotonically increasing, while sin and cos functions change between monotonically increasing and monotonically decreasing on intervals of width π.

We can also apply IA to *boolean* behaviors, if we consider booleans to be ordered with `False < True`. There are three nonempty boolean intervals, corresponding to the behavior being true never, sometimes, or always. For example, the interval form of equality checks whether its two interval arguments overlap. If not, the answer is uniformly false. If both intervals are the same *singleton* interval, then the answer is uniformly true. Otherwise, IA only knows that the answer may be true or false throughout the interval. Specifically:

```
(lo1 `Upto` hi1) ==# (lo2 `Upto` hi2)
  | hi1 < lo2 || hi2 < lo1              =
      False `Upto` False
  | lo1==hi1 && lo2==hi2 && lo1==lo2  =
      True  `Upto` True
  | otherwise                           =
      False `Upto` True
```

Similarly, it is straightforward to define interval versions of the inequality operators and logical operators (conjunction, disjunction, and negation).

With this background, detection of *predicate* events through IA is straightforward. Given a start time t_1, choose a time $t_2 > t_1$, and evaluate the boolean behavior over $[t_1, t_2]$, yielding one of the three boolean intervals listed above. If the result is uniformly false, then t_2 is guaranteed to be a lower bound for the event time. If uniformly true, then the event time is t_1 (which is the infimum of times after t_1). Otherwise, the interval is split in half, and the two halves are considered, starting with the earlier half (because we are looking for the *first* time the boolean behavior is true). At some point in this recursive search, the interval being divided becomes smaller than the desired degree of temporal accuracy, at which point event detection claims a success.

This event detection algorithm is captured in the definition of `predicate` given in Appendix A. This function uses the above divide-and-conquer strategy in narrowing down the interval, but also, a *double*-and-conquer strategy in searching the right-unbounded time interval. The idea that if the event was not found in the next w seconds, then perhaps we should look a bit further into the future—$2w$ seconds—the next time around.

It is also possible to apply IA to positional user input. The idea is to place bounds on the rate or acceleration of the positional input, and then make a worst-case analysis based on these bounds. We have not yet implemented this idea.

5 Related Work

Henderson's *functional geometry* [12] was one of the first purely declarative approaches to graphics, although it does not deal with animation or reactivity. Several other researchers have also found declarative languages well-suited for modeling pictures. Examples include [15, 23, 3, 10].

Arya used a lazy functional language to model 2D animation as lazy lists of pictures [1, 2], constructed using list combinators. While this work was quite elegant, the use of lists implies a discrete model of time, which is somewhat unnatural. Problems with a discrete model include the fact that time-scaling becomes difficult, requiring throwing away frames or interpolation between frames, and rendering an animation requires that the frame rate match the discrete representation; if the frames cannot be generated fast enough, the perceived animation will slow down. Our continuous model avoids these problems, and has the pleasant property that animations run at precisely the same speed, regardless of how fast the underlying hardware is (slower hardware will generate less smooth animations, but they will still run at the same rate).

The *TBAG* system modeled 3D animations as functions over *continuous* time, using a "behavior" type family [8, 19]. These behaviors are built up via combinators that are automatically invoked during solution of high level constraints. Because it used continuous time, TBAG was able to support derivatives and integrals. It also used the idea of elevating functions on static values into functions on behaviors, which we adopted. Unlike our approach, however, reactivity was handled imperatively, through constraint assertion and retraction, performed by an application program.

CML (Concurrent ML) formalized synchronous operations as first-class, purely functional, values called "events" [18]. Our event combinators "`.|.`" and "`==>`" correspond to CML's `choose` and `wrap` functions. There are substantial differences, however, between the meaning given to "events" in these two approaches. In CML, events are ultimately used to perform an *action*, such as reading input from or writing output to a file or another process. In contrast, our events are used purely for the values they generate. These values often turn out to be behaviors, although they can also be new events, tuples, functions, etc.

Concurrent Haskell [14] extends the pure lazy functional programming language Haskell with a small set of primitives for explicit concurrency, designed around Haskell's monadic support for I/O. While this system is purely functional in the technical sense, its semantics has a strongly imperative feel. That is, expressions are evaluated without side-effects to yield concurrent, imperative computations, which are executed to perform the implied side-effects. In contrast, modeling entire behaviors as implicitly concurrent functions of continuous time yields what we consider a more declarative feel.

Haskore [13] is a purely functional approach to constructing, analyzing, and performing computer music, which has much in common with Henderson's functional geometry, even though it is for a completely different medium. The Haskore work also points out useful algebraic properties that such declarative systems possess. Other computer music languages worth mentioning include *Canon* [5], *Fugue* [6], and a language being developed at GRAME [16], only the latter of which is purely declarative. Fugue also highlights the utility of lazy evaluation in certain contexts, but extra effort is needed to make this work in its Lisp-based context, whereas in a non-strict language such as Haskell it essentially comes "for free."

DirectX Animation is a library developed at Microsoft to support interactive animation. Fran and DirectX Animation both grew out of the ideas in an earlier design called *ActiveVRML* [7]. DirectX Animation is used from more mainstream imperative languages, and so mixes the functional and imperative approaches.

There are also several languages designed around a *synchronous data-flow* notion of computation. The general-purpose functional language Lucid [21] is an example of this style of language, but more importantly are the languages Signal [11] and Lustre [4], which were specifically designed for control of real-time systems.

In Signal, the most fundamental idea is that of a *signal*, a time-ordered sequence of values. Unlike Fran, however, time is not a value, but rather is implicit in the ordering of values in a signal. By its very nature time is thus discrete rather than continuous, with emphasis on the relative ordering of values in a data-flow-like framework. The designers of Signal have also developed a clock calculus with which one can reason about Signal programs. Lustre is a language similar to Signal, rooted again in the notion of a sequence, and owing much of its nature to Lucid.

6 Conclusions

Writing rich, reactive animations is a potentially tedious and error-prone task using conventional programming methodologies, primarily because of the attention needed for issues of *presentation*. We have described a system called Fran that remedies this problem by concentrating on issues of *modeling*, leaving presentation details to the underlying implementation. We have given a formal semantics and described an implementation in Haskell, which runs acceptably fast using the Hugs interpreter. Future work lies in improving performance through the use of standard compilation methods as well as domain-specific optimization techniques; extending the ideas to 3D graphics and sound; and investigating other applications of this modeling approach to software development.

Our implementation of Fran currently runs under the Windows '95/NT version of Hugs, a Haskell implementation being developed collaboratively by Yale, Nottingham, and Glasgow Universities. It is convenient for developing animation programs, because of quick turn-around from modification to execution, and it runs with acceptable performance, for a byte-code interpreter. We expect marked performance improvement once Fran is running under GHC (the Glasgow Haskell Compiler). Even better, when these two Haskell implementations are integrated, Fran programs will be convenient to develop and run fast. The Hugs implementation, which includes the entire Fran system, may be retrieved from `http://www.haskell.org/hugs`. Although this paper will give the reader an understanding of the technical ideas underpinning Fran, its power as an animation engine (and how much fun it is to play with!) can only be appreciated by using it.

Acknowledgements We wish to thank Jim Kajiya for early discussions that stimulated our ideas for modeling reactivity; Todd Knoblock who helped explore these ideas as well as many other variations; John Peterson and Alastair Reid for experimental implementations; Philip Wadler for thoughtful comments that resulted in simplifying the semantic model; and Sigbjørn Finne for helping with the implementation of Fran. We also wish to acknowledge funding of this project from Microsoft Research, DARPA/AFOSR under grant number F30602-96-2-0232, and NSF under grant number CCR-9633390.

References

[1] Kavi Arya. A functional approach to animation. *Computer Graphics Forum*, 5(4):297–311, December 1986.

[2] Kavi Arya. A functional animation starter-kit. *Journal of Functional Programming*, 4(1):1–18, January 1994.

[3] Joel F. Bartlett. Don't fidget with widgets, draw! Technical Report 6, DEC Western Digital Laboratory, 250 University Avenue, Palo Alto, California 94301, US, May 1991.

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

[5] R.B. Dannenberg. The Canon score language. *Computer Music Journal*, 13(1):47–56, 1989.

[6] R.B. Dannenberg, C.L. Fraley, and P. Velikonja. A functional language for sound synthesis with behavioral abstraction and lazy evaluation. In Denis Baggi, editor, *Computer Generated Music*. IEEE Computer Society Press, 1992.

[7] Conal Elliott. A brief introduction to ActiveVRML. Technical Report MSR-TR-96-05, Microsoft Research, 1996. `ftp://ftp.research.microsoft.com/pub/tech-reports/Winter95-96/tr-96-05.ps`.

[8] Conal Elliott, Greg Schechter, Ricky Yeung, and Salim Abi-Ezzi. TBAG: A high level framework for interactive, animated 3D graphics applications. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida)*, pages 421–434. ACM Press, July 1994.

[9] John Peterson et. al. Haskell 1.3: A non-strict, purely functional language. Technical Report YALEU/DCS/RR-1106, Department of Computer Science, Yale University, May 1996. WWW version at `http://haskell.cs.yale.edu/haskell-report`.

[10] Sigbjorn Finne and Simon Peyton Jones. Pictures: A simple structured graphics model. In *Glasgow Functional Programming Workshop*, Ullapool, July 1995.

[11] 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.

[12] Peter Henderson. Functional geometry. In *ACM Symposium on LISP and Functional Programming*, pages 179–187, 1982.

[13] Paul Hudak, Tom Makucevich, Syam Gadde, and Bo Whong. Haskore music notation – an algebra of music, September 1994. To appear in the Journal of Functional Programming; preliminary version available via `ftp://nebula.systemsz.cs.yale.edu/pub/yale-fp/papers/haskore/hmn-lhs.ps`.

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

[15] Peter Lucas and Stephen N. Zilles. Graphics in an applicative context. Technical report, IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120-6099, July 8 1987.

[16] O. Orlarey, D. Fober, S. Letz, and M. Bilton. Lambda calculus and music calculi. In *Proceedings of International Computer Music Conference*. Int'l Computer Music Association, 1994.

[17] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing (2nd ed.)*. Cambridge University Press, Cambridge, 1992. ISBN 0-521-43108-5.

[18] 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.

[19] Greg Schechter, Conal Elliott, Ricky Yeung, and Salim Abi-Ezzi. Functional 3D graphics in C++ - with an object-oriented, multiple dispatching implementation. In *Proceedings of the 1994 Eurographics Object-Oriented Graphics Workshop*. Eurographics, Springer Verlag, 1994.

[20] John M. Snyder. Interval analysis for computer graphics. In Edwin E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 121–130, July 1992.

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

[22] Philip Wadler. Comprehending monads. In *Proceedings of Symposium on Lisp and Functional Programming*, pages 61–78, Nice, France, June 1990. ACM.

[23] S.N. Zilles, P. Lucas, T.M. Linden, J.B. Lotspiech, and A.R. Harbury. The Escher document imaging model. In *Proceedings of the ACM Conference on Document Processing Systems (Santa Fe, New Mexico)*, pages 159–168, December 5–9 1988.

Appendix A: Haskell Code for Predicate Event Detection

```
type BoolB = Behavior Bool
type TimeI = Ivl Time

predicate :: BoolB -> Time -> Event ()

predicate cond t0 = predAfter cond t0 1
  where
    predAfter cond t0 width =
      predIn    cond  (t0 `Upto` t0+width) ( \ cond' ->
      predAfter cond' (t0+width) (2*width) )

    predIn :: BoolB -> TimeI -> (BoolB -> Event ()) -> Event ()
    predIn cond iv tryNext =
      case valI of
        False `Upto` False    ->                   -- no occurrence
          -- Note lower bound and try the next condition.
          timeIsAtLeast hi (tryNext cond')
        False `Upto` True     ->                   -- found at least one
          if   hi-mid <= eventEpsilon
          then constEv mid ()
          else predIn cond    (lo `Upto` mid) ( \ midCond ->
               predIn midCond (mid `Upto` hi) tryNext )
        True `Upto` True -> constEv lo ()          -- found exactly one
      where
        lo `Upto` hi   =  iv
        mid            =  (hi+lo)/2
        ivLeftTrimmed  =  lo + leftSkipWidth `Upto` hi
        (valI,cond')   =  cond `during` ivLeftTrimmed

-- Interval size limit for temporal subdivision
eventEpsilon  = 0.001  :: Time
-- Simulate left-open-ness via a small increment
leftSkipWidth = 0.0001 :: Time
```