# Embedding Dynamic Dataflow in a Call-by-Value Language*

Gregory H. Cooper and Shriram Krishnamurthi

Brown University, Providence, RI 02912, USA
{greg, sk}@cs.brown.edu

**Abstract.** This paper describes FrTime, an extension of Scheme designed for writing interactive applications. Inspired by functional reactive programming, the language embeds dynamic dataflow within a call-by-value functional language. The essence of the embedding is to make program expressions evaluate to nodes in a dataflow graph. This strategy eases importation of legacy code and permits incremental program construction. We have integrated FrTime with the DrScheme programming environment and have used it to develop several novel applications. We describe FrTime's design and implementation in detail and present a formal semantics of its evaluation model.

## 1 Introduction

This paper describes FrTime (pronounced "father time"), a programming language built atop the DrScheme environment [9]. FrTime is an exploration of an important point in the design space of dynamic dataflow, or functional reactive [7, 16, 19], programming.

To make FrTime as familiar as possible to current programmers, the language reuses much of the infrastructure, including the syntax, of an existing call-by-value language. In this case the host language is a purely functional subset of Scheme, although the strategy we describe could be applied to other call-by-value languages as well. The embedding strategy reuses the host language's evaluator to make program execution construct a graph of dataflow dependencies; a dataflow *engine* subsequently reacts to events and propagates changes through this graph. FrTime conservatively extends basic language constructs to trigger graph creation when used in the context of time-varying values. Pure Scheme programs are also FrTime programs with the same meaning they have in Scheme and may be incorporated into FrTime programs without modification.

The design of FrTime reflects a desire to satisfy three main goals.

1. Programs should be able to respond to and process events from external sources. For example, one application of FrTime is as a scripting language for a debugger [15]. A debugger script must respond to events from the program under investigation, which arrive at an unspecified frequency that cannot be known *a priori*. This suggests that the language should embrace a push-driven implementation strategy, where the arrival of an event triggers a computation that propagates up a tree of dependencies.

* This work is partially supported by NSF grant CCR-0305949.

P. Sestoft (Ed.): ESOP 2006, LNCS 3924, pp. 294–308, 2006.

2. FrTime programs should be able to make maximal use of a programming environment for incremental development. This especially means that programmers must be able to write expressions in the read-eval-print loop (REPL), observe and name their values, use them to build larger expressions, and so on. With such support, the REPL can serve as one of the primary interfaces for many programs, thereby saving programmers from having to construct a separate, explicit interface. For example, the paper about the scriptable debugger [15] discusses the debugging of a shortest-paths algorithm by interactively adding script fragments that provide increasingly better clues. FrTime's support for dynamic dataflow graph construction saves us the need to *build* an interaction mode for the debugger. Instead we reuse the DrScheme REPL, inheriting its many features and also its careful handling of many subtle issues [11].
3. As a practical matter, FrTime needs to reuse as much of an existing evaluator as possible. In particular, the underlying evaluator in DrScheme is quite complex and supports a large legacy codebase. Ideally, therefore, FrTime needs an evaluation strategy that can reuse this evaluator and seamlessly integrate as much legacy code as possible to inherit a large library of useful functionality. (For example, by inheriting DrScheme's graphics library, the scriptable debugger supports graphical display of the target program's state.) A corollary is that legacy programs should be incrementally convertible into FrTime. That is, it should be possible to begin with an existing Scheme application and run it under FrTime, then gradually change fragments of it to use dataflow features. This must not require a significant source transformation (such as conversion into continuation-passing or monadic style).

In this paper we present the semantics and implementation of FrTime. In particular, we describe the language's embedding strategy and how it satisfies the goals stated above. We also provide an operational semantics that specifies the language's evaluation model. FrTime has been distributed with the DrScheme programming environment since 2003 and has been used to develop several non-trivial applications, including a scriptable debugger [15], a spreadsheet, and a version of the Slideshow [10] presentation system enhanced with interactive animations.

## 2 The FrTime Language

FrTime extends the language of DrScheme [9] with support for dynamic dataflow through a notion of *signals*, or time-varying values. The language is inspired and informed by work on functional reactive programming (FRP) [7, 16, 19], which extends Haskell [14] with similar features.

The most basic signals are those that represent time itself. For example, there is a signal called *seconds*, which counts the number of seconds elapsed since a specific point in the past. *Seconds* is an example of a *behavior*—a signal that is defined at every point in time, or *continuous*. If we apply a primitive function $f$ to a behavior, the result is a new behavior, whose value is computed by applying $f$ to the argument at (conceptually) every point in time.[1] In other words, FrTime *lifts* primitive functions to the domain of

[1] Operationally, the language only applies $f$ to the argument initially and each time it changes.

behaviors. For example, if we evaluate (*even? seconds*), the result is a new behavior that indicates, at every moment, whether the current value of *seconds* is even.

In addition to behaviors, there are signals called event streams that carry sequences of discrete values. For example, we have built an interface to the DrScheme window toolkit that provides an event stream called *key-strokes*, which carries key events. Unlike with behaviors, primitive procedures cannot be applied to event sources. FrTime instead provides a collection of event-processing combinators that are analogous to common list-processing routines. For example, the raw *key-strokes* stream contains events for key presses and releases. Applications that don't care about the releases can elide them with (*filter-e char? key-strokes*). This produces a new event stream that only carries the events whose values are characters.

There is similarly an analog of *map* called *map-e*, which we could use to convert all of the alphabetic characters to upper case. Another combinator, called *collect-e*, resembles Haskell's *scanl*; it consumes an event stream, an initial accumulator, and a transformer. For each event occurrence, *collect-e* applies the transformer to the new event and the accumulator, yielding a new accumulator which is emitted on the resulting event stream. By passing **empty** and **cons** as the second and third arguments, we can build a list of all the occurrences of a given event.

FrTime provides primitives for converting between behaviors and event streams. One is *hold*, which consumes an event stream and an initial value and returns a behavior that starts with the initial value and changes to the last event value each time an event occurs. Conversely, *changes* consumes a behavior and returns an event stream that emits the value of the behavior each time it changes.

On the surface, signals bear some similarity to constructs found in other languages. Behaviors change over time, like mutable data structures or the return values of impure procedures, and event streams resemble the infinite lazy lists (also called streams) common to Haskell and other functional languages. The key difference is that FrTime tracks dataflow relationships between signals and automatically recomputes them to maintain programmer-specified invariants.

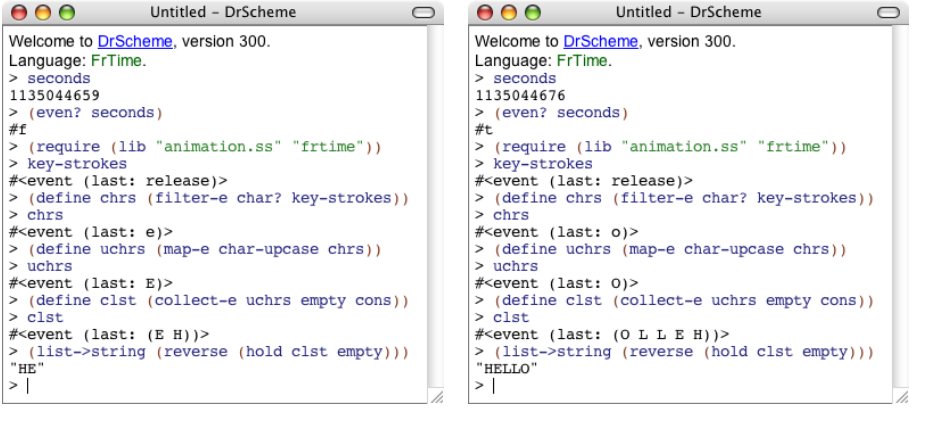


**Fig. 1.** Screenshots of a single interactive FrTime session, taken 17 seconds apart

FrTime runs in the DrScheme programming environment. Figure 1 presents two screenshots from the same interactive session in DrScheme, taken about seventeen seconds apart. In this session we first evaluate *seconds* and (*even? seconds*). Then we load the FrTime animation library, which creates a new, empty window (not shown). As we type into this new window, key press and release events arrive on the *key-strokes* event stream. We create *chrs* by filtering out the release events, and *uchrs* by converting these to upper-case with *map-e*. We make *collect-e* accumulate a list of characters, then apply *hold* to produce a behavior, which we reverse and convert to a string.

Evaluating a signal at the DrScheme prompt registers a dependency between that signal and the graphical object that represents it in the interactions window. Thus, when the signal's value changes, FrTime automatically triggers an update of the display. This explains why the two screenshots in Fig. 1 show different values for many expressions, even though they are taken from the same session. This is an important example of integrating the language with the environment in order to respect the language's unconventional abstractions. Conversely, the language supports the environment's notion of interactive, incremental program construction. For example, as we build up the string of key-strokes, we can name and observe each intermediate result, checking that it behaves as we expect before adding the next piece.

## 3 Evaluation Strategy

In this section we describe FrTime's evaluation strategy, which satisfies the goals set forth in the Introduction. Firstly, it employs a push-driven update mechanism: events initiate computation, and changes cause dependent parts of the program to recompute. Secondly, the language supports incremental program construction; the programmer can interleave program construction, evaluation, and observation. Finally, it reuses the Scheme evaluator and permits reuse of existing Scheme library code, which supports incremental conversion of Scheme programs to use FrTime's dataflow features.

FrTime is a collection of syntactic abstractions and value definitions implemented in Scheme. Executing a FrTime program means running the Scheme evaluator in an environment containing the FrTime definitions. These definitions *make executing the program build a graph of its dataflow dependencies*. The nodes of this graph correspond to program expressions, and the arcs indicate flow of values from one expression to another. An expression that does not utilize any dataflow elements evaluates as a standard, pure Scheme expression, yielding the same value it would have in Scheme.

Because evaluation is push-driven, a program's reactivity originates through dependence on primitive event sources, for example a timer, a keyboard, a mouse, or a network data stream. The FrTime *engine* listens to events from these sources and routes them to the interested parts of the program's dataflow graph. Values change at the corresponding nodes of the dataflow graph and propagate along the dependency arcs.

In the remainder of this section, we explain how evaluating a FrTime expression constructs a graph of dataflow dependencies, and how the language implements reactivity through subsequent traversal of this graph. We discuss some of the difficulties that arise from a push-driven update model and how we solve them.

### 3.1 Dataflow Graph Construction and Manipulation

Suppose the programmer enters the expression (+ 3 4) at the FrTime REPL. Its evaluation proceeds in the traditional call-by-value fashion, first reducing subexpressions to values, then applying the specified operation to them.

FrTime is meant to extend pure Scheme with a notion of signals, so if we start with a pure Scheme expression and replace some constant values with signals, the result should be a legal FrTime program. For example, we should be able to refer to "the time 3 seconds from now" by writing (+ 3 *seconds*). However, evaluating such an expression in a standard Scheme evaluator does not yield the desired dataflow semantics. Scheme primitives like + only know how to process ordinary, constant Scheme values (in this case numbers). At best, the result might be to add 3 to the *current value* of *seconds*. This would produce a constant value reflecting the state of the system at the moment of evaluating the expression, but it would fail to update with the passage of time. In reality, the situation is worse; FrTime's signals are implemented as data structures, so passing *seconds* to + is a type mismatch and causes a runtime exception.

Clearly, *ordinary* Scheme evaluation does not work for FrTime. This means that we must either write a new evaluator for FrTime, or extend Scheme evaluation to accommodate FrTime's novel features. Since we want to reuse as much of Scheme as possible, we take the latter approach by interposing a mechanism that prevents the direct application of Scheme primitives to signals. Specifically, we define the FrTime evaluation environment so that the names of Scheme primitives refer to lifted versions of the same. Lifting wraps a primitive with code that checks for signal arguments and, if there are any, constructs and returns a new signal. For example, the FrTime expression (+ 3 *seconds*) reduces to the following Scheme code:

```
(if (or (signal? 3) (signal? seconds))
    (make-signal (λ () (+ (current-value 3) (current-value seconds))) 3 seconds)
    (+ 3 seconds))
```

This first tests for signals among the argument subexpressions. Since *seconds* is a signal, the conditional selects the first branch. The procedure **make-signal** consumes a thunk (nullary procedure), boxed above, and any number of *producer* values to which the thunk refers. It returns a new signal whose value is defined, at any point in time, by the result of calling the thunk. In this case, it applies the addition primitive to the current values of the constant 3 and the signal *seconds*. The procedure *current-value* acts like the identity function on constant values, so (*current-value* 3) reduces to 3. On signals, *current-value* projects the signal's current value, an ordinary Scheme constant. Thus the addition primitive inside the thunk sees only constants, so there are no errors. The fact that signals like *seconds* change over time underscores the necessity of the thunk: the language needs to re-evaluate the procedure to update the signal when any of the producers change.

The additional arguments to **make-signal** (here 3 and *seconds*) are the producers on which the new signal depends. They may include both constants and signals; **make-signal** ignores the constants and registers a dependency with each of the signals. Registration gives the producers explicit references to the new signal (instead of the other way around, as a reader might initially assume). These *reverse* references are essential to

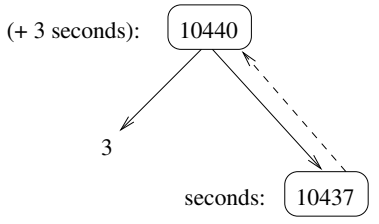


**Fig. 2.** Dataflow graph for (+ 3 *seconds*)

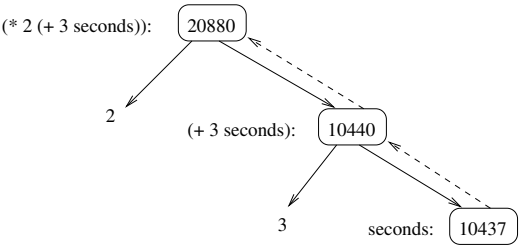


**Fig. 3.** Dataflow graph for (* 2 (+ 3 *seconds*))

implementing push-driven evaluation: when a signal changes, the runtime system follows them to determine which signals need recomputation. Figure 2 shows the resulting signal graph. Rounded boxes depict signals, solid arrows are normal data references, and dashed arrows represent the reverse references needed for push-driven updates.

The addition of these reverse references would ordinarily expand reachability into a symmetric relation, to the detriment of effective memory management. To solve this problem, we make the reverse references *weak*, which tells the memory manager to ignore them when computing reachability. If a signal is no longer reachable from the application, it will be reclaimed. Since there may be a significant delay between objects' becoming unreachable and their reclamation by the garbage-collector, it is possible that the engine will continue recomputing such *dead* signals for some time. This strategy is unacceptable in general, so we need a mechanism for deleting signals when they cease to belong in the system. We describe such a mechanism in Sect. 3.3.

The FrTime evaluation model applies to all expressions, even those that do not use signals. For example, the FrTime expression (+ 3 4) reduces to the following Scheme expression:

```
(if (or (signal? 3) (signal? 4))
    (make-signal (λ () (+ (current-value 3) (current-value 4))) 3 4)
    (+ 3 4))
```

Because the constants 3 and 4 are not signals, the entire expression is clearly equivalent to its raw Scheme counterpart, and yields the constant 7. This illustrates one of our design goals: that pure Scheme expressions should evaluate in FrTime as they would in standard Scheme. This means that programmers can easily mix pure Scheme and FrTime code, which creates a smooth migration path for porting Scheme code.

Because user-defined signals like (+ 3 *seconds*) are indistinguishable from primitive signals like *seconds*, evaluation of FrTime expressions works even when operations on signals nest. For example, if a programmer writes (∗ 2 (+ 3 *seconds*)), the inner (+ ...) subexpression evaluates first, yielding a signal like the one described above. The evaluation of the (∗ ...) application proceeds in an analogous manner, constructing a new signal that depends upon the value of (+ 3 *seconds*). We show the resulting graph in Fig. 3.

Expressions can arbitrarily nest and mix computations involving constants and signals. For example, if we write (+ (+ 1 2) *seconds*), the (+ 1 2) evaluates as in Scheme, reducing to the constant 3, after which evaluation proceeds exactly as above for

(+ 3 *seconds*). Only one new signal is created, and the resulting dataflow graph is identical to the one shown in Fig. 2.

The dataflow graph construction that occurs when a FrTime program runs is just the first step in its evaluation. The interesting part—the program's reactivity—begins once the graph is constructed and continues as long as the system runs and events arrive. This involves primitive signals changing in response to external events and propagating through the dataflow graph. For example, once every second, a timer triggers a change in *seconds*, which in turn triggers recomputation of every signal that *depends* on *seconds*, such as (+ 3 *seconds*) in our example above. Changes then propagate to transitive dependents, such as (∗ 2 (+ 3 *seconds*)).

When the engine recomputes a signal, it compares the new value with the previous one. If they are the same (according to Scheme's **eq?** procedure), the engine does not schedule the signal's dependents. For example, a signal defined by an expression like (*quotient seconds* 10) depends on *seconds* but only changes after *seconds* increases by ten. Consumers of this signal, like (> (*quotient seconds* 10) 100), only recompute every ten seconds, not every second.

### 3.2 Glitch Prevention

Scheduling recomputation is an important semantic issue. For example, consider the expression (< *seconds* (+ 1 *seconds*)). This evaluates to a signal that should always have the value **true**, since $n$ is always less than $n + 1$.

However, life is not so simple in a push-driven update model. Each change in *seconds* triggers recomputation of the overall expression and the inner (+ 1 *seconds*) signal, and the order in which FrTime recomputes these signals affects the answer. If it updates the (+ 1 *seconds*) signal first, then the top-level < compares up-to-date versions of *seconds* and (+ 1 *seconds*), yielding **true**. On the other hand, if it updates the top-level signal first, it then compares the up-to-date *seconds* with the stale (+ 1 *seconds*)—which is equal to the new value of *seconds*—yielding **false**.

This situation, where a signal is recomputed before all of its subordinate signals are up-to-date, is called a *glitch* [5]. Such behavior is unacceptable as it results in redundant computation and, much worse, causes signals to violate invariants.

We need a traversal strategy that prevents glitches. The crucial property is that no signal should update until everything on which it depends is also up-to-date. Unfortunately, the obvious candidates of depth-first and breadth-first search are susceptible to glitches, as the preceding example shows. However, a slightly modified breadth-first search achieves the goal. Specifically, we approximate the graph's structure by assigning each signal a *height*, which exceeds that of all its producers. To make a valid depth assignment possible, the dataflow graph must be acyclic. This restriction has the benefit of guaranteeing that update propagation terminates, but it also seems to impose a severe limit on the language's the expressive power. We explain in Sect. 3.5 how FrTime supports programs with cyclic dependencies.

Computing signal heights is relatively simple. Since **make-signal** receives all of the new signal's producers, it only needs to compute their maximum and add 1 to it. Instead of a standard first-in-first-out queue, the engine uses a priority queue to process nodes in order of increasing height. Since each signal is higher than everything on which it depends, this strategy guarantees the absence of glitches and redundant computation.

### 3.3 Dynamic Reconfiguration

The height-guided recomputation strategy works under the assumption that the dataflow graph does not change in the middle of an update cycle. Unfortunately, this is an unreasonable assumption: the need to reconfigure the graph dynamically arises naturally from combining behaviors with basic Scheme features.

We illustrate some of the intricacies of dynamic reconfiguration through a simple example involving the use of a time-varying condition in an **if** expression:

```
(let* ([len (modulo seconds 4)]
       [lst (build-list len add1)])
  (if (zero? len)
      0
      (list-ref lst (sub1 len))))
```

In this program, *len* cycles through the values 0, 1, 2, 3, and *lst* is the list (1 ... *len*). When *len* is 0, the value of the whole expression is 0, and otherwise it is the last element of *lst*, which is also equal to *len*.

Evaluating this program proves to be somewhat tricky. Since the **if**'s condition is time-varying, the result of the whole expression needs to switch dynamically between the branches (either of which may also be time-varying), forwarding the value of whichever branch the condition currently selects.

In general, evaluating a branch is only legal when the condition selects it. For example, when the first branch is selected above, evaluating the second branch would raise an exception by attempting to extract the element of *lst* at position $-1$. The threat of such problems means that, when the condition changes, a new branch must be constructed and the old one disabled, or deleted, before evaluation proceeds. Thus the structure of the dataflow graph must change in the middle of an update cycle. In this example, the need arises from the use of behaviors in conditionals; an analogous situation arises when the function position of an application is time-varying.

Changing the structure of the dataflow graph in the middle of an update cycle creates a number of hazards that must be handled carefully. Constructing new dataflow graph fragments is precarious because the existing graph may be in an inconsistent state, with some signals updated but others stale. In this example, *lst* has a large height because *build-list* is a recursive procedure that constructs a complex fragment of dataflow graph. However, since *zero?* is a primitive, (*zero? len*) has height 2, and when it becomes **false** (triggering construction of the second branch), *lst* still has the stale value **empty**. Evaluating the new branch (which tries to extract an element from *lst*) would raise an exception. To avoid this problem, FrTime constructs the new branch *without computing the initial node values*. Instead it enqueues the new nodes for update, and the recomputation algorithm initializes them after reaching the proper height.

Another problem is that the new fragment's height may exceed that of the old one. To prevent glitches, the engine needs to adjust height assignments to reflect the new graph topology before performing any more updates. It must also notify the priority queue of any changes in heights of signals that are already enqueued for update.

Deleting a fragment of the dataflow graph is also subtle. To prevent any unwanted evaluation, FrTime must delete all of the signals in the dead branch, including those

already enqueued for recomputation. This means that the height of all of these signals must strictly exceed that of the condition. Deleting a signal involves removing all edges incident on it, which makes it unreachable and ensures that it will not be scheduled for recomputation again. However, since a change may have already scheduled the deleted signal for recomputation, deletion replaces the signal's update procedure with a no-op, preventing ill effects from any final update attempt. (This technique is more efficient than the alternative of removing the deleted signals from the priority queue.)

Determining which signals to delete can be tricky, too. It is not simply the set of all signals reachable from the root of the deleted fragment; this would include many signals merely referenced within the branch (in the example, signals like *len* and *lst*). However, taking only the signals directly created by evaluation of the branch yields an underapproximation. This is because the branch may contain other dynamic expressions, which in turn constructed signals after the creation of the enclosing branch. FrTime needs to track construction within this extended notion of the expression's dynamic extent. The semantics presented in Sect. 4 provides an abstract model of this mechanism, but the details involved in implementing it efficiently are beyond the scope of this paper.

### 3.4 Incremental Construction

FrTime's evaluation model differs from the approaches taken in the Haskell FRP systems [7, 16]. In those, a program specifies the structure of a dynamic dataflow computation, but the actual reactivity is implemented in an interpreter called *reactimate*. This interpreter runs in an infinite loop, blocking interaction through the REPL until the computation is finished. In many applications, we need to support REPL-style interaction in the middle of the reactive program's execution.

FrTime supports REPL interaction by implementing reactivity in a separate thread. The user is assigned one thread, typically corresponding to the DrScheme REPL, while the FrTime dataflow engine, which constructs and manipulates the program's dataflow graph, runs in a separate thread. These threads communicate through a message queue; at the beginning of each update cycle, the engine empties the queue and processes the messages. Each message corresponds either to an event occurrence or to a request for construction of a new dataflow graph fragment. When the user enters an expression at the REPL prompt, the REPL sends a message to the dataflow engine, which evaluates it and responds with the root of the resulting graph. Control returns to the REPL, which issues a new prompt for the user, while in the background the engine continues processing events and updating signals.

On the surface, it may appear that the Haskell systems could achieve similar behavior simply by spawning a new thread to evaluate the call to *reactimate*. Control flow would return to the REPL, apparently allowing the user to extend or modify the program. However, this background process would still not return a value or offer an interface for probing or extending the running dataflow computation. The values of signals running inside a *reactimate* session, like the dataflow program itself, reside in the procedure's scope and hence cannot escape or be affected from the outside. In contrast, FrTime's message queue allows users to submit new program fragments dynamically, and *evaluating an expression returns a live signal* which, because of the engine's background execution, reflects part of a running computation.

$$x \in \langle \text{var} \rangle ::= \text{(variable names)} \qquad p \in \langle \text{prim} \rangle ::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{<} \mid \texttt{>} \mid \ldots$$

$$\sigma \in \langle \text{loc} \rangle ::= \text{(store locations)} \qquad t, n \in \langle \text{num} \rangle ::= \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots$$

$$u, v \in \langle \text{v} \rangle ::= \bot \mid \texttt{true} \mid \texttt{false} \mid \langle \text{num} \rangle \mid \langle \text{prim} \rangle \mid \texttt{(}\lambda\texttt{(} \langle \text{var} \rangle^* \texttt{)} \langle \text{e} \rangle \texttt{)} \mid \langle \text{loc} \rangle$$

$$e \in \langle \text{e} \rangle ::= \langle \text{v} \rangle \mid \langle \text{var} \rangle \mid \texttt{(} \langle \text{e} \rangle \langle \text{e} \rangle^* \texttt{)} \mid \texttt{(delay} \langle \text{e} \rangle \langle \text{num} \rangle \texttt{)} \mid \texttt{(if} \langle \text{e} \rangle \langle \text{e} \rangle \langle \text{e} \rangle \texttt{)}$$

$$E \in \langle \text{E} \rangle ::= \texttt{[]} \mid \texttt{(} \langle \text{v} \rangle^* \langle \text{E} \rangle \langle \text{e} \rangle^* \texttt{)} \mid \texttt{(delay} \langle \text{E} \rangle \langle \text{num} \rangle \texttt{)} \mid \texttt{(if} \langle \text{E} \rangle \langle \text{e} \rangle \langle \text{e} \rangle \texttt{)}$$

$$s \in \langle \text{sig-type} \rangle ::= \texttt{(lift} \langle \text{prim} \rangle \langle \text{v} \rangle^* \texttt{)} \mid \texttt{(delay} \langle \text{loc} \rangle \langle \text{num} \rangle \langle \text{loc} \rangle \texttt{)} \mid \texttt{input}$$
$$\mid \texttt{(dyn(}\lambda\texttt{(} \langle \text{var} \rangle \texttt{)} \langle \text{e} \rangle \texttt{)} \langle \text{loc} \rangle \langle \text{loc} \rangle \texttt{)} \mid \texttt{(fwd} \langle \text{loc} \rangle \texttt{)} \mid \texttt{const}$$

**Fig. 4.** Grammars for FrTime values, expressions, evaluation contexts, and signal types

### 3.5 Cycles

We explain in Sect. 3.2 how our height assignment strategy restricts the dataflow graph to be acyclic. However, programs with cyclic signal networks arise naturally in many applications. For example, in user interfaces, we often want two sets of widgets that display and control the same underlying model, such as RGB and HSV views in a color-selection window. Since either set of widgets must be able to influence the other, they are mutually dependent. Forbidding cycles altogether would disallow expression of such patterns, making the language unacceptably weak.

In the current implementation, we make a compromise consistent with that made by other dataflow languages [4, 5, 16, 18, 19]. We provide a **delay** operator that reflects the value that its argument had at a specific interval in the past. If a cycle includes a signal created by **delay**, then that cycle cannot cause the system to enter a tight loop, since the delay halts update propagation until the future. We therefore assign a height of 0 to **delay**-ed signals. As long as each cycle passes through a **delay**, a consistent height assignment is possible, and evaluation is safe.

## 4 Semantics

We have developed a formal semantics of FrTime's evaluation model, which highlights the push-driven update strategy and the embedding in a call-by-value functional host language. Figure 4 shows the grammars for values, expressions, evaluation contexts, and signal types. Values include the undefined value ($\bot$), booleans, numbers, primitive procedures, $\lambda$-abstractions, and store locations (which identify signals). Expressions include values, procedure applications, `delay`s, and conditionals. Evaluation contexts [8] enforce a left-to-right, call-by-value order on subexpression evaluation. Signal types, which we explain in detail below, describe the different signal variants.

Figure 5 presents semantic domains and operations over them. $\delta$, a parameter to the system, defines reduction for primitives. $\Sigma$ denotes a set of signal locations and $X$ means a set of *external events*, each of which contains a location, a value, and an occurrence time (when it enters the system). $I$ refers to a set of *internal events*, which contain only target locations and (optionally) values. A store $S$ maps signal locations to triples containing a *current value*, a *signal type*, and a *set of dependents*. For notational

$$
\begin{array}{rll}
\delta & : \langle \text{prim} \rangle \times \langle \text{v} \rangle \times \ldots \rightarrow \langle \text{v} \rangle & \text{(primitive evaluation)} \\
\Sigma & \subset \langle \text{loc} \rangle & \text{(store location set)} \\
I & \subset \langle \text{loc} \rangle \cup (\langle \text{loc} \rangle \times \langle \text{v} \rangle) & \text{(internal event set)} \\
X & \subset \langle \text{loc} \rangle \times \langle \text{v} \rangle \times \langle \text{num} \rangle & \text{(external event set)} \\
S & : \langle \text{v} \rangle \rightarrow \langle \text{v} \rangle \times \langle \text{sig-type} \rangle \times 2^{\langle \text{loc} \rangle} & \text{(signal in store)} \\
\mathcal{V}_S(v) & = v', \text{ where } S(v) = (v', \_, \_) & \text{(current value projection)} \\
A(\Sigma, v_0, v) & = \begin{cases} \Sigma & \text{if } v \neq v_0 \\ \emptyset & \text{otherwise} \end{cases} & \text{(signals affected by change)} \\
\text{reg}(\sigma, \Sigma, S) & = S[\sigma' \mapsto (v, s, \Sigma' \cup \{\sigma\})]_{\forall \sigma' \in \Sigma | S(\sigma') = (v,s,\Sigma')} & \text{(dependency registration)} \\
\mathcal{D}_S(\Sigma) & = \bigcup_{\sigma \in \Sigma} \Sigma', \text{ where } S(\sigma) = (\_, \_, \Sigma') & \text{(dependency lookup)} \\
\text{dfrd}_S(I) & = \mathcal{D}_S^+(\{\sigma \mid \sigma \in I \vee (\sigma, \_) \in I\}) & \text{(deferred recomputations)} \\
\text{del}(S, \Sigma) & = \left( \begin{array}{l} S[\sigma \mapsto (v, s, \Sigma' \setminus \Sigma)]_{\forall \sigma | S(\sigma) = (v,s,\Sigma')}, \\ \bigcup_{\sigma \in \Sigma} \begin{cases} \Sigma' & \text{if } S(\sigma) = (\_, (\texttt{dyn} \ \_ \ \_ \ \_), \Sigma') \\ \emptyset & \text{otherwise} \end{cases} \end{array} \right) & \text{(dependency removal)}
\end{array}
$$

**Fig. 5.** Semantic domains and operations

convenience when dealing with behaviors and constants, the store permits lookup of constants, which `const` signals. This simplifies the definition of $\mathcal{V}_S$, which projects the current value of any signal or constant. Other important operations include *reg*, which registers one signal's dependence on a set of other signals, and $\mathcal{D}_S$, which computes the set of signals dependent upon any of a set of signals. *dfrd* computes the set of stale signals that are deferred, or not ready for immediate update (the $^+$ indicates transitive, irreflexive closure). Finally, *del* eliminates references to deleted signals from a given store. As explained in Sect. 3.3, FrTime needs to delete signals recursively from nested dynamic branches. To facilitate this, *del* not only returns the modified store but also finds all the nested `dyn` signals, whose children must be deleted.

FrTime's evaluation model divides naturally into two layers. One is the context-sensitive rewriting system that captures the call-by-value functional core and the extension that constructs the dataflow graph. Figure 6 shows the transformation rules that comprise this layer. These *construction* rules reduce expressions in the context of a store and a set of internal events. The $\delta$, $\beta_v$, and IF reductions are standard for languages derived from the $\lambda$-calculus; they neither read nor change any of the additional elements in the tuple. The LIFTed versions of these rules describe how the system extends the dataflow graph when behaviors are used with primitive procedures, user-defined procedures, and conditionals.

The LIFTed rules explain only the construction of the dataflow graph. The reactivity is described by the layer of *update* rules, which are presented in Fig. 7. These specify how the system evolves when each variety of signal updates:

**lift** Application of a primitive to one or more behaviors results in the lifting of the application (rule $\delta$-LIFT). This yields a new `lift` signal that records the primitive and its arguments. The new signal is enqueued for update, which invokes rule U-LIFT after all the arguments are up-to-date. The rule computes the signal's value by applying the primitive to the current values of the arguments. If the new value differs from the old one, the signal's dependents are enqueued for update.

$$\frac{\{v_1, \ldots, v_n\} \wedge \langle \text{loc} \rangle = \emptyset}{\langle S, I, E[(p\ v_1 \ldots v_n)] \rangle \rightarrow \langle S, I, E[\delta(p, v_1, \ldots, v_n)] \rangle} \quad (\delta)$$

$$\frac{\{v_1, \ldots, v_n\} \wedge \langle \text{loc} \rangle = \{\sigma_1, \ldots, \sigma_k\} \neq \emptyset \qquad \forall i \in [1..k].S(\sigma_i) = (v_i, s_i, \Sigma_i) \quad S' = \text{reg}(\sigma, \{\sigma_1, \ldots, \sigma_k\}, S[\sigma \mapsto (\bot, (\texttt{lift}\ p\ v_1 \ldots v_n), \emptyset)])}{\langle S, I, E[(p\ v_1 \ldots v_n)] \rangle \rightarrow \langle S', I \cup \{\sigma\}, E[\sigma] \rangle} \quad (\delta\text{-LIFT})$$

$$\langle S, I, E[((\lambda\ (x_1 \ldots x_n)\ e)\ v_1 \ldots v_n)] \rangle \rightarrow \langle S, I, E[e[v_1/x_1] \ldots [v_n/x_n]] \rangle \quad (\beta_v)$$

$$\frac{S' = S[\sigma_1 \mapsto (\bot, (\texttt{dyn}\ (\lambda\ (x)\ (x\ v_1 \ldots v_1))\ \sigma\ \sigma_2), \emptyset)][\sigma_2 \mapsto (\bot, (\texttt{fwd}\ \bot), \emptyset)]}{\langle S, I, E[(\sigma\ v_1 \ldots v_n)] \rangle \rightarrow \langle \text{reg}(\sigma_1, \{\sigma\}, S'), I \cup \{\sigma_1\}, E[\sigma_2] \rangle} \quad (\beta_v\text{-LIFT})$$

$$\langle S, I, E[(\texttt{if true}\ e_1\ e_2)] \rangle \rightarrow \langle S, I, E[e_1] \rangle$$
$$\langle S, I, E[(\texttt{if false}\ e_1\ e_2)] \rangle \rightarrow \langle S, I, E[e_2] \rangle \quad (\text{IF})$$

$$\frac{S' = S[\sigma_1 \mapsto (\bot, (\texttt{dyn}\ (\lambda\ (x)\ (\texttt{if}\ x\ e_1\ e_2))\ \sigma\ \sigma_2), \emptyset)][\sigma_2 \mapsto (\bot, (\texttt{fwd}\ \bot), \emptyset)]}{\langle S, I, E[(\sigma\ v_1 \ldots v_n)] \rangle \rightarrow \langle \text{reg}(\sigma_1, \{\sigma\}, S'), I \cup \{\sigma_1\}, E[\sigma_2] \rangle} \quad (\text{IF-LIFT})$$

$$\frac{S' = \text{reg}(\sigma_2, \{\sigma\}, S[\sigma_1 \mapsto (\bot, \texttt{input}, \emptyset)][\sigma_2 \mapsto (\bot, (\texttt{delay}\ \sigma\ n\ \sigma_1), \emptyset)])}{\langle S, I, E[(\texttt{delay}\ \sigma\ n)] \rangle \rightarrow \langle S', I \cup \{\sigma_2\}, E[\sigma_1] \rangle} \quad (\text{DELAY})$$

**Fig. 6.** Construction rules

**delay, input** Delaying a signal requires two new signals: a *consumer* (of type `delay`) observes changes in the argument and directs events to a *producer* that arrive after the given interval (rule U-DELAY). The producer has type INPUT and simply forwards the delayed value carried by the latest event (rule U-INPUT). Because communication passes through the external event mechanism, there is no direct dependence; this is why **delay** breaks cycles. In general, input signals can channel values into the system from the external event queue. They are thus useful not only for **delay** but can also model events from all manner of input sources, such as a mouse or a network port.

**dyn, fwd** Signals of type `dyn` modify the structure of the dataflow graph in response to changes in a given *trigger* signal. These signals are used to implement both conditionals (**if** expressions) and applications with a signal in the function position. For conditionals, the trigger is the condition, while for applications the trigger is the function. Each `dyn` signal contains an update procedure (the $u$ field in rule U-DYN), which FrTime applies to the current value of the trigger ($\sigma_1$) to yield a new branch of dataflow graph (rooted at $\sigma_3$). The branch is connected to the rest of the graph by a permanent `fwd` signal, which forwards the value of the current branch. The `dyn` signal's $\Sigma$ field, normally used to track dependents, tracks all the signals created by the most recent invocation of the update procedure. These are the signals that must be deleted when a change in the trigger invalidates the existing branch. Each application of *del* removes references to these signals in the store and accumulates the set of signals created by nested `dyn` signals. These also must be deleted and may in turn have children requiring deletion. The language thus applies *del* repeatedly until no deletions remain.

The rules described above leave the precise scheduling of updates non-deterministic. However, they enforce a topological order, which guarantees the absence of glitches and

$$\frac{I \ni \sigma \notin \mathrm{dfrd}_S(I) \qquad S(\sigma) = (v_0, (\texttt{lift}\ p\ v_1 \ldots), \Sigma) \qquad \delta(p, \mathcal{V}_S(v_1), \ldots) = v}{\langle X, S, I, t\rangle \hookrightarrow \langle X, S[\sigma \mapsto (v, (\texttt{lift}\ p\ v_1 \ldots), \Sigma)], I \setminus \{\sigma\} \cup A(\Sigma, v_0, v), t\rangle} \text{ (U-LIFT)}$$

$$\frac{\sigma \in I \qquad S(\sigma) = (\bot, (\texttt{delay}\ \sigma\ n\ \sigma_1), \Sigma)}{\langle X, S, I, t\rangle \hookrightarrow \langle X \cup \{(\sigma_1, \mathcal{V}_S(\sigma), t+n)\}, S, I \setminus \{\sigma\}, t\rangle} \text{ (U-DELAY)}$$

$$\frac{(\sigma, v) \in I \qquad S(\sigma) = (v_0, \texttt{input}, \Sigma)}{\langle X, S, I, t\rangle \hookrightarrow \langle X, S[\sigma \mapsto (v, \texttt{input}, \Sigma)], I \setminus \{\sigma\} \cup A(\Sigma, v_0, v), t\rangle} \text{ (U-INPUT)}$$

$$\frac{\begin{array}{c}\sigma \in I \qquad S(\sigma) = (\bot, (\texttt{dyn}\ u\ \sigma_1\ \sigma_2), \Sigma) \\ S(\sigma_2) = (v, (\texttt{fwd}\ \_), \Sigma_2) \qquad (S^*, \emptyset) = \mathrm{del}^*(S, \Sigma) \\ \langle S^*, I, (u\ \mathcal{V}_S(\sigma_1))\rangle \rightarrow^* \langle S', I', \sigma_3\rangle \qquad \Sigma' = \mathrm{dom}(S') \setminus \mathrm{dom}(S) \\ S_1 = \mathrm{reg}(\sigma_2, \{\sigma_3\}, S'[\sigma \mapsto (\bot, (\texttt{dyn}\ u\ \sigma_1\ \sigma_2), \Sigma')][\sigma_2 \mapsto (v, (\texttt{fwd}\ \sigma_3), \Sigma_2)])\end{array}}{\langle X, S, I, t\rangle \hookrightarrow \langle X, S_1, (I' \setminus \Sigma) \setminus \{\sigma\}, t\rangle} \text{ (U-DYN)}$$

$$\frac{\sigma \in I \qquad S(\sigma) = (v_0, (\texttt{fwd}\ \sigma'), \Sigma) \qquad S(\sigma') = (v, \_, \_)}{\langle X, S, I, t\rangle \hookrightarrow \langle X, S[\sigma \mapsto (v, (\texttt{fwd}\ \sigma'), \Sigma)], I \setminus \{\sigma\} \cup A(\Sigma, v_0, v), t\rangle} \text{ (U-FWD)}$$

$$\langle X, S, \emptyset, t\rangle \hookrightarrow \langle X, S, \{(\sigma, v) \mid (\sigma, v, t+1) \in X\}, t+1\rangle \text{ (U-SHIFT)}$$

**Fig. 7.** Update rules

makes the state at the end of each update cycle well-defined. When there are no more internal update events to process, the system is stable and awaits the arrival of new events. Time advances to the next step, and any external events scheduled for the new time shift into the set of internal events (rule U-SHIFT).

## 5 Related Work

There is a large body of research on dataflow programming. An early language was Lucid [18], a pure, first-order dataflow language based on synchronous streams. Lustre [4] offers a similar programming model to that of Lucid, but with restrictions that support compilation to finite automata and real-time performance guarantees. Lustre also adds a notion of user-defined clocks, allowing streams to compute at different rates. Lucid Synchrone [12] extends Lustre with ML-style type inference, pattern-matching, and first-class functions. Signal [2] is similar to Lustre but is based on relations rather than functions, so the evaluation model is non-deterministic. There are other synchronous languages, such as Esterel [3], whose programming models are imperative.

Functional reactive programming (FRP) [7, 16, 17, 19] merges the model of synchronous dataflow programming with the expressive power of Haskell, a statically-typed, higher-order functional language. In addition, it adds support for *switching* (dynamically reconfiguring a program's dataflow structure) and introduces a conceptual separation of signals into (continuous) *behaviors* and (discrete) *events*.

There has been significant work on implementation models for FRP. Real-time FRP [20] FRP is close in spirit to the synchronous dataflow languages, where the focus is on bounding resource consumption. Parallel FRP [17] adds a notion of non-determinism and explores compilation of FRP programs to parallel code. Elliott discusses several functional implementation strategies for FRP systems [6], which suffer from various

practical problems such as time- and space-leaks. A newer version, Yampa [16], fixes these problems at the expense of some expressive power: while Fran [7] extended Haskell with first-class signals, the Yampa programmer builds a network of *signal functions* in a custom syntax, through a set of *arrow* combinators [13]. FrTime's linguistic goals are more in line with those of Fran—integrating signals with the Scheme language in as seamless a manner as possible. Importantly, because Scheme is eager, the implementation has precise control over when signals begin evaluating, which helps to prevent time-leaks. In addition, the use of state in the implementation allows more control over memory usage, which helps to avoid space-leaks. The evaluation model leads to several other differences, as described in Section 3.

Frappé [5] is a Java library for building FRP-style dynamic dataflow graphs. Its evaluation model is similar to FrTime's, in the sense that computation is driven by external events, not by a central clock. However, the propagation strategy is based on a "hybrid push-pull" algorithm, whereas FrTime's is entirely push-driven, which makes conditional evaluation more challenging. A more important difference from FrTime is that Frappé is a library, not a language. It is intended less for end-user programming than as runtime support for an FRP compiler that targets Java.

Adaptive functional programming (AFP) [1] supports incremental recomputation of function results when their inputs change. As in FrTime, execution occurs in two stages. First the program runs, constructing a graph of its data dependencies. The user then changes input values and tells the system to recompute their dependents. The key difference from FrTime is that AFP requires transforming the program into *destination-passing style*. This prevents the easy import of legacy code and complicates the task of porting existing libraries. The structure of AFP also leads to a more linear recomputation process, where the program re-executes from the first point affected by the changes.

## 6 Conclusions and Future Work

We have presented FrTime, an implementation of functional reactive programming for a call-by-value language. We have described its novel evaluation model, which accomplishes the goals set forth in the Introduction. We have also provided a formal semantic model for reasoning about FrTime evaluation more abstractly. The language is integrated and distributed with the DrScheme programming environment. We have developed interfaces for various libraries and built several non-trivial applications with it.

Our primary focus for future research is to improve performance of the update strategy. Currently, there is significant overhead involved when moving from Scheme's top-down, stack-based execution model to FrTime's push-driven, queue-based update algorithm. In particular, we have noticed severe degradations in performance when running code from existing Scheme libraries under FrTime. FrTime permits fine control (not described in this paper) over the boundary between the two execution strategies, and we are interested in developing mechanical techniques for optimizing the decision.

**Acknowledgements.** We are grateful to Antony Courtney, Paul Hudak, Guillaume Marceau, and John Peterson for valuable discussions about this work. We also thank the anonymous reviewers for their suggestions.

## References


1. U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. In *ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 247–259, 2002.
2. A. Benveniste, P. L. Guernic, and C. Jacquemot. Synchronous programming with events and relations: the signal language and its semantics. *Science of Computer Programming*, 16(2):103–149, 1991.
3. G. Berry. *The Foundations of Esterel*. MIT Press, 1998.
4. P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 178–188, 1987.
5. A. Courtney. Frappé: Functional reactive programming in Java. In *Practical Aspects of Declarative Languages*, pages 29–44, 2001.
6. C. Elliott. Functional implementations of continuous modeled animation. In *Programming Languages: Implementations, Logics, and Programs*, pages 284–299, 1998.
7. C. Elliott and P. Hudak. Functional reactive animation. In *ACM SIGPLAN International Conference on Functional Programming*, pages 263–277, 1997.
8. M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 102(2):235–271, 1992.
9. R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, 2002.
10. R. B. Findler and M. Flatt. Slideshow: Functional presentations. In *ACM SIGPLAN International Conference on Functional Programming*, pages 224–235, 2004.
11. M. Flatt, R. B. Findler, S. Krishnamurthi, and M. Felleisen. Programming languages as operating systems (*or*, Revenge of the Son of the Lisp Machine). In *ACM SIGPLAN International Conference on Functional Programming*, pages 138–147, 1999.
12. G. Hamon and M. Pouzet. Modular Resetting of Synchronous Data-flow Programs. In *ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 289–300, 2000.
13. J. Hughes. Generalizing monads to arrows. *Science of Computer Programming*, 37(1-3):67–111, 2000.
14. S. P. Jones and J. Hughes, editors. *Report on the Programming Language Haskell 98*. 1999.
15. G. Marceau, G. H. Cooper, S. Krishnamurthi, and S. P. Reiss. A dataflow language for scriptable debugging. In *IEEE International Symposium on Automated Software Engineering*, pages 218–227, 2004.
16. H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *ACM SIGPLAN Workshop on Haskell*, pages 51–64, 2002.
17. J. Peterson, V. Trifonov, and A. Serjantov. Parallel functional reactive programming. In *Practical Aspects of Declarative Languages*, pages 16–31, 2000.
18. W. W. Wadge and E. A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press U.K., 1985.
19. Z. Wan and P. Hudak. Functional reactive programming from first principles. In *ACM Conference on Programming Language Design and Implementation*, pages 242–252, 2000.
20. Z. Wan, W. Taha, and P. Hudak. Real-time FRP. In *ACM SIGPLAN International Conference on Functional Programming*, pages 146–156, 2001.