Integrating Dataflow Evaluation into a Practical Higher-Order Call-by-Value Language

by
Gregory Harold Cooper
B. S., University of Rhode Island, 2000
Sc. M., Brown University, 2002

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in the
Department of Computer Science at Brown University

Providence, Rhode Island
May 2008

This dissertation by Gregory Harold Cooper is accepted in its present form by the Department of Computer Science as satisfying the dissertation requirement for the degree of Doctor of Philosophy.

Date _____ _____

Shriram Krishnamurthi, Director

Recommended to the Graduate Council

Date _____ _____

Steven P. Reiss, Reader

Date _____ _____

John Peterson, Reader
(Western State College of Colorado)

Approved by the Graduate Council

Date _____ _____

Sheila Bonde
Dean of the Graduate School

Vita

Gregory Harold Cooper was born on New Year's Day of 1978 in South County, Rhode Island. He has enjoyed mathematics and logic since he can remember and was addicted to computer programming by age 7. He was valedictorian of the class of 1996 at North Kingstown High School and a National Merit Scholar and Barry M. Goldwater Scholar at the University of Rhode Island. He also received an honorable mention in the National Science Foundation Graduate Research Fellowship competition in 2001 and 2002.

Acknowledgements

This dissertation would not have been possible without the help of many people. Thanks are in order first to my advisor, Shriram Krishnamurthi, and readers, Steve Reiss and John Peterson. I'd also like to thank those who collaborated on various parts of the work: Kim Burchett, Dan Ignatoff, Guillaume Marceau, and Jono Spiro. In addition, a number of people provided useful feedback and participated in helpful discussions, including Ezra Cooper, Antony Courtney, Matthias Felleisen, Paul Hudak, Henrik Nilsson, Manuel Serrano, Mike Sperber, Phil Wadler, and many others whom I'm sure I've forgotten. Finally, I am indebted to Melissa Chase, Manos Renieris, and Dave Tucker for their enduring support and encouragement.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This dissertation explores the design of linguistic support for reactive programs. By *reactive*, I mean programs like word processors, Web browsers, and programming environments—programs whose "inputs" are unbounded sequences of *events* (e.g., key strokes, mouse clicks, network messages, etc.) that arrive from a variety of sources at times beyond the program's control. In contrast to a *transformational* program, which runs uninterrupted and controls when it reads its input and produces its output, a reactive program must respond immediately to each event by updating its internal state and emitting a representation of it. In essence, a reactive program must be designed to allow its environment to control its execution.

1.1 Motivation

Reactive programs constitute the majority of software systems deployed in the world, so it is important for programmers to be able to build such systems easily and reliably. Unfortunately, the programming languages and paradigms in current common use were developed for writing transformational programs, and they make reactive programs awkward to express.

As a concrete example, consider the implementation of a simple reactive program that counts the elapsed time (in seconds) up to some user-controlled time, rendering it as a textual string. At any point, the user can click a *Reset* button to start the count over again.

```
(define frame
  (new frame% [label "Timer"] [height 80] [width 300]))
(send frame show #t)

(define duration 60)
(define elapsed 0)

(define elapsed-display
  (new message% [parent frame] [label "0 s"] [min-width 60]))

(define clock
  (new timer% [interval 1000]
              [notify-callback
                (λ (t e)
                  (set! elapsed (min (add1 elapsed) duration))
                  (send elapsed-display set-label (format "~a s" elapsed)))]))

(define duration-control
  (new slider% [parent frame] [label "Duration (s)"]
               [min-value 10] [max-value 120] [init-value 60]
               [callback (λ (s e)
                           (set! duration (send slider get-value))
                           (set! elapsed (min elapsed duration))
                           (send elapsed-display set-label (format "~a s" elapsed)))]))

(define reset-button
  (new button% [parent frame] [label "Reset"]
               [callback
                 (λ (b e)
                   (set! elapsed 0)
                   (send elapsed-display set-label "0 s"))]))
```

Figure 1.1: A simple timer application in PLT Scheme

Figure 1.1[1] shows how someone might implement such a program in the Scheme [55] programming language, following a conventional programming style. The first definition creates a top-level window, called a *frame%*.[2] In DrScheme's object system, a **new** expression constructs an object of a given type with a set of named arguments, in this case the

[1]The code is executable under the Pretty Big language level in the v300 series of DrScheme revisions.

[2]By convention, class names in DrScheme end with a % sign, suggesting **o**bject-**o**rientation.

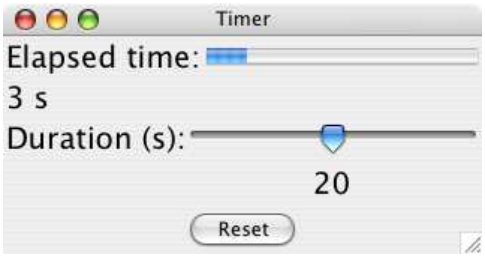


Figure 1.2: A screenshot of a graphical timer

label, *width*, and *height*. After displaying the frame, the program defines a variable to hold the *elapsed* time (initially zero) and creates a *message* control in which to display it.

The remaining code makes the program reactive: the *clock* advances the elapsed time every second, the *duration-control* lets the user adjust the duration, and the *reset-button* starts the elapsed time over at 0. Since the program is reactive, events from these three sources can arrive in any order and at any time. In order to react to whichever event occurs next, the program defines *callback* procedures and registers them with the user interface toolkit. The toolkit's event-handling loop calls the appropriate callback whenever the associated event occurs.

1.2 Callbacks: an Imperative of Imperatives

The code in Figure 1.1 illustrates an interesting pattern: although most of the program is functional, all of the callback procedures perform destructive side effects (either directly, via **set!**, or by invoking a mutator method in an object). This is no coincidence, and the explanation derives from the fact that callbacks are designed to let the event loop *call back* into the application. If callbacks were functional, all they could do is compute values and return them to the event loop, in which case they could not possibly affect the state of the application. Hence, in order for the program to progress, *callbacks must perform side effects*.

While side effects are necessary in some cases, they generally have undesirable consequences. For example, because the *elapsed-display* is updated via side effects, its definition

```
(define frame
  (new frame% [label "Timer"] [height 80] [width 300]))
(send frame show #t)

(define duration 60)
(define elapsed 0)

(define elapsed-display
  (new gauge% [parent frame] [label "Elapsed: "] [range 60]))

(define clock
  (new timer% [interval 1000]
              [notify-callback
                (λ (t e)
                  (set! elapsed (min (add1 elapsed) duration))
                  (send gauge set-value elapsed))]))

(define duration-control
  (new slider% [parent frame] [label "Duration (s)"]
               [min-value 10] [max-value 120] [init-value 60]
               [callback (λ (s e)
                           (set! duration (send slider get-value))
                           (set! elapsed (min elapsed duration))
                           (send gauge set-value elapsed)
                           (send gauge set-range duration))]))

(define reset-button
  (new button% [parent frame] [label "Reset"]
               [callback
                 (λ (b e)
                   (set! elapsed 0)
                   (send gauge set-value 0))]))
```

Figure 1.3: A timer with graphical display

does not provide a complete specification of its behavior, the way it would in a purely functional program. Conversely, the *clock*'s callback refers to *elapsed-display* even though the latter has no bearing on the time.

In general, callbacks result in a programming style in which the definition of an object does not fully express that object's behaviors in terms of other objects' values. Instead,

each object is responsible for tracking changes in its state and updating other objects that depend on it. This is precisely the opposite of how functional programs work, and this structural inversion makes programs more difficult to understand. For example, to reason about the temporal behavior of *elapsed-display*, a programmer (or tool) needs to find all the code that *changes* the object. The problem amplifies when this code refers to other values (e.g., *elapsed*) that are also mutated from various parts of the program.

Structural inversion also makes programs more difficult to write and modify. For example, suppose that the programmer wants to change the display of the elapsed time from the textual message to a graphical progress bar, or *gauge*. Figure 1.3 shows the code that might result. There are three places in the code that update the display, and all of them must be changed. Moreover, in order for the gauge to display the fraction of elapsed time correctly, its *range* must be kept consistent with the *duration*. There is no analog to this logic in the original version of the program, so the programmer might easily overlook the need to make this change. In a functional program, this oversight would likely manifest as a missing procedure argument, which would be an error. However, in the imperative version, there is simply the absence of a side-effect, which is much more difficult to detect automatically..

1.3 Dataflow Evaluation: an Alternative to Callbacks

This dissertation explores an alternative to callbacks that allows programmers to develop reactive programs in a functional style. The key idea is to use *dataflow evaluation* [20, 98], a programming model in which a program's values may change over time, but instead of using explicit mutation, they recompute automatically when their inputs change.

Languages based on the idea of dataflow evaluation (so-called dataflow languages) have existed for decades and have been applied in various specialized domains, most notably real-time, safety-critical embedded systems. Historically, such languages have been designed to support formal reasoning about safety properties and resource requirements. To that end, these languages have traditionally been restrictive, omitting such features as recursive datatypes, dynamic recursion, and higher-order functions. Such limitations are necessary to provide the guarantees required by real-time systems, but they are not appropriate for writing modern general-purpose applications, which must be able to perform complex

operations over dynamic data structures.

Within the past decade, researchers have developed a model called functional reactive programming (FRP) [39, 74, 102, 104], which embeds dataflow evaluation within general-purpose functional languages. Functional reactive programming has proven expressive enough to support a variety of applications, such as animation [39], graphical user interfaces [30, 85], robotics [78], and vision [79].

The work described in this dissertation follows in the FRP vein but departs from prior work in several important ways. Specifically, my thesis states that **a practical notion of dataflow evaluation can be embedded within a general-purpose, higher-order call-by-value language, integrating seamlessly with all of the (non-imperative) features in the original language**. The thesis is supported by a working implementation of such an embedding: the language FrTime [17, 26, 52], which builds upon the dialect of Scheme [55] used in the DrScheme [42] programming environment.

1.4 A Brief Introduction to FrTime

The essence of FrTime is to extend Scheme with a notion of time-varying values called *signals*. For example, the language provides a signal called seconds, whose value at every point in time is equal to the result of Scheme's built-in procedure *current-seconds*. Because its value is defined at every point in time, seconds is said to be *continuous* and is called a *behavior*. If a program applies a primitive function to a behavior, the result is a new behavior, whose value at every point in time is computed by applying the function to the argument's current value.[3] For example, the value of (*even?* seconds) is a behavior whose value alternates between *true* and *false*, changing once every second.

The generalization of Scheme's primitives to operate over behaviors is called *lifting*. Lifting allows a program to use existing purely functional Scheme code in the context of reactive values, a property known as *transparent reactivity*. For example, the following Scheme procedure consumes a time in seconds and formats it as a human-readable string like "10:25:43":

(**define** (*format-time t*)

[3] Operationally, the language only re-evaluates the application each time the argument value changes.

```
(let* ([date (seconds→date t)]
       [hours (date-hour date)]
       [minutes (date-minute date)]
       [seconds (date-second date)])
  (format "~a:~a:~a" hours minutes seconds))
```

In Scheme, one might apply it to (*current-seconds*), producing a value that reflects the time at which the program called (*current-seconds*). One can also use this definition *verbatim* in FrTime and apply it to seconds, creating a simple clock.

In addition to applying primitive functions to them, programs can delay behaviors by any (non-negative) amount of time, and they can compute time integrals over numeric behaviors. There is also a procedure called *changes* that lets a program see the sequence of discrete changes that a behavior experiences over time. This produces a different kind of signal, called an *event stream*. Event streams are a natural abstraction for modeling many inputs to a reactive program, such as the sequence of keys a user types or the clicks of a button in a graphical interface.

Primitive procedures cannot be applied to event streams, but FrTime provides a collection of event-processing operators that are analogous to standard list-processing functions. For example, if an application is only interested in key strokes corresponding to digits, it can use *filter-e* to select them:

```
(define figures
  (filter-e (λ (x) (member x '(#\0 #\1 #\2 ... #\9))) key-strokes))
```

To convert these characters to actual numbers, it can use *map-e* to transform each event:

```
(define digits
  (map-e (λ (ch) (- (char→integer ch) 48)) digits-typed))
```

It can then use *collect-e* to accumulate the sequence of digits into a decimal integer:

```
(define number
  (collect-e digits 0 (λ (digit num) (+ digit (* 10 num)))))
```

Finally, it can use *hold*, the dual of *changes*, to convert *number* to a behavior by "holding" onto the value of its most recent event (using 0 until the first one occurs).

```
(hold number 0)
```

```
(define frame
  (new ft-frame% [label "Timer"] [width 200] [height 80] [shown true]))

(define duration-control
  (new ft-slider% [label "Duration"] [min-value 10] [max-value 120])

(define reset-button
  (new ft-button% [label "Reset"])

(define duration (send slider get-value-b))
(define last-click-time
  (hold (map-e second (snapshot-e (send reset-button get-value-e) seconds))
        (value-now seconds)))

(define elapsed (min duration (- seconds last-click-time)))

(define elapsed-display
  (new ft-message% [label (format "~a s" elapsed)]
                   [parent frame] [min-width 60]))
```

Figure 1.4: Implementation of the timer in FrTime

As the user enters the characters 2, 1, 3, 6, this behavior takes on the values 2, 21, 213, and 2136.

One other important event-processing operator is *merge-e*, which combines several event streams into a single one. For example, if the above program took its input from a pad of graphical buttons instead of the keyboard, it would merge all of the event streams and apply *collect-e* to the result.

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.

```
(define frame
  (new ft-frame% [label "Timer"] [width 200] [height 80] [shown true]))

(define duration-control
  (new ft-slider% [label "Duration"] [min-value 10] [max-value 120])

(define reset-button
  (new ft-button% [label "Reset"])

(define duration (send slider get-value-b))
(define last-click-time
  (hold (map-e second (snapshot-e (send reset-button get-value-e) seconds))
        (value-now seconds)))

(define elapsed (min duration (- seconds last-click-time)))

(define elapsed-display
  (new ft-gauge% [label "Elapsed time:"] [range duration]
                 [parent frame] [value elapsed]))
```

Figure 1.5: The FrTime timer with graphical display

1.5 Declarative Reactive Programming with Signals

FrTime's signal abstractions offer a way to rewrite reactive programs without callbacks or side effects. Figure 1.4 shows the code for a FrTime implementation of the interval timer discussed above. In it, there are no callbacks or destructive side effects, and every definition provides a complete description of the object's behavior over time. The clicks of the reset button, instead of triggering an imperative callback, produce an event stream. The program uses *snapshot-e* to pair each click with the time at which it occurred, then projects out just the occurrence times with *map-e*. It uses *hold* to lock on to the last click time, then subtracts this value from the current time to compute the elapsed time. The elapsed time then automatically updates as time passes or the user clicks the button.

Other aspects of the user interface are also defined in terms of signals. The *duration-control* slider exposes its value as a behavior, from which the program defines the timer's duration. Likewise, the program specifies the message's content as a string behavior that depends on the elapsed time. The language automatically keeps all of the program's state consistent as changes occur.

Figure 1.5 shows a variation of this program in which the elapsed time is displayed graphically instead of textually. Unlike in the callback-based version, where making this change involved modifying code throughout the program, in FrTime the changes are confined to the definition of *elapsed-display*.

1.6 Design Principles and Challenges

A fundamental goal in FrTime is to reuse as much of Scheme as possible, including not only its evaluation mechanism but also its libraries, environment, and tools. These latter artifacts are responsible for much of the cost of developing a language, and without them a language can have little practical value. Thus their reuse offers an important strategic advantage. In addition to conserving development resources, such reuse also makes the extended language more accessible to programmers familiar with the original language, thereby encouraging adoption.

To achieve such reuse, the techniques I have developed are based on lightweight, context-free transformations of the language's core constructs. A key property of these transformations is that they result in a conservative extension of the language, so in the absence of signals, the evaluator defers to the base language's semantics. This means that pure Scheme programs are also FrTime programs, having the same meaning as in Scheme, and they may be incorporated into FrTime programs without modification.

The challenge that arises from this approach is to make the dataflow extension interact seamlessly with the many features available in a rich general-purpose language like Scheme. Some of the tricky features include:

- higher-order functions,
- control-flow constructs,
- structured data,
- automatic memory management,
- legacy libraries (containing both functional and imperative code),
- an interactive read-eval-print loop, and

- tools for understanding program behavior.

Making the dataflow mechanism interact with all of these features places considerable constraints on the language's design, and the main contribution of my work has been to design a strategy that can accommodate all of them.

An underlying theme in FrTime's evaluation model is the use of recomputation, under a standard call-by-value regime, as the mechanism for keeping state consistent. While behaviors are conceptually values that change over over continuous time, in practice they are mutable structures whose contents are recomputed in response to discrete changes.

The main problem this model creates is to find the appropriate level of granularity at which to recompute things. For example, any part of a purely functional (side-effect free) expression can be evaluated repeatedly without affecting the program's semantics. Thus, a simple but naïve strategy would be to re-evaluate the whole program each time anything changed. However, doing so would be incorrect, since operators like *delay-by* need to remember state over time. The basic model for FrTime is therefore one in which expressions are recomputed at the finest grain—primitive procedure applications and core syntactic forms like **if**. This strategy, which is described in detail in Chapter 2, never computes more than is necessary to bring the system to a consistent state.

The basic evaluation model is also presented in Chapter 3 as a three-part formal semantics. One layer is an extension of the call-by-value λ-calculus; it specifies the reduction of FrTime programs to plain values and signals. A slight modification of this layer defines how the value of a signal recomputes in response to a change. The remaining layer specifies how the dataflow evaluator schedules updates to guarantee consistency.

FrTime's dataflow update mechanism itself incurs significant overhead, so achieving optimal performance requires more than simply minimizing the number of atomic operations. Working in a call-by-value host language, it is generally much more efficient to recompute a complex expression in a large, monolithic call-by-value step, than to break it down to atomic steps and evaluate each one separately. Chapter 4 describes a static analysis for identifying maximal fragments of purely functional code, which can then be executed in a single call-by-value step. This refinement of the evaluation strategy results in considerable performance improvements.

While recomputation is key to the mechanism for keeping a program's internal state up-to-date, FrTime also uses it to keep its internal state synchronized with the outside world, through things like graphics libraries and user interface toolkits. In these cases, the re-executed code is not purely functional (rather, its chief purpose is to perform side-effects), so considerable care is needed to ensure that the effects leave the world in a consistent state. Chapter 6 addresses this problem from one angle: when the external libraries are object-oriented class hierarchies whose objects encapsulate flat values like numbers, booleans, and strings. The canonical example is a user interface toolkit, with simple widgets like messages, check boxes, scroll bars, and gauges.

For more complicated applications, programs need a richer assortment of data structures, such as lists, trees, and vectors. Keeping the world consistent with such data is a non-trivial task, since changes can occur at any time or place within the data and can even change the structure of the data. Chapter 5 describes strategies for tracking changes within structured data and communicating them to the world in a logically consistent manner.

Another important aspect of a language is having a programming environment that respects the language's abstractions and provides tools to ease program development and understanding. A benefit of FrTime's evaluation model is that it naturally supports incremental program development in an interactive read-eval-print loop REPL.

The DrScheme environment also provides a collection of tools that are sensitive to control flow, including an error tracer and an execution profiler. Because FrTime is an embedding in Scheme, the tools report information about the execution of the FrTime evaluator as it processes a program. This information is at the wrong level of detail for a user to understand the behavior of the original program. The tools' output thus violates FrTime's linguistic abstractions and only indirectly reflects the original program's behavior. To address this problem, Chapter 7 describes a straightforward technique for manipulating the Scheme tools so that they provide meaningful information about FrTime programs. The technique, which is justified by the formal semantics presented in Chapter 3, applies to any host language with a suitable control-flow introspection mechanism.

To demonstrate the utility of FrTime, Chapter 8 presents the language's use in the novel and non-trivial context of scripting a debugger. Chapter 9 provides a discussion of related work, and Chapter 10 concludes and proposes possible directions for future work.

Chapter 2

Core Evaluation Model

This chapter describes in detail the approach for embedding dataflow in a call-by-value language.[1]

The following presentation refers to a purely functional subset of Scheme, whose grammar is shown in Figure 2.1. The set of values includes basic constants (e.g., numbers and booleans), primitive operators, linked lists, and user-defined procedures (λ-abstractions). Programs are fully expression-based, consisting of procedure applications, conditionals, and recursive binding constructs.

The conceptual goal of FrTime is to extend this call-by-value core language with a notion of *behaviors*, or continuous time-varying values. The remaining sections in this chapter describe how FrTime treats behaviors in each of these forms. The only exception is data structures, which involve enough complexity to warrant their own chapter (Chapter 5). Section 2.10 explains how a notion of behaviors can also be used to model discrete event streams.

Figure 2.2 shows the grammar for the extended language. The new constructs include simply

- a set of primitive behaviors (e.g., `seconds`),
- the undefined value ($\perp$), and
- the `prev` operator, which delays a behavior by a single time step.

[1] This chapter expands on previously published material [26].

$$
\begin{array}{rcl}
x \in \langle \text{var} \rangle & ::= & \text{(variable names)} \\
p \in \langle \text{prim} \rangle & ::= & \text{(primitive operators)} \\
u, v \in \langle \text{v} \rangle & ::= & \texttt{true} \mid \texttt{false} \mid \texttt{empty} \mid \texttt{(cons} \ \langle \text{v} \rangle \ \langle \text{v} \rangle \ \texttt{)} \mid \langle \text{prim} \rangle \mid \\
 & & \texttt{(}\lambda\texttt{(} \ \langle \text{var} \rangle^* \ \texttt{)} \ \langle \text{e} \rangle \ \texttt{)} \mid 0 \mid 1 \mid 2 \mid \ldots \\
e \in \langle \text{e} \rangle & ::= & \langle \text{v} \rangle \mid \langle \text{var} \rangle \mid \texttt{(} \ \langle \text{e} \rangle \ \langle \text{e} \rangle^* \ \texttt{)} \mid \texttt{(rec} \ \langle \text{var} \rangle \ \langle \text{e} \rangle \ \texttt{)} \mid \texttt{(if} \ \langle \text{e} \rangle \ \langle \text{e} \rangle \ \langle \text{e} \rangle \ \texttt{)}
\end{array}
$$

Figure 2.1: Grammar for purely functional subset of core PLT Scheme

$$
\begin{array}{rcl}
u, v \in \langle \text{fv} \rangle & ::= & \texttt{true} \mid \texttt{false} \mid \texttt{empty} \mid \texttt{(cons} \ \langle \text{fv} \rangle \ \langle \text{fv} \rangle \ \texttt{)} \mid \langle \text{prim} \rangle \\
 & & \texttt{(}\lambda\texttt{(} \ \langle \text{var} \rangle^* \ \texttt{)} \ \langle \text{fe} \rangle \ \texttt{)} \mid 0 \mid 1 \mid 2 \mid \ldots \mid \text{(primitive behaviors)} \mid \bot \\
e \in \langle \text{fe} \rangle & ::= & \langle \text{fv} \rangle \mid \langle \text{var} \rangle \mid \texttt{(} \ \langle \text{fe} \rangle \ \langle \text{fe} \rangle^* \ \texttt{)} \mid \texttt{(rec} \ \langle \text{var} \rangle \ \langle \text{fe} \rangle \ \texttt{)} \mid \texttt{(if} \ \langle \text{fe} \rangle \ \langle \text{fe} \rangle \ \langle \text{fe} \rangle \ \texttt{)} \mid \\
 & & \texttt{(prev} \ \langle \text{fe} \rangle \ \texttt{)}
\end{array}
$$

Figure 2.2: Grammar for FrTime (extends grammar in Figure 2.1)

This grammar defines a new language that allows a behavior to appear in any context in which Scheme would permit a (constant) value. Since Scheme's constructs are only defined to operate on constant values, they must be extended in order to work meaningfully with behaviors. The rest of this chapter discusses what it means to use behaviors with each of these constructs and how FrTime defines the extended semantics.

Before going into the details of the evaluation model, I will discuss some high-level concerns.

2.1 Discrete Modeling of Continuous Phenomena

Behaviors are intended to model phenomena that vary over continuous time. However, their implementation on digital hardware requires a discrete approximation. In FrTime, each behavior maintains an approximation of its current value, which updates in discrete

steps.

In most cases, the accuracy of the approximation depends on the frequency with which the updates occur; smaller sampling intervals result in smaller jumps between steps and therefore smoother curves that more closely follow the continuous ideal. Signals that have this property (e.g., the results of all stateless transformations and some stateful ones like *integral*) are said to be *convergent*, since they converge on their ideal values as the sampling interval tends toward zero.

It is possible, however, to define behaviors that do not converge. For example, the following expression counts the number of changes in a fine-grained timer called milliseconds:

(*collect-b* (*changes* milliseconds) 0 (λ (_ *n*) (*add1 n*)))

The value of this expression grows without bound, and at any given time t, its value is proportional to t times the rate at which milliseconds changes. On a slow machine, it might update only 20 times per second, while on a faster machine, it could update 100 times per second or more. As the sampling rate increases, the value of this expression grows more rapidly. This behavior is therefore said to *diverge*.

The existence of approximation errors and divergent behaviors are unfortunate but necessary consequences of the use of a discrete model. However, there are also benefits of using a discrete notion of time. Most importantly, like the presence of a clock in a digital circuit, discrete time supports synchronous execution, allowing the language to avoid glitch- and hazard-like conditions and to support extraction of consistent snapshots of a program's state.

2.2 Push-Driven Recomputation

A typical evaluator performs a pre-order traversal of an expression's abstract syntax tree, recursively reducing subexpressions to values until reaching the root. In FrTime, evaluation results in a graph of signals that exhibit different values over time, and the dataflow evaluator needs to recompute these values as new inputs become available.

A natural way to compute values over time is to re-apply the standard top-down tree-traversal whenever any of the input values change. Unfortunately, this *pull*-based approach

has several drawbacks. For one thing, many subexpressions' values may not change often (because the values they depend on change rarely), so the evaluator will perform many unnecessary recomputations. One key technique in avoiding this inefficiency is to maintain a cache with each subexpression's most recent value, and only recompute it when something upstream has changed.

Another problem is that several subexpressions may share the same values. A pull-based evaluator will naturally recompute these once for each subexpression that depends on them, which is problematic for non-idempotent updates (e.g., state accumulators). To deal with such nodes correctly, the evaluator needs to ensure that it doesn't update any node more than once in a given time step, for example by keeping a timestamp at each node and comparing it to the current time before recomputing.

FrTime avoids these complications by instead using a bottom-up, push-based recomputation strategy. Its evaluator maintains a graph that captures the flow of data between behaviors. Each node represents a behavior and caches that behavior's most recent value. When a node's value changes, it ensures that each node that refers to it is scheduled for recomputation. By traversing the graph in topological order, the update scheduler guarantees that each node updates at most once, and only when something on which it depends changes. When a value changes, everything that depends on it is recomputed within that time step. Thus values are always kept fresh, even if they are not needed. This prevents the *time leaks* that can arise in demand-driven implementations, where deferred computations accumulate until their results are needed.

2.3 Defining the Dataflow Extension

I now present the essence of this evaluation model through an executable interpreter. The interpreter consists of several distinct pieces:

- an *evaluator*, shown in Figure 2.15, which consumes FrTime expressions and reduces them to values. Values may include both constants and signals. The evaluator connects the signals into a graph that captures their data dependencies. The evaluator is a variation on a standard meta-interpreter for Scheme, with eager substitutions.

- a *snapshot evaluator*, shown in Figure 2.6, which consumes an expression and a

store, and returns the instantaneous value of the expression given the mappings in the store.

- an *updater*, shown in Figure 2.16, which iteratively updates a program's state once it has been evaluated.

A program's state consists of the following:

1. the current time,[2]
2. the store, represented as an association list of signals and their current values,
3. the set of dependencies, and
4. the set of signals in need of recomputation

This interpreter is presented for purposes of illustration. The real FrTime implementation uses DrScheme's integrated macro and module system [44] (instead of an interpreter) to extend the definitions of Scheme's core syntax and primitive operators and achieve the same semantics. Thus it reuses Scheme's evaluation mechanism to an even greater extent than the interpreter.

2.4 Primitive Operators and Constant Values

To explain how FrTime interprets programs, I will step through the evaluation of expressions that exercise various features. To begin, consider what happens if a user enters

(+ (− 2 1) (*modulo* seconds 3))

at the REPL. The expression is first evaluated by the *evaluate* procedure, the necessary fragments of which are shown in Figure 2.3.

At the top level, this expression is a function application, so it matches the (*f* . *args*) clause in the evaluator, which begins by recursively evaluating all of the subexpressions (i.e., the function expression and each of the arguments).

The function position contains the identifier +, which matches the last clause in the evaluator. The evaluator returns a structure containing three fields: the expression's value,

[2]The time does not actually influence evaluation; it is only present as a debugging aid.

```
;; <expr> → (make-result <val> <new deps> <new signals>)
(define (evaluate expr)
  (match expr
    ;; — code for most expression types elided —
    [(f . args) ; procedure applications:
     (match-let* ([($ result vs deps sigs) (evaluate-list (cons f args))]
                  [fv (first vs)] [argvs (rest vs)])
       (cond
         ;; — code for time-varying procedures elided —
         [(prim? fv) ; application of primitive procedures:
          (if (ormap signal? vs)
              (let ([new-sig (new-signal vs)])
                (make-result
                 new-sig (union deps (map (λ (d) (list d new-sig)) (filter signal? argvs)))
                 (union (list new-sig) sigs)))
              (make-result (apply (eval fv) args) deps sigs))]))]
    ;; values:
    [_ (make-result expr empty empty)]))
```

Figure 2.3: FrTime evaluator (excerpts for primitive procedures)

the new data dependencies, and the new signals. In this case, the identifier evaluates to itself, introducing no new data dependencies or signals, so the last two fields are empty.

The first argument to + is the function application (− 2 1). The next step is to recursively evaluate each of its subexpressions. In this case all of the arguments are already values. The function is a primitive, and none of the arguments are signals, so it defers to Scheme to perform the raw function application, yielding the value 1.

The second argument to + is the function application (*modulo* seconds 3). All of the arguments are values, but seconds is a signal, so the result is a new signal.

Signals are represented as structures of type *sig*. For example, the result of (*modulo* seconds 3) is:

#(*struct:sig* 0 (*modulo* seconds 3))

The 0 is its identification number, and (*modulo* seconds 3) is the expression that computes its value.

Because the new signal refers to (and depends on) seconds, the evaluator extends the dataflow graph with an edge from seconds to this new signal. Note that the evaluator does not compute signals' current values; that is done in a separate *update* step, described later.

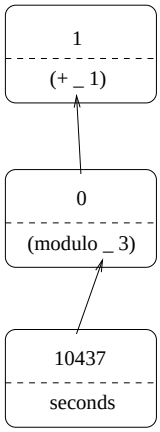


Figure 2.4: Dataflow graph for (+ (*modulo seconds* 3) (- 2 1))

Having evaluated (*modulo* seconds 3) to produce the signal

#(*struct:sig* 0 (*modulo* seconds 3))

the evaluator is ready to proceed with the evaluation of the top-level expression, which has by now been reduced to (+ 1 #(*struct:sig* 0 (*modulo* seconds 3))). One of the arguments is a signal, so the overall result is again a new signal, this time as follows:

#(*struct:sig* 1 (+ 1 #(*struct:sig* 0 (*modulo* seconds 3))))

I will subsequently abbreviate signals as just #(*struct:sig n* ...) when the expression is either unimportant or already shown.

Figure 2.4 shows the graph for the expression discussed above.

2.5 Updating Signal Values

The evaluator constructs a dataflow graph for an expression, but it does not compute the current values of any signals. This is done in a separate step by the *updater*. Signals values are kept in a *store*, which initially maps all signals to the undefined value, $\perp$.

The implementation of the updater's basic functionality is shown in Figure 2.5. It consumes a time *t*, a store, a set of dependencies (*deps*), and a set of *stale* signals.

Initially, the stale set contains all of the signals constructed during evaluation of the expression. The updater selects one of them for update, say #(*struct:sig* 0 ...). It then

```
;; update: <number> <store>
(define (update t store deps stale)
  (if (empty? stale)
      ;; — empty case elided —
      (let* ([ready-for-update (set- stale (transitive-deps stale deps))]
             [sig0 (first ready-for-update)])
        (match sig0
          [($ sig _ expr)
           (let ([val (snapshot expr store)])
             (values t (store-update sig0 val store) deps
                     (set- (if (eq? (store-lookup sig0 store) val)
                               stale
                               (union stale (immediate-deps (list sig0) deps)))
                           (list sig0))))]
          ;; — other cases elided —
          ))))
```

Figure 2.5: FrTime updater (excerpts for primitive procedures)

```
(define (snapshot expr store)
  (match expr
    [`(if ,c ,e1 ,e2) (if (snapshot c store)
                          (snapshot e1 store)
                          (snapshot e2 store))]
    [($ sig _ _) (snapshot (store-lookup expr store) store)]
    [($ switch _ _ _) (snapshot (store-lookup expr store) store)]
    [($ prev _ _) (snapshot (store-lookup expr store) store)]
    [`((λ ,vars ,body) . vals)
     (snapshot (foldl (λ (var arg body) (subst var arg body)) vars args))]
    [('λ . _) expr]
    [(p . vals)
     (apply (eval p) (map (λ (v) (snapshot v store)) vals))]
    [x x]))
```

Figure 2.6: A snapshot evaluator for the essence of FrTime

computes its value by a *snapshot* evaluation of its expression, (*modulo* seconds 3).

Snapshot evaluation, defined in Figure 2.6, computes the current value of a signal-containing expression, given a store containing the current values of the signals to which it refers. Suppose that the current store maps seconds to 0, so the expression's snapshot

value is 0. The store is updated to map #(*struct:sig* 0 ...) to 0. This differs from the previous value of $\perp$, so the updater adds all of #(*struct:sig* 0 ...)'s dependents to the set of stale signals. In our example, the only dependent is #(*struct:sig* 1 ...), which is already in the stale set, so this has no effect. However, in subsequent iterations, this mechanism keeps all of the signals consistent with each other as they change.

The updater next processes #(*struct:sig* 1 ...) in a store that maps #(*struct:sig* 0 ...) to 0. Its update proceeds similarly to that of the previous signal, except that its expression is (+ 1 #(*struct:sig* 0 ...)), so its snapshot evaluation yields the value 1. This signal has no dependents, so the set of stale signals becomes empty, and the update cycle is complete.

The updater next allows time to advance to the next step, then repeats the process described above. (In order to avoid monopolizing the CPU in the actual implementation, FrTime waits for a given amount of real time before proceeding.)

2.6 Scheduling of Updates

Consider the expression

(< seconds (+ 1 seconds))

This evaluates to a behavior that should always have the value **true**, since n is less than $n + 1$. However, when seconds changes, it makes two signals stale: the one associated with (< seconds (+ 1 seconds)) and the one associated with (+ 1 seconds). If the former is updated first, then the comparator will see the *new* value of seconds and the *old* value of (+ 1 seconds) (which is the same as seconds!), resulting in a **false** comparison. Once the (+ 1 seconds) signal updates, the (< seconds (+ 1 seconds)) signal becomes stale again, and when it updates the second time, it yields the expected value of **true**. This situation, in which a behavior momentarily exhibits an incorrect value, is called a *glitch*. Glitches are unacceptable because they hurt performance (by causing redundant computation) and, more importantly, because they produce incorrect values.

To eliminate glitches, the language must guarantee that, whenever it updates a signal, everything on which that signal depends is up-to-date. FrTime ensures this by updating signals in an order that respects the topology of the dataflow graph. The relevant line of code in the updater is the one that binds *ready-for-update* to (*set- stale* (*transitive-deps*

stale deps)). This computes the set of signals that depend transitively (but not reflexively) on any stale signal and removes them from consideration. The signals in the remaining set are guaranteed not to interfere, so the updater can choose any of them to process next. In fact, they could all be updated safely in parallel, but the sequential updater here simply picks the first signal in the list.

In practice, computing sets of transitive dependents and set differences like this would be too expensive. In the real implementation, each signal has an extra field that stores its *height* in the dataflow graph, which is greater than that of anything on which it depends. The updater uses a priority queue to process signals in increasing order of height, adding to each step a number of operations only logarithmic in the number of enqueued signals.

2.6.1 Subtleties of Memory Management

In the dataflow graph, there is a reference from each behavior to each other behavior whose definition refers to it. These dataflow references, which point from referent to referrer, are in the opposite direction from the references that occur naturally in the program. If implemented naïvely, these references would make reachability in FrTime programs a symmetric relation, preventing garbage collection of any part of the dataflow graph. To avoid this problem, the real implementation uses *weak* references for the dataflow graph, which are not counted by the garbage collector.

Using weak references in the dataflow graph eliminates one source of memory leakage. However, it is still possible for a dead signal to survive collection; for example, if it is being updated when the garbage collector runs, then it will temporarily appear live and therefore not be collected. In addition, its references to other signals will keep them alive as well, and so on.

While there is no way to eliminate such transient liveness from the system completely, it is important to reduce it as much as possible. For a simple example of why this is crucial, consider the following program:

```
(collect-b (changes seconds) empty
           (λ (_ lst)
             (append (take 4 lst)
                     (list (− milliseconds (current-milliseconds))))))
```

This program produces a steady stream of new behaviors (one per second) that count the number of milliseconds since their birth. The last five such behaviors are retained in a list, and any older ones are garbage. Since all of these behaviors depend directly on seconds, as soon as milliseconds updates, they are all placed on the update queue. Assuming these are the majority of the signals in the system, on average about half of them will be on the update queue when the collector runs, so if the update queue uses strong references, half of the (dead) signals will survive. Therefore, the expected number of signals remaining after a collection grows continually over time, eventually choking the system. To prevent such scenarios from arising in practice, the update queue must also use weak references.

2.7 Time-Varying Control Flow

The previous sections have explored examples in which behaviors are provided as arguments to primitive procedures. However, this is not the only way in which a program might reasonably use behaviors. For example, consider the following procedure definition:

```
(λ (x)
  (if (zero? x)
      (add1 seconds)
      (/ 6 x)))
```

Suppose that a program applies this function to the constant 0. Its evaluation begins similarly to that of a primitive procedure application, except that the function expression is a λ-abstraction, which evaluates to itself as shown in Figure 2.7. While interpreting the application, *fv* matches ('λ *vars body*), so the evaluator performs a β-substitution, yielding the following expression:

```
(if (zero? 0)
    (add1 seconds)
    (/ 6 0))
```

This expression matches the ('if *c t e*) clause in Figure 2.7. It reduces (*zero?* 0) to **true** and selects the first branch for evaluation. This is exactly how a conditional expression would be evaluated under a standard call-by-value semantics. The fact that the branch evaluates

```
;; <expr> → (make-result <val> <new deps> <all signals> <stale signals>)
(define (evaluate expr)
  (match expr
    ;; — other cases elided —
    [('if c t e) ; conditional expressions:
     (match-let ([($ result cv deps all-sigs stale-sigs) (evaluate c)])
       (cond [(signal? cv)
              (let* ([swc (make-switch '(if ,□ ,t ,e) cv)]
                     [fwd (make-signal swc)])
                (make-result fwd (union deps '((,cv ,swc) (,swc ,fwd)))
                             (union '(,swc ,fwd) all-sigs) (union '(,swc) stale-sigs)))]
             [cv   (evaluate t)]
             [else (evaluate e)]))]
    [('λ . _) (make-result expr empty empty empty)]
    [(f . args) ; procedure applications:
     (match-let* ([($ result vs deps sigs stale) (evaluate-list (cons f args))]
                  [fv (first vs)] [argvs (rest vs)])
       (match fv
         [('λ vars body) ; application of lambda abstractions:
          (match-let ([($ result v deps1 all-sigs1 stale-sigs1)
                       (evaluate (foldl (λ (var arg body) (subst arg var body))
                                        body vars argvs))])
            (make-result v (union deps deps1) (union sigs all-sigs1)
                         (union stale stale-sigs1)))]
         ;; — other cases elided —
         ))]))
```

Figure 2.7: FrTime evaluator (excerpts for **if** expressions and λ-abstractions)

to a signal is irrelevant, since once the branch is selected, the *conditional* aspect of the evaluation is complete.

The interesting case is when the value of the condition is a behavior. Since the condition's value might change, the evaluator cannot simply select one branch or the other for evaluation. Rather, it must dynamically take the value of whichever branch is selected by the current value of the condition.

A naïve strategy for such dynamic switching would be to evaluate both branches and simply choose between their values according to the condition. However, this strategy fails in general. For example, in the function above, the second branch would raise a *division-by-zero* error if evaluated while the condition were **true**. The evaluator could work around this

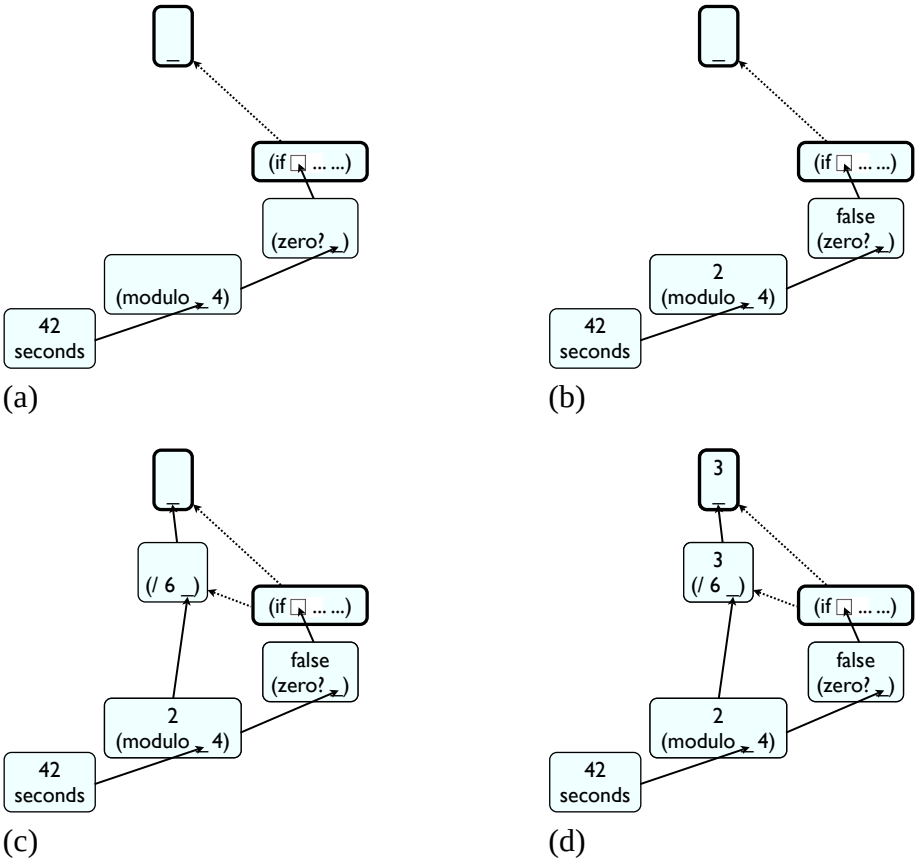


Figure 2.8: Dataflow graphs for a conditional expression

problem by catching and ignoring errors within the inactive branch, but it would still have problems in other situations, such as when the conditional is used to terminate recursion.

To avoid these potential problems, FrTime implements conditionals by dynamically re-evaluating the branches as their values are needed. Each time the condition acquires a new value, FrTime evaluates the appropriate branch and uses its (potentially time-varying) value until the condition changes again.

For example, consider the application of the above procedure to (*modulo* seconds 4). Evaluation initially proceeds according to the rules presented above. The argument reduces to #(*struct:sig* 2 (*modulo* seconds 4)), which is substituted into the body of the λ abstraction to yield the following:

(**if** (*zero?* #(*struct:sig* 2 . . .))
(*add1* seconds)

(/ 6 #(*struct:sig* 2 . . .)))

The condition reduces to the signal #(*struct:sig* 3 . . .). Because this is a signal, the evaluator's (*signal? cv*) test succeeds, and it creates two new signals: a *switch* and a *forwarder*, abbreviated *swc* and *fwd* respectively in the code. These are the two signals shown in bold in Figure 2.7 (a).

A switch is a special kind of signal with the following structure:

#(*struct:switch* 4 (**if** □
(*add1* seconds)
(/ 6 #(*struct:sig* 2 . . .)))
#(*struct:sig* 3 . . .))

The first field is an identification number, as in an ordinary signal. However, while an ordinary signal would contain a single expression, a switch's expression is split into a *context* and a *trigger*. For conditionals the trigger is the condition, and the context is the conditional expression with a hole (□) in place of the condition.

The essential idea is that the trigger's values populate the context's hole, and the resulting expression is re-evaluated each time the trigger changes in order to produce a new, potentially time-varying, value. The forwarder is dynamically connected to the most recent result and just copies the result's current value as it changes. A forwarder is an ordinary signal whose defining expression is simply another signal:

#(*struct:sig* 5 #(*struct:sig* 4 . . .))

Recall that the evaluation step does not compute signals' values, and without knowing the trigger's current value, the evaluator cannot determine the switch's current branch. This step is therefore deferred until the update step.

During the update step, the signals created by the evaluator are assigned values. The ordinary signals (e.g., the ones numbered 2 and 3) update as described above. Suppose that signal 2's value is 2, so signal 3's is **false**. The graph now looks like the one shown in Figure 2.7 (b). When the updater processes the *switch* (signal 4), it matches the #(*struct:switch* . . .) case shown in Figure 2.9. The most important step is in the code that reads:

(*evaluate* (*subst* (*store-lookup trigger store*) □ *ctxt*))

```
(define (update t store deps stale)
  (if (empty? stale)
      ;; — empty branch elided —
      (let* ([ready-for-update (set- stale (transitive-deps stale deps))]
             [sig (first ready-for-update)])
        (match sig
          ;; — other cases elided —
          [($ switch _ ctxt trigger) ; updating a switch:
           (match-let*
             ([fwd (first (filter (λ (sig1) (and (sig? sig1) (eq? (sig-expr sig1) sig0)))
                                  (map first store)))]
              [old (set- (transitive-deps (list sig0) deps)
                         (cons fwd (transitive-deps (list fwd) deps)))]
              [deps1 (filter (λ (dep) (not (member (second dep) old))) deps)]
              [($ result v new-deps new-sigs)
               (evaluate (subst (store-lookup trigger store) □ ctxt))])
             (values t (append (map (λ (s) (list s ⊥)) new-sigs)
                               (filter (λ (m) (not (member (first m) old)))
                                       (store-update sig0 v store)))
                     (union (map (λ (s) (list sig0 s)) new-sigs)
                            deps1 new-deps (if (signal? v) (list (list v fwd)) empty))
                     (set- (union (list fwd) new-sigs stale) (cons sig0 old))))])))
```

Figure 2.9: The part of the updater that handles switching

This substitutes the trigger's current value—**false**—for the hole in the context and evaluates the result. Since this is the first time updating this switch, there is no previous subgraph. Evaluation creates a new signal, #(*struct:sig* 6 (*/* 6 #(*struct:sig* 2 ...))). It also updates the store to map the switch's value to this new signal and adds a dependency from this signal to the forwarder, and between the new signal and the switch. The graph then looks like the one shown in Figure 2.7 (c). As the update cycle proceeds, signal 6 acquires the value 3, and the forwarder (signal 5) copies it. The graph is then stable, and its state is as in Figure 2.7 (d).

In the next update cycle, *seconds* increments, and the change propagates through the rest of the graph. Because the trigger does not change, the switch does not update, and the graph's structure remains the same. Its state is shown in Figure 2.7 (e).

In the third update cycle, shown in Figure 2.7 (f), the trigger changes and causes the switch to update. The old signal, which would erroneously divide by zero if left to update,

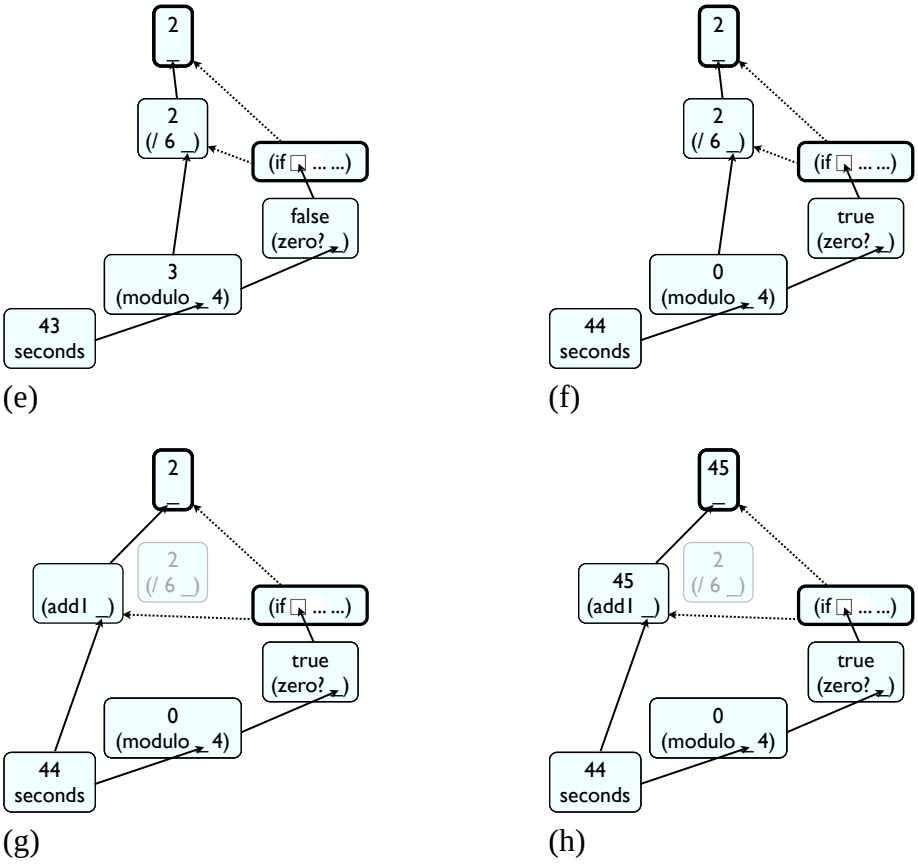


Figure 2.10: Dataflow graphs for a conditional expression

is removed from the graph, and the evaluation of the switch's expression yields a new graph fragment, which is spliced into the rest of the graph. The result is depicted in Figure 2.7 (g). The new signal updates, and its value propagates to the forwarder, as shown in Figure 2.7 (h).

The dashed arcs from the switch to its current branch are crucial for the timely removal of old signals. These ensure that, when the trigger changes, all of the obsolete signals will be destroyed before having a chance to update. They are drawn with dashes because they represent control, rather than data, dependencies.

The switching mechanism also plays a crucial role in the implementation of time-varying procedures. In that case, the trigger is the procedure signal, and the context is the procedure application with a hole in the function position. Each time the procedure

```
;; <expr> → (make-result <val> <new deps> <all signals> <stale signals>)
(define (evaluate expr)
  (match expr
    ;; — other cases elided —
    [(f . args)
     (match-let* ([($ result vs deps sigs) (evaluate-list (cons f args))]
                  [fv (first vs)] [argvs (rest vs)])
       (match fv
         ;; — other cases elided —
         ;; signal in function position:
         [($ signal _)
          (let* ([swc (new-switch (cons □ argvs) fv)]
                 [fwd (new-signal swc)])
            (make-result fwd (union deps `((,fv ,swc) (,swc ,fwd))) (union sigs `(,swc ,fwd))))]
         ))]))
```

Figure 2.11: FrTime evaluator (excerpts for time-varying procedures)

changes, its current value is substituted into the hole, and the resulting expression is evaluated to produce a new fragment of dataflow graph. The similarity between conditionals and time-varying procedures is clear from the amount of code shared between Figures 2.7 and 2.11.

In the actual implementation, there is no interpreter, so the values that represent behaviors must be directly applicable as procedures. In FrTime, this is made possible by PLT Scheme's support for the use of data structures as procedures. Other languages provide different features that can achieve the same purpose. For example, in a language like C++, behaviors can be instances of a class with a function-call operator. In JavaScript, behaviors can inherit from the class `Function`. As a fallback, in any dynamically typed functional language, behaviors can simply be represented as procedures.

2.8 Remembering Past Values

The constructs presented so far only allow processing of behaviors' values within a single instant. In practical applications, however, we need the ability to remember the past. For example, to record the sequence of characters that have been typed into a text box, the language needs a way to accumulate state.

```
(define (update t store deps stale)
  (if (empty? stale)
      (let* ([prevs (map first (filter prev? store))])
        (values (add1 t) (foldl (λ (prev new-store)
                                  (store-update prev (store-lookup (prev-signal prev) store)
                                                new-store))
                                store prevs)
                deps (immediate-deps prevs deps)))
      ;; — non-empty branch elided —
      )))
```

Figure 2.12: An update engine for the essence of FrTime (excerpt for `prev` statements)

FrTime provides this capability by means of a *prev* operator, which consumes a signal and returns a new signal whose value is initially undefined ($\perp$) and subsequently equal to the value its argument had in the previous time step. As Figure 2.12 shows, all *prev* signals are updated while the system is in a stable state (i.e., when the set of stale signals is empty). The *prev* signals therefore effectively update between two time steps, when all signals are consistent for the previous step, and none of them have changed in the next step. This property is what allows them to copy values soundly from one time step to the next. Moreover, this peculiar update protocol means that *prev* signals need not depend on their arguments, and therefore need not update in response to changes in their arguments. Since *prev*s may refer to each other, it is crucial that all of their updates occur atomically. This is why the *store-lookup*s in Figure 2.12 all use the original store.

While there are several operators that accumulate history, they can all be expressed in terms of *prev*. For example, to keep track of the value a signal had k time steps ago, one can simply compose k uses of *prev*. Similarly, to compute a numerical approximation of a time integral, one can define a signal whose value is initially zero, then the sum of its own previous value and the current value of its argument times the duration of the time step.

Of course, in a real implementation, constructing long delays through chains of atomic delays would be wasteful, as it would require many updates in each time step. Moreover, in many interactive applications (e.g., games), the programmer may want to delay a signal by a specific amount of real time, which may not correspond to any fixed number of FrTime's update cycles. (The rate at which FrTime processes an update cycle may vary according to the hardware configuration, the number of active signals in the system, the behavior of the

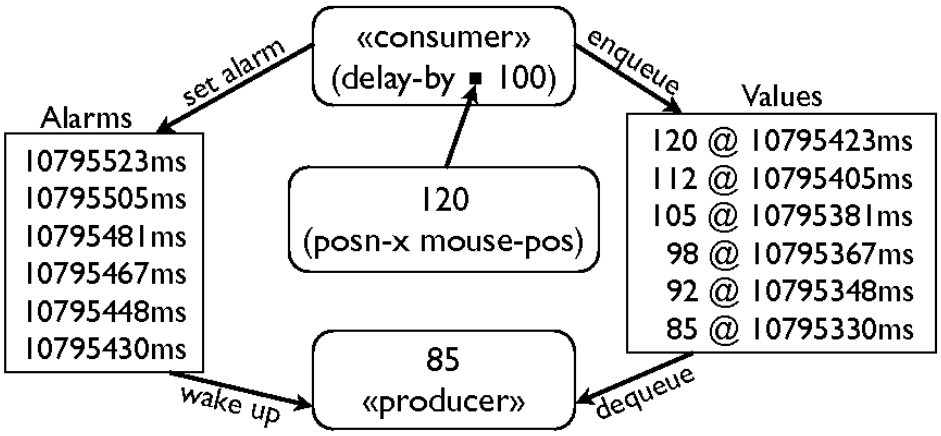


Figure 2.13: The implementation of *delay-by*

garbage collector, and other factors beyond the program's control.)

To support real-time interaction, FrTime provides a *delay-by* operator that efficiently delays a signal by a given amount of real time dt (or at least, by as close an approximation as it can achieve). This *delay-by* operator constructs a pair of signals called a *consumer* and a *producer*. The two signals communicate through a shared queue that maintains a sliding window of the argument's recent values.

Figure 2.13 illustrates how the consumer and producers cooperate to delay a behavior. The consumer (top) depends on the argument (middle), so it updates every time the argument changes, adding an entry to the queue (right) with the current time t and the argument's current value.

Alarms. The dataflow engine provides an *alarm* mechanism, whereby a signal may be scheduled for update at (or shortly after) a given point in real time. Internally, the engine maintains an ordered collection of pairs, $(\text{signal}, \text{wake-time})$, and at the beginning of each update cycle, it enqueues for update all signals whose wake-times have elapsed. To ensure that signals are updated as soon as possible after their requested wake-times, the engine sets a timeout at the earliest wake-time before asking the runtime to wait for external events.

The consumer uses this alarm mechanism (left) to schedule the producer for update at time $t+dt$. Thus, when the producer (bottom) updates, at least dt milliseconds have elapsed since some change in the argument's value. The producer dequeues the appropriate value

```
;; <expr> → (make-result <val> <new deps> <all signals>)
(define (evaluate expr)
  (match expr
    [('rec x e)
     (match-let* ([sig (make-signal ⊥)]
                  [($ result val deps sigs) (evaluate (subst sig x e))])
       (vector-set! sig 2 val)
       (make-result val (union deps (if (signal? val) (list (list val sig)) empty))
                    (cons sig sigs)))]
    ;; — other cases elided —
    ))
```

Figure 2.14: FrTime evaluator (excerpts for recursive bindings)

and emits it.

2.9 Recursion

An important application of history is in the expression of self-referential signal networks, which are necessary for many practical programs. As an illustration of the power of self-reference, observe that time itself can be expressed as a signal whose value starts as zero and is then equal to one greater than its previous value:

(**rec** *t* (*init* 0 (*add1* (*prev t*))))

Here, *rec* is a fixed point operator, and the *init* operator uses the value of its first argument as long as the second argument is ⊥. (Typically, primitive procedures like *add1* are lifted in a ⊥-strict fashion, so they return ⊥ if any of their arguments are ⊥.)

Figure 2.14 shows how FrTime implements the **rec** construct. The body is evaluated with the variable bound to a placeholder signal, whose value starts out undefined (⊥). Then the signal's value is rebound to the result of the evaluation. In the general case, the result is a signal, so the placeholder signal depends on the result, and the result also refers to the signal. If this chain of reference is not broken by a *prev*, then a cycle in the dataflow graph results, rendering topological evaluation impossible. In the expression above, a *prev* intervenes, so the graph remains acyclic.

If one were to define an ill-founded behavior, such as

(**rec** *x* (*init* **true** (*not x*)))

then the evaluator would err when it tried to select the next signal for update. Since every signal would depend transitively on another signal with an update pending, it would not be safe to update any signal. In the actual implementation, FrTime discovers the tight cycle when it rebinds the placeholder and traverses the graph to update the height assignment.

2.10 Event Streams

The preceding sections have explained how behaviors interact with all of the features in standard (purely functional) Scheme. FRP, however, also supports the modeling of sequences of discrete phenomena, which are called *event streams*. Behaviors and events are in some sense duals, and either one can be used as a basis to implement the other, at least in a model where time is fundamentally discrete. In FrTime, behaviors are primitive, and event streams are modeled as structured behaviors carrying the time at which the last event occurred, and the collection of events that occurred at that time. It is easy to define the standard event-processing combinators as lifted functions over these behaviors. For example:

```
(define (map-e f e)
  (make-event (event-time e) (map f (event-occurrences e))))

(define (filter-e p e)
  (make-event (event-time e) (filter p (event-occurrences p))))

(define (merge-e . es)
  (let ([last-time (apply max (map event-time es))])
    (make-event
      last-time
      (apply append
             (map event-occurrences
                  (filter (λ (e) (= (event-time e) last-time))
                          es))))))
```

Stateful event processors (e.g., *accum-b*, *collect-b*) can be expressed via recursion and *prev*.

2.11 Support for REPL-based Interactive Programming

Another concern for FrTime is supporting a read-evaluate-print loop (REPL) for interactive, incremental program development. While not technically part of the language itself, this is an important feature from the standpoint of usability, and one that Scheme and other Lisp dialects have long provided.

FrTime supports REPL interaction by allowing the dataflow computation to run concurrently with the REPL. The language dedicates one thread of control, called the *dataflow engine*, to the update algorithm, while another thread manages the DrScheme REPL, allowing the user to enter expressions at the prompt. The threads communicate through a message queue; at the beginning of each update cycle, the engine empties its queue and processes the messages. When the user enters an expression at the 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.

This approach differs from the one taken by the Haskell FRP systems [39, 74]. In those, a program specifies the structure of a dynamic dataflow computation, but the actual reactivity is implemented in an interpreter called *reactimate*. The use of *reactimate* is an artifact of a particular implementation strategy, in which an external entity was required to drive reactive computations by pulling values from streams. *Reactimate* runs in an infinite loop, blocking interaction with the REPL until the computation is finished. It therefore creates a closed world, which is problematic for applications that need to support REPL-style interaction in the middle of their execution.

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.

```
;; <expr> → (make-result <val> <new deps> <all signals> <stale signals>)
(define (evaluate expr)
  (match expr
    [('rec x e)
     (match-let* ([sig (make-signal ⊥)]
                  [($ result val deps sigs stale) (evaluate (subst sig x e))])
       (set-sig-expr! sig val)
       (make-result val (union deps (if (signal? val) (list (list val sig)) empty))
                    (cons sig sigs) (cons sig stale)))]
    [('prev e)
     (match-let* ([($ result val deps sigs stale) (evaluate e)]
                  [new-sig (make-prev val)])
       (make-result new-sig deps (cons new-sig sigs) stale))]
    [('if c t e)
     (match-let ([($ result cv deps all-sigs stale-sigs) (evaluate c)])
       (cond [(signal? cv)
              (let* ([swc (make-switch `(if ,□ ,t ,e) cv)]
                     [fwd (make-signal swc)])
                (make-result fwd (union deps `((,cv ,swc) (,swc ,fwd)))
                             (union `(,swc ,fwd) all-sigs) (union `(,swc) stale-sigs)))]
             [cv  (evaluate t)]
             [else (evaluate e)]))]
    [('λ . _) (make-result expr empty empty empty)]
    [(f . args)
     (match-let* ([($ result vs deps sigs stale) (evaluate-list (cons f args))]
                  [fv (first vs)] [argvs (rest vs)])
       (match fv
         [('λ vars body)
          (match-let ([($ result v deps1 all-sigs1 stale-sigs1)
                       (evaluate (foldl (λ (var arg body) (subst arg var body))
                                        body vars argvs))])
            (make-result v (union deps deps1) (union sigs all-sigs1)
                         (union stale stale-sigs1)))]
         [($ signal _)
          (let* ([swc (new-switch (cons □ argvs) fv)]
                 [fwd (new-signal swc)])
            (make-result fwd (union deps `((,fv ,swc) (,swc ,fwd)))
                         (union sigs `(,swc ,fwd)) (union stale `(,swc))))]
         [(? prim?)
          (if (ormap signal? vs)
              (let ([new-sig (new-signal vs)])
                (make-result
                 new-sig (union deps (map (λ (d) (list d new-sig)) (filter signal? argvs)))
                 (union (list new-sig) sigs) (union (list new-sig) stale)))
              (make-result (apply (eval fv) args) deps sigs stale))]))]
    [_ (make-result expr empty empty empty)]))
```

Figure 2.15: An evaluator for the essence of FrTime

```
(define (update t store deps stale)
  (if (empty? stale)
      (let* ([prevs (map first (filter prev? store))])
        (values (add1 t)
                (foldl (λ (prev new-store)
                          (store-update prev (store-lookup (prev-signal prev) store)
                                        new-store))
                       store prevs)
                deps (immediate-deps prevs deps)))
      (let* ([ready-for-update (set- stale (transitive-deps stale deps))]
             [sig (first ready-for-update)])
        (match sig
          [($ sig _ expr)
           (let ([val (snapshot expr store)])
             (values t (store-update sig val store)
                     deps
                     (set- (if (eq? (store-lookup sig store) val)
                               stale
                               (union stale (immediate-deps (list sig) deps)))
                           (list sig))))]
          [($ switch _ inner-ctxt trigger)
           (match-let* ([(fwds) (filter (λ (sig1) (and (sig? sig1)
                                                       (eq? (sig-expr sig1) sig)))
                                        (map first store))]
                        [(fwd) (first fwds)]
                        [(old) (set- (transitive-deps (list sig) deps)
                                     (cons fwd (transitive-deps (list fwd) deps)))]
                        [(deps1) (filter (λ (dep) (not (member (second dep) old))) deps)]
                        [($ result v new-deps new-sigs new-stale)
                         (evaluate (subst (store-lookup trigger store)
                                          □ inner-ctxt)
                                   outer-ctxt)])
             (values t (append (map (λ (s) (list s ⊥)) new-sigs)
                               (filter (λ (m) (not (member (first m) old)))
                                       (store-update sig v store)))
                     (union (map (λ (s) (list sig s)) new-sigs)
                            deps1 new-deps (if (signal? v) (list (list v fwd)) empty))
                     (set- (union (list fwd) new-stale stale) (cons sig old))))])))))
```

Figure 2.16: An update engine for the essence of FrTime

Chapter 3

Semantics

This chapter presents 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 3.1 shows the grammars for values, expressions, and evaluation contexts. A system is parameterized over a set of base constants and primitive operators (which operate on these base constants). Values include the undefined value ($\perp$), booleans, primitive procedures, λ-abstractions, and signals. Expressions include values, procedure applications, conditionals, and `prev` statements. Evaluation contexts [41] enforce a left-to-right, call-by-value order on subexpression evaluation.

Figure 3.2 shows the types of various elements of the semantics, along with the letters we use conventionally to represent them. The δ function, a parameter to the language, specifies how primitive operators evaluate. P represents the program, of which one (possibly trivial) expression is evaluated at each moment in time. D denotes a dependency relation: the set of values on which various signals depend. Σ and I name sets of signals, and S identifies a store, which maps signals to values.

Figure 3.3 describes how a FrTime program evaluates to produce a dataflow graph. Transitions operate on triples containing a signal set I, a dependency relation D, and an expression. I (the set of *internal* updates) represents the set of signals whose values will need to be computed once the expression has fully evaluated. During graph construction, the I parameter accumulates newly created signals, and D accumulates their dependencies. Sometimes a single rule creates multiple signals, some of which depend on the others, so that I need not accumulate all of the new signals.

$$
\begin{array}{rcl}
x \in \langle \mathrm{var} \rangle & ::= & \text{(variable names)} \\
p \in \langle \mathrm{prim} \rangle & ::= & \text{(primitive operators)} \\
t \in \langle \mathrm{time} \rangle & ::= & \text{(moments in time)} \\
\sigma \in \langle \mathrm{loc} \rangle & ::= & \texttt{(prev}\ \langle \mathrm{E} \rangle\ \langle \mathrm{loc} \rangle\ \texttt{)} \mid \texttt{(sig}\ \langle \mathrm{E} \rangle\ \langle \mathrm{e} \rangle\ \texttt{)} \mid \texttt{(switch}\ \langle \mathrm{E} \rangle\ \langle \mathrm{E} \rangle\ \langle \mathrm{loc} \rangle\ \texttt{)} \\
u, v \in \langle \mathrm{v} \rangle & ::= & \bot \mid \texttt{true} \mid \texttt{false} \mid \langle \mathrm{prim} \rangle \mid \texttt{(}\lambda\texttt{(}\ \langle \mathrm{var} \rangle^*\ \texttt{)}\ \langle \mathrm{e} \rangle\ \texttt{)} \mid \langle \mathrm{loc} \rangle \mid \ldots \\
e \in \langle \mathrm{e} \rangle & ::= & \langle \mathrm{v} \rangle \mid \langle \mathrm{var} \rangle \mid \texttt{(}\ \langle \mathrm{e} \rangle\ \langle \mathrm{e} \rangle^*\ \texttt{)} \mid \texttt{(rec}\ \langle \mathrm{var} \rangle\ \langle \mathrm{e} \rangle\ \texttt{)} \mid \texttt{(prev}\ \langle \mathrm{e} \rangle\ \texttt{)} \mid \texttt{(if}\ \langle \mathrm{e} \rangle\ \langle \mathrm{e} \rangle\ \langle \mathrm{e} \rangle\ \texttt{)} \\
E \in \langle \mathrm{E} \rangle & ::= & \texttt{[]} \mid \texttt{(}\ \langle \mathrm{v} \rangle^*\ \langle \mathrm{E} \rangle\ \langle \mathrm{e} \rangle^*\ \texttt{)} \mid \texttt{(prev}\ \langle \mathrm{E} \rangle\ \texttt{)} \mid \texttt{(if}\ \langle \mathrm{E} \rangle\ \langle \mathrm{e} \rangle\ \langle \mathrm{e} \rangle\ \texttt{)}
\end{array}
$$

Figure 3.1: Grammars for values, expressions, evaluation contexts, and signal types

$$
\begin{array}{rcll}
\delta & : & \langle \mathrm{prim} \rangle \times \langle \mathrm{v} \rangle \times \ldots \to \langle \mathrm{v} \rangle & \text{(primitive evaluation)} \\
P & : & \langle \mathrm{time} \rangle \to \langle \mathrm{e} \rangle & \text{(program fragments)} \\
\Sigma, I & \subset & \langle \mathrm{loc} \rangle & \text{(signal set)} \\
D & \subset & \langle \mathrm{v} \rangle \times \langle \mathrm{loc} \rangle \times \{\texttt{true}, \texttt{false}\} & \text{(tagged dependency relation)} \\
S & : & \langle \mathrm{v} \rangle \to \langle \mathrm{v} \rangle & \text{(signal value)}
\end{array}
$$

Figure 3.2: Semantic domains and operations

The first three rules ($\underline{\delta}$, $\underline{\beta_v}$, and $\underline{\text{IF}}$) are special cases for evaluation steps that only involve constants. The underlining of these rule names distinguishes them from their *lifted*, or signal-aware, counterparts. The underlined rules behave as would be expected in a standard call-by-value semantics.

The next three rules (δ, β_v, and IF) define the behavior of primitive application, application of λ-abstractions, and conditional expressions (respectively) in the presence of signals. Understanding them requires a knowledge of the different varieties of signals and what they mean.

It is instructive to view signals from two different perspectives. One view is as abstract entities that have an associated (constant) value at each point in time. This view is embodied semantically by the store (S), which explicitly maps signals to their values at a given point during evaluation. This view naturally gives rise to the notion of *snapshot evaluation* (Figure 3.4), in which signals evaluate to their projections in a given store.

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

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

$$\begin{array}{c} \langle I, D, E[(\texttt{if true}\ e_1\ e_2)] \rangle \rightarrow \langle I, D, E[e_1] \rangle \\ \langle I, D, E[(\texttt{if false}\ e_1\ e_2)] \rangle \rightarrow \langle I, D, E[e_2] \rangle \end{array} \quad (\underline{\text{IF}})$$

$$\frac{\{v_1, \ldots, v_n\} \cap \langle \text{loc} \rangle = \Sigma \neq \emptyset \qquad \sigma = (\texttt{sig}\ E\ (p\ v_1 \ldots v_n))}{\langle I, D, E[(p\ v_1 \ldots v_n)] \rangle \rightarrow \langle I \cup \{\sigma\}, D \cup (\Sigma \times \{\sigma\}), E[\sigma] \rangle} \quad (\delta)$$

$$\frac{\sigma_g = (\texttt{switch}\ E\ ([\,]\ v_1 \ldots v_n)\ \sigma) \qquad \sigma_f = (\texttt{sig}\ E\ \sigma_g)}{\langle I, D, E[(\sigma\ v_1 \ldots v_n)] \rangle \rightarrow \langle I \cup \{\sigma_g\}, D \cup \{(\sigma, \sigma_g), (\sigma_g, \sigma_f)\}, E[\sigma_f] \rangle} \quad (\beta_v)$$

$$\frac{\sigma_g = (\texttt{switch}\ E\ (\texttt{if}\ [\,]\ e_1\ e_2)\ \sigma) \qquad \sigma_f = (\texttt{sig}\ E\ \sigma_g)}{\langle I, D, E[(\texttt{if}\ \sigma\ e_1\ e_2)] \rangle \rightarrow \langle I \cup \{\sigma_g\}, D \cup \{(\sigma, \sigma_g), (\sigma_g, \sigma_f)\}, E[\sigma_f] \rangle} \quad (\text{IF})$$

$$\frac{\sigma_g = (\texttt{switch}\ E\ ((\lambda\ (x)\ e)\ [\,])\ \sigma_f) \qquad \sigma_f = (\texttt{sig}\ E\ \sigma_g)}{\langle I, D, E[(\texttt{rec}\ x\ e)] \rangle \rightarrow \langle I \cup \{\sigma_g\}, D \cup \{(\sigma_g, \sigma_f)\}, E[\sigma_f] \rangle} \quad (\text{REC})$$

$$\frac{\sigma = (\texttt{prev}\ E\ \sigma')}{\langle I, D, E[(\texttt{prev}\ \sigma')] \rangle \rightarrow \langle I, D, E[\sigma] \rangle} \quad (\text{PREV})$$

Figure 3.3: Evaluation rules

$$E[(p\ v_1 \ldots v_n)] \rightarrow_S E[\delta(p, v_1, \ldots, v_n)] \quad (\delta_S)$$

$$E[((\lambda\ (x_1 \ldots x_n)\ e)\ v_1 \ldots v_n)] \rightarrow_S E[e[v_1/x_1] \ldots [v_n/x_n]] \quad (\beta_{vS})$$

$$\begin{array}{c} E[(\texttt{if true}\ e_1\ e_2)] \rightarrow_S E[e_1] \\ E[(\texttt{if false}\ e_1\ e_2)] \rightarrow_S E[e_2] \end{array} \quad (\text{IF}_S)$$

$$E[\sigma] \rightarrow_S E[S(\sigma)] \quad (\sigma_S)$$

Figure 3.4: Snapshot rules

Since the underlying model is call-by-value, another sensible view of signals is as expressions that return different values to their contexts over time. This view is embodied semantically in the representation of each signal as an algebraic structure containing an evaluation context E and the information needed to compute the signal's current value. This view helps to maintain and elucidate the relationship between the dataflow and call-by-value aspects of the language's evaluation model. These signal structures are as follows:

$$\frac{I \ni \sigma = (\texttt{sig}\ E\ e) \notin D^+(I) \qquad e \rightarrow_S^* v}{\langle t, S, D, I\rangle \hookrightarrow \langle t, S[\sigma \mapsto v], D, (I \setminus \{\sigma\}) \cup \left\{ \begin{array}{ll} D(\sigma) & \text{if } v \neq S(\sigma) \\ \emptyset & \text{otherwise} \end{array} \right. \rangle} \quad \text{(U-SIG)}$$

$$\frac{\begin{array}{c}\sigma_g = (\texttt{switch}\ E_0\ E\ \sigma) \in I \qquad \sigma_f = (\texttt{sig}\ E_0\ \sigma_g) \qquad \Sigma = D^+(\sigma_g) \setminus D^*(\sigma_f) \\ \langle I \setminus \Sigma, D \setminus \langle \text{loc}\rangle \times \Sigma, E_0[E[S(\sigma)]]\rangle \rightarrow^* \langle I', D', E_0[v]\rangle \qquad \Sigma' = I' \setminus (I \setminus \Sigma)\end{array}}{\langle t, S, D, I\rangle \hookrightarrow \langle t, S[\sigma_g \mapsto v], D' \cup \{(v, \sigma_f)\} \cup (\{\sigma_g\} \times \Sigma'), (I' \setminus \{\sigma_g\}) \cup \{\sigma_f\}\rangle} \quad \text{(U-SWC)}$$

$$\frac{\begin{array}{c}P(t) = e \qquad \langle \emptyset, D, e\rangle \rightarrow^* \langle I, D', v\rangle \\ S' = S[(\texttt{prev}\ E\ \sigma') \mapsto S(\sigma')]_{(\texttt{prev}\ E\ \sigma') \in \text{dom}(S)}\end{array}}{\langle t-1, S, D, \emptyset\rangle \overset{v}{\hookrightarrow} \langle t, S', D', I \cup D'(\{\sigma \in \text{dom}(S) \mid S(\sigma) \neq S'(\sigma)\})\rangle} \quad \text{(U-ADV)}$$

Figure 3.5: Update rules

`sig` These signals are simple time-varying values, defined by a purely functional transformation over other time-varying values. Each `sig` signal identifies the evaluation context that created it and the expression whose snapshot-evaluation produces its values over time. Most `sig` signals are created by the δ rule, which deals with application of primitive procedures to signals.

`prev` Such a signal takes on exactly the values of some other signal, but delayed by one instant. By chaining together several `prev`s, one can implement longer delays, and in general `prev` supports the construction of signals that compute arbitrarily complex functions of the system's history.

`switch` These signals correspond to places where the value of some *trigger* signal influences the program's control flow. Whenever the trigger changes, the evaluator plugs its new value into a given *inner* context, the result of which generally evaluates to the root of a fragment of dataflow graph. This node becomes associated with an *outer* context, which is the overall context for the `switch` signal. In essence, the `switch` allows different fragments of graph to be constructed and switched into graph for the whole program.

The δ rule, applied to a primitive procedure application $(p\ v\ldots)$ in evaluation context E, creates a `sig` signal with context E and expression $(p\ v\ldots)$. This new signal depends on all of the signals in $v\ldots$, so the rule adds all of the corresponding pairs to the dependency

relation. Since a proper value for the new signal is unknown (and cannot be computed without a suitable store—see Section 2.7), the rule ensures that the new signal is a member of the resulting I.

The β_v and IF rules create `switch` signals. This is because, in each case, the value of a signal affects the control flow of a computation that defines the overall value of the expression. In the case of the IF rule, the value of the test expression determines which branch is evaluated, while in the β_v rule, the function varies over different primitive operations and λ-abstractions, which are applied to the (possibly time-varying) arguments. Each `switch` signal cooperates with another signal that forwards the value of the current subgraph.

The REC rule also works by creating a `switch` signal, albeit in a slightly different way from the other rules. In this case, the switch is not recorded as a dependent of the trigger, so no actual switching ever occurs. The rule just uses the switching mechanism to defer evaluation of the body. This deferral is necessary in order to preserve the small-step nature of this layer of the semantics. (Evaluating the body would require a full reduction, i.e., a large step.)

The PREV rule produces a `prev` signal, which contains the signal to be delayed and the context awaiting the delayed value. `prev` signals have no dependencies and are not added to the set I, so they always have the undefined value ($\perp$) at the end of the instant in which they are created.

Figure 3.4 defines snapshot evaluation. It is essentially pure call-by-value evaluation of FrTime expressions, except that signals are no longer considered values. Instead, they must be looked up in the given store. It is worth noting that one signal's value may be another signal, so the lookup process must recur until it reaches an ordinary (constant) value.

Figure 3.5 shows the set of rules that define the scheduling of dataflow updates. These rules operate on 4-tuples containing a time t, a store S, a dependency relation D, and a set I of signals with potentially inconsistent values. The main purpose of these rules is to define legal orderings of individual signal updates such that, within a given unit of time, each signal updates at most once, after which its value remains consistent for the remainder of the time unit. We will make these notions more precise in the rest of this chapter.

There are only three update rules, each of which focuses on a particular variety of signal:

U-SIG The rule for updating a `sig` signal is straightforward. Its expression is snapshot-evaluated in the current store, and the store is updated with the signal's new value. If this value has changed, then all of the signal's dependents are added to the set of potentially inconsistent signals.

U-ADV This is the rule for advancing to the next time step, which fires whenever all signals are up-to-date (the set I of out-of-date signals is empty). The rule first evaluates the new expression for this timestep, extending the dependency relation and computing an initial set of out-of-date signals. It then seeds the system with change by updating any `prev` signals whose inputs have changed and adding all of their dependents to the new initial I.

U-SWC This is the rule for updating `switch` signals. It is significantly more complicated than the others. It can only fire when a switch's trigger has changed, and it completely removes any signals previously constructed by this switch before constructing a fresh fragment of dataflow graph. The new fragment is the result of composing the switch's inner and outer evaluation contexts, filling the hole with the trigger's new value, and evaluating until the outer context contains a value.

A note on memory management. In this idealized model, signals only become superfluous when they are switched out, in which case they are explicitly removed from the graph by rule U-SWC. In the actual implementation, the use of additional features (e.g., imperative updates) can cause fragments of the graph to become unreachable without notifying the language. This explains why the implementation requires weak references for the data dependency references, but this notion is unnecessary in the semantics.

Lemma 3.1. *If* $\langle I, D, e\rangle \rightarrow^* \langle I', D', e'\rangle$*, then* $I \subseteq I'$ *and* $D \subseteq D'$*.*

Proof. By induction on the length of the reduction sequence, with case analysis on the reduction rules. The dependency relation and set of internal updates are either extended or passed through unchanged in every transition, a fact easily verified by inspection of the rules. □

Definition 3.1. *We say that an expression* e **refers** *to a signal* σ *if* σ *is a subexpression of* e*. We write* $R[\![e]\!]$ *for the set of all signals to which* e *refers.* R *is defined recursively by case*

analysis on the abstract syntax for expressions:

$$\begin{aligned}
R[\![(\texttt{sig}\ E\ e)]\!] &= \{(\texttt{sig}\ E\ e)\} \cup R[\![e]\!] \\
R[\![(\texttt{switch}\ E_0\ E\ \sigma)]\!] &= R[\![E[\sigma]]\!] \\
R[\![(\texttt{prev}\ E\ \sigma)]\!] &= \{\sigma\} \cup R[\![\sigma]\!] \\
R[\![(\lambda\ (x \ldots)\ e)]\!] &= R[\![e]\!] \\
R[\![(e_1\ \ldots\ e_n)]\!] &= \bigcup_{i=1}^{n} R[\![e_i]\!] \\
R[\![(\texttt{rec}\ v\ e)]\!] &= R[\![e]\!] \\
R[\![(\texttt{if}\ e_c\ e_t\ e_f)]\!] &= R[\![e_c]\!] \cup R[\![e_t]\!] \cup R[\![e_f]\!] \\
R[\![e]\!] &= \emptyset \text{ (for all other cases)}
\end{aligned}$$

We also generalize R to operate on sets of signals in the natural way:

$$R[\![\Sigma]\!] = \bigcup_{\sigma \in \Sigma} R[\![\sigma]\!]$$

Lemma 3.2. *The value of an expression is independent of the values of any signals to which it refers.*

Proof. Evaluation occurs in the absence of a store. □

Lemma 3.3. *The snapshot-evaluation of an expression is completely determined by the values of the signals to which it refers. Formally: for all stores S and S', if $S(\sigma) = S'(\sigma)$ for all $\sigma \in R[\![e]\!]$, then $e \rightarrow_S^n e' \Leftrightarrow e \rightarrow_{S'}^n e'$.*

Proof. By induction on n with case analysis on the snapshot evaluation rules. The only case that refers to the store at all is σ_S. □

Definition 3.2 (natural dependence)**.** *A signal σ_2* **naturally depends** *on another signal σ_1 if and only if for some values of the variables below, where $v_1 \neq v_1'$ and $v_2 \neq v_2'$:*

$$\begin{aligned}
\langle t, S[\sigma_1 \mapsto v_1], D_0, \{\sigma_2\}\rangle &\hookrightarrow \langle t, S[\sigma_1 \mapsto v_1, \sigma_2 \mapsto v_2], D, D(\sigma_2)\rangle \text{ and} \\
\langle t, S[\sigma_1 \mapsto v_1'], D_0, \{\sigma_2\}\rangle &\hookrightarrow \langle t, S[\sigma_1 \mapsto v_1', \sigma_2 \mapsto v_2'], D', D'(\sigma_2)\rangle
\end{aligned}$$

In other words, it is possible to choose different values for σ_1 such that the outcome of updating σ_2 is different for each.

Theorem 3.1. *If σ_2 naturally depends on σ_1, then σ_2 refers to σ_1 ($\sigma_1 \in R[\![\sigma_2]\!]$).*

Proof. By case analysis on the kind of σ_2.

`sig` The value of $(\texttt{sig}\ E\ e)$ is computed by snapshot evaluation of e. By Lemma 3.3, in order for e to evaluate to two different values under two different stores, the stores must map some $\sigma \in R[\![e]\!]$ to two different values. However, in the definition of influence, the two stores differ only on the value of σ_1. Thus, $\sigma_1 \in R[\![e]\!]$.

`prev` The value of $(\texttt{prev}\ E\ \sigma_1)$ is just the value of σ_1 in S. Thus σ_1 (and only σ_1) influences it. Moreover, that $\sigma_1 \in R[\![(\texttt{prev}\ E\ \sigma_1)]\!]$ follows directly from Definition 3.1.

`switch` The value of $\sigma_2 = (\texttt{switch}\ E_0\ E\ \sigma_1)$ is computed by (partially) evaluating $e = E_0[E[S(\sigma_1)]]$, so changing the value of σ_1 may yield different results. However, by Lemma 3.2, this evaluation does not depend on the values of any signals to which e refers. Thus, the largest set of signals on which σ_2 can naturally depend is $\{\sigma_1\}$. By Definition 3.1, $R[\![e]\!] \supseteq \{\sigma_1\}$, from which the implication follows.

□

Related to natural dependence is the notion of *control*, defined as follows:

Definition 3.3. *A signal σ_1* **controls** *another signal σ_2 if and only if for any time t, there exist stores S and S', dependency relations D and D', and signals sets I and I' such that:*

$$\begin{aligned} \langle t, S, D, I\rangle &\hookrightarrow \langle t, S', D', I'\rangle \\ \sigma_1 &\in I \\ \sigma_1 &\notin I' \\ \sigma_2 &\in \mathrm{dom}(D) \cup \mathrm{rng}(D) \\ \sigma_2 &\notin \mathrm{dom}(D') \cup \mathrm{rng}(D') \end{aligned}$$

Intuitively, control refers to the ability for one signal to terminate another. This relationship exists so that, when conditions arise in which updating a signal might cause evaluation to get stuck, the signal can be removed before it has a chance to update. As with natural dependence, when this relationship holds, it is important that the controlling signal updates first, a property that should be enforced by the dependency relation D, as follows:

Definition 3.4 (soundness of dependency relation)**.** *A dependency relation D is* **sound** *with respect to a signal set Σ if and only if, for all pairs of distinct signals $(\sigma_1, \sigma_2) \in R[\![\Sigma]\!]^2$ (where $\sigma_2 \neq (\texttt{prev}\ E\ \sigma_1)$), if σ_2 naturally depends on σ_1, then $(\sigma_1, \sigma_2) \in D$. Also, if σ_1 controls σ_2, then $(\sigma_1, \sigma_2) \in D^+$ (the transitive closure of D).*

Signals in a `prev` relationship are specifically excluded because the dependency relation works contrary to normal in the case of `prev`s. This is to ensure that a `prev` signal always sees the observed signal's value *before* it updates, so that at the end of the time step it will reflect the stable value from the *previous* time step.

The call-by-value rules produce sound dependency relations.

Theorem 3.2 (preservation of soundness by evaluation)**.** *If D is sound with respect to $I \supseteq R[\![e]\!]$ and $\langle I, D, e\rangle \rightarrow^* \langle I', D', v\rangle$, then D' is sound with respect to $R[\![v]\!] \cup I' (\supseteq I)$.*

Proof. By induction on the length of the reduction sequence. The base case, where e is a value, holds trivially. The induction step involves case analysis of the reduction rules (Figure 3.3). Rules $\underline{\delta}$, $\underline{\beta_v}$, and $\underline{\text{IF}}$ are trivial because they do not affect the dependency relation. The other rules behave as follows:

δ The rule refers only to σ and the elements of Σ, so it cannot affect any other signals. The new signal σ will update according to the rule U-SIG; this performs snapshot evaluation of the expression e in store S, which is only affected by S's mappings for signals in Σ. Thus, σ depends on at most the signals in Σ. The addition of $\Sigma \times \{\sigma\}$ to the dependency relation therefore reflects all possible additional dependency relationships, thereby preserving soundness.

β_v The only signals involved in the transition are σ, σ_g, and σ_f. Clearly, σ_f naturally depends on σ_g. By Theorem 3.1 and Lemma 3.2, σ_g may only naturally depend on σ.

σ_f updates according to U-SIG, so it has no other influences. Thus, the dependencies reflected in the extension of D cover all the new influences. Although σ_g will eventually control all of the signals created during its update (this is the subject of Theorem 3.3), at the point of its evaluation it controls nothing.

IF This case is analogous to that of β_v.

REC This case is again analogous to that of the previous two. The difference is that there is no σ, or rather that $(\texttt{prev}\ \sigma_f)$ takes the place of σ. The interposition of the `prev` breaks the strict influence of σ_f on σ_g, so it suffices to add only the single dependency.

PREV This case follows directly from the definition of soundness, which explicitly excludes pairs of signals in a `prev` relationship.

□

We now show that the update rules (Figure 3.5) preserve soundness.

Theorem 3.3. *If D is sound with respect to $I \cup \mathrm{dom}(D) \cup \mathrm{rng}(D)$ and*

$$\langle t, S, D, I\rangle \hookrightarrow^* \langle t', S', D', I'\rangle$$

then D' is sound with respect to $I' \cup \mathrm{dom}(D') \cup \mathrm{rng}(D')$.

Proof. By induction on the length of the reduction sequence. The base case (no reduction steps) is trivial. The induction step involves case analysis on the update rules (Figure 3.5). Rule U-SIG does not affect the dependency relation, and the other two cases proceed as follows:

U-ADV We assume that P only returns expressions that refer to signals in D, so D is sound with respect to $R[\![e]\!]$. By the induction hypothesis, D is sound with respect to its domain and range. By Theorem 3.2, D' is sound with respect to $R[\![v]\!]$ and its own domain and range.

U-SWC Initially, the set of signals controlled by σ_g is Σ. As these are defined by $D^+(\sigma_g) \setminus D^*(\sigma_f)$, they are all certainly in $D^+(\sigma_g)$. Before constructing the replacement graph fragment, the rule removes the signals in Σ from every element of the semantics. By

Theorem 3.2, the evaluation of $E_0[E[S(\sigma)]]$ preserves the soundness of the resulting dependency relation. All of the signals constructed by this evaluation are controlled by σ_g, and the resulting dependency relation reflects this fact.

□

Definition 3.5 (local consistency)**.** *A signal $\sigma = (\texttt{sig}\ E\ e)$ is* **locally consistent** *in store S if and only if $e \to_S^* S(\sigma)$. Similarly, $\sigma = (\texttt{switch}\ E_0\ E\ \sigma_1)$ is locally consistent in S if and only if there exist D and I such that $\langle \emptyset, \emptyset, E_0[E[\sigma_1]] \rangle \to \langle I, D, E_0[v] \rangle$ and $S(\sigma) = v$. The notion of local consistency for signals of the form $\sigma = (\texttt{prev}\ E\ \sigma')$ cannot be defined solely with respect to a store S. We observe, however, that the rule* U-ADV *assigns the desired value at the beginning of each update cycle, and as long as this value does not change within the cycle, it remains consistent.*

Theorem 3.4. *If D is sound with respect to all signals in its own domain and range, and $\langle t, S, D, \emptyset \rangle \to^* \langle t+1, S', D', I \rangle$, then every signal $\sigma \in (\mathit{dom}(D') \cup \mathit{rng}(D')) \setminus I$ is locally consistent in S'. In other words, if an update cycle starts with a sound dependency relation, then within that cycle, every signal is always either locally consistent or enqueued in I for recomputation.*

Proof. By induction on the length of the sequence of update rules, with case analysis on the rules.

U-ADV This rule enqueues for update all of the signals that depend on any signal whose value is changed by application of the rule. By the assumption of D's soundness, these are the only signals that can become locally inconsistent. Thus the rule preserves the conclusion of the theorem.

U-SIG At most one signal (σ) changes, and if it does then all of its dependents are enqueued in I for update. By D's soundness, no other signals can be made locally inconsistent, so again the property is preserved.

U-SWC Everything that σ_g controls before the rule fires is deleted from the system. All of the newly constructed signals are enqueued for update at the end of the rule (unless they depend on another new signal, in which case they are still consistent), as is σ_f, which is the only signal that naturally depends on σ_g.

□

Corollary 3.1 (global consistency at quiescence)**.** *If D is sound with respect to itself, and $\langle t, S, D, \emptyset\rangle \rightarrow^* \langle t+1, S', D', \emptyset\rangle$, then every signal $\sigma \in dom(D') \cup rng(D')$ is locally consistent in S'. In other words, if an update cycle starts with a sound dependency relation, then when there are no signals enqueued for update (the end of that cycle, a quiescent state), all signals are locally consistent, making the system globally consistent.*

Proof. Let $I' = \emptyset$ in Theorem 3.4, and the result follows directly. □

Corollary 3.2 (full consistency)**.** *If D is sound with respect to itself, then for all $t' \geq t$, if $\langle t, S, D, \emptyset\rangle \rightarrow^* \langle t' > t, S', D', \emptyset\rangle$, then every signal $\sigma \in dom(D') \cup rng(D')$ is locally consistent in S'. In other words, if an update cycle starts with a sound dependency relation, then all future quiescent states are globally consistent.*

Proof. By induction on t', using Corollary 3.1 to prove the induction step. The base case is trivial. □

Chapter 4

Optimization by Lowering

Chapter 2 explained how FrTime induces construction of a dataflow graph by redefining operations through an implicit *lifting* transformation. Lifting takes a function that operates on constant values and produces a new function that performs the same operation on time-varying values. Each time the program applies a lifted function to time-varying arguments, it builds a new node and connects it to the nodes representing the arguments. Core Scheme syntactic forms are redefined to extend the graph when used with time-varying values.

Dynamic dataflow graph construction offers several benefits. For example, it permits incremental development of reactive programs in, for instance, a read-eval-print loop (REPL). The implicit lifting also allows programmers to write in exactly the same syntax as a purely functional subset of Scheme. Because lifting is conservative, FrTime programs can reuse Scheme code without any syntactic changes, a process we call *transparent reactivity*.

Unfortunately, the fine granularity of implicit graph construction can result in significant inefficiency. Every application of a lifted function may create a new dataflow node, whose construction and maintenance consume significant amounts of time and space. As a result, large legacy libraries imported into FrTime may be slowed down by two orders of magnitude or more. One experiment, for example, involved attempting to reuse an image library from PLT Slideshow [43], but the result was unusably slow.

This chapter[1] presents an optimization technique designed to eliminate some of the inefficiency associated with FrTime's evaluation model, while still giving programmers the same notion of transparent reactivity. The technique works by collapsing regions of the

[1]This chapter expands on previously published joint work [17] with Kimberley Burchett.

dataflow graph into individual nodes. This moves computation from the dataflow model back to traditional call-by-value, which the runtime system executes much more efficiently. Because this technique undoes the process of lifting, we call it *lowering*. Of course, lowering must not alter the semantics of the original program or sacrifice the advantages of FrTime's evaluation strategy. I present a static analysis that determines when the optimizer can safely lower an expression. The lowering analysis and its implementation yield a significant reduction in time and space usage for real programs.

Lifting and Projection

This chapter is concerned only with behaviors. In FrTime (as in other FRP systems), the programmer processes events through a special set of operators. Since these have no natural analogs in the non-dataflow world, they cannot be used transparently and do not involve implicit lifting. They therefore do not suffer from the associated performance problems, so I do not consider the problem of optimizing them. While the ideas may generalize to events, I have not explored such an extension.

FrTime extends Scheme by replacing its primitives with *lifted* versions. The inverse of lifting is *projection*, which samples a behavior at the current instant, retrieving a raw Scheme value. FrTime uses the operator *value-now* to perform projection, but in this chapter I consistently refer to it as *project*. Formally, these two operations have the following types:

$$\begin{aligned} \textit{lift}_n &: (t_1 \ldots t_n \to u) \to (\mathrm{sig}(t_1) \ldots \mathrm{sig}(t_n) \to \mathrm{sig}(u)) \\ \textit{project} &: \mathrm{sig}(t) \to t \end{aligned}$$

In these definitions, t and u are type variables that can stand for any base (non-signal) type, and $\mathrm{sig}(t)$ is either a base type t, or a signal of base type t. That is, t is a subtype of $\mathrm{sig}(t)$. This means that lifted functions are polymorphic with respect to the time-variance of their arguments, so they can consume an arbitrary combination of constants and signals. Likewise, *project*ing the current value of a constant simply yields that constant.

Note that there is a separate function $\textit{lift}_n$ for each possible n. Scheme supports the

definition of a single procedure *lift* that implements the union of these functions for all n. In the rest of the chapter, I shall refer simply to *lift*, leaving the arity implicit.

Lift and *project* are related through the following identity:

(*project* ((*lift f*) *s* ...)) ≡ (*f* (*project s*) ...)

At any point in time, the current value of the application of a lifted function is equal to the result of applying the original, unlifted function to the projections of the arguments.

Throughout the rest of the chapter, I will refer to constants and signals as inhabiting separate *layers*. Specifically, I will talk about constants as belonging to a *lower* layer, and I will underline the names of lower functions, which can only operate on constants. In contrast, I will say that signals belong to an *upper* layer, and I will put a hat over the names of upper functions, which can operate on signals.

Since lifting generalizes the behavior of raw Scheme functions, it is always safe to substitute a lifted function for its lower counterpart. FrTime does exactly this, so programmers rarely need to worry about accidentally applying lower functions to signals. (The exception is when they import raw Scheme libraries, whose procedures must be explicitly lifted.) In the next section, we shall see that this extreme conservatism takes a toll on performance, which will motivate an exploration of ways to avoid it when possible.

The Need for Optimization

In other FRP systems like Yampa [74], programmers can manually choose the granularity at which to lift operations. It is in their interest, in terms of both human and machine time, to do this as little as possible, which means placing the *lift*s at the highest level possible. Regardless of where the programmer decides to put *lift*s, Haskell's static type system ensures that behaviors are never passed to unlifted primitives, and operations are never lifted twice.

In contrast, FrTime handles reactivity in a dynamic and implicit manner. All primitives are lifted, and every application of a lifted function to time-varying arguments results in a new dataflow graph node. For example, Fig. 4.2 (left) shows the dataflow graph for the relatively simple function in Fig. 4.1. To evaluate this function, six signal objects must be allocated on the heap and connected together: one for each $-$, $+$, *sqr* (square), and *sqrt* (square root) in the expression. Each signal object requires nearly one hundred bytes of

```
(define (distance)
  (λ (x1 y1 x2 y2)
    ((sqrt) (+ ((sqr) (- x1 x2))
               ((sqr) (- y1 y2))))))
```

Figure 4.1: Definition of distance function.

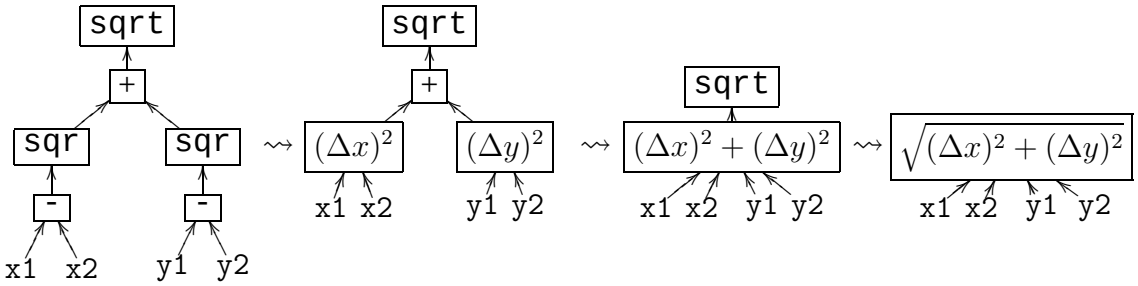


Figure 4.2: Left: Unoptimized dataflow graph for the distance function. Right: optimized equivalent. Various stages of optimization are shown in-between. Inter-procedural optimization can improve the result even further. Each box is a heap-allocated signal object.

memory on the heap.

Whenever one of the inputs to the $\widehat{distance}$ function changes, FrTime has to update the four signals along the path from that input to the root. (If multiple inputs change simultaneously, then it must update everything along the union of their paths.) Each update requires:

1. extracting the node from a priority queue,
2. retrieving the current value of its input signals,
3. invoking a closure to produce an updated value,
4. storing the new value in the signal object, and
5. iterating through a list of dependent signals and enqueueing them for update.

Thus every invocation of the $\widehat{distance}$ function introduces a significant cost in three different areas: the time required to initially *construct* the dataflow graph, the amount of memory

required to *store* the dataflow graph, and the time required to *propagate changes* along the dataflow graph.

Figure 4.3 shows another definition of the $\widehat{distance}$ function, this time with the upper and lower layers made explicit. Note that each of the functions called by $\widehat{distance}$ is actually a lifted version of the lower function by the same name. In other words, they are just lower functions that FrTime has wrapped (like how *frtime:*+ wrapped the primitive + function, above). When lifted functions are composed to form expressions, every intermediate value is lifted to the upper layer, only to be immediately projected back to the lower layer by the next function in line.

The goal is to reduce the use of the expensive dataflow evaluator by eliminating some of the intermediate nodes from the dataflow graph. The key observation is that it is unnecessary to use the dataflow mechanism for every step of a stateless computation. That is, if an expression consists entirely of applications of lifted primitives, then its graph can be replaced with a single node that projects the inputs once, performs the whole computation under call-by-value, and lifts the result. I call this transformation *lowering*, since it removes intermediate lifting steps. Lowering is conceptually similar to Wadler's work on listlessness [99] and deforestation [100], which transform programs to eliminate intermediate list and tree structures.

By moving computation from the dataflow model back into a call-by-value regime, lowering eliminates the overhead of repeatedly transferring values between the upper and lower layers. It also allows the use of the call stack to transfer control and data, which is much more efficient than using the dataflow graph for the same purpose.

In the $\widehat{distance}$ example above, lowering can collapse the entire graph into a single node, yielding an order of magnitude improvement in both speed and memory usage. Section 4.9 shows experimental results on substantial programs.

In general, programs use stateful signal-processing operations, which cannot be combined directly with call-by-value code. The strategy presented here simply stops lowering when it encounters a stateful operation. Since there is a fixed vocabulary of such operations (e.g., *delay-by*, *integral*), it may be possible to develop specific techniques for dealing with them in the optimizer. For example, Nilsson's work [73] on GADT-based optimization for Yampa includes support for combining stateful operations.

```
(define sqrt^ (lift sqrt))
(define sqr^ (lift sqr))
(define +^ (lift +))
(define -^ (lift -))
(define distance^
  (λ (x1 y1 x2 y2)
    (sqrt^ (+^ (sqr^ (-^ x1 x2)))
               (sqr^ (-^ y1 y2))))))
```

Figure 4.3: Definition of the distance function with upper and lower layers made explicit.

4.1 Dipping and Lowering

I now introduce a new syntactic form called **dip**. **Dip** is like *lift* and *project* in that it bridges the two layers, but it does so in a different way.

Dip operates on two syntactic entities: a list of variables whose values are assumed to be signals, and an expression which is assumed to be lower code. **Dip** expands into an expression that, at runtime, projects the variables, evaluates the code, and lifts the resulting value. In this way **dip** allows an entire subexpression of lower code to be embedded inside a section of upper code; whereas *lift* operates on functions, **dip** operates on expressions.

(**dip** (*x*. . .) *e*) $\stackrel{\text{def}}{=}$ ((*lift* (λ (*x*. . .) *e*)) *x*. . .)

Each time a **dip** expression is evaluated, it adds a single node to the dataflow graph that depends on all the variables. Note that the list of variables is a co-environment for the **dip**'s body; it contains all the free variables to which the expression actually refers.

In order to optimize a whole program, the compiler dips as many subexpressions as possible. Dipping a subexpression involves extracting its set of free variables and replacing the code with its lower counterpart. To perform this translation, the optimizer needs to know the *lower counterpart* of each function it calls.

The lower counterpart of each lifted primitive is simply the original (unlifted) primitive. Initially, primitives are the only functions with known lower counterparts, but as the optimizer processes the program, it generally discovers user-defined functions that also have lower counterparts. A compound expression has a lower counterpart if its top-level operation is purely combinatorial and all of its subexpressions have lower counterparts. The

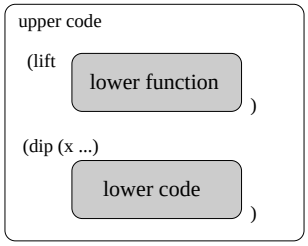


Figure 4.4: Allowed containment relationships for code.

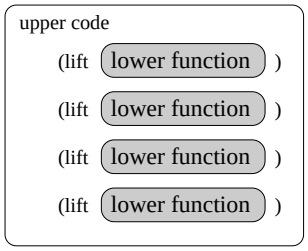


Figure 4.5: Unoptimized FrTime code.

optimizer maintains an explicit mapping between functions and their lower counterparts; entries in this mapping are denoted by $\langle \widehat{func}, \underline{func} \rangle$.

Not all functions have lower counterparts. For example, the function $\widehat{delay\text{-}by}$, which time-shifts a signal's value, needs to remember the history of its changing input signal. It cannot do anything useful if it is called afresh with each new value the signal takes. In general, any function that depends on history has no meaning in the lower layer of constant values. For expressions that involve such functions, it is critical that the optimizer not erroneously dip them, as the resulting program would behave incorrectly.

In the following sections, I will distinguish between lowering, which *replaces* an upper expression with a corresponding lower expression, and dipping, which takes values from the upper layer to the lower layer and back, with some computation in between. The following summarizes the three varieties of code that result from these transformations:

Lower code consists entirely of pure Scheme expressions. All the functions it calls are lower versions, so it cannot operate on time-varying values.

Upper code is standard FrTime. Each primitive operation constructs a dataflow node that

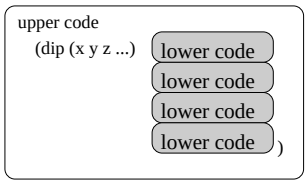


Figure 4.6: Optimized FrTime code.

---

recomputes its value whenever its input values change.

Dipped code is observationally equivalent to upper code, but operates very differently. Instead of producing *many* dataflow nodes, each of which performs *one* primitive operation, dipped code produces *one* dataflow node that evaluates a complex expression involving *many* primitive operations.

Figure 4.4 shows the allowed containment relationships for these different varieties of code. At the top-level, the program consists of upper code (assume that it actually involves signals). This code can refer to lifted functions and dipped expressions, but not to bare lower code. The lifts and dips wrap lower code with logic that protects it from time-varying values. In contrast, lower code never contains upper code of any form (including lifted functions or dipped expressions), since it has no need to process signals. In essence, the optimizer exploits the fact that upper and lower counterparts are *twin* versions of the same code; the lower version can be viewed as a special entry point that allows skipping over the extra checking and wrapping needed by the more general upper version.

Figures 4.5 and 4.6 illustrate the goal of optimization. Figure 4.5 represents unoptimized FrTime code. In it, the upper program refers to a large number of small fragments of lifted[2] code. In comparison, Figure 4.6 represents code of the sort that we would like the optimizer to produce. The fragments of dipped code have been combined into a small number of larger blocks, reducing the overhead associated with constructing and maintaining a signal for each atomic operation.

[2]Because the application of a lifted primitive yields the same result as dipping, everything could just be expressed in terms of **dip**. However, lifting is an established term within the FRP community, so I use it for clarity.

4.2 The Lowering Algorithm

The optimization algorithm works by rewriting expressions to semantically equivalent expressions that contain fewer *dip*s, each of which contains a larger body of code. This rewriting is an application of equational reasoning and is justified by the definition of *dip* and the *lift/project* identity.

The algorithm works in a bottom-up fashion. It begins with the leaves of the abstract syntax (variables and constants) and proceeds to their parent expressions, their grandparent expressions, and so on.

Formally, the algorithm is guided by a set of rewrite rules. I write $\Gamma \vdash e \rightsquigarrow (\mathbf{dip}\ (\vec{x})\ e')$ to indicate that e' is the dipped version of e, where the environment Γ associates function names with the names of their lower counterparts, and $\vec{x}$ is the set of all signals on which the value of e may depend. For example, dipping of literals c simply involves wrapping them in a **dip** expression. Since the value of a literal is always a constant, its dipped equivalent does not depend on anything:

$$\vdash c \rightsquigarrow (\mathbf{dip}\ ()\ c)$$

The optimizer treats identifiers similarly, but since they may refer to signals, it includes them in the list of dependencies:

$$\vdash id \rightsquigarrow (\mathbf{dip}\ (id)\ id)$$

For example, in the case of the $\widehat{distance}$ function, the optimizer arrives at the identifiers *x1* and *x2* and applies this rule, resulting in the following expression:

```
(define distance^
  (λ (x1 y1 x2 y2)
    (sqrt^ (+^ (sqr^ (−^ (dip (x1) x1)
                         (dip (x2) x2))))
               (sqr^ (−^ y1 y2))))))
```

The optimizer proceeds by combining dipped subexpressions into larger code fragments. In the case of function applications, it computes the union of the arguments' dependencies and replaces the lifted function with its lower counterpart:

$$\frac{\langle \widehat{f}, \underline{f} \rangle \in \Gamma \qquad \Gamma \vdash e_i \rightsquigarrow (\mathbf{dip}\ (\vec{x_i})\ e_i')}{\Gamma \vdash (\widehat{f}\ e_i \ldots) \rightsquigarrow (\mathbf{dip}\ (\vec{x_i} \ldots)\ (\underline{f}\ e_i' \ldots))}$$

Continuing the $\widehat{distance}$ example, one application of this rule produces the following result:

(**define** $\widehat{distance}$
  (λ (*x1 y1 x2 y2*)
    ($\widehat{sqrt}$ ($\widehat{+}$ ($\widehat{sqr}$ (**dip** (*x1 x2*) ($\underline{-}$ *x1 x2*)))
                ($\widehat{sqr}$ ($\widehat{-}$ *y1 y2*))))))

Applying this rule once more produces:

(**define** $\widehat{distance}$
  (λ (*x1 y1 x2 y2*)
    ($\widehat{sqrt}$ ($\widehat{+}$ (**dip** (*x1 x2*) ($\underline{sqr}$ ($\underline{-}$ *x1 x2*)))
                ($\widehat{sqr}$ ($\widehat{-}$ *y1 y2*))))))

Next, the optimizer dips the second argument to $\widehat{+}$, which is transformed identically to the left branch:

(**define** $\widehat{distance}$
  (λ (*x1 y1 x2 y2*)
    ($\widehat{sqrt}$ ($\widehat{+}$ (**dip** (*x1 x2*) ($\underline{sqr}$ ($\underline{-}$ *x1 x2*)))
                (**dip** (*y1 y2*) ($\underline{sqr}$ ($\underline{-}$ *y1 y2*)))))))

Since dipping does not change the observable semantics of an expression, it is safe to stop optimizing at any time. In this case the bottom-up traversal will continue until it reaches the λ, at which point it must stop because of subtleties involved with lambda abstractions (explained below).

The final optimized result contains only a single **dip** expression, which means that when evaluated, it creates only a single dataflow graph node instead of the six nodes required for the original function. Figure 4.2 shows the final dataflow graph, along with some intermediate graphs. The final code is as follows:

(**define** $\widehat{distance}$
  (λ (*x1 y1 x2 y2*)

(**dip** (*x1 x2 y1 y2*)
  (*sqrt* (+ (*sqr* (− *x1 x2*)))
    (*sqr* (− *y1 y2*))))))

Though the above example does not contain any **let** expressions, dipping them is also straightforward. The newly-introduced binding (id) is excluded from the body's dependency list ($\vec{x_e}$) because it is guaranteed to be subsumed by the bound value's dependency list ($\vec{x_v}$).

$$\frac{\Gamma \vdash v \rightsquigarrow (\mathbf{dip}\ (\vec{x_v})\ v') \qquad \Gamma \vdash e \rightsquigarrow (\mathbf{dip}\ (\vec{x_e})\ e')}{\Gamma \vdash (\mathbf{let}\ ((id\ v))\ e) \rightsquigarrow (\mathbf{dip}\ (\vec{x_v} \cup (\vec{x_e} \setminus id))\ (\mathbf{let}\ ((id\ v'))\ e'))}$$

The following subsections describe the details of optimizing the language's remaining syntactic forms.

4.3 Lambda Abstractions

Dipping a λ expression is somewhat subtle. For example, suppose the optimizer encounters the following expression:

(λ (*x*) ($\widehat{+}$ *x* 3))

So far, it has dipped expressions by wrapping their lowered counterparts in the **dip** form. If it does that here, the result is:

(**dip** () (λ (*x*) (+ *x* 3)))

This is clearly unsafe, because if the resulting closure were applied to a signal, the lowered $+$ operator would cause a type error. To prevent such errors, it can only dip the *body*, instead of the whole λ expression. Then the result is:

(λ (*x*) (**dip** (*x*) (+ *x* 3)))

In general, the rule is as follows:

$$\frac{\Gamma \vdash e \rightsquigarrow (\mathbf{dip}\ (\vec{x})\ e')}{\Gamma \vdash (\lambda\ (\vec{v})\ e) \rightsquigarrow (\lambda\ (\vec{v})\ (\mathbf{dip}\ (\vec{x})\ e'))}$$

If the optimizer never lowered function bodies, then it would be incapable of discovering lower counterparts of user-defined functions. This would make the analysis purely intraprocedural, greatly reducing the number of opportunities for optimization and therefore the utility of the technique.

The ability to achieve interprocedural optimization takes advantage of the fact that a **dip** expression's body is the original expression's lower counterpart. Therefore, if the optimizer successfully dips a function, then it knows the function's lower counterpart. I write $\Gamma \vdash e \overset{\lambda}{\rightsquigarrow} (\mathbf{dip}\ (\vec{x})\ e')$ to indicate not only that e' is the dipped version of e, but that in addition e is a λ expression whose body can be lowered:

$$\frac{\Gamma \vdash e \rightsquigarrow (\mathbf{dip}\ (\vec{x})\ e')}{\Gamma \vdash (\lambda\ (\vec{v})\ e) \overset{\lambda}{\rightsquigarrow} (\mathbf{dip}\ (\vec{x} \setminus \vec{v})\ (\lambda\ (\vec{v})\ e'))}$$

References to variables bound by the lambda's argument list are removed from the list of dependencies, since in a lower context they cannot be signals.

When the $\overset{\lambda}{\rightsquigarrow}$ transformation applies, the optimizer adds a top-level definition for the lower counterpart of $\widehat{f}$, called $\underline{f}$, and remembers the association $\langle \widehat{f}, \underline{f} \rangle$:

$$\frac{\begin{array}{c}\Gamma \cup \langle \widehat{f}, \underline{f} \rangle \vdash e \rightsquigarrow (\mathbf{dip}\ (\vec{x})\ e') \\ \Gamma \cup \langle \widehat{f}, \underline{f} \rangle \vdash e \overset{\lambda}{\rightsquigarrow} (\mathbf{dip}\ ()\ e'')\end{array}}{\begin{array}{l}\Gamma \vdash (\mathbf{define}\ \widehat{f}\ e) \rightsquigarrow (\mathbf{begin}\ (\mathbf{define}\ \widehat{f}\ (\mathbf{dip}\ (\vec{x})\ e')) \\ \qquad\qquad\qquad\qquad (\mathbf{define}\ \underline{f}\ e''))\end{array}}$$

The above rule expands the scope of the optimization to include interprocedural optimization. On the other hand, if a definition does *not* have a lower counterpart then only the dipped version is defined:

$$\frac{\begin{array}{c}\Gamma \vdash e \rightsquigarrow (\mathbf{dip}\ (\vec{x})\ e') \\ \Gamma \cup \langle \widehat{f}, \underline{f} \rangle \vdash e \overset{\lambda}{\not\rightsquigarrow} (\mathbf{dip}\ ()\ e'')\end{array}}{\Gamma \vdash (\mathbf{define}\ \widehat{f}\ e) \rightsquigarrow (\mathbf{define}\ \widehat{f}\ (\mathbf{dip}\ (\vec{x})\ e'))}$$

If a program contains a sequence of definitions, each definition is dipped separately:

$$\frac{\Gamma \vdash e_i \rightsquigarrow (\mathbf{dip}\ (\vec{x_i})\ e_i')}{\Gamma \vdash (\mathbf{begin}\ e_i \ldots) \rightsquigarrow (\mathbf{dip}\ (\vec{x_i} \ldots)\ (\mathbf{begin}\ e_i' \ldots))}$$

For concision and clarity, the above judgements do not describe the full mechanism for interprocedural optimization. Adding this would be straightforward but would increase the size of the judgements considerably.

4.4 Conditionals

The criterion for dipping **if** expressions is the same as for all other expression types: all of their subexpressions must have lower counterparts. Moreover, the consequence is also the same, namely that the resulting node depends on the union of the subexpressions' dependencies.

$$\frac{\begin{array}{c}\Gamma \vdash c \rightsquigarrow (\mathbf{dip}\ (\vec{x_c})\ c') \\ \Gamma \vdash t \rightsquigarrow (\mathbf{dip}\ (\vec{x_t})\ t') \\ \Gamma \vdash f \rightsquigarrow (\mathbf{dip}\ (\vec{x_f})\ f')\end{array}}{\Gamma \vdash (\mathbf{if}\ c\ t\ f) \rightsquigarrow (\mathbf{dip}\ (\vec{x_c} \cup \vec{x_t} \cup \vec{x_f})\ (\mathbf{if}\ c'\ t'\ f'))}$$

Conditional evaluation in FrTime is relatively expensive, so dipping conditionals can improve performance significantly. Moreover, dipping of conditionals is necessary in order to define lower counterparts for recursive functions, which makes it possible to collapse a potentially long chain of graph fragments into a single node.

4.5 Higher Order Functions

Higher order function applications, which evaluate a closure passed as an argument, cannot be dipped using only the strategy defined in this paper. For example, consider the type of $\widehat{map}$:

$\widehat{map} : \mathrm{sig}(\mathrm{sig}(t) \rightarrow \mathrm{sig}(u)) \times \mathrm{sig}(\mathrm{list}(t)) \rightarrow \mathrm{sig}(\mathrm{list}(u))$

$\widehat{map}$'s first argument is a signal, which can be called to produce another signal. That is, the choice of which function to apply can change over time, as can the result of applying the function. Dipping only removes the first kind of time dependency, not the second. If $\langle \widehat{map}, \underline{map} \rangle$ were a valid upper/lower pair, then the type of $\underline{map}$ would have to be:

$\underline{map} : (\mathrm{sig}(t) \rightarrow \mathrm{sig}(u)) \times \mathrm{list}(t) \rightarrow \mathrm{list}(u)$

Clearly this could cause a problem at runtime, since the actual $\underline{map}$ doesn't support functions that may produce signals. In order to avoid this problem, the optimizer never associates a lower counterpart with a higher order function. For the built-in higher order functions such as $\widehat{map}$ and $\widehat{apply}$, it just omits them from its initial mapping. However, this still leaves the question of higher order functions written by users.

The only way a user-defined function can be assigned a lower counterpart is if its body can be completely lowered; no higher order function can satisfy this requirement, since at some point it must call the procedural argument. Lexical scoping guarantees that the function's arguments will have fresh names, so the optimizer cannot possibly know of a lower counterpart for the argument closure. Since the function makes a call with no known lower counterpart, the body is not lowerable.

A static dataflow analysis could address this weakness by identifying closures that have known lower counterparts. However, the need for such an extension has not yet arisen, and, in any case, it is always safe to assume the absence of lower counterparts. It just means that certain expressions cannot be optimized.

4.6 Inter-Module Optimization

DrScheme's module framework makes it easy to write the optimizer in such a way that it processes each module individually. An unfortunate consequence of this approach is that the associations between user-defined functions and their lower counterparts is not shared between modules. Unless the optimizer can recover these associations, it will lose many opportunities for optimization. It will be unable to optimize any expression containing a call to a function whose entry was forgotten, even if that call is in a deeply nested subexpression. For commonly used functions such as those that manipulate low-level data types, this effect can cascade throughout much of the program.

In order to recover the lost associations, the FrTime optimizer uses a consistent naming convention to identify the lower counterpart of an upper function (Scheme doesn't understand the underline and overline annotations, so some amount of name mangling is necessary in any case). Because of this naming convention, the optimizer can recover the forgotten associations simply by inspecting a module's list of exported identifiers. Thus it can perform inter-module optimization.

$$\vdash c \rightsquigarrow (\mathbf{dip}\ ()\ c) \qquad \text{(CONST)}$$

$$\vdash id \rightsquigarrow (\mathbf{dip}\ (id)\ id) \qquad \text{(VAR)}$$

$$\frac{\langle \widehat{f}, \underline{f} \rangle \in \Gamma \qquad \Gamma \vdash e_i \rightsquigarrow (\mathbf{dip}\ (\vec{x_i})\ e_i')}{\Gamma \vdash (\widehat{f}\ e_i \dots) \rightsquigarrow (\mathbf{dip}\ (\vec{x_i} \dots)\ (\underline{f}\ e_i' \dots))} \qquad \text{(APP)}$$

$$\frac{\Gamma \vdash v \rightsquigarrow (\mathbf{dip}\ (\vec{x_v})\ v') \qquad \Gamma \vdash e \rightsquigarrow (\mathbf{dip}\ (\vec{x_e})\ e')}{\Gamma \vdash (\mathbf{let}\ ((id\ v))\ e) \rightsquigarrow (\mathbf{dip}\ (\vec{x_v} \cup (\vec{x_e} \setminus id))\ (\mathbf{let}\ ((id\ v'))\ e'))} \qquad \text{(LET)}$$

$$\frac{\Gamma \vdash e \rightsquigarrow (\mathbf{dip}\ (\vec{x})\ e')}{\Gamma \vdash (\lambda\ (\vec{v})\ e) \rightsquigarrow (\lambda\ (\vec{v})\ (\mathbf{dip}\ (\vec{x})\ e'))} \qquad \text{(LAMBDA)}$$

$$\frac{\Gamma \vdash e \rightsquigarrow (\mathbf{dip}\ (\vec{x})\ e')}{\Gamma \vdash (\lambda\ (\vec{v})\ e) \overset{\lambda}{\rightsquigarrow} (\mathbf{dip}\ (\vec{x} \setminus \vec{v})\ (\lambda\ (\vec{v})\ e'))} \qquad \text{(LAMBDA-BODY)}$$

$$\frac{\begin{array}{c}\Gamma \cup \langle \widehat{f}, \underline{f} \rangle \vdash e \rightsquigarrow (\mathbf{dip}\ (\vec{x})\ e') \\ \Gamma \cup \langle \widehat{f}, \underline{f} \rangle \vdash e \overset{\lambda}{\rightsquigarrow} (\mathbf{dip}\ ()\ e'')\end{array}}{\begin{array}{l}\Gamma \vdash (\mathbf{define}\ \widehat{f}\ e) \rightsquigarrow (\mathbf{begin}\ (\mathbf{define}\ \widehat{f}\ (\mathbf{dip}\ (\vec{x})\ e')) \\ \qquad\qquad\qquad\qquad (\mathbf{define}\ \underline{f}\ e''))\end{array}} \qquad \text{(DEFINE-LOWER)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e \rightsquigarrow (\mathbf{dip}\ (\vec{x})\ e') \\ \Gamma \cup \langle \widehat{f}, \underline{f} \rangle \vdash e \overset{\lambda}{\not\rightsquigarrow} (\mathbf{dip}\ ()\ e'')\end{array}}{\Gamma \vdash (\mathbf{define}\ \widehat{f}\ e) \rightsquigarrow (\mathbf{define}\ \widehat{f}\ (\mathbf{dip}\ (\vec{x})\ e'))} \qquad \text{(DEFINE-NO-LOWER)}$$

$$\frac{\Gamma \vdash e_i \rightsquigarrow (\mathbf{dip}\ (\vec{x_i})\ e_i')}{\Gamma \vdash (\mathbf{begin}\ e_i \dots) \rightsquigarrow (\mathbf{dip}\ (\vec{x_i} \dots)\ (\mathbf{begin}\ e_i' \dots))} \qquad \text{(BEGIN)}$$

$$\frac{\begin{array}{c}\Gamma \vdash c \rightsquigarrow (\mathbf{dip}\ (\vec{x_c})\ c') \\ \Gamma \vdash t \rightsquigarrow (\mathbf{dip}\ (\vec{x_t})\ t') \\ \Gamma \vdash f \rightsquigarrow (\mathbf{dip}\ (\vec{x_f})\ f')\end{array}}{\Gamma \vdash (\mathbf{if}\ c\ t\ f) \rightsquigarrow (\mathbf{dip}\ (\vec{x_c} \cup \vec{x_t} \cup \vec{x_f})\ (\mathbf{if}\ c'\ t'\ f'))} \qquad \text{(IF)}$$

Figure 4.7: Complete description of the lowering transformation

The flexibility of this mechanism provides an additional usability benefit: the programmer can define hand-coded lower counterparts for functions the optimizer is not sophisticated enough to lower automatically.

4.7 Macros

Since macros must be fully expanded before runtime, they can have no time-varying semantics. They are therefore easy to support; the optimizer simply expands all macros before attempting to apply the lowering optimization.

4.8 Pathological Cases

In most cases, lowering reduces execution time and memory requirements, but there are instances in which it can have the opposite effect. The reason is that lowering combines several small fragments of code, each depending on a few signals, into a large block that depends on many signals. For example, consider the following simple expression:

(*expensive-operation* (*quotient* milliseconds 10000))

Though milliseconds changes frequently, the *quotient* changes relatively rarely. If run under the standard FrTime evaluator, the *quotient* node will stop propagation when its result doesn't change, thus short-circuiting the recomputation of the *expensive-operation* most of the time. However, in the "optimized" version, this whole computation (and perhaps more) is combined into a single node, which must recompute *in its entirety* each time milliseconds changes.

As discussed in Section 4.3, interprocedural optimization requires that the optimizer produce two versions of each lowerable procedure definition: one that is merely dipped, and one that is actually lowered. Lowering thus has the potential to double the size of a program's code. I have so far chosen not to worry about this because the optimized code is static and, in most cases, accounts for a relatively small fraction of a program's overall dynamic memory usage. However, for large programs, this may become a concern. In particular, recent versions of DrScheme employ a just-in-time compiler, which generates native code for each executed procedure body. Since native code occupies considerably

	Count	Needles	S'sheet	TexPict
Size (exprs)	7	62	2,663	13,022
Start_{orig} (sec)	9.5	89.0	9.2	35.2
Start_{opt} (sec)	$<$0.1	35.3	11.8	28.9
Mem_{orig} (MB)	204.7	581.4	34.8	170.7
Mem_{opt} (MB)	0.2	240.5	50.9	119.4
Shrinkage (ratio)	*971*	*2.4*	*0.7*	*1.4*
Run_{orig} (sec)	4.8	5.6	19.3	273.4
Run_{opt} (sec)	$<$0.1	2.0	20.5	3.5
Speedup (ratio)	*16,000*	*2.8*	*0.94*	*78.1*

Table 4.1: Experimental benchmark results for lowering optimization

more space than expression data structures, lowering has the potential to increase a program's memory usage significantly.

4.9 Evaluation

This section presents the impact of optimization on several FrTime benchmarks. It also contains a discussion of the optimizer's impact on the usability of FrTime.

4.9.1 Performance

Four different benchmarks evaluate the effect of the optimization on the resource requirements of various programs. Other than the Count microbenchmark, none of these applications was written with lowering in mind, so the findings should be broadly representative.

Table 4.1 summarizes the performance results.[3] *Size* denotes the program's size measured by the number of expressions ("parentheses"). The $_{orig}$ and $_{opt}$ subscripts denote the original and optimized versions. *Start* is the initial graph construction time, while *Run* is reaction time, i.e., the time for a change to propagate through the graph. Times $<$0.1 are too small to be measurable. *Mem* denotes memory footprint beyond that of DrScheme (which is 72MB). *Speedup* denotes the ratio between the unoptimized run-time and the optimized run-time, and *Shrinkage* denotes the analogous ratio for memory usage.

[3]Measured on a Dell Latitude D610 with 2Ghz Pentium M processor and 1GB RAM, running Windows XP Pro SP2 with SpeedStep disabled. The numbers are the mean over three runs from within DrScheme version 360, restarting DrScheme each time.

The Count microbenchmark consists of a function that takes a number, recursively decrements it until reaching zero, and then increments back up to the original number, i.e., an extremely inefficient implementation of the identity function for natural numbers. The purpose of the benchmark is to quantify the potential impact of lowering for code that involves a large number of very simple operations (in this case only addition, subtraction, comparison, and conditionals). The results are dramatic: for inputs around 600, the unoptimized version takes several seconds to start and then takes nearly five seconds to recompute whenever the input value changes. In contrast, even for inputs in the hundreds of thousands, the optimized version starts in a fraction of a second and updates even more quickly.

The Needles program (due to Robb Cutler) displays a 60×60 grid of unit-length vectors. Each vector rotates to point at the mouse cursor, and its color depends on its distance from the mouse cursor. The main effect of optimization is to collapse the portions of the dataflow graph that calculate each vector's color and angle (these consist entirely of numerical operations, which are an easy case for lowering). Since these constitute a significant portion of the code, optimization has a significant effect. The optimized version runs nearly three times faster and uses about half as much memory.

The Spreadsheet program implements a standard 2D spreadsheet. Formulas are evaluated by calling Scheme's built-in *eval* procedure in the FrTime namespace. The startup phase has several calculations for drawing the grid, setting the size of scroll-bars, etc., which are optimized. Somewhat surprisingly, the "optimized" spreadsheet requires more time and space than the original version. This is indeed counterintuitive, if not disappointing. One reasonable explanation is that, because the spreadsheet was designed from the beginning to run in FrTime, its dataflow graphs already work efficiently under the default FrTime evaluator. Also, as explained in Section 4.8, there are known scenarios in which lowering can make programs less efficient. In most cases this inefficiency is more than outweighed by the reduction in dataflow evaluation, but apparently not in this case.

TexPict is the image-compositing subsystem of Slideshow, whose unacceptable execution performance (under FrTime) motivated this work. As can be seen from the experimental results, lowering yields a speedup of almost two orders of magnitude. The result is still significantly slower than a raw Scheme analog, but fast enough to make it usable for many applications. This offers strong evidence in support of the hypothesis that large

dataflow graphs arising from implicit, fine-grained lifting can lead to a significant slowdown. Moreover, it demonstrates that lowering makes transparent reactivity feasible for real legacy programs.

The TexPict benchmark is also interesting because it frequently uses higher order functions. The fact that a first order analysis yields a dramatic improvement even in this case indicates that the current approach is sufficient for a broad range of applications, even those that use higher order functions extensively.

4.9.2 Usability

In DrScheme, any collection of syntax and value definitions can be bundled into a module that comprises a "language". For example, the FrTime language is a set of lifted primitives, along with special definitions for certain syntactic forms (e.g., conditionals). The optimized language is defined similarly, except that it defines a syntax transformer for a whole FrTime program. FrTime programmers enable optimization simply by changing the module language declaration from `frtime` to `frtime-opt`. The optimizer will be shipped with the next standard DrScheme release, so no additional installation or configuration is necessary.

Even though the FrTime optimizer works by performing a source-to-source transformation, it does not adversely affect the programmer's ability to understand the original program. In particular, the optimizer preserves source location information within the transformed code, so the runtime system reports errors in terms of the original source code [36, 44]. Furthermore, if optimization fails for some section of code (perhaps due to the use an unsupported feature, or even due to a bug in the optimizer itself), the optimizer will silently fall back to using the original code, and continue the optimization at the next top-level definition.

Users can discover whether or not a particular piece of code was optimized by examining the fully expanded result; unoptimized code is preceded by a literal string explaining what went wrong during optimization. On the other hand, code that *is* optimized will stand out because the names of upper functions will have been replaced with the mangled names of their lower counterparts.

The overhead of the optimization pass is quadratic in the nesting depth of function definitions and linear in the size of the code base. This makes it practical to apply optimization

to large systems, such as the TexPict benchmark presented above. Furthermore, an optimized module can be precompiled so that the overhead of static analysis does not need to be repeated when the module is used later.

4.10 Future Directions

Achieving acceptable runtime performance in FrTime required the development of a novel optimization technique called *lowering*. This technique works by processing the source program in a bottom-up fashion, recursively combining calls of lifted primitives into larger *dipped* expressions. This has the effect of shifting significant computation from the dataflow mechanism back to the underlying (in this case) call-by-value evaluator. Though the analysis is still unable to handle certain language features, such as higher order functions, experimental results indicate that the technique can achieve a significant reduction in a program's time and space needs, making transparent reactivity a viable approach for realistic systems.

The notion of lowering applies outside of FRP, for example to any monad [101] where the *lift* operator distributes over function composition. Specifically, wherever $(lift\ g) \circ (lift\ f) \equiv lift\ (g \circ f)$, and *lift* is expensive, it is beneficial to rewrite to reduce the number of *lift*s. Lowering may therefore be useful in general for languages that use monads extensively. For example, the Glasgow Haskell Compiler [83] optimizes code by rewriting expressions according to such identities.

One limitation of this technique is that if a subexpression has no lower counterpart, then the enclosing expression cannot be lowered either. This limitation could be avoided by hoisting the problematic subexpression out and storing its result in a temporary variable; however, in a call-by-value language like Scheme, such a transform must take care not to affect evaluation order. Translating to continuation-passing style would make evaluation order easier to deal with, but would make it more difficult to identify dippable subexpressions.

For languages that support runtime code generation, it would be possible to explicitly build the dataflow graph first, and then collapse nodes into call-by-value subexpressions. This approach would trivially support inter-procedural optimization, and would be able to collapse arbitrary nodes in the dataflow graph, whether or not they contained unlowerable subexpressions in the original program text. This approach would depend on the ability of

the runtime environment to compile dynamically-generated subexpressions into efficient code.

I anticipate the application of the lowering optimization to Flapjax. Since the language provides transparent reactivity and employs a FrTime-like evaluation model, I would expect to see similar results.

Chapter 5

Implicitly Reactive Data Structures

The preceding chapters have discussed the core features and evaluation model of FrTime. However, they have avoided one important issue that any practical language must address, which is how to support structured data. This is a significant omission, since writing non-trivial programs requires the ability to organize state into such structures. In this chapter, I discuss the design problems that arise when attempting to add structured data types to a language like FrTime, along with the solutions I have developed.

Since FrTime is an embedding in Scheme, the goal is to support the same kinds of structures that Scheme provides. These include lists, trees, vectors, and user-defined record types called *struct*s. For example, Scheme provides the following primitives for manipulating lists:

cons : any $\times$ **list** $\rightarrow$ **list** constructs a non-empty list with a given first element and list of remaining elements.

cons? : any $\rightarrow$ **boolean** determines whether a given value is a non-empty list.

first : list $\rightarrow$ **any** returns the first element of a non-empty list.

rest : list $\rightarrow$ **list** returns the rest of a non-empty list.

It also provides a mechanism for defining custom datatypes, such as a *posn* for storing positions in 2-space:

(**define-struct** *posn* (*x y*))

This expression results in the following collection of definitions:

make-posn : number × **number** → **posn** constructs a *posn* from two numbers.

posn? : any → **boolean** determines whether a given value is a *posn*.

posn-x : posn → **number** extracts the *x*-component of a *posn*.

posn-y : posn → **number** extracts the *y*-component of a *posn*.

In general, the interface to a Scheme data structure consists of three kinds of procedures:

constructors such as *cons* and *make-posn*.

discriminators like *cons?* and *posn?*.

accessors like *first*, *rest*, *posn-x*, and *posn-y*.

FrTime needs a way of letting all of these procedures work meaningfully in the presence of behaviors. The rest of this chapter explores the issues involved in achieving this goal.

The first observation is that FrTime needs to lift any discriminators that it imports. If it did not, then expressions like

```
(cons? (build-list (modulo seconds 3) add1))
```

or

```
(posn? (if (even? seconds)
           (make-posn 3 4)
           0))
```

would not work as intended. In the first example, the (*build-list* . . .) expression evaluates to a behavior whose value is sometimes a *cons*. Thus, we should expect the whole expression to evaluate to a behavior whose value is sometimes **true**. However, at the level of Scheme, behaviors are custom structures that are distinct from any other types (including *cons*es), regardless of their current values. Thus, the value of the whole expression would be the constant **false**, which is not very satisfying. The same problem would also arise in the second expression.

Like discriminators, accessors cannot be imported directly into FrTime, or expressions like

```
(posn-x (if (even? seconds)
            (make-posn 3 4)
            (make-posn 5 12)))
```

would fail with type errors.[1]

The situation for constructors is more complicated than for discriminators and accessors. The reason for the complexity is that constructors, at least in Scheme, are oblivious to their arguments: they just blindly store them in structures. Thus, unlike other primitives, constructors need not be lifted in order to prevent errors. For example, importing *make-posn* directly and evaluating

```
(make-posn (modulo seconds 100) 50)
```

yields a *posn* whose *x*-component is a time-varying integer.

On the other hand, there is no obvious harm in lifting the constructor. In that case, the above expression evaluates to a behavior whose value at each point in time is a *posn* with an *x*-component equal to the current value of seconds.

I call these two approaches respectively the use of *raw* and *lifted* constructors. The difference between them is quite subtle, and the rest of this chapter will explore the tradeoffs between the two approaches. With raw constructors, the above evaluates to a structure containing a behavior, while with lifted constructors it reduces to a behavior containing a structure. Though these are not the same types, they support the same sets of operations, and so they are essentially interchangeable. For example, the result of the following program is the same whether *make-posn* is lifted or not.[2]

```
;; compute the Euclidean distance between two posns
(define (distance p1 p2)
  (sqrt (+ (sqr (- (posn-x p1) (posn-x p2)))
           (sqr (- (posn-y p1) (posn-y p2))))))

(distance (make-posn x1 y1) (make-posn x2 y2))
```

[1] It may appear that the accessors could simply be lifted like other primitives, but the situation is a bit more subtle, as I will discuss later.

[2] However, the performance characteristics of the two approaches may differ, as Section 5.3.1 discusses.

5.1 An Application of Structured Data: Animation

Where the difference between raw and lifted constructors really becomes apparent is when data structures are used as the medium for communicating with the world. To see this, it is useful to consider a concrete application, for example a library for defining functional animations. Since animations are time-varying images, a natural approach is to start with a library for static images, then generalize it to support image behaviors.

A functional image library allows a program to manipulate images as objects, without calling imperative drawing procedures. The application program builds a data structure representing an image, and the library implicitly performs the side-effecting operations required to display it.

In Scheme, the following imperative drawing procedures are available:

(open-viewport title width height) opens a new window with the given width, height, and title, returning a *viewport* object.

(clear-viewport viewport) clears the contents of a given viewport.

(make-rgb r g b) constructs an *rgb* color structure with the given red, green, and blue components (as real numbers in $[0, 1]$).

(draw-solid-ellipse viewport top-left width height color) draws a solid ellipse of the given width, height, and color, such that the upper left corner of its bounding box is at *top-left*.

(draw-solid-rectangle viewport top-left width height color) is analogous to *draw-solid-ellipse*.

To make a functional image library, I define a set of data structures that capture the types and parameters of the shapes that can be drawn:

(**define-struct** *ellipse* (*center width height color*))
(**define-struct** *rectangle* (*top-left width height color*))

Thus, an image consists of an ellipse, a rectangle, or a collection (i.e., a list) of other images.

The library defines a mechanism for drawing the images represented by these data structures:

```
(define (show! image title width height)
  (render! image (open-viewport title width height)))

(define (render! scene viewport)
  (clear-viewport viewport) ; start with a clean canvas
  (draw! viewport scene))

(define (draw! scene viewport)
  (match scene
    [($ ellipse ($ posn x y) width height color)
     (draw-solid-ellipse viewport
      (make-posn (- x (/ width 2)) (- y (/ height 2))) width height color)]
    [($ rectangle top-left width height color)
     (draw-solid-rectangle viewport top-left width height color)]
    [scenes (for-each (λ (scene) (draw! scene viewport)) scenes)]))
```

So, for example,

```
(show! (list (make-ellipse (make-posn 100 200) 40 30 (make-rgb 0 0 1))
             (make-rectangle (make-posn 200 100) 20 15 (make-rgb 1 0 0))))
```

opens a new window and draws a blue ellipse and a red rectangle in it.

The next step is to allow a FrTime program to use a behavior anywhere the corresponding Scheme program would use a constant, and to have the resulting program exhibit the expected reactivity. For example, one might want to make the ellipse move back and forth on the screen, or the color of the rectangle pulsate. According to FrTime's principle of transparent reactivity, one ought to be able to produce such an animation by writing something like the following:

```
(define cycle (modulo (quotient milliseconds 10) 100))
(define oscillate (/ (+ 1.0 (sin (/ milliseconds 300))) 2.0))
(show! (list (make-ellipse (make-posn cycle 200) 40 30 (make-rgb 0 0 1))
             (make-rectangle (make-posn 200 100) 20 15 (make-rgb oscillate 0 0))))
```

5.2 Reactivity with Raw Constructors

If the constructors are imported raw from Scheme, then running this program will result in an error, not an animation. The reason is as follows. The argument to *show!* is an ordinary Scheme list containing ordinary *ellipse* and *rectangle* structures, each of which contains an ordinary *posn* and *rgb* structure. In the case of the ellipse, the *posn* contains a behavior, and in that of the rectangle, the *rgb* contains a behavior. When *show!* calls *render!* to display the image, it determines that its argument is a list and maps itself over the elements. In the recursive call, the first argument is the *ellipse*, so *render!* calls *draw-solid-ellipse* on the *posn* with a time-varying *x*-component. The underlying Scheme implementation requires that the fields of the *posn* be ordinary numbers, not behaviors, so it raises a type error.

The preceding example illustrates one key point: the language must not allow behaviors to flow to raw Scheme code. Any values that do flow to Scheme code must therefore be projected. One way to do so is to apply *value-now* to all such values. For example, we could rewrite the drawing procedure as follows:

```
(define (draw! scene viewport)
  (match scene
    [($ ellipse ($ posn x y) width height ($ rgb r g b))
     (let ([x (value-now x)] [y (value-now y)]
           [width (value-now width)] [height (value-now height)]
           [r (value-now r)] [g (value-now g)] [b (value-now b)]])
       (draw-solid-ellipse
         viewport
         (make-posn (- x (/ width 2)) (- y (/ height 2)))
         width height (make-rgb r g b)))]
    ;; — other cases adapted similarly —
```

Now an ellipse with a time-varying center is no problem, since the *draw!* procedure projects everything to a constant before calling the low-level drawing procedure. Unfortunately, this also means that there is no animation, since projecting the current value of the behaviors eliminates their reactivity.

Interestingly, if we were willing to employ a polling-based strategy here, then this approach would be viable. In that case, we could just re-execute the call to *render!* at a regular

interval, which would repeatedly draw the image in its current state. Unfortunately, polling would mean, on the one hand, recomputing values even when they haven't changed, and on the other, possibly failing to render states if values change faster than the polling rate. Therefore, we reject the use of polling, so we need to find a way of making push-driven recomputation work.

5.2.1 Choosing the Granularity of Lifting

Lifting is the obvious technique for adapting Scheme code to react to changes in behaviors. For example, by lifting the drawing procedures (e.g., *draw-solid-ellipse*), we protect them from behaviors without neutralizing their reactivity. In fact, by lifting them, we also ensure that shapes will be redrawn whenever any of their properties changed.

Unfortunately, lifting the individual drawing procedures does not achieve the desired effect. Each time a shape's property changes, that shape is drawn again. However, drawing is a side-effecting operation, so maintaining a graphical rendering of a shape is not as simple as redrawing it every time it changes. In general, the effects of drawing the shape's previous state must also be undone. (Otherwise, the result is a trail of old shapes.)

The preceding paragraph emphasizes an important point about the use of lifting to communicate the state of a dataflow program to an external system. Since such communication necessarily involves side effects, the granularity of the lifted procedures must be carefully chosen so that the side effects from their repeated evaluation always leave the world in a consistent and desirable state.

In the case of animation, lifting must encompass at least the *render!* procedure: *render!* begins by clearing the canvas, which is an easy way of undoing all side effects from the previous rendering. The other option would be to keep track somehow of the previous state of the shape and try to undo only the effects from drawing it. However, this would be significantly more complicated than just clearing the canvas, especially in the face of overlapping shapes, and would offer little advantage.

If we lift at a higher level than *render!*) (say *show!*), then we stil get a consistent rendering of the shape each time it changes, but we also get something else that we probably don't want—a new window for each image. Hence the result is not so much an animation as a filmstrip, which is a less intuitive user interface, not to mention a significant leak of

```
(define (deep-project struct/bhvr)
  (cond
    [(behavior? struct/bhvr)
     (deep-project (behavior-value struct/bhvr))]
    [(cons? struct/bhvr)
     (cons (deep-project (first struct/bhvr))
           (deep-project (rest struct/bhvr)))]
    ...
    [else struct/bhvr]))
```

Figure 5.1: A deep projection procedure

system resources.

Thus, the only reasonable level at which to lift drawing code is the *render!* procedure. However, if we just lift *render!*, it won't actually solve the problem described above. Since the constructors aren't lifted, the argument to *render!* may contain behaviors without being a behavior itself.

5.2.2 Deep Lifting

What we need for *render!* is a mechanism akin to lifting, except that it

1. reacts to changes nested arbitrarily deep within its argument, and

2. projects the current values of any behaviors within the argument.

The second of these requirements, computing what I call *deep projections*, is straightforward. Figure 5.1 shows the essence of the implementation. It mainly involves walking and copying each node of the structure, projecting the current value of each behavior encountered, and recurring on its contents. In FrTime, the actual implementation also uses a table to prevent infinite loops when projecting cyclic data, along with special logic to avoid returning copies of substructures that don't contain any behaviors.

Reacting to changes that occur within a structure is a bit more complicated. In particular, since the program's dataflow graph can change dynamically, there is not necessarily a fixed set of signals that need to be watched. Figure 5.2 shows the heart of the implementation of this operation, which I call *deep-lift*ing. Before calling *proc*, the update procedure

```
(define (all-nested-behaviors struct/bhvr known-bhvrs)
  (cond
    [(memq struct/bhvr known-bhvrs)
     known-bhvrs]
    [(behavior? struct/bhvr)
     (all-nested-behaviors (behavior-value struct/bhvr)
                           (cons struct/bhvr known-bhvrs))]
    [(cons? struct/bhvr)
     (let ([bhvrs (all-nested-behaviors (first struct/bhvr)
                                        known-bhvrs)])
       (all-nested-behaviors (rest struct/bhvr) bhvrs))]
    ...
    [else known-bhvrs]))

(define (deep-lift proc)
  (λ (struct/bhvr)
    (rec result
      (new-behavior
        (λ ()
          (let ([bhvrs (all-nested-behaviors struct/bhvr empty)])
            ;; ensure the dataflow graph reflects this behavior's
            ;; dependence on each of the nested behaviors

              ;; if depth has changed, reschedule this behavior
              ;; for a later update and escape
              ...)
            (proc (deep-project struct/bhvr))) ...))))
```

Figure 5.2: Deep lifting

traverses the argument structure completely to find all of the behaviors it currently contains, then ensures that the dataflow graph reflects dependencies on all of these behaviors. The implementation of *all-nested-behaviors* is also shown in Figure 5.2.

There is a somewhat subtle point about this step: if *result* now depends on something new, then it may not be safe to continue processing right away. In particular, in some cases the new behavior may not have been updated yet, so calling *proc* would result in an inconsistent result (i.e., a glitch). In such cases it is necessary to abort the current update operation and reschedule according to the new topology of the dataflow graph. This rescheduling could occur several times within an update cycle (if updating the new behavior

results in further changes to the graph's topology). When *result*'s height in the graph stops increasing, it is safe to proceed by calling *proc* on a deep projection of the argument, *struct/bhvr*.

The *deep-lift* operator provides precisely the varied notion of lifting that is needed to turn *render!* into an animator.

5.3 Reactivity with Lifted Constructors

The preceding section describes a strategy for using data structures built with raw constructors. Although this strategy works, it involves quite a bit of complexity, including primarily the definition of a deep-lifting operator, several aspects of which are fairly subtle.

In fact, we can avoid all of this complexity simply by lifting constructors. Returning to the animation example, if we lift all of the constructors (i.e., *make-posn*, *make-rgb*, *make-ellipse*, and *cons*), then we can just lift *render!*, and it will produce animations. In particular, because *make-posn* is lifted and *cycle* is a behavior,

(*make-posn cycle* 200)

produces a behavior. Thus, since *make-ellipse* is lifted,

(*make-ellipse* (*make-posn cycle* 200) ...)

produces another behavior. Likewise, because *make-rgb* is lifted and *oscillate* is a behavior,

(*make-rgb oscillate* 0 0)

creates a behavior, and so on for *make-rectangle* and *list*. Whenever either *cycle* or *oscillate* changes, the chain of behaviors causes it to propagate all the way to the top-level image list. Finally, *show!* creates a window and delegates to *render!* to perform the drawing. Because *render!* is lifted, any change in the image list causes it to re-execute, clearing the screen and rendering a fresh snapshot of the image's current state. Thus, by lifting constructors, we obtain a simple mechanism for transforming a static renderer into an animator.

5.3.1 Consequences of Lifted Constructors

Figure 5.3 illustrates how lifted constructors cause behaviors to spread to any structures that (transitively) contain them. This propagation of behaviors is what makes lifting so

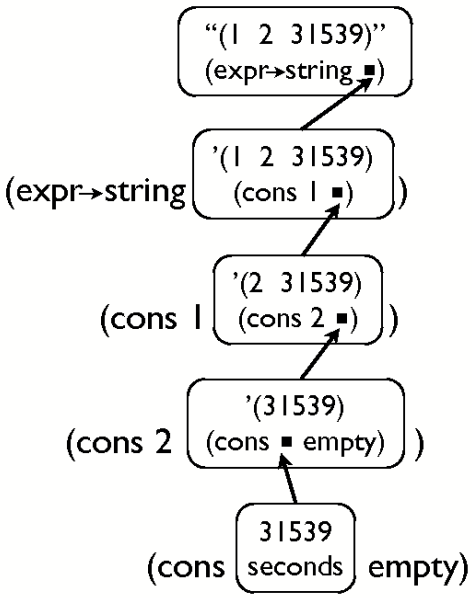


Figure 5.3: Use of lifted constructors

convenient. However, it also incurs a cost.

Creating a behavior is a relatively heavy-weight operation, involving allocation of a structure, a closure, lists of references to other behaviors, and various other sorts of bookkeeping information. When a lifted constructor is used to make a structure, a change in any of the structure's fields results in the re-evaluation of the constructor and hence the allocation of fresh storage for its new state. In most cases, the program has no need for the old structure, so it just becomes garbage. If the program uses many time-varying structures, or even just a few that change rapidly, then the effect on garbage-collection pressure, and on performance in general, can be significant.

For comparison, the use of a raw constructor incurs no overhead whatsoever: each call to a constructor allocates a single data object, exactly as it would in Scheme. Raw constructors do not propagate behaviors, so they eliminate all of the costs associated with creating and updating behaviors.

While constructors are responsible for significant overhead, accessors are also problematic. Because a behavior only exposes its current value and the fact that it is time-varying,[3]

[3]It would be possible to modify the language to support such introspective capabilities, possibly enabling better dynamic optimization. However, exploring such an avenue is beyond the scope of this paper and, in

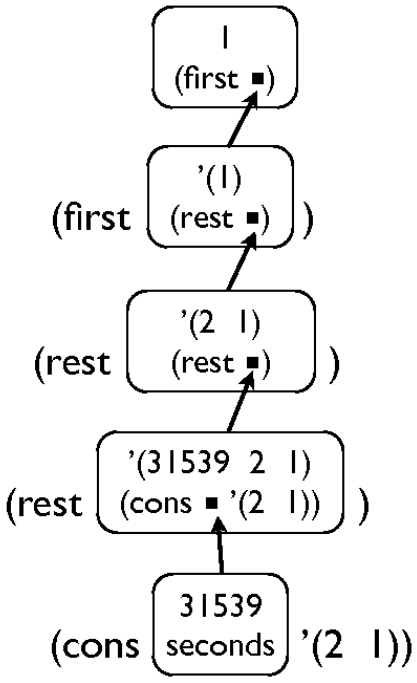


Figure 5.4: Creation of additional behaviors by lifted accessors

an accessor (e.g., *first*) cannot distinguish between the following two behaviors (whose current values are always the same):

(*cons* (*modulo* seconds 2) empty)

(**if** (*even?* seconds)
  (*cons* 0 empty)
  (*cons* 1 empty))

Thus, even in the first case, *first* cannot magically return the original (*modulo* seconds 2) behavior; it must create a new behavior whose value is computed by selecting the first element of its argument's current value.

Because lifted accessors construct new behaviors, a lifted accessor cannot traverse a structure in the traditional sense. Instead of deconstructing the time-varying structure itself, it builds additional behaviors that traverse projections of the time-varying structure. In Figure 5.4, we see a particularly pathological manifestation of this phenomenon: the tail

any case, not necessary to achieve the linguistic goals we seek here.

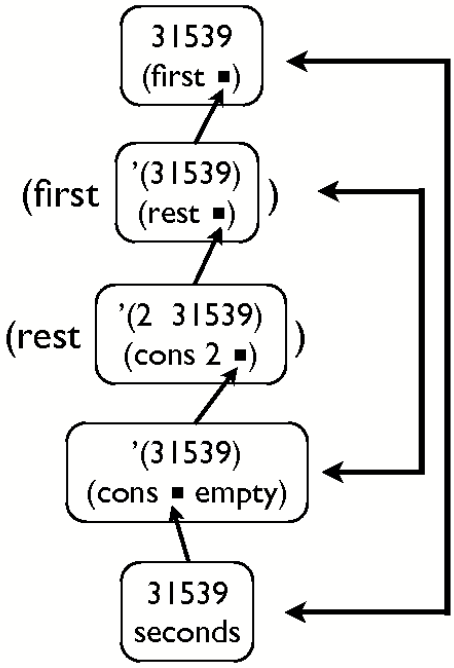


Figure 5.5: Loss of intensional equality from lifted constructors

of the list was originally a constant, but because there's a behavior at the front, code that traverses it perceives everything as a behavior, even though most of the values are actually constants.

In general, the combination of lifted constructors and accessors results in the conflation of reactivity from different sources. As mentioned previously, every time the value of *any* field in a structure changes, a new structure is created. Then, *every* accessor applied to the structure perceives the change in the structure and must recompute its result.

While all of these extra behaviors and updates result in some inefficiency, there are also semantic consequences of using lifted constructors. Most notably, with raw constructors, (*posn-x* (*make-posn x y*)) actually evaluates to *x*. However, with lifted constructors, it results in a new behavior whose *current value* is always the same as *x*'s. In other words, raw constructors preserve intensional equality across storage in data structures, but lifted constructors respect only extensional equality. Figure 5.5 illustrates the problem: the behaviors connected by thick arrows are extensionally equal, but without lifted constructors they would be the same physical value (and therefore intensionally equal).

Substituting extensional for intensional equality essentially means replacing a constant with a behavior whose value never changes. We know that the constant cannot change, but all we know about the behavior is that whenever we've happened to look at it, it's had the same value. We can't be sure that it won't change in the next time step, or that it if we were able to sample it at a finer interval, we wouldn't notice it changing back and forth to some other value very rapidly. In many applications, such as animations, extensional equality is sufficient; the program can sample things faster (and for longer) than a human can perceive (and endure), so from the user's perspective there is no difference. However, it is important in general because of the temporal nature of FrTime values. Time is conceptually continuous and infinite, while the program's execution is a discrete, finite approximation.

This distinction becomes important for programs that manipulate collections of behaviors. For instance, recall the simple program described above for rendering collections of time-varying points. Suppose that we wished to extend this to allow a user to manipulate the points, perhaps by clicking and dragging to create, move, and delete points. A natural representation for the state of such a program is a time-varying list of time-varying *posn*s.

Attempting to model such a program with lifted constructors is awkward. Because lifted constructors bring all reactivity to the top level, the state can only be a time-varying list of *posn*s. That is, the reactivity of each point is conflated with that of each other, and with that of the list itself. Hence the program can only operate meaningfully on the state as a whole. The state must then be a single, monolithic entity, which is processed by a single set of top-level event-handling procedures. Ultimately, such a design is non-modular, unnatural to implement, and difficult to maintain.

While raw constructors lead to various complications, they can all be addressed through deep lifting. In contrast, the drawbacks of lifted constructors cannot be remedied through any means. I therefore conclude that, despite the attractive simplicity that lifted constructors bring in the common case, raw constructors are ultimately the only viable approach.

5.4 Improvements to Deep Lifting

Deep lifting eliminates the large dataflow graphs that lifting would otherwise create to mirror large data structures in the program. Thus it drastically reduces the overhead of

```
(define (draw-point/proj p)
  (let ([p (value-now posn)])
    (draw-point drawing-window
                (make-posn (posn-x p) (posn-y p)))))

(define (for-each/proj proc lst)
  (let ([lst (value-now lst)])
    (when (cons? lst)
      (proc (first lst))
      (for-each/proj proc (rest lst)))))

(define (render/proj list-of-points)
  (clear drawing-window)
  (for-each/proj (λ (p)
                   (draw-point/proj drawing-window p))
                 list-of-points))
```

Figure 5.6: Interleaving projection with traversal

propagating changes through the dataflow graph. It could be characterized as a dynamic, constructor-specific analog of the *lowering* optimization described in Chapter 4.

However, there are still problems with deep lifting. In particular, it still incurs significant overhead in the form of allocation. Every time a change occurs anywhere within a large structure, it constructs a fresh deep snapshot of the structure. In some cases this can actually be worse than using lifted constructors, which only force reconstruction along paths from changed nodes to the root.

One way to avoid so much deep projection is to make external interface procedures project behaviors as they encounter them. For example, instead of using the version of *render* from above, which assumes that its arguments are constants, we could write a version that expects to find behaviors and projects the value of each one it encounters.

Such an implementation is shown in Figure 5.6. The basic idea is that, before applying any primitive operation to a value, we wrap it in a call to *value-now*. If the value is a constant, then *value-now* has no effect, and if it's a behavior, it projects its current value. Either way, there is no allocation, and it is safe to proceed with the resulting value.

The one exception is when calling a library procedure that operates on structured data.

For example, if *draw-point* consumes a *posn* containing a behavior, a type error will result. In Figure 5.6, *render/proj* constructs a new *posn* with projections of the original's fields. This prevents the type error but brings us back to the original problem of doing extra allocation each time something changes.

There are a couple of options available for eliminating this last bit of allocation:

1. Find a lower-level drawing procedure that operates on plain integers, and wrap that instead.

2. Assuming *draw-point* is never called recursively, or from more than thread at a time, allocate a single *posn* structure and set its fields imperatively before calling *draw-point*.

Once we've written our interface procedures in this style, we can write a refined version of deep lifting, called something like *deep-lift/no-project*. This would be identical to *deep-lift*, except that instead of calling *deep-project* on the argument before passing it to *proc*, it would call *proc* directly on the argument.

This approach, which we call *incremental projection*, can eliminate most if not all of the allocation due to deep-projection. The effect is analogous to what listlessness [99] and deforestation [100] achieve.

As Figure 5.6 shows, making each operation call *value-now* on its argument would be a tedious task to perform by hand. Fortunately, this task can be performed mechanically. (Since *value-now* is idempotent, one easy but safe approach is simply to wrap it around every argument to every function call.) Scheme provides a macro system for defining syntactic abstractions, which we can use to perform such a transformation automatically.

One thing that is still unsatisfying about this solution is that it involves traversing the structure twice: once to detect changes in its topology, and another time to process its current value. An worthwhile question is whether we can merge these two traversals into one. This certainly sounds plausible, but unfortunately it is more subtle than it seems. As described above in the context of defining *deep-lift*, if the structure's shape changes, then it may contain new of behaviors, and it cannot be processed safely until all of them have been updated. This means that the signal that processes the structure must update its dependencies to reflect the new set of behaviors. If it now depends on something at its level or higher, then we must delay evaluation or risk the occurrence of a glitch.

However, if we interleave the two steps of processing, then we discover the potential inconsistency after we have already processed some of the structure. If we abort and retry later, then we need to be sure that we won't put the system in an inconsistent state by performing destructive side beffects more than once. Fortunately, experience indicates that FrTime's external interface procedures (at least for libraries like graphics and user interface widgets) are idempotent, so it is safe to abort part way through a traversal and later restart from the beginning. The only potential problem with this approach is performance: in theory, an adversarially chosen program could cause FrTime to perform the traversal a number of times linear in the size of the structure. Since each traversal also takes linear time, the time taken to achieve a full, successful traversal could become quadratic. It's not clear whether this possibility is a serious cause for concern, but it does suggest that the language should take care only to abort the traversal when there is a real risk of inconsistency.

Another way to work around this problem is to treat all external interface behaviors specially and force their heights to be greater than those of any ordinary signals. For example, we could simply assign them a height of *infinity*. The disadvantage is that, if we needed to impose ordering constraints among these signals, such a strategy would make that more difficult. We have not yet determined which is the better tradeoff.

5.4.1 Defining the *Apply* Operator

The preceding discussion assumed that Scheme primitives operate only on flat values, not structured data, so all primitives can simply be lifted. Unfortunately, there is an important exception to this rule: Scheme's *apply* operator allows (some suffix of) the arguments to a function to be packaged up in a list. In order to support full transparency, FrTime needs to provide a version of *apply* that works with any list-like value that FrTime might pass to it (e.g., lists of behaviors, behaviors whose values are lists, and even nestings of these). Doing so sensibly turns out to be somewhat subtle, although much of the machinery we've developed so far is reusable.

First, consider what happens if we try to use a raw *apply*. Since *cons* is not lifted, this works correctly for simple lists of behaviors. For example, there is no problem with something like:

(*apply* + 2 3 (*list* 5 (*modulo* seconds 7) 11))

Here, *apply* consumes the list of arguments and calls (lifted) + on its elements. However, raw *apply* fails on behaviors whose values are lists. For example, in

```
(apply + (if (even? seconds) (list 1 2) (list 3 4 5)))
```

apply is expecting a list but receives a behavior, which results in a type error. If we instead use a lifted *apply*, then it will work correctly in the above cases: in the first case, its arguments appear constant, so it behaves like the raw *apply*, and in the second case, it calls the + operator afresh each time the argument list changes.

Unfortunately, simple lifting of *apply* doesn't work in general. For example, consider the following expression:

```
(apply + (if (even? seconds)
             (list (modulo milliseconds 1000) 2)
             (list 3 4 5)))
```

In this case, *apply*'s argument is a behavior whose current value is sometimes a list containing another behavior. This means that the result of applying + to the current list of arguments may result in a behavior. Thus, in order for its result to reflect the correct value, *apply* needs to use a *switch* to react to changes in its argument list and track the changes in the current result.

However, even with switching there are some problems. For example, in

```
(apply + (cons 1 (if (even? seconds)
                     (list 1 2 3)
                     (list 4 5))))
```

the argument list starts with an ordinary *cons* cell, but its tail is a behavior, which causes a type error in *apply* as it traverses the list looking for arguments. In order to work properly, such nested reactivity needs to be raised to the top-level before *apply* switches over it.

This strategy is almost correct, but it can have undesirable consequences when the applied function is not primitive. For example, consider:

```
(apply (λ (x y z)
          (+ (delay-by x y) z))
       (list seconds 200 1))
```

If we raise the reactivity of *seconds* to the top-level, then each time *seconds* changes, the argument list will change, causing *apply* to call the function again, which wipes out the state of the *delay-by* operator. To avoid unnecessary switching like this, we only want to raise reactivity that affects the *structure* of the argument list, not the elements themselves. To achieve this, we use a modified implementation of the *deep lifting* operation described earlier in this section, which ignores behaviors in the *first* field of each *cons* cell, only paying attention to behaviors in the *rest* field.

Of course, if the structure of the list changes, it still causes the function to be re-applied afresh, resulting in a loss of state for operators like *delay-by* and *integral*. Unfortunately, there is no way to solve this problem without making modifications to the language.

5.5 The Efficiency of Traversal

Using raw constructors eliminates the ttop-down spread of behaviors which, by reducing the total number of behaviors, also reduces amount of top-down propagation. However, it's still possible to have top-down propagation, since structure behaviors can exist even in the absence of lifted constructors. For example, in

```
(define y
  (if (even? (quotient seconds 10))
      (cons seconds empty)
      (cons 3 (cons 4 empty))))
```

y becomes bound to a behavior whose value is a list of (sometimes time-varying) numbers. This means there can still be some spread of behaviors due to the use of accessors.

Note that we can now have ordinary *cons* cells that contain behaviors. This means that if we just lift accessors, then (*first y*) evaluates to a behavior whose value is sometimes another behavior (whose value is equal to that of *seconds*). Having behaviors nested inside each other violates a key invariant of FrTime, so it is preferable to make the accessors more sophisticated instead. Fortunately, this is easy to do. We just put a *switch* inside them so they can merge the two sources of reactivity into one. Then (*first y*) returns a behavior whose value is always just a number.

However, a problem becomes apparent when we evaluate a program like the following:

```
(map sqr (if (even? seconds)
             (build-list 1000 identity)
             (build-list 2000 add1))))
```

where *map* might be loosely defined as

```
(define (map f lst)
  (if (cons? lst)
      (cons (f (first lst)) (map f (rest lst)))
      empty))
```

The first subexpression to be evaluated is (*cons? lst*). Since *lst* is a behavior, the result is a boolean behavior (which will always be **true**), so the **if** expression evaluates to a switch and selects the first branch for evaluation. The branch evaluates (*first lst*), which results in another switch, then passes the result to *sqr*, which creates yet another behavior. The next step is to call *map* recursively on (*rest lst*), which is yet another switch. The recursive call proceeds in a manner analogous to the top-level call. In particular, it constructs another switch for the conditional expression, and the branch constructs a *cons* whose *rest* is computed by another recursive call on a behavior, and so on. By the time it has finished, the program has constructed 3000 switch nodes and 1000 other behaviors for the list elements. After construction, each of these 4000 nodes must be updated.

When *seconds* changes from even to odd, the dataflow graph is extended to accommodate the longer input list. This requires repeating all of the work described above, including the creation of another 3000 switches and 1000 other behaviors, as well as updating each of the 8000 nodes in the overall graph. When *seconds* changes from odd to even, the 4000 nodes comprising the latter half of the list need to be destroyed (and disconnected from the graph), and changes propagated through the other 4000 nodes. Both operations are very expensive and, over time, result in the creation of large amounts of garbage.

In fact, there are only two lists, each with a static structure (and static content), and a single conditional that switches between them. Thus nearly all of the work described above is unnecessary. If we instead write:

```
(if (even? seconds)
    (map sqr (build-list 1000 identity))
    (map sqr (build-list 2000 add1)))
```

the result is the same as above, but it is computed in a very different way. In particular, neither branch refers to any behaviors, so each *map* call executes like ordinary Scheme code (very quickly, without creating any behaviors at all).

The inefficiency comes about because *map* uses a conditional to test the type of *lst* and select the corresponding branch. This is problematic because conditionals are oblivious to their test expressions, so the knowledge the test extracts about the input's current value is not available in the branch. The branch must therefore apply accessors to *lst*, building additional switches that repeat work already done in the test. Essentially, the problem is one of computational leakage [60].

There's no way we can change the definitions of *cons?*, *first*, and *rest* to eliminate the inefficiency; the problem arises from the fact that we use them at all. The solution is to combine all of these operations, including conditionals, into a single abstraction that encapsulates the desired pattern without leaking computation. That abstraction is pattern-matching. For example, in the case of lists, we would provide a **match-list** construct, with which we could rewrite *map* as follows:

```
(define (map f lst)
  (match-list lst
    (cons (a d) (cons (f a) (map f d)))
    (empty () empty)))
```

This is vaguely similar to the above definition, but it means something subtly different. Now, each time *lst* changes, the switch discards its entire old branch and evaluates a new one according to whether *lst* is empty or a cons. Since it restarts the branch each time *lst* changes, the branch can be evaluated with *a* and *d* bound to the *current* contents of *lst*, which in this case are constants. From this point on, evaluation proceeds as in ordinary Scheme (like the transformed example above). In general, using this approach instead of conditionals and accessors results in the creation of one switch per switch in the input, as opposed to several switches for each node encountered below the first switch in the input.

It is worth noting that one can contrive examples in which the semantics of these two approaches differ. These essentially involve programs like the following:

```
(let ([lst (if (even? seconds)
               (cons 1 empty)
```

```
                (cons −1 (cons 0 empty)))])
    (+ (integral (first lst)) (length (rest lst))))
```

which we might try to rewrite using **match-list** as follows:

```
(match-list (if (even? seconds)
                (cons 1 empty)
                (cons −1 (cons 0 empty)))
  (cons (a d) (+ (integral a) (length d)))
  (empty () 0))
```

The key thing is that the list is never empty, and its first element is always *seconds*. Thus, in the first version, the call to *integral* is evaluated only once and accumulates state until it is no longer needed. In the second version, each change to *seconds* causes the *cons* case to be re-evaluated afresh, restarting the *integral* and losing whatever state it had accumulated previously.

While this difference in semantics of **match-list** is perhaps a bit disturbing, we argue that the drastic performance increase that it enables more than outweighs any minor loss in expressiveness. For purposes of transparency, FrTime needs to continue supporting conditionals and accessors anyway, so it just provides the **match-list** construct as well, offering more attractive performance properties for those willing to rewrite list-processing procedures in terms of it.

5.6 Performance Evaluation

This section presents experimental measurements of the relative performance of the strategies discussed above for dealing with structured data.

Table 5.1 presents the results of a microbenchmark involving the construction and update of various constant-length lists, each of whose last element is time-varying. The lists have length 50, 100, and 200. Each measured value is the amount of time in milliseconds required to perform 1000 FrTime update cycles (an average over ten runs, with the standard deviation following in parentheses).[4] The *baseline* involves simply incrementing the value

[4]The experiments were run in DrScheme 3.99.0.13 on a MacBook Pro with 2.0GHz Intel Core Duo processor and 1GB of RAM.

of a behavior in each cycle (no list).

The *raw* times are for lists built with raw *cons*; since raw constructors do not propagate changes, these are expected to be very close to the baseline (independent of list length). The experimental values agree with this prediction.

The *lifted* times are for lists built with lifted *cons*, which does propagate changes. Therefore, these times should increase with list length, which is again what the experiments show. The *Lifted - raw* numbers show the overhead of lifting, which grows nearly linearly with list length.

The *deep-lifted* times are for lists built with raw constructors whose reactivity is then deep-lifted to the top-level in a single step. These exhibit the same allocation overhead as lifting, but not the propagation overhead, since all of the propagation occurs in one update. The *Deep-lifted - raw* numbers show the overhead of raising, which also grows with list length, albeit considerably more slowly than the lifting overhead (as expected). In general, raising appears to outperform lifting by 35 to 45%.

Of course, the point of deep-lifting is to provide a more efficient implementation of lifting that works at the interface between FrTime and the world. To get a realistic measurement of the improvement, we also need to account for the cost of traversing and processing the data structures. The *Lift/traverse* (*Deep-lift/traverse*) measurements show the times taken to traverse and compute the sums of the elements in the lifted (deep-lifted) lists. Raising is still faster (as expected), but only by 15 to 25%.

Finally, the *Incremental* measurements show the times taken for incremental projection and processing of raw lists, which avoids the need for raising. This eliminates the allocation overhead from the raising strategy (which already eliminates the allocation overhead from lifting). This results in an additional 40 to 50% savings over raising, making the overall time more than twice as fast as the lifting-based approach.

Table 5.2 shows the performance of two animation applications. *Needles* animates a field of vectors that aim at a given time-varying point. In this experiment, the grid contains 676 vectors, arranged in a 26×26 grid. The measurements include the times (again in milliseconds) to start up and to update, using both raw and lifted constructors. The startup time for raw constructors is about twice as long as for lifted constructors, but updates occur about 15% faster, and after only about 20 updates the raw version overtakes the lifted one. One reason that raw constructors only improve performance by about 15% is that the cost

Type \ Size	50		100		200	
	avg	(dev)	avg	(dev)	avg	(dev)
Raw	52.1	(3.5)	54.2	(3.2)	55.2	(5.0)
Lifted	191.1	(17.8)	317.9	(22.5)	610.1	(20.3)
(Lifted - raw)	139.0	-	263.7	-	554.9	-
Raised	110.5	(30.7)	173.1	(15.6)	396.9	(25.8)
(Raised - raw)	58.4	-	118.9	-	341.7	-
Raised vs. lifted	**42.2%**	-	**45.5%**	-	**34.9%**	-
Lift/traverse	403.3	(31.2)	703.8	(14.6)	1432.5	(28.7)
Raise/traverse	307.9	(19.1)	593.4	(30.3)	1205.7	(16.0)
Raised vs. lifted	**23.7%**	-	**15.7%**	-	**15.8%**	-
Incremental	199.1	(13.2)	338.5	(15.1)	615.7	(16.9)
Incr. vs. raise	**37.9%**	-	**43.0%**	-	**48.9%**	-
Incr. vs. lift	**52.6%**	-	**51.9%**	-	**57.0%**	-

Table 5.1: Micro-benchmark results for lifting, raising, and incremental projection

Type \ Size	Needles		Oscillation	
	avg	(dev)	avg	(dev)
Startup (lifted)	242.2	6.9	3652.8	107.0
Startup (raw)	558.4	15.1	1442.5	52.8
Raw vs. lifted	**-130.6%**	-	**60.5%**	-
Update (lifted)	127.4	4.2	59.0	2.4
Update (raw)	110.6	1.4	30.7	1.7
Raw vs. lifted	**13.2%**	-	**48.0%**	-

Table 5.2: Performance of animation programs using data structures and graphics

of updating everything else and rendering the vectors dominates that of updating a few hundred *cons* behaviors.

The *oscillation* example makes more extensive use of lists. It models a time-varying graph as a 200-element list of behaviors, of which any subset may be active at a given time. In these tests, the number of active points is relatively small, so in the lifted case the list-related updates constitute a more significant fraction of the time. The raw version starts up about 2.5 times more quickly and updates about twice as fast.

Chapter 6

Integration with Object-Oriented Toolkits

Chapter 2 described, among other things, the use of a *lifting* transformation to adapt existing purely functional operations so they could work with time-varying values. However, to support realistic applications, the language also needs access to libraries for capabilities like graphics, user interfaces, networking, and so on. These sorts of libraries interact with the world and therefore depend on the ability to maintain state and perform actions. Unfortunately, the simple notion of lifting presented earlier assumes procedures are free of side effects, so it can re-apply them anytime without risk of destructive effects. This assumption is clearly invalid in the case of such imperative libraries. The purpose of this chapter is therefore to develop a variation of lifting that works with stateful entities.[1]

The kinds of libraries in which I'm interested have several main characteristics. First, they tend to be large and detailed, so it is impractical to rewrite them. Second, they are maintained by third-party developers, so they should be integrated with a minimum of modification to enable easy upgrading. Third, these libraries—especially for GUIs—are often written in object-oriented (OO) languages. The integration process must therefore handle this style, and ideally exploit it. An important subtlety is that OO and FRP languages have different ways of handling state: OO *makes state explicit but encapsulates it, whereas state in* FRP *is hidden from the programmer by the temporal abstractions of the language*. Somehow, these two paradigms must be reconciled.

[1] This chapter expands on previously published joint work [52] with Daniel Ignatoff.

This chapter describes considerable progress on this integration problem for the specific case of GUIs. The DrScheme environment provides a large and robust GUI library called MrEd [45], based on the wxWindows framework, which is used to build DrScheme's interface itself. The environment is a good representative of a library that meets the characteristics listed above; furthermore, its integration is of immediate practical value. I have discovered several useful abstractions based on *mixins* [13] (classes parameterized over their super-classes) that enable a seamless integration. I have further found that there are patterns to these mixins and abstracted over them using *macros* [57]. As a consequence, the adapter for MrEd is under 400 lines of code.

This chapter is organized as follows. I first discuss the design philosophy that governs the adaptation of MrEd to a signal-based programming interface. What follows is the heart of the chapter: a description of the implementation of this interface and of the abstractions that capture the essence of the adaptation. I also discuss a spreadsheet application built with the adapted toolkit.

6.1 Adapting MrEd to FrTime

In adapting any existing library to become reactive, the main goal is to reuse the existing library implementation as much as possible and perform a minimum of manual adaptation. In order to minimize the manual effort, we need to uncover patterns and abstract over them. In this case, the problem is how to maintain a consistent notion of state between the object-oriented and functional reactive models.

The functional reactive world represents state implicitly through time-varying values, and the dataflow mechanism is responsible for keeping it consistent. In contrast, the object-oriented world models state with mutable fields, and programmers are responsible for writing methods that keep them consistent. We presume that the toolkit implementors have done this correctly, so our job is simply to translate state changes from the dataflow program into appropriate method invocations. However, since GUI toolkits also mediate changes coming from the user, they provide a callback mechanism by which the application can monitor state changes. The interface between the GUI and FrTime must therefore also translate callbacks into state changes in the dataflow world.

Not surprisingly, the nature of the adaptation depends primarily upon the direction of

communication. We classify each widget property according to whether the application or the toolkit changes its state. The most interesting case, naturally, is when both of them can change the state. We now discuss each case separately.

6.1.1 Application-Mutable Properties

MrEd allows the application to change many of a widget's properties, including its value, label, cursor, margins, minimum dimensions, and stretchability. A widget provides an accessor and mutator method for each of these properties, but the toolkit never changes any of them itself, so we classify these properties as "application-mutable."

In a functional reactive setting, we can manipulate time-varying values directly, so it is natural to model such properties with behaviors. For example, we would use a behavior to specify a gauge's value and range and a message's label. This sort of interface renders accessors and mutators unnecessary, since the property automatically updates whenever the behavior changes, and the application can observe it by reading whatever behavior it used for initialization.

To implement a behavior-based interface to such widget properties, the first step is to derive a subclass from the original MrEd widget. For example, we can define a *ft-gauge%* from the MrEd gauge.

(**define** *ft-gauge%*
  (**class** *gauge%* . . .))

In the new class, we want to provide constructor arguments that expect behaviors for all of the application-mutable properties. In FrTime, behaviors extend the universe of values, and any constant may be taken as a special case of a behavior (that never changes); i.e., behaviors are supertypes of constants. Thus the application may safely supply constants for any properties that it wishes not to change. Moreover, if we use the same property names as the superclass, then we can construct an *ft-gauge%* exactly as we would construct an ordinary gauge. This respects the principle of contravariance for function subtyping: our extension broadens the types of legal constructor arguments.

In fact, the DrScheme class system allows us to override the superclass's initialization arguments, or *init-field*s. Of course, the superclass still refers to the original fields, so

its behavior remains unchanged, but this lets us extend the constructor interface to permit behaviors. The code to add these initialization arguments is as follows:

(*init-field value label range vert-margin horiz-margin min-width* . . .)

Next, we need code to enforce consistency between these behavioral fields and the corresponding fields in the superclass. The first step is to perform superclass initialization, using the current values of the new fields as the initial values for the old ones. Although the old and new versions of the fields have the same names, there is no ambiguity in the superclass instantiation expression; in each name/value pair, the name refers to a field in the superclass, and the value expression uses the subclass's scope.

(**super-instantiate** () [*label* (*value-now label*)] [*range* (*value-now range*)] . . .)
(**send** *this set-value* (*value-now value*))

(Since there is no initial *value* field in the superclass, we need to set it separately after super-class initialization.)

Having set appropriate initial values for the fields, we need to ensure that they stay consistent as the behaviors change. That is, we need to translate changes in state from the dataflow program to the object-oriented "real world." This is a central problem in building an interface between the two models.

The basic idea behind our translation is straightforward: detect changes in a behavior and update the state of the corresponding object through an appropriate method call. We use the FrTime primitive *changes* to detect changes in a behavior and expose them on an event stream. Then we convert the event stream into a series of method invocations. This second step is somewhat unusual, since the methods have side effects, unlike the operations found in a typical dataflow model. However, in this case we are concerned not with *defining* the model but with *communicating* its state to the outside world. The effects are therefore both safe (they do not interfere with the purity of the model) and necessary (there is no other way to tell the rest of the world about the system's changing state).

The invocation of imperative methods is technically trivial. Since FrTime is built atop Scheme, any procedure that updates a signal is free to execute arbitrary Scheme code, including operations with side effects. Of course, we ordinarily avoid the practice of performing side effects in signal processors, since it could lead to the violation of program invariants. As mentioned above, it is safe when the effects are restricted to communication

with the outside world (as they are in this case). In particular, we use the primitive *map-e*, passing a procedure that invokes the desired method:

(*map-e* (λ (*v*) (**send** *this set-value v*)) (*changes value*))
(*map-e* (λ (*v*) (**send** *this set-label v*)) (*changes label*))
...

Each call above to *map-e* creates a new event stream, whose occurrences all carry the *void* value—the return value of the imperative method call—but are accompanied by the method's side effects. Because the event values are all *void*, they have no meaningful use within a larger dataflow program.

The above expressions are static initializers in the widget classes, so they are evaluated whenever the application constructs a new instance. Using static initializers allows the adapter to automatically forward updates without the developer having to invoke a method to initiate this. Because the code constructs signals, which participate in the dataflow computation, it therefore has a dynamic effect throughout the life of the widget, unlike typical static initializers.

Subtleties Involving Side-Effecting Signals

We have resolved the interface for communicating state changes from the dataflow to the object-oriented model. However, a more serious concern is the mismatch between their notions of *timing*. In a typical object-oriented program, method invocations are synchronous, which fixes the ordering of operations within each thread of control. However, FrTime processes updates according to their data dependencies, which does not necessarily correspond to a sequential evaluation order. This makes it difficult for programmers to reason about when effects occur.

Fortunately, the functional reactive model and interface are designed in such a way as to prevent operations from occurring unpredictably. Firstly, there is at most one signal associated with any given widget property. If the programmer wishes to control a widget with several different signals, he must define a composite signal that mediates explicitly between the individual signals. Thus, there can be no implicit modifications or contention over who is responsible for keeping it up-to-date.

Secondly, FrTime processes updates in order of data dependencies, so if one property's

signal depends on another's, then it will be updated *later*. If the order of updates were significant, and if the dependencies in the toolkit were reflected by dependencies in the application, then this would yield a "safe" order in which to update things.

There is, however, a problem with the strategy described above that is difficult to diagnose and debug. The symptoms are as follows: at first, the program seems to work just fine. Sometimes it may run successfully to completion. Other times, depending upon what else is happening, it runs for a while, then suddenly and seemingly without explanation the gauge's properties stop updating when the behaviors change. The point at which it stops varies from run to run, but there are never any error messages.

The problem results from an interaction with the memory manager. An ordinary FRP application would use the event source returned by the *map-e*, but in this case we only care about side effects, so we neglect to save the result. Since there are no references to the updating event source, the garbage collector eventually reclaims it, and the gauge stops reacting to changes in the behavior.

To avoid these problems, we define a new abstraction specifically for side-effecting event processors. This abstraction, called *for-each-e!*, works just like *map-e*, except that it ensures its result will not be collected. It also lends itself to a more efficient implementation, since it can throw away the results of the procedure calls instead of enqueuing them on a new event stream.

The *for-each-e!* implementation stores references to the imperative event processors in a hash table, indexed by the objects they update. It is important that this hash table hold its keys with weak references so that, if there are no other references to the widget, both it and the event processor may be reclaimed.

6.1.2 Toolkit-Mutable Properties

Some widget properties are controlled primarily by the user or the toolkit rather than the application. For example, when the user resizes a window, the toolkit adjusts the locations and dimensions of the widgets inside. Since the application cannot control these properties directly, the widgets provide accessor methods but no mutators. Additionally, the application may want to be notified of changes in a property. For example, when a drawing canvas changes size, the application may need to update its content or recompute parameters for its

scrollbars. For such scenarios, accessor methods alone are insufficient, and toolkits provide callback interfaces as described in the previous section. However, we saw that callbacks lead to an imperative programming style with various pitfalls, so we would like to support an alternative approach.

For such "toolkit-mutable" properties, we can remove the dependency on callbacks by adding a method that returns the property's time-varying value as a behavior. For example, instead of allowing registration *on-size* and *on-move* callbacks, the toolkit would provide methods that return behaviors reflecting the properties for all subsequent points in time.

The implementation of such methods is similar to that for application-mutable properties. However, in this case we cannot just override the existing *get-width*, *get-height*, *get-x*, and *get-y* methods and make them return behaviors. Though FrTime allows programmers to use behaviors just like constants, an application may need to pass a widget to a library procedure written in raw Scheme. (For example, the widget may need to invoke methods in its superclass, which is implemented in Scheme.) If a Scheme expression invokes an accessor and receives a behavior, there is nothing FrTime can do to prevent a type-mismatch error. Since behaviors are supertypes of constants, overriding in this manner would violate the principle of covariance for procedure return values.

To preserve type safety, we must define the new signal-aware methods so as not to conflict with the existing ones. We choose the new names by appending *-b* to the existing names, suggesting the behavioral nature of the return values. Again, we derive a subclass of the widget class we want to wrap. For example, continuing with the *ft-gauge%*, we would add methods called *get-width-b get-height-b*, *get-x-b*, and *get-y-b*.

We need to determine how to construct the behaviors returned by these methods. We want these behaviors to change with the corresponding widget properties, and we know that the widget's *on-size* or *on-move* method will be called when the properties change. So, we are now faced with the converse of the previous problem—converting a imperative procedure call into an observable FrTime event.

FrTime provides an interface for achieving this goal, called *make-event-receiver*. This procedure returns two values: an event source *e* and a unary procedure **send-event**$_e$. Whenever the application executes (**send-event**$_e$ *v*), the value *v* occurs on *e*. In the implementation, **send-event**$_e$ sends a message to the FrTime dataflow engine indicating that *v* should

occur on *e*, which leads to *v*'s being enqueued on the stream of *e*'s occurrences. By overriding the widget's callbacks and calling *make-event-receiver*, we can create an event source carrying changes to the widget's properties:

```
(define-values (width-e send-width) (make-event-receiver))
(define-values (height-e send-height) (make-event-receiver))
(define/override (on-size w h)
  (super on-size w h)
  (send-width w)
  (send-height h))
;; similarly for position
```

Once we have the changes to these properties in the form of FrTime event sources, we convert them to behaviors with *hold*:

```
(define/public (get-width-b) (hold width-e (send this get-width)))
(define/public (get-height-b) (hold height-e (send this get-height)))
...
```

6.1.3 Application- and Toolkit-Mutable Properties

We have discussed how to adapt properties that are mutable by *either* the toolkit or the application, but many properties require mutability by *both* the toolkit and the application. This need usually arises because there are several ways to change the same property, or several views of the same information. For example, a text editor provides scrollbars so the user can navigate a long document, but the user can also navigate with the keyboard, in which case the application needs to update the scrollbars accordingly.

All widgets that allow user input also provide a way to set the value from the application. Several other properties may be set by either the toolkit or the user:

focus When the user clicks on a widget, it receives *focus* (meaning that it hears key strokes) and invokes its *on-focus* callback method. This is the common mode of operation, but the application can also explicitly send focus to a widget. For example, when a user makes a choice to enter text, the application may automatically give the text field focus for the user's convenience.

visibility The application may hide and show widgets at various stages of an interactive computation. Since *show*ing a widget also shows all of its descendents, the toolkit provides an *on-enable* callback so the application does not need to track ancestry. In addition, the user can affect visibility by, for example, closing a window, which hides all of its children.

ability Similar to visibility, the application can selectively enable and disable widgets depending upon their necessity to various kinds of interaction. Enabling also works transitively, so the toolkit invokes the *on-enable* method for all children of a newly-enabled widget.

One might naturally ask, since we have already discussed how to adapt application- and toolkit-mutable properties, why we cannot simply combine the two adaptation strategies for these hybrid properties. The reason is that the application specifies a property's time-varying value through a behavior, which defines the value at every point in the widget's lifespan. This leaves no gaps for another entity to specify the value.

Our solution to this problem is to use event sources in addition to behaviors. Recall that in the implementation of toolkit-mutable properties, we first constructed an event source from callback invocations, then used hold to create a behavior. In this case, both the application and toolkit provide event streams, and instead of holding directly, we merge the streams and hold the result to determine the final value:

```
(init-field app-focus app-enable app-show)
(define-values (user-focus send-focus) (make-event-receiver))
(define/public (has-focus-b?)
  (hold (merge-e app-focus user-focus) (send this has-focus?)))
(define/override (on-focus on?)
  (super on-focus on?)
  (send-focus on?))
...
```

This code completely replaces the fragments shown previously for properties that are mutable by only the application or the toolkit.

6.1.4 Immutable Properties

MrEd does not allow certain properties to change once a widget is created. For example, every non-window widget has a parent, and it cannot be moved from one parent to another. In theory, we could build a library atop MrEd in which we simulated the mutability of these properties. However, this would be a significant change to not only the toolkit's interface but also its functionality, and we would have to implement it ourselves. Since our goal is to reify the existing toolkit through a cleaner interface, we have not attempted to extend the underlying functionality.

6.2 Automating the Transformation

We have so far discussed how to replace the imperative interface to object-oriented widget classes with a more elegant and declarative one based on behaviors and events. The problem is that there is a large number of such widgets and properties, and dealing with all of them by hand is a time-consuming and tedious task. Thus we look to reduce the manual effort by automating as much as possible of the transformation process.

The reader may have noticed that the code presented in the previous section is highly repetitive. There are actually two sources of repetition. The first is that we need to perform many of the same adaptations for all of the MrEd widget classes, of which there are perhaps a dozen. The second is that the code used to adapt each property is essentially the same from one property to the next. We now discuss how to remedy these two forms of duplication individually, by abstracting first over multiple widget classes, then over multiple properties within each class.

6.2.1 Parameterized Class Extensions

In Sect. 6.1 we adapted a collection of widget properties by sub-classing. Since most of the code in the subclasses is essentially the same across the framework, we would like to be able to reuse the common parts without copying code. In other words, we would like a class extension parameterized over its superclass.

The DrScheme object system allows creation of *mixins* [13, 46], which are precisely such parameterized subclasses. We write a mixin to encapsulate the adaptation of each

property, then apply the mixins to all classes possessing the properties. For example, instead of defining an *ft-gauge%* like we did before, we define a generic class extension to adapt a particular property, such as the label:

```
(define (adapt-label a-widget)
  (class a-widget
    (init-field label)
    (super-instantiate () [label (value-now label)])
    (for-each-e! (changes label) (λ (v) (send this set-label v)) this)))
```

In the code snippet above, we box the superclass position of the class definition to highlight that it is a variable rather than the literal name of a class. This parameterization makes it possible to abstract over the base widget class and thus to apply the adaptation to multiple widgets.

We write mixins for other properties in a similar manner. Since there are several properties common to all widget classes, we compose all of them into a single mixin:

```
(define (adapt-common-properties a-widget)
  (foldl (λ (mixin cls) (mixin cls)) a-widget (list adapt-label adapt-enabling ...)))
```

Although this procedure contains no explicit **class** definitions, it is still a mixin: it applies a collection of smaller class extensions to the input class. This *compound* mixin takes a raw MrEd widget class and applies a mixin for each standard property. The resulting class provides a consistent FrTime interface for all of these properties. For example, we can use this mixin to adapt several widget classes:

```
(define pre-gauge% (adapt-common-properties gauge%))
(define pre-message% (adapt-common-properties message%))
...
```

We call the resulting widget classes "pre-" widgets because they still await the adaptation of widget-specific properties. Most importantly, each widget supports manipulation of a particular kind of value (e.g., boolean, integer, string) by either the application or the toolkit, and the various combinations give rise to different programmer interfaces.

6.2.2 A Second Dimension of Abstraction

Mixins allow us to avoid copying code across multiple classes. However, there is also code duplication across mixins. In Sect. 6.1, we develop patterns for adaptation that depend on whether the property is mutable by the application, the toolkit, or both. Once we determine the proper pattern, instantiating it only requires identification of the field and method names associated with the pattern. However, in Sect. 6.1 we duplicated the pattern for each property.

In most programming languages, we would have no choice but to copy code in this situation. This is because languages don't often provide a mechanism for abstracting over field and method names, as these are program syntax, not values. However, Scheme provides a *macro system* [57] with which we can abstract over program syntax. For example, with application-mutable properties we only need to know the name of the field and mutator method, and we can generate an appropriate mixin:

```
(define-syntax adapt-app-mutable-property
  (syntax-rules ()
    [(_ field mutator)
     (λ (widget)
       (class widget
         (init-field field)
         (super-instantiate () [field (value-now field)])
         (for-each-e! (changes field) (λ (v) (send this mutator v)) this)))]))
```

With this macro, we can generate mixins for the application-mutable properties:

```
(define adapt-label (adapt-app-mutable-property label set-label))
(define adapt-vert-margin (adapt-app-mutable-property vert-margin vert-margin))
...
```

Of course, we write similar macros that handle the other two cases of mutability and instantiate them to produce a full set of mixins for all of the properties found in MrEd's widget classes. At this point, we have fully abstracted the principles governing the toolkit's adaptation to a functional reactive interface and captured them concisely in a collection of macros. By instantiating these macros with the appropriate properties, we obtain mixins that adapt

the properties for actual widgets. We compose and apply these mixins to the original MrEd widget classes, yielding new widget classes with interfaces based on behaviors and events.

The ability to compose the generated mixins safely depends upon two properties of the toolkit's structure. Firstly, most properties have distinct names for their fields and methods and hence are non-interfering by design. Secondly, in cases where two properties *do* share a common entity (for example, the single callback *on-size* affects the width and height), the disciplined use of inheritance (i.e., always calling **super**) ensures that one adaptation will not conflict with the other.

To save space and streamline the presentation, we have simplified some of the code snippets in this paper. The full implementation has been included with the DrScheme distribution since release version 301. We provide a catalog of adapted widgets in an appendix. The core contains about 80 lines of macro definitions and 300 lines of Scheme code. This is relatively concise, considering that the MrEd toolkit consists of approximately 10,000 lines of Scheme code, which in turn provides an interface to a 100,000-line C++ library. Moreover, our strategy satisfies the criteria set forth in the Introduction: it is a pure interface extension and does not require modifications to the library.

6.2.3 Language Independence of the Concepts

Some of the ideas presented in this chapter are specific to DrScheme. For example, DrScheme's object system supports features like mixins and keyword constructor arguments, which more common languages like C++ and Java do not provide. Likewise, DrScheme's macro system offers a more sophisticated metaprogramming system than is found in most languages. Because I have made use of these less common features, a reader might argue that the ideas are not portable.

However, while the implementation techniques are somewhat specific to DrScheme, I argue that the essential concepts apply to a wide array of call-by-value languages, much like the embedding techniques of FrTime in general. For example, categorizing state transfers as *application to toolkit* versus *toolkit to application* (or both) is a necessary first step, and determining whether state is *discrete* or *continuous* is also important for any functional reactive toolkit adaptation. Once these characterizations are made, the same basic approaches may be used to translate state changes between the functional and imperative subsystems.

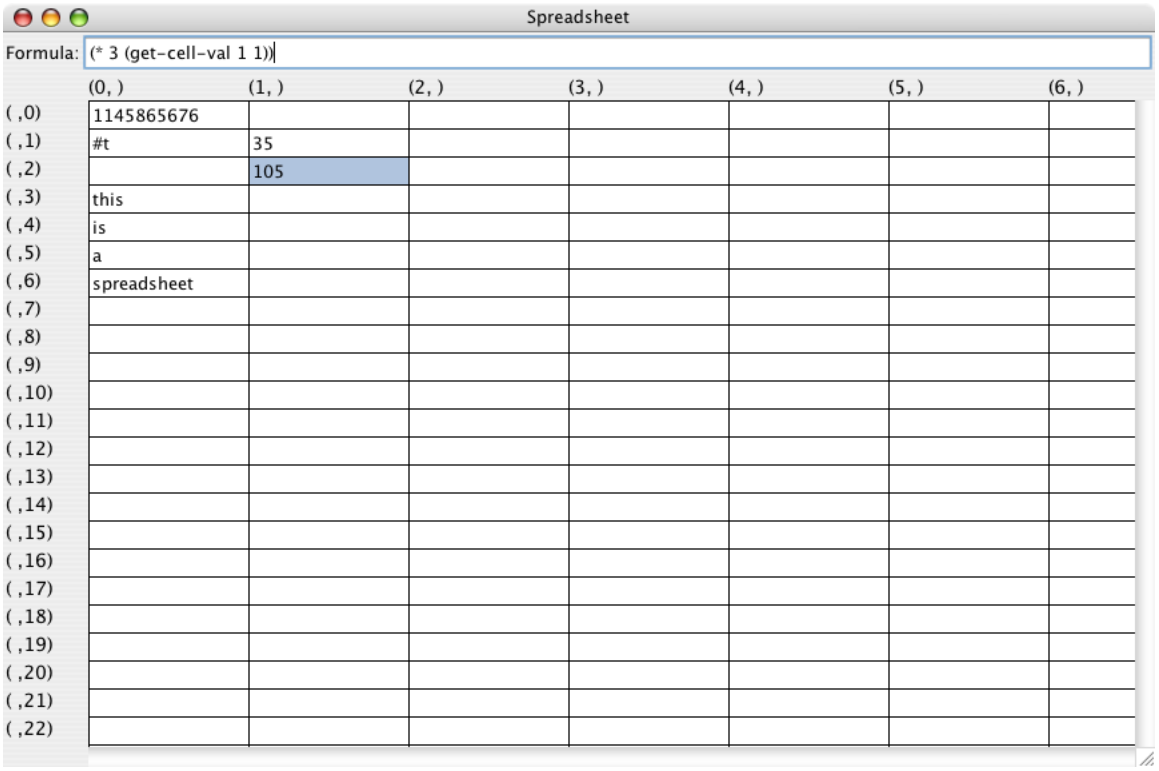


Figure 6.1: Screenshot of the FrTime spreadsheet application

These necessarily include impure event-based mechanisms (e.g., *send-event*, *for-each-e!*) for defining the bridge between these subsystems. Once there is a way for them to communicate, the patterns underlying the adaptation may be abstracted using whatever techniques are available within the host language. DrScheme happens to provide powerful features (mixins and macros) that allow a high level of abstraction. In other languages, different features may be available (e.g., multiple inheritance, static overloading) to support alternate approaches to this problem.

6.3 A Spreadsheet Application

To evaluate the adapted version of MrEd, I have applied it to a realistic spreadsheet application. The major challenges in building a spreadsheet, in my experience, are implementing a language with its dataflow semantics, and managing and displaying a large scrollable array of cells. Fortunately, FrTime makes the linguistic problem relatively straightforward,

since its dataflow evaluator can be reused to implement update propagation. This leaves the representation and display of the cell grid.

The core of the spreadsheet user interface is an extension of the MrEd *canvas* widget. A canvas is a region in which the application can listen to key and mouse events and perform arbitrary drawing operations. The application renders the cell content into a canvas and processes mouse events to perform selection. When the user selects a cell, he can enter a formula into a text field, and the selected cell receives the value of the formula.

The functional reactivity helps greatly, for example, in managing the scrolling of the grid content. The canvas includes a pair of scrollbars, which must be configured with ranges and page sizes. These parameters depend upon the number of cells that fit within the physical canvas, which in turn depends upon the size of the canvas relative to the size of the cells. The cell size depends in turn upon the font and margins used when rendering the text. Since the user can resize the window or change the font, these parameters must be kept up-to-date dynamically. In raw MrEd, all of this recomputation would need to be managed by hand, but with the FrTime adaptation, we simply specify the functional relationships between the time-varying values, and the various widget properties update automatically.

For example, the following expression defines the number of characters that fit horizontally in the canvas at one time:

```
(define v-cells-per-page
  (quotient (- canvas-height top-margin) cell-height))
```

Both *cell-height* and *canvas-height* are time-varying, and *v-cells-per-page* always reflects their current state. The range on the scroll bar is equal to the difference between the total number of cells (rows) and the number that can be displayed on a single page:

```
(define v-scroll-range
  (max 0 (- total-rows v-cells-per-page)))
```

If the user resizes the window or changes the font, the range on the scrollbar updates automatically.

When the user clicks at a particular position in the canvas, the application needs to map the position to a cell so it can highlight it and allow the user to edit its formula. The following code expresses the mapping:

```
(define (y-pos->row-num y)
```

```
(if (> y top-margin)
    (+ v-scroll-pos (quotient (- y top-margin) cell-height))
    -1))
```

By applying this function to the y component of the mouse position within the canvas, the application obtains the row number (if any) over which the mouse is hovering. It uses this information to implement a roll-over effect, shading the cell under the mouse cursor, and to determine which cell to select when the user clicks the mouse.

The following code shows the definition of the text field in which the user enters cell formulas:

```
(define formula
  (new ft-text-field%
       [label "Formula:"]
       [content-e (map-e (λ (addr) [(value-now (cell-text (addr->key addr)))]1)
                         select-e))]
       [[focus-e select-e]]2))
```

When the user clicks on a cell, the cell's address appears on an event stream called *select-e*. The occurrence of the selection event affects *formula* in two ways. First, the code in box 1 retrieves the selected cell's text from the spreadsheet; this text becomes *formula*'s new content. Second, the code in box 2 specifies that selection events send focus to *formula*, allowing the user to edit the text. When the user finishes editing and presses the *enter* key, *formula* emits its content on an output event stream; the application processes the event and interprets the associated text (code not shown).

The spreadsheet experiment has proven valuable in several respects. First, by employing a significant fragment of the MrEd framework, it has helped us exercise many of our adapters and establish that the abstractions do not adversely affect performance. Second, as a representative GUI program, it has helped us identify several subtleties of FRP and the adaptation of state, some of which we have discussed in this paper. Finally, the spreadsheet is an interesting application in its own right, since the language of the cells is FrTime itself, enabling the construction of powerful spreadsheet programs.

6.4 Catalog of Adapted User Interface Widgets

ft-frame% These objects implement top-level windows. They support all of the standard signal-based property interfaces (label, size, position, focus, visibility, ability, margins, minimum dimensions, stretchability, and mouse and keyboard input). As in the underlying *frame%* objects, the *label* property specifies the window's title.

ft-message% These objects contain strings of text that are mutable by the application but not editable by the user. They support all of the standard signal-based property interfaces. In this case, the *label* property specifies the content of the message.

ft-menu-item% These objects represent items in a drop-down or pop-up menu. In addition to the standard properties, each widget exposes an event stream that fires whenever the user chooses the item.

ft-button% These objects represent clickable buttons. In addition to the standard properties, each widget exposes an event stream that fires each time the user clicks it.

ft-check-box% These objects represent check-box widgets, whose state toggles between **true** and **false** with each click. In addition to the standard properties, each *ft-check-box%* widget exposes a boolean behavior that reflects its current state. The application may also specify an event stream whose occurrences set the state.

ft-radio-box% These objects allow the user to select an item from a collection of textual or graphical options. In addition to the standard properties, each *ft-radio-box%* object exposes a numeric behavior indicating the current selection.

ft-choice% These objects allow the user to select a subset of items from a list of textual options. In addition to the standard properties, each *ft-choice%* object exposes a list behavior containing the currently selected elements.

ft-list-box% These objects are similar to *ft-choice%*, except that they support an additional, immutable *style* property that can be used to restrict selections to singleton sets or to change the default meaning of clicking on an item. Otherwise, the application's interface is the same as that of *ft-choice%*.

ft-slider% These objects implement slider widgets, which allow the user to select a number within a given range by dragging an indicator along a track. In addition to the standard properties, each *ft-slider%* object allows the application to specify the range through a time-varying constructor argument called *range*, and it exposes a numeric behavior reflecting the current value selected by the user.

ft-text-field% These objects implement user-editable text fields. In addition to the standard properties, each widget exposes the content of its text field as a behavior, as well as an event stream carrying the individual edit events. The application can also specify an event stream whose occurrences replace the text field content.

Chapter 7

Programming Environment Reuse

Building a new language is no modest undertaking, so there is value in reducing the manual development effort as much as possible. Importantly, a language is more than just an interpreter or compiler. To compete with established systems it must also have extensive libraries and an array of tools that support program understanding and development. These comprise the overall programming environment in which users work. By helping to automate the tasks of finding bugs and elucidating program behavior, they contribute greatly to the power of languages as software engineering tools.

Unfortunately, developing and maintaining a high-quality tool suite demands significant time and effort. Especially for domain-specific languages, where the target market is small (at least initially), adequate resources may not be available to develop all of the needed infrastructure from scratch.

One way to reduce the cost of developing a new domain-specific language is to *embed* it within an existing general-purpose language. Typically, this involves implementing the core functionality as a library within the host language and supporting it with lightweight syntactic extensions. Since embedded DSL programs are essentially host-language programs, they can reuse any interpreters, compilers, runtime systems, libraries, and tools that have been written for the host language.

Embedding has another appeal: if programmers are already familiar with the general-purpose host language and its environment, then they can adopt the DSL with significantly greater ease than something completely new and foreign. This "cognitive reuse" also increases a language's utility and popularity.

Although embedding offers extensive reuse opportunities and greatly reduces the manual effort required to implement a DSL, there is a subtle but important pitfall that accompanies tool reuse. In particular, if one applies a host-language tool to an embedded-language program, what it shows us is the program's *implementation*, in terms of the host-language constructs that the tool understands. Depending upon the complexity of the DSL, this may not match the abstractions of the DSL or meaningfully reflect the original program's behavior. It may instead expose implementation details that confuse or even mislead the user.

In this chapter, I address the problem of meaningfully reusing a language's tools. The solution strategy depends upon the host language's providing a suitable interface for implementing control-oriented tools, i.e., a mechanism for reflecting on a program's control flow. By manipulating such a mechanism, I develop the notion of *effective evaluation contexts*, which underlie the tool-adaptation technique for embedded languages. I have applied this technique to several existing DSLs (including most notably FrTime) and tools, and the results are encouraging. The modifications are straightforward, and the result is more helpful and appropriate feedback from the tools.

Since programming tools play an important role in software engineering, it is critical that they be reliable and trustworthy. The manipulation of tool interfaces creates opportunities for reuse but also introduces possibilities for subtle errors. While raw tool reuse gives results that violate the DSL's abstractions, at least they present, in some sense, a ground truth about the execution of the system. In contrast, a complex new tool that gives correct answers sometimes but fails in other contexts may lead users further astray and result in heightened frustration. To avoid such problems, I develop a formal model that specifies the intended behavior of a specific tool-language combination.

7.1 Background

I am concerned with the reuse of control-oriented tools in deep DSL embeddings. Before I discuss the problem, or my solution, I explain the concept of deep embedding, and what I mean by control-oriented tools.

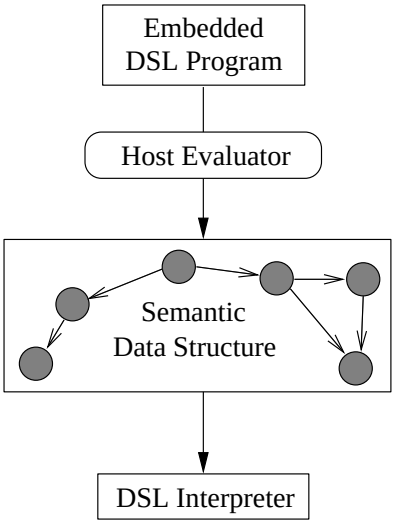


Figure 7.1: Structure of a deep embedding

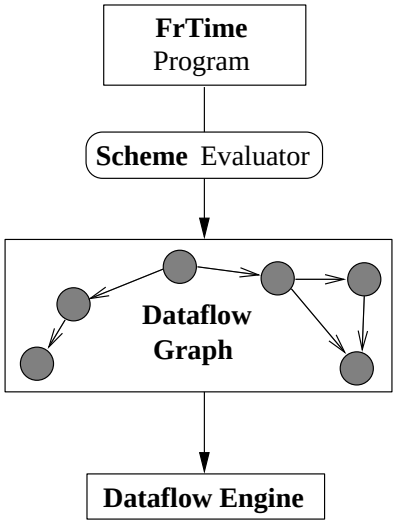


Figure 7.2: Embedding FrTime

7.1.1 Domain-Specific Embedded Languages

Domain-specific languages (DSLs) are programming languages designed specifically to handle the pervasive concerns of specific problem domains. Because they can abstract away concerns that cannot be encapsulated modularly in a general-purpose language, DSLs are extremely powerful software-engineering tools.

Developing any new language is a non-trivial task. Though tools like scanner and parser generators have been around for decades, these only help with a small part of the problem. Such concerns as type-checking, code-generation, and runtime support constitute a much more significant part of the task, for which there is less support.

One technique for reducing the burden of language development is to *embed* the new language within an existing "host" language. Embedding supports reuse of much of the host language's infrastructure, including its type system, runtime system, libraries, and interpreter and/or compiler.

An embedding can be either *shallow* or *deep* [12]. In a shallow embedding, constructs in the DSL are implemented directly as host-language abstractions. A shallow embedding is essentially a library; it can add new functionality, but not new features, to the host language.

In contrast, a deep embedding represents DSL constructs as host-language data structures, to which an explicit interpreter assigns meaning. Figure 7.1 shows a diagram of this

model. The host language evaluates the DSL program, yielding a data structure that encodes the program's meaning; an interpreter processes the data structure and implements the semantics.

A deep embedding requires more effort to implement, since it involves explicit definitions of the embedded language's constructs. However, it offers more flexibility and power to the language implementor; the DSL's features are not confined by those of the host language. Also, since there is an explicit representation of the embedded program, the implementation can analyze, optimize, or otherwise manipulate the embedded program.

7.1.2 Examples

My experience primarily involves the language FrTime, the focus of this dissertation. Since FrTime's notion of dataflow lacks direct support from Scheme, its embedding must be (at least partially) deep. The semantic data structure is a graph of the program's dataflow dependencies, and the interpreter is a dataflow engine that traverses this graph and recomputes signals in response to changes in the environment and other signals (see Figure 7.2).

7.1.3 Control-Oriented Tools

A control-oriented tool is one that observes points in the control-flow of an executing program. For example, a profiler counts how many times expressions execute, and how much they contribute to the total execution time. Likewise, an error-tracer catches errors and shows the user where they came from (e.g., the immediate expression that raised the error and its context of execution).

Building a control-oriented tool requires an interface for extracting information about the state of a running program. In many language implementations, the compiler generates such information and represents it in a proprietary, extra-lingual manner. In such cases, the tools are tightly coupled to the compiler.

Other implementations provide an open, linguistic mechanism for reflecting on control flow. For example, Java compilers provide information in class files so that runtime systems can track the file, class, method, and line number of each activation record. This information is available through a public method in the *Exception* class, so applications can use it for their own purposes.

PLT Scheme provides an even more general and open stack inspection mechanism based on continuation marks [23]. With this mechanism, an application can associate a *mark* with the evaluation of an expression. The mark is an arbitrary Scheme value and resides on the runtime stack, in the expression's activation record, until the expression finishes evaluating. An application can, at any point in its execution, introspect on its control flow by requesting the set of marks currently on the stack.

The continuation mark interface exists primarily for tools, not applications. To work properly, the tools need a way to install whatever information they require in the marks. PLT Scheme supports this capability by means of a lightweight interface to its compiler, through which tools can syntactically transform, or *annotate*, target programs before they execute. For example, the error-tracer works by installing, for each expression, a mark containing the expression's source location. The runtime system automatically captures the continuation marks when an exception arises and stores them in an *exception* object. If the application fails to handle the exception, the error-tracer catches it at the top-level, extracts the source location information from its continuation marks, and presents a trace to the user.

The combination of annotation and continuation marks allows for the implementation of a wide variety of tools. Some that have been developed include a profiler, an algebraic stepper [25], and a scriptable debugger [63]. These tools insert annotations to perform such tasks as timing execution, checking for and suspending at breakpoints, and installing continuation marks that not only identify the source locations of active procedures but also provide access to the names and values of lexical variables.

More importantly, continuation marks offer a clean interface for control-flow introspection that is backed by a formal model [23]. The use of explicit syntactic transformations also allows us to capture the notion of a *debugging compiler* that communicates with tools through this continuation-mark interface. These abstractions prove useful when we formalize the interaction between control-oriented tools and deep DSLs embeddings.

7.2 The Tool-Reuse Problem

A deep DSL embedding can reuse much of the host language's infrastructure directly. However, if we try to reuse the host language's control-oriented tools, the feedback they provide

```
LiftB.evalMethod: Exception raised when invoking method foo
java.lang.reflect.InvocationTargetException
        at sun.reflect.GeneratedMethodAccessor2.invoke(Unknown Source)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:324)
        at edu.yale.cs.frp.FRPUtilities$LiftedB.evalMethod(FRPUtilities.java:458)
        at edu.yale.cs.frp.FRPUtilities$LiftedB.recomputeValue(FRPUtilities.java:486)
        at edu.yale.cs.frp.FRPUtilities$LiftedB.propertyChange(FRPUtilities.java:472)
        at java.beans.PropertyChangeSupport.firePropertyChange(PropertyChangeSupport.java:252)
        at edu.yale.cs.frp.BehaviorAdapter.firePropertyChange(BehaviorAdapter.java:65)
        at edu.yale.cs.frp.BehaviorAdapter.setValue(BehaviorAdapter.java:90)
        at edu.yale.cs.frp.Stepper.eventOccured(Stepper.java:42)
        at edu.yale.cs.frp.AbstractEventSource.setEvent(AbstractEventSource.java:62)
        at edu.yale.cs.frp.AccumE.eventOccured(AccumE.java:57)
        at edu.yale.cs.frp.AbstractEventSource.setEvent(AbstractEventSource.java:62)
        at edu.yale.cs.frp.EventBind.eventOccured(EventBind.java:42)
        at edu.yale.cs.frp.AbstractEventSource.setEvent(AbstractEventSource.java:62)
        at edu.yale.cs.frp.FRPUtilities$EventObserver.invoke(FRPUtilities.java:256)
        ...
```

Figure 7.3: An error trace from a Java FRP implementation

by default will be misleading. This is because the DSL program itself does not perform any interesting computation—it only creates a representation of itself as a data structure, which the DSL implementation interprets in order to realize the program's semantics. Thus, control-oriented tools observe and analyze the process of interpreting the data structure, although this may have no apparent connection to the original DSL program.

For example, executing a FrTime program constructs a graph of dataflow dependencies. The behavior the user cares about begins afterward, when the language's dataflow engine traverses and recomputes values on the graph. Regardless of the structure of the FrTime program, the dataflow engine is an infinite loop with a relatively shallow stack.

If a program is syntactically well-formed, then construction of its dataflow graph completes without any problems, but a logical bug may cause a runtime error (e.g., division-by-zero, index out-of-bounds) to arise later while recomputing a signal. Since the dataflow engine performs this recomputation, it experiences the error, and any tool that observes control flow—even a full-featured interactive debugger—will blame the update algorithm. A typical user will probably be perplexed by the error trace, since it has nothing to do with his program. A more advanced user might even suspect that there is a bug in the language's implementation.

Although, strictly speaking, the error-tracer is telling the truth, the feedback it gives is problematic for two key reasons:

1. It fails to identify the actual source of the error, which is the FrTime expression that constructed the problematic signal. (In fact, since the engine's control structure is so simple, the tool produces essentially the same trace for most errors.)

2. It exposes the DSL's implementation to the user, which violates a basic principle of language design.

The use of deep embedding may also cause other tools to produce misleading results. For example, consider the output that a profiler gives when run on a FrTime. The user's program executes once (and only once), consuming a short burst of processing time to construct a semantic data structure. After that, the interpreter (dataflow engine or slide renderer) performs the repetitive and intensive computation necessary to implement the program's semantics. A profiler observes and reports this low-level phenomenon. However, this is neither surprising nor useful to a programmer looking to understand which parts of his program are responsible for consuming the most resources.

These tool interactions are inherent to this style of embedding and are not specific to Scheme. For example, Frappé is a Java implementation of functional reactive programming which, like FrTime, builds an explicit graph of the program's dataflow dependencies and employs an external recomputation algorithm to keep signals up-to-date. Once the graph is constructed, the update algorithm performs all of the interesting computation and is what any tool will observe. As in FrTime, there is no connection to the original program (that constructed the graph), which is ultimately responsible for the program's behavior.

Figure 7.3 shows a specific example of the problem, an error trace for a *division-by-zero* error in a Frappé [29] program. The trace reveals many details about the implementation instead of indicating the source of the error. We show this to emphasize that the problem we describe is neither an artifact of our choice of host language, nor simply the result of our own carelessness as language implementors. Moreover, it demonstrates that other implementors of similar systems have not already solved the problem.

Control-oriented tools are not the only ones that can have misleading interactions. In any embedded DSL, the implementation may use a low-level host-language data structure to implement a high-level domain-specific abstraction. In such cases, the host language's data-oriented tools, which are unaware of the encoding, will present the data in host-language terms. This breaks the DSL's abstractions in an analogous manner to control-oriented tools. However, the solution in this case is relatively simple; the language provides the tool with a custom display translator that renders the data in an appropriate manner. We therefore focus on control-oriented tools for the rest of the paper and do not discuss data

interactions any further.

Admittedly, control-oriented tools are less important for DSELs implemented in statically-typed languages, since the type checker catches a significant fraction of the mistakes before the program has a chance to run. In theory, the type checker might report a misleading unification failure within the DSEL implementation, thereby exhibiting the same problem we've described about runtime tools. In practice, however, this does not seem to be problem, most likely because the DSEL implementation may be type-checked separately from the user program, allowing the type-checker to divert blame from the (internally consistent) implementation.

7.3 Solution Techniques

Chapter 3 presented a formal model of FrTime. The model separates conceptually into two "layers", a low-level evaluation semantics that resembles that of Scheme or the λ_v-calculus [84] and a high-level dataflow semantics that captures FrTime's reactive recomputation strategy.

The low-level layer, shown in Figure 3.3, is a small-step operational semantics based on evaluation contexts [41], the grammar for which appears in Figure 3.1, along with the syntax for expressions and values.

The high-level evaluation rules (Figure 3.5) define the operation of FrTime's dataflow propagation mechanism. They operate on 4-tuples $\langle t, S, D, I, t\rangle$, where D and I are the same as in the low-level semantics, S is a store containing the current values of signals, and t is the current time.

A key element of the semantic model is the evaluation context E in the `signal` structure. This represents the evaluation context that created the signal. For example, suppose a user were to evaluate the following expression (where *x* is a behavior):

$(+\ 4\ (-\ 3\ (*\ 2\ (/\ 1\ x))))$

In its evaluation, the first step would be to apply the δ rule, which would match the original triple as:

$$\langle I, D, (+\ 4\ (-\ 3\ (*\ 2\ [\ (/\ 1\ x)\ ]\ )))\rangle$$

and take a small step to the tuple:

$$\langle I \cup \{\sigma\}, D \cup \{(x, \sigma)\}, (+\ 4\ (-\ 3\ (*\ 2\ [\ \sigma\ ]))) \rangle$$

where

$$\sigma = (\texttt{sig}\ \ (+\ 4\ (-\ 3\ (*\ 2\ [\ ])))\ \ (/\ 1\ x))$$

Thus, the new signal σ captures the evaluation context from the step that created it: $(+\ 4\ (-\ 3\ (*\ 2\ [\ ])))$. This means that, if *x* takes on the value 0, resulting in a division-by-zero error, then the dataflow evaluator knows both the specific operation that caused the error, i.e., $(/\ 1\ x)$, and the context, $(+\ 4\ (-\ 3\ (*\ 2\ [\ ])))$, in which it *would have occurred* had it been evaluated in its original context. Thus it has enough information to report a meaningful error message.

7.3.1 Higher-Order Signals

An important feature of FrTime is that the update of one signal can result in the creation of new signals. For example, consider the following expression:

$(-\ 3\ (*\ 2\ (\textbf{if}\ b\ (f\ x)\ (g\ y))))$

As *b* takes on different values, $(f\ x)$ and $(g\ y)$ are evaluated alternately, and in general this evaluation creates new signals. One might naturally think to evaluate $(f\ x)$ or $(g\ y)$ by itself, but this would mean dropping the part of their context that lies outside the **if** expression, i.e., $(-\ 3\ (*\ 2\ [\ ]))$. Any signals created by this evaluation would likewise contain only the local context from inside the **if**.

However, because every signal captures the context of its creation, it is possible to preserve the proper contextual information even in cases where a signal results from another signal's update. For example, the signal created for the **if** expression is:

$$(\texttt{sig}\ \ (-\ 3\ (*\ 2\ [\ ]))\ \ (\textbf{if}\ [\ ]\ (f\ x)\ (g\ y))\ \ \mathrm{b})$$

As shown in rule U-SWC, whenever the value of *b* changes, the inner context—$(\textbf{if}\ [\ ]\ (f\ x)\ (g\ y))$—will be applied to the new value to construct a new branch. The important thing is that the evaluation of the resulting expression occurs within the outer context, $(-\ 3\ (*$

2 [ ])), so any new signals will contain the full context that would have been present in a traditional call-by-value evaluation. Evaluation only proceeds until the original (inner) expression has been reduced to a value. Thus the outer context is just present to record the origin of any signals.

7.3.2 Implementation

The next step is to apply this idea of capturing evaluation contexts to a real implementation. Of course, realistic programming languages (including Scheme) don't have evaluation contexts that programs can capture and manipulate. However, many of them provide a mechanism for inspecting the control stack, which is the implementation's representation of the evaluation context.

In DrScheme, the stack inspection mechanism is based on the notion of *continuation marks*.

To understand why DrScheme's error-tracer fails in our embedded implementation, we must look more closely at its interface for control-oriented tools. We use a formal model of our implementation language, based on the model of an algebraic stepper presented by Clements et al. [25]. The language is a simple version of Scheme that includes a mechanism for control-flow introspection. This mechanism, based on the manipulation of *continuation marks*, is essentially a formal model of the stack-inspection capabilities provided by languages like Java and C#. It allows a programmer to associate data with each activation record on the runtime stack, just as the Java runtime associates source location information with each of its stack frames. Due to space limitations, we omit the specification of the implementation language from the paper.

In the formal model, the analog of a "stack" is an evaluation context, where each stack frame corresponds to one application of a production rule in the context's grammar. This gives rise to a linear structure, as in a stack, because each production (except for the empty terminal context "[ ]") contains exactly one sub-context (representing the next stack frame).

As mentioned in Section 2, DrScheme's [42] control-sensitive tools employ two main elements: (1) reading and interpreting the continuation marks from the running program, and (2) *annotating* the program prior to execution with instructions that install the necessary marks. Fortunately, we can also model the annotation process formally. For example,

Clements et al. present an annotator for their algebraic stepper, and the one we use for FrTime is a straightforward extension of theirs that handles a slightly richer implementation language. The key property of the annotator is that it preserves semantics while supporting reflection on the program's control flow. We write $A[\![\langle \text{EXPR} \rangle]\!]$ to indicate the application of such debugging annotations to the expression $\langle \text{EXPR} \rangle$, the result being a semantically equivalent expression save that, at any point in its evaluation, a tool can extract its continuation marks and reconstruct its evaluation context.

Another way to think of this annotator is as a compiler with a "debug" flag enabled. It generates code with extra information that tools like debuggers can use to elucidate the program's runtime behavior. Although our presentation is based on annotation and continuation marks, the specific mechanism is not essential; Pettyjohn et al. [82] show how to use Java's exceptions and a suitable compiler to obtain the same basic functionality.

Now that we have a formal model of the implementation language, we can *formally* express the deep embedding of FrTime. Specifically, we can implement FrTime's primitives (e.g., *make-signal*, **send-event**, *value-now*, etc.) and dataflow engine as Scheme procedures[1]. A FrTime program, then, is simply a Scheme program evaluated in a context in which all of these elements are defined. We write $\text{FRTIME}[\langle \text{EXPR} \rangle]$ to denote the result of filling a FrTime context with a program. The result is a Scheme expression that, when evaluated under Scheme's semantics, implements the behavior that the original $\langle \text{EXPR} \rangle$ would exhibit under the FrTime semantics shown in Figures 3.1 through 3.5.

This formalism allows us to capture precisely the problem with the naïve error-reporting in the FrTime implementation described in the previous section. For one thing, Scheme's error-tracer, by default, annotates the whole program, yielding $A[\![\text{FRTIME}[\langle \text{EXPR} \rangle]]\!]$. This explains why error traces contain references to the implementation. What we want instead is to evaluate $\text{FRTIME}\ [A[\![\langle \text{EXPR} \rangle]\!]]$, so only contextual information from the user's program appears in the traces. However, this is only half of the problem, since the FrTime semantics demands an error-trace containing more than local information (it composes additional context from the signal whose update threw the exception). Our original (buggy) implementation neglects this requirement, with the consequence that error reports fail to identify the responsible fragments of the user's program.

[1]The implementation is included with the DrScheme distribution.

7.3.3 Effective Evaluation Contexts

To fix this problem, the implementation needs to be able to capture and store a representation of its evaluation context. Fortunately, this is exactly what our control-flow reflection mechanism allows us to do. In particular:

1. We first need a variable to simulate the E element in the low-level semantics—the additional context associated with the signal being recomputed. We call this variable *e-e-c*, as it holds the *effective evaluation context*. Its value is initially empty, representing the empty context, since top-level evaluation does not occur on behalf of any particular signal.

2. Within the implementation of the *make-signal* function, we capture the *current* evaluation context, compose it with the effective evaluation context, and store it with the signal. This corresponds exactly to the behavior of MAKE-SIGNAL in the semantics; it saves the full program context that is responsible for the resulting signal.

3. Inside the main loop of the dataflow engine, before recomputing a signal, we set the effective evaluation context to the context saved from the signal's creation. This corresponds to the operation of the UPDATE-INT and ERROR rules in the high-level semantics: when they invoke the low-level evaluator on a signal's update procedure, they provide that signal's additional context as E.

4. When the top-level error-handler catches an error and reports it to the user, we make it build a trace by composing the effective and actual evaluation contexts. This mimics the behavior of the rule ERROR, which detects low-level evaluation errors and reports them at the higher level with contexts enriched by the responsible signals.

7.3.4 Generalizing the Solution Strategy

We have gone into a lot of detail about a specific application of effective evaluation contexts—to make an error-tracing tool for Scheme work for a particular embedded DSL. We claim, however, that the ideas we have developed are not specific to FrTime or the error-tracing tool. In fact, to apply them to a different deep DSL embedding, we just need to replace

signal with the new language's semantic data structure and *update* with its notion of interpretation.

For example, suppose we wish to apply the same solution to fix the error-tracing in Slideshow:

1. Just as for FrTime, we need to extend our tool interface with an *e-e-c* variable. However, if we have already made this extension for another language, we don't need to do it again.

2. In the constructor for the *pict* data structure (Slideshow's semantic data structure), we capture and save the current evaluation context.

3. In the slide navigator, around the interpreter that renders each *pict*, we extract the context associated with the *pict* to be drawn and copy it into the *e-e-c*.

4. We make the top-level error-handler compose the contents of the *e-e-c* with the local context. That way, if an error occurs while rendering a slide, the trace will automatically include the code that created the faulty *pict*. This change, like the addition of the *e-e-c* variable, is only necessary if this is the first time we are adapting the tool.

If we wish to reuse a different tool, then we must similarly extend it with an awareness of effective evaluation contexts. For example, DrScheme's profiler uses an annotator that inserts calls to read the CPU time before and after each procedure application, counting the elapsed time against the actual procedure (in its evaluation context). This default behavior is not very useful for deep DSL embeddings, since it ends up telling us the obvious—that the interpreter consumes the majority of the processing time. To get more meaningful measurements, we should attribute the time spent in the interpreter toward *the context that created the interpreted data*. Fortunately, this information is readily available: in modifying our languages to work meaningfully with the error-trace tool, we made the interpreter copy the context from the interpreted datum's creation to the *e-e-c* variable. The profiler only needs to be aware of the contents of this variable and adjust its accounting accordingly.

% Time	Msec	Calls	Function
31.765	58405	4933	inner
30.565	56199	378610	loop
18.572	34149	373647	<< unknown >>
4.582	8425	1576095	/home/greg/v208/plt/collects/frtime/frp.ss: 711.17

Figure 7.4: Output from original profiler on FrTime program

% Time	Msec	Calls	Function
38.446	216827	13451187	/home/greg/v208/plt/collects/frtime/demos/piston-b.ss: 30.5
10.564	59583	5647678	/home/greg/v208/plt/collects/frtime/demos/piston-b.ss: 45.5
21.623	121949	5647676	/home/greg/v208/plt/collects/frtime/demos/piston-b.ss: 47.5
10.477	59092	3080557	/home/greg/v208/plt/collects/frtime/demos/piston-b.ss: 26.5

Figure 7.5: Output from adapted profiler on FrTime program

7.3.5 Transformation of the Semantic Data Structure

Sometimes, a semantic data structure does not feed directly into a DSL interpreter, but instead first undergoes a number of transformations. For example, the language may convert it to a more efficient intermediate representation, annotate it with the results of various static analyses, and apply a number of optimizations to it before finally running it. So, the value that finally flows into the rendering engine is several stages removed from the original program and its corresponding effective evaluation context. Without some additional book-keeping, the system loses its connection to the original DSL program.

Effective evaluation contexts can help us with this problem as well. The idea is to instrument each of the transformation steps so that it copies the effective context from its input to its output. This way, the final representation—the one that the interpreter sees—maps to the same effective evaluation context as the raw data structure from which it is derived. This gives the interpreter access to the effective evaluation context that it would have if it interpreted the original data structure directly, without the intervening transformations.

7.4 Implementation Status

I have implemented the strategy described above for both FrTime and Slideshow, in conjunction with DrScheme's error-tracer and profiler, and the modifications have improved the quality of the feedback. In each case, the implementation requires less than thirty lines of code. For the languages, the bulk of this involves instrumenting the semantic data structure to capture and store the evaluation context upon construction. For the tools, the only

extension is to read and take into account the additional information stored in the effective evaluation context.

I show the impact of these modifications on the profiler. Figure 7.4 shows the original profiler output from a FrTime program. In this case, the tool's statistics come from actual evaluation contexts. The expressions *inner* and *loop* are the main loops inside the FrTime dataflow engine. It is neither interesting nor informative that they perform the bulk of the computation in the system. On the other hand, Figure 7.5 shows the profiler output for the same program after modifying it to use effective evaluation contexts. Here, the top contributors are those expressions in the user's program that construct signals for which recomputation is expensive.

Chapter 8

Extended Application: Scriptable Debugging

Debugging is a laborious part of the software development process.[1] Indeed, even with the growing sophistication of visual programming environments, the underlying debugging tools remain fairly primitive.

Debugging is a complex activity because there is often a good deal of knowledge about a program that is not explicitly represented in its execution. For instance, imagine a programmer trying to debug a large data structure that appears not to satisfy an invariant. He might set a breakpoint, examine a value, compare it against some others and, not finding a problem, resume execution, perhaps repeating this process dozens of times. This is both time-consuming and dull; furthermore, a momentary lapse of concentration may cause him to miss the bug entirely.

The heart of automated software engineering lies in identifying such repetitive human activities during software construction and applying computational power to ameliorate them. For debuggers, one effective way of eliminating repetition is to make them *scriptable*, so users can capture common patterns and reuse them in the future. The problem then becomes one of designing effective languages for scripting debuggers.

[1] This chapter expands on previously published joint work [63, 64] with Guillaume Marceau, Jonathan P. Spiro, and Steven P. Reiss. Guillaume deserves credit for the original idea of using a functional reactive language to script a debugger.

Debugging scripts must easily capture the programmer's intent and simplify the burdensome aspects of the activity. To do this, they must meet several criteria. First, they must match the temporal, event-oriented view that programmers have of the debugging process. Second, they must be powerful enough to interact with and monitor a program's execution. Third, they should be written in a language that is sufficiently expressive that the act of scripting does not become onerous. Finally, the scripting language must be practical: users should, for instance, be able to construct *program-specific* methods of analyzing and comprehending data. For example, users should be able to create redundant models of the program's desired execution that can be compared with the actual execution. This calls for a library of I/O and other primitives more commonly found in general-purpose languages than in typical domain-specific languages.

In this paper, we present the design and implementation of an interactive scriptable debugger called MzTake (pronounced "miz-take"). Predictably, our debugger can pause and resume execution, and query the values of variables. More interestingly, developers can write scripts that automate debugging tasks, even in the midst of an interactive session. These scripts are written in a highly expressive language with a dataflow evaluation semantics, which is a natural fit for processing the events that occur during the execution of a program. In addition, the language has access to a large collection of practical libraries, and evaluates in an interactive programming environment, DrScheme.

8.1 A Motivating Example

Figure 8.1 shows a Java transcription of Dijkstra's algorithm, as presented in *Introduction to Algorithms* [27]. Recall that Dijkstra's algorithm computes the shortest path from a source node to all the other nodes in a graph. It is similar to breadth-first search, except that it enqueues the nodes according to the total *distance* necessary to reach them, rather than by the number of *steps*. The length of the shortest path to a node (so far) is stored in the *weight* field, which is initialized to the floating point infinity. The algorithm relies on the fact that the shortest-path estimate for the node with the smallest weight is provably optimal. Accordingly, the algorithm removes that node from the pool (via *extractMin*), then uses this optimal path to improve the shortest path estimate of adjacent nodes (via *relax*). The algorithm makes use of a priority queue, which we also implemented.

Figure 8.2 shows a concrete input graph (where S, at location $\langle 100, 125 \rangle$, denotes the source from which we want to compute distances) and the output that results from executing this algorithm on that graph. The output is a set of nodes for which the algorithm was able to compute a shortest path. For each node, the output presents the node's number, its coordinates, and its distance from the source along the shortest path.

As we can see, this output is incorrect. The algorithm fails to provide outputs for the nodes numbered 4, 5 and 6, even though the graph is clearly connected, so these are a finite distance from S.

Since the implementation of Dijkstra's algorithm is a direct transcription from the text (as a visual comparison confirms), but *we* implemented the priority queue, we might initially focus our attention on the latter. Since checking the overall correctness of the priority queue might be costly and difficult, we might first try to verify a partial correctness criterion. Specifically, if we call *extractMin* to remove two elements in succession, with no insertions in-between, the second element should be at least as large as the first.

Unfortunately, most existing debuggers make it difficult to automate the checking of such properties, by requiring careful coordination between breakpoint handlers. For example, in `gdb` [90] we can attach conditional breakpoint handlers—which are effectively callbacks—to breakpoints on *insert* and *extractMin*, and so observe values as they enter and leave the queue. Figure 8.3 illustrates the control flow relationship between the target and the debugging script when we use callbacks to handle events. Starting at the top left, the target program runs for a while until it reaches the *extractMin* function; control then shifts to the debugger, which invokes the callback. The callback makes a decision to either pause or resume the target. Eventually, the target continues and runs until it reaches the breakpoint on the *extractMin* function for a second time. If we are monitoring a temporal property, such as the ordering of elements taken out of a priority queue, the decision to pause or resume the target on the second interruption will depend on data from the first callback invocation. Observe that, for the program on the left, it is natural to communicate data between the parts of execution, because it consists of one single thread of control. In contrast, the "program" on the right is broken up into many disjoint callback invocations, so we need to use mutable shared variables or other external channels to communicate data from one invocation to the next.

All this is simply to check for pairs of values. Ideally, we want to go much further than

simply checking pairs. In fact, we often want to create a redundant model of the execution, such as mirroring the queue's intended behavior, and write predicates that check the program against this model. Upon discovering a discrepancy, we might want to interactively explore the cause of failure. Moreover, we might find it valuable to abstract over these models and predicates, both to debug similar errors later and to build more sophisticated models and predicates as the program grows in complexity.

In principle, this is what scriptable debugging should accomplish well. Unfortunately, this appears to be difficult for existing scriptable debuggers. For example, Coca [35] offers a rich predicate language for identifying interesting data and points in the execution, but it does not offer a facility for relating values across different points in time, so the programmer would still need to monitor this criterion manually. UFO [6] supports computation over event-streams, but does not support interaction. Dalek [76] is interactive and offers the ability to relate execution across time, but provides limited abstractions capabilities, so we could not use it to build the predicates described in this paper. In general, existing scriptable debuggers appear to be insufficient for our needs; we discuss them in more detail in section 9.

This chapter presents a new system that addresses the weaknesses found in existing debuggers. In section 8.2, we describe the goals and observations that have guided our work. We reflect on lessons learned from this example in section 8.5. In Section 8.6 and Section 8.7, we describes the design and the implementation, respectively. Section 8.8 discusses strategies to control the execution of a target program. Section 8.9 provides additional, illustrative examples of the debugger's use.

8.2 Desiderata

We believe that users fundamentally view debugging as a temporal activity with the running program generating a stream of events (entering and exiting methods, setting values, and so on). They use constructs such as breakpoints to make these events manifest and to gain control of execution, at which point they can inspect and set values before again relinquishing control to the target program. To be maximally useful and minimally intrusive, a scriptable debugger should view the debugging process just as users do, but make it easy to automate tedious activities.

Concretely, the scripting language must satisfy several important design goals.

1. While debuggers offer some set of built-in commands, *users often need to define problem-specific commands*. In the preceding example, we wanted to check the order of elements extracted from a queue; for other programs, we can imagine commands such as "verify that this tree is balanced". While obviously a debugger should not offer commands customized to specific programs, it should provide a powerful enough language for programmers to capture these operations easily. Doing so often requires a rich set of primitives that can model sophisticated data, for instance to track the invariants of a program's data.

2. Programs often contain implicit invariants. Validating these invariants requires maintaining auxiliary data structures strictly for the purpose of monitoring and debugging. In our example, although Dijkstra's algorithm depends on nodes being visited in order of weight, there is no data structure in the program that completely captures the ordered list of nodes (a priority heap satisfies only a weaker ordering relation). Lacking a good debugging framework, the developer who wants to monitor monotonicity therefore needs to introduce explicit data structures into the source. These data structures may change the space- and time-complexity of the program, so they must be disabled during normal execution. All these demands complicate maintenance and program comprehension. Ideally, *a debugger should support the representation of such invariants outside the program's source*. (In related work, we explain why approaches like contracts and aspects [5] are insufficient.)

3. Debugging is often a process of generating and falsifying hypotheses. *Programmers must therefore have a convenient way to generate new hypotheses while running a program.* Any technique that throws away the entire debugging context between each attempt is disruptive to this exploratory process.

4. Since the target program is a source of events and debugging is an event-oriented activity, *the scripting language must be designed to act as a recipient of events*. In contrast, traditional programming languages are designed for writing programs that are "in control"—i.e., they determine the primary flow of execution, and they

provide cumbersome frameworks for processing events. This poses a challenge for programming language design.

5. As a pragmatic matter, *debuggers should have convenient access to the rich* I/O *facilities provided by modern consoles* so they can, for instance, implement problem-specific interfaces. A custom language that focused solely on the debugging domain would invariably provide only limited support for such activities. In contrast, the existence of rich programming libraries is important for the widespread adoption of a debugging language.

To accomplish these goals, a debugging language must address a conflict central to all language design: balancing the provision of powerful abstractions with restrictions that enable efficient processing. This has been a dominant theme in the prior work (see section 9). Most prior solutions have tended toward the latter, while this paper begins with a general-purpose language, so as to explore the space of expression more thoroughly. This results in some loss of machine-level efficiency, but may greatly compensate for it by saving users' time. Furthermore, the functional style we adopt creates opportunities for many traditional compiler optimizations.

8.3 Language Design Concerns

FrTime supports the development of a scriptable debugger in several ways. Firstly, the rich libraries of DrScheme are available for FrTime, and are automatically lifted to the time domain, so they recompute when their arguments update. Secondly, the DrScheme prompt recognizes behaviors and automatically updates the display of their values as they change over time. Finally, FrTime upholds a number of guarantees about a program's execution, including the order in which it processes events and the space required to do so:

- **Ordering of event processing**: Since FrTime must listen to multiple concurrent event sources and recompute various signals in response, we might worry about the possibility of timing and synchronization issues. For example, if signal *a* depends on signal *b*, we would like to know that FrTime will not recompute *a* using an out-of-date value from *b*. Fortunately, FrTime's recomputation algorithm is aware

of dataflow dependencies between signals and updates them in a topological order, starting from the primitive signals and working towards their dependents.

- **Space consumption**: FrTime only remembers the current values of behaviors and the most recent occurrences of events. Thus, if the program's data structures are bounded, then the program can run indefinitely without exhausting memory. If the application needs to maintain histories of particular event streams, it can use FrTime primitives like *history-e* or *accum-b* for this purpose. The application writer must apply these operations explicitly and should therefore be aware of their cost.

8.4 Debugging the Motivating Example

We are now ready to return to our example from section 8.1. As we explained previously, our implementation of Dijkstra's algorithm employs a priority queue coded by us. In addition, we noted that our implementation of *DijkstraSolver* is a direct transcription of the pseudocode in the book. We hypothesized that the bug might be in the implementation of the priority queue, and that we should therefore monitor its behavior. Recall that the partial correctness property we wanted to verify was that consecutive pairs of elements extracted from the queue are in non-decreasing order.

Figure 8.4 presents a debugging script that detects violations of this property. In the script, the variable *c* is bound to a debugging session for *DijkstraTest*, a class that exercises the implementation of Dijkstra's algorithm. The invocation of *start-vm* initiates the execution of the Java Virtual Machine (JVM) on this class, and immediately suspends its execution pending further instruction.

The expression (*jclass c PriorityQueue*) creates a FrTime proxy for the *PriorityQueue* class in Java. Since Java dynamically loads classes on demand, this proxy is a time varying value: its value is $\perp$ at first, and stays so until the class is loaded into the JVM. The operator *jclass* treats its second argument specially: *PriorityQueue* is not a variable reference, but simply the name of the target class. In Lisp terminology, *jclass* is a *special form*.

Next, we install tracing around the methods *add* and *extractMin* of the priority queue. A *tracepoint* is a FrTime event-stream specifically designed for debugging: the stream contains a new value every time the Java program's execution reaches the location marked

by the tracepoint. Concretely, the expression

```
(define inserts
  (trace (queue.add entry)
         (bind (item) item.weight)))
```

installs a tracepoint at the entry of the *add* method of *queue*.[2] The result of **trace** is an event stream of values. There is an event on the stream each time the target program reaches the *add* method. To generate the values in the stream, the **trace** construct evaluates its body; this body is re-evaluated for each event. In this instance, we use the *bind* construct to reach into the stack of the target, find the value of the variable *item* (in the target), and bind it to the identifier *item* (in the body of the *bind*). In turn, the body of the *bind* extracts the *weight* field from this item. This weight becomes the value of the event.

The identifier *inserts* is therefore bound to a FrTime event-stream consisting of the weights of all nodes inserted into the priority queue. The identifier *removes* is bound correspondingly to the weights of nodes removed from the queue by *extractMin*.

We initially want to perform a lightweight check that determines whether consecutive *remove*s (not separated by an *insert*) are non-decreasing. To do this, we merge the two event-streams, *inserts* and *removes*. Since we are only interested in consecutive, uninterrupted removals, the monitor resets upon each insertion. The following FrTime code uses the combinator -=> to map the values in the *inserts* stream to the constant 'reset, which indicates that the monitor should reset:

```
(merge-e removes (inserts . -=> . 'reset))
```

The result of this expression is illustrated in Figure 8.5. In this graph, time flows towards the right, so earlier events appear to the left. Each circle represents one event occurrence on the corresponding stream. The first three lines show the streams we just discussed: *inserts*, *removes*, and the mapped *inserts*. The fourth timeline of the figure shows that the *merge-e* expression evaluates to an event-stream whose events are in the order they are encountered during the run. The insert events have been mapped to the constant, while the remove events are represented by the weight of the node.

The last two timelines in Figure 8.5 depict the next two streams created by the script.

[2] Here and in the rest of this paper, we use the infix notation supported by FrTime: (*x . op . y*) is the same as (*op x y*) in traditional Lisp syntax.

The merged stream is passed to the core monitoring primitive, *not-in-order*, shown in Figure 8.6. This uses *history-e* to extract the two most recent values from the stream and processes each pair in turn. It filters out those pairs that do not exhibit erroneous behavior, namely when one of the events is a 'reset or when both events reflect extracted weights that are in the right order. The result is a stream consisting of pairs of weights where the weightier node is extracted first, violating the desired order. We call this stream *violations*.

The FrTime identifier *latest-violation* is bound to a behavior that captures the last violation (using the FrTime combinator *hold*). If the priority queue works properly, this behavior will retain its initial value, **false** (meaning "no violation so far"). If it ever changes, we want to pause the JVM so that we can examine the context of the violation. To do this, we use the primitive *set-running-e!*, which consumes a stream of boolean values. Calling *set-running-e!* launches the execution of the target program proper, and it will keep on consuming future events on the given stream: when an event with the value **false** occurs the JVM pauses, after which, when an event with a true value occurs the JVM resumes.[3] Since we anticipate wanting to observe numerous violations, we define the (concisely named) abstraction *nv*, which tells the JVM to run until the **n**ext **v**iolation occurs.

At the interactive prompt, we type (*nv*). Soon afterward, the JVM stops, and we query the value of *latest-violation*:

```
> (nv)
short pause
> latest-violation
(+inf.0  55.90169943749474)
```

This output indicates that the queue has yielded nodes whose weights are out of order. This confirms our suspicion that the problem somehow involves the priority queue.

Continuing Exploration Interactively

To identify the problem precisely, we need to refine our model of the priority queue. Specifically, we would like to monitor the queue's complete black-box behavior, which might provide insight into the actual error.

[3] In Scheme, any value other than **false** is true.

With the JVM paused, we enter the code in figure 8.7 to the running FrTime session. This code duplicates the priority queue's implementation using a sorted list. While slower, it provides redundancy by implementing the same data structure through an entirely different technique, which should help identify the true cause of the error.[4]

We now explain the code in figure 8.7. The identifier *model* is bound to a list that, at every instant, consists of the elements of the queue in sorted order. We decompose its definition to improve readability. The value *inserters* is an event-stream of FrTime procedures that insert the values added to the priority queue into the FrTime model (==> applies a given procedure to each value that occurs in an event-stream); similarly, *removers* is bound to a stream of procedures that remove values from the queue. The code

(*accum-b* (*merge-e inserters removers*)
(*convert-queue-to-list* (*bind* (*q*) *q*)))

merges the two streams of procedures using *merge-e*, and uses *accum-b* to apply the procedures to the initial value of the model. *accum-b* accumulates the result as it proceeds, resulting in an updated model that reflects the application of all the procedures in order. *accum-b* returns a behavior that reflects the model after each transformation. We must initialize the model to the current content of the queue. The user-defined procedure *convert-queue-to-list* (elided here for brevity) converts *q*'s internal representation to a list.

Having installed this code and initialized the model, we resume execution with *nv*. At the next violation, we interactively apply operations to compare the queue's content against its FrTime model (the list). We find that the queue's elements are not in sorted order while those in the model are. More revealingly, the queue's elements are not the same as those in the model. A little further study shows that the bug is in our usage of the priority queue: we have failed to account for the fact that the assignment to *dest.weight* in *relax* (figure 8.1) *updates* the weights of nodes already in the queue. Because the queue is not sensitive to these updates, what it returns is no longer the smallest element in the queue. (Of course, these steps—of observing the discrepancy between the model and the phenomenon, then mapping it to actual understanding—require human ingenuity.)

On further reading, we trace the error to a subtle detail in the description of Dijkstra's algorithm in Cormen, et al.'s book [27, page 530]. The book permits the use of a binary

[4]Since the property we are monitoring depends only on the nodes' weights, not their identities, the model avoids potential ordering discrepancies between equally-weighted nodes.

heap (which is how we implemented the priority queue) for sparse graphs, but subsequently amends the pseudocode to say that the assignment to *dest.weight* must explicitly invoke a key-decrement operation. Our error, therefore, was not in the implementation of the heap, but in using the (faster) binary heap implementation without satisfying its (stronger) contract.

8.5 Reflections on the Example

While progressing through the example, we encounter several properties mentioned in the desiderata that make FrTime a good substrate for debugging. We review them here, point by point.

1. The DrScheme environment allows the user to keep and reuse abstractions across interactive sessions. For instance, to monitor the priority queue, we define procedures such as *not-in-order* and *convert-queue-to-list*. Such abstractions, which manipulate program data structures in a custom fashion, may be useful in finding and fixing similar bugs in the future. They can even become part of the program's distribution, assisting other users and developers. In general, debugging scripts can capture some of the *ontology* of the domain, which is embedded (but not always explicated) in the program.

2. We discover the bug by monitoring an invariant not explicitly represented in the program. Specifically, we keep a sorted list that mirrors the priority queue, and we observe that its behavior does not match the expectations of Dijkstra's algorithm. However, the list uses a linear time insertion procedure, which eliminates the performance benefit of the (logarithmic time) priority queue. Fortunately, by expressing this instrumentation as a debugging script, we cleanly separate it from the program's own code, and hence we incur the performance penalty only while debugging.

3. The interactive console of DrScheme, in which FrTime programs run, enables users to combine scripting with traditional interactive debugging. In the example, we first probe the priority queue at a coarse level, which narrows the scope of the bug. We then extend our script to monitor the queue in greater detail. This ability to explore

interactively saves the programmer from having to restart the program and manually recreate the conditions of the error.

4. The dataflow semantics of FrTime makes it well suited to act as a recipient of events and to keep models in a consistent state, even as the script is growing. During the execution of the Dijkstra solver, FrTime automatically propagates information from the variables *inserts* and *removes* to their dependents, the *violations* variable and the *set-running-e!* directive. Also, when we add the variable *model*, FrTime keeps it synchronized with *violations* without any change to the previous code.

5. The libraries of FrTime are rich enough to communicate with external entities. The programmer also has access to the programming constructs of DrScheme (higher-order functions, objects, modules, pattern-matching, etc.), which have rigorously defined semantics, in contrast to the ad-hoc constructs that populate many scripting languages. Further, since FrTime has access to all the libraries in DrScheme [42], it can generate visual displays and so on, as we will see in section 8.9.1.

8.6 Design

The design of MzTake contains four conceptual layers that arise naturally as a consequence of the goals set forth in the desiderata (Section 8.2).

First, we need abstractions that capture the essential functionality of a debugger. These are: observing a program's state, monitoring its control path, and controlling its execution. MzTake captures them as follows: *bind* retrieves values of program variables, **trace** installs trace points, and *set-running-e!* lets the user specify an event stream that starts and stops the program.

Second, we need a way to navigate the runtime data structures of the target program. For a Java debugger, this means providing a mechanism for enumerating fields and looking up their values.

Third, and most importantly, we need to be able to write scripts that serve as passive agents. Most general-purpose languages are designed to enable writing programs that control the world, starting with a "main" that controls the order of execution. In contrast, a debugging script has no "main": it cannot anticipate what events will happen in what order,

and must instead faithfully follow the order of the target program's execution. Therefore we believe that a semantic distance between the scripting language and the kind of target language we are addressing is a necessary part of the solution.[5] Since the script's execution must be driven by the arrival of values from the program under observation, a dataflow language is a natural choice.

Once we have chosen a dataflow evaluation semantics, we must consider how broad the language must be. It is tempting to create a domain-specific debugging language that offers only a small number of primitives, such as those we have introduced here. Unfortunately, once the script has gained control, it may need to perform arbitrary computational tasks, access libraries for input/output, and so forth. This constant growth of tasks makes it impractical to build and constantly extend this domain-specific language, and furthermore it calls into question the strategy of restricting it in the first place. In our work, we therefore avoid the domain-specific strategy, though we have tried to identify the essential elements of such a language as a guide to future language designers.

Having chosen a general-purpose strategy, we must still identify the right dataflow language. Our choice in this paper is informed by one more constraint imposed by debugging: the need to extend and modify the dataflow computation interactively without interrupting execution. Among dataflow languages, this form of dynamicity appears to be unique to FrTime.

We present the grammar of the MzTake language in Figure 8.8. The grammar is presented in layers, to mirror the above discussion. The first layer, represented by <*debug-expr*>, presents the most essential language primitives. The second layer, consisting of <*inspect-expr*> and <*loc-expr*>, represents primitives for obtaining information about the target program. The third layer describes the FrTime language.

8.7 Implementation

The examples we have seen so far describe a debugger for Java programs. However, the same principles of scriptable debugging should apply to most control-driven, call-by-value

[5] Our work additionally introduces a *syntactic* difference when the target language is Java, but this can be papered over by a preprocessor.

programming languages, with changes to take into account the syntactic and semantic peculiarities of each targeted language. To investigate the reusability of our ideas, we have implemented a version of MzTake for Scheme [55] also.

Not surprisingly, both the Java and Scheme versions share the design of the debugging constructs **trace**, *bind*, and *set-running-e!*. They differ in the operators they provide for accessing values in the language: because FrTime's data model is closer to Scheme's than to Java's, the Java version of the debugger requires a *jdot* operator to dereference values, but the Scheme version does not need the equivalent. Furthermore, because Java (mostly) names every major syntactic entity (such as classes and methods) whereas Scheme permits most values to be anonymous, the two flavors differ in the way they specify syntactic locations.

8.7.1 Java

The overall architecture of the Java debugger is shown in Figure 8.9.

On the left, we have the target Java program running on top of the virtual machine. The Java standard provides a language-independent debugging protocol called the Java Debug Wire Protocol (JDWP), designed to enable the construction of out-of-process debuggers. We have adapted a JDWP client implementation in Ruby [1] to DrScheme by compiling its machine-readable description of JDWP packets. We use this implementation to connect to the virtual machine over TCP/IP.

On the right of the figure, we have the stack of programming languages that we used to implement the debugger. FrTime is implemented on top of DrScheme, the debugging language is implemented on top of FrTime, and debugging scripts are themselves implemented in the debugging language.

The communication between the low-level debugger and the script proceeds in three stages. The first stage translates JDWP packets to a callback interface, the second dispatches these callbacks to their respective tracepoints, and the third translates them to FrTime event occurrences.

The second of these stages must handle subtleties introduced because the JDWP does not provide guarantees about the order in which messages arrive. For example, the following is a legal but troublesome sequence of messages. First, MzTake sends a message requesting a

new tracepoint B. While MzTake waits for a reply, the target program reaches an existing tracepoint, A, generating an event that appears on the port before the virtual machine's reply to the request to install B. MzTake must either queue the trace at A while awaiting the acknowledgment of B or dispatch the A trace concurrently; it does the latter.

A trickier situation arises when a trace event at B appears even before the acknowledgment of installing that tracepoint. This is problematic because every trace event is tagged with a label that identifies which tracepoint generated it. This label is generated by the JDWP and communicated in the tracepoint installation acknowledgment. Therefore, until MzTake receives this acknowledgement, it cannot correctly interpret trace events labeled with a new tag. In this case, MzTake is forced to queue these events, and revisits the queue upon receipt of an acknowledgment.

We also need to translate the event callbacks into FrTime's event streams. Each usage of **trace** becomes associated with a callback. When the target reaches the traced location, its callback evaluates the **trace** expression's body and adds the result to FrTime's queue of pending events. It then posts on a semaphore to awaken the FrTime evaluator thread and waits. The event's value automatically propagates to all expressions that refer to the **trace** statement, directly or indirectly, in accordance with FrTime's dataflow semantics. When the FrTime dataflow graph reaches quiescence, the evaluator posts on a semaphore, which releases the callback thread and subsequently resumes the Java process. This provides synchronization between the debugging script and the debugged program. If the Java target program uses multiple threads, MzTake handles each event in a stop-the-world manner, to ensure that the script observes a consistent view of the program's state.

We found that the JDWP provides most of the functionality needed to build a scriptable debugger. Beyond implementing the packets and the dispatching as we mentioned above, we also needed to write two more components. The first was to duplicate Java's scoping rules in the implementation of *bind*: looking up *x* at a location first finds a local variable, if any, otherwise the field named *x* in the enclosing class, then in the super class, and so on. The second was to cache the results of JDWP queries pertaining to the fields of classes and the line numbers of methods, and flush the cache whenever the cached value might be invalidated; this is necessary to achieve both quick startup and acceptable runtime performance.

There are some other debugging events and inspection functions available in MzTake

that we mentioned very briefly, or not at all, during the example. These include facilities for traversing the stack, enumerating local variables, and so on. There are also other events and functionality available through the JDWP that are not accessible in the debugger, such as class-loading and unloading events, static initializers, etc. What we have described so far is a conservative minimal extension of the programming language FrTime; it is easy to continue in the same vein to include support for the remaining events.

The inspection functions we provide pertain only to the data present in the target. We might like to reflect on the program's syntactic structure as well, for example to trace all assignments to a variable or all conditional statements. However, the JDWP does not provide support for such inspection, so we would need to build it on our own. In a sense, such capabilities are orthogonal to our work, since dataflow offers no new insight on processing of static syntax trees.

The quality of the JDWP implementation varied across virtual machines, and many versions were prone to crashes; we tested against the Sun JVM, the IBM JVM, and the Blackdown JVM, ultimately settling on the Sun implementation.

8.7.2 Scheme

The Scheme version employs source annotation. We instrument the Scheme program so that it mirrors the functionality of a process under the control of a debugger. The annotation mirrors the content of the lexical environment and introduces a procedure that determines when to invoke the debugger.

For example, suppose the original target program contains the expression

(**define** (*id x*) *x*)
(*id* 10)

The output of the annotator would be (approximately)

```
(define env empty)
(define (id x)
  (set! env (cons (list "x" x) env))
  (invoke-debugger 1 15 env)
  (begin0  ;; perform steps in order, then return value of the first expression
    x
    (set! env (rest env))))
(invoke-debugger 2 1 env)
(id 10)
```

When the annotated version executes, the *env* variable recreates the lexical environment. In particular, it tracks the *names* of variables in conjunction with their values, enabling inspection. The *invoke-debugger* procedure receives source location information (e.g., the arguments 2 and 1 refer to line two, column one). Each invocation of the procedure tests whether a tracepoint has been installed at that location and accordingly generates an event.

There are several important details glossed over by this simplified notion of annotation. We discuss each in turn:

thread-safety This annotation uses a mutable global variable for the environment. The actual implementation instead uses thread-local store.

tail-calls This annotation modifies the environment at the end of the procedure, thereby destroying tail-call behavior. The actual implementation uses *continuation-marks* [24], which are specifically designed to preserve tail-calls in annotations.

communication This annotation appears to invoke a procedure named *invoke-debugger* that resides in the program's namespace. Because FrTime runs atop the DrScheme virtual machine, the target Scheme program and the MzTake debugging environment share a common heap. Therefore, the annotation actually introduces a reference to the *value* of the debugging procedure, instead of referring to it by name.

The procedure *invoke-debugger* generates a FrTime event upon reaching a tracepoint, and then waits on a semaphore. From there, the evaluation of the script proceeds as in the Java case, since both implementations share the same FrTime evaluation engine. When the evaluation reaches quiescence, it releases the semaphore.

The implementation is available from

```
http://www.cs.brown.edu/research/plt/software/mztake/
```

8.7.3 Performance

We analyze the performance of the Dijkstra's algorithm monitor shown in figures 8.4 and 8.6. This example has a high breakpoint density (approximately 500 events per millisecond), so the time spent monitoring dominates the overall computation. In general, the impact of monitoring depends heavily on breakpoint density, and on the amount of processing performed by each breakpoint. All time measurements are for a 1.8GHz AMD Athlon XP processor running Sun's JVM version 1.4 for Linux.

We measure the running time of the the Dijkstra's algorithm monitor shown in figures 8.4 and 8.6, when it executes in the Java version of the debugger. Excluding the JVM startup time, it takes 3 minutes 42 seconds to monitor one million heap operations (either *add* or *extractMin*), which represents 2.217 milliseconds per operation. We partition this time into four parts: First, the virtual machine executes the call to either *add* or *extractMin* (0.002 milliseconds per operation). Second, the JDWP transmits the context information, FrTime decodes it, and FrTime schedules the recomputation (1.364 milliseconds per operation). Third, FrTime evaluates the script which monitors the partial correctness property, in figure 8.4 (0.851 milliseconds per operation).

According to these measurements, nearly one-third of the debugging time is devoted to JDWP encoding and decoding and to the context-switch. This is consistent with the penalty we might expect for using an out-of-process debugger. The time spent in FrTime can, of course, be arbitrary, depending on the complexity of the monitoring and debugging script.

In the Scheme implementation, the target and the debugger execute in the same process (while still preserving certain process-like abstractions [45]). As a result, whereas the Java implementation incurred a high context-switch cost, but no per-statement cost, the Scheme implementation incurs a small cost for each statement, but no operating system-level cost for switching contexts. Per operation, the annotation introduces a 0.126 milliseconds overhead. Thanks to the absence of a cross-process context-switch, dispatching an event costs 0.141 milliseconds per operation (compared with 1.3 milliseconds in the Java version of the debugger). The remaining times stay the same.

Obviously, MzTake is not yet efficient enough for intensive monitoring. A two millisecond response time is, however, negligible when using MzTake interactively.

8.8 Controlling Program Execution

Debuggers not only inspect a program's values, but sometimes also control its execution. Some of the abstractions we defined in our running example were of the former kind (*not-in-order*, *convert-queue-to-list*). In contrast, we also defined a custom-purpose rule for deciding when to execute and when to pause, namely the function *nv*.

These *start-stop policies* represent a general pattern of debugger use. These policies can differ in subtle but important ways, especially when the same line has several breakpoints, each with its own callback. The start-stop policy used by most scripted debuggers consists of running the callbacks in order of their creation, until one of them requests a pause. Once this happens, the remaining breakpoints on the same line are not executed at all.

One might wonder if this is the right rule for all applications. In particular, preventing the execution of the subsequent callbacks creates a dependency between breakpoints (if the first breakpoint decides to suspend the execution, the second does not get to run at all). These dependencies are problematic if these breakpoints monitor implicit invariants or implicit data structures, as we did during the example. During our debugging session, we created a mirror model of the queue so that it would elucidate the problem with the state of the real queue. In order to be of any debugging help, the model and the state must remain synchronized. If the event that detected the state violation prevented the execution of the event that updates the model, the program and model would cease to be synchronized. Worse, this would happen exactly when we need to look at the model, namely when we begin to explore the context of the violation.

By using a combination of first class events and *set-running-e!*, it is easy to define start-stop policies which are both custom-purpose and reusable. We implement the problematic start-stop policy just described with the code in figure 8.10. In the code, *breakpoints* is a hash table that maps locations to event streams. The *break* function sets or adds a breakpoint on a given line. The first time it is called on a given location, it installs a **trace** handler at that location, which simply sends the value **true** on the event stream each time the target program reaches that location. On subsequent invocations, it accumulates a cascade

of events where each event is subordinate to the event that was in that location previously. When the execution of the target program reaches one of the locations, the script invokes each callback function in the cascade until the first one that returns false. The condition (**if** *i* ...) ensures that the other callbacks are not called afterwards.

With MzTake, it is straightforward to define a different policy. Figure 8.11 shows the code for a break policy that executes all the breakpoints at one location before pausing the target program.

8.9 Additional Examples

In this section, we present some additional examples that further illustrate the power of our language.

8.9.1 Minimum Spanning Trees

Because MzTake has the full power of FrTime, users can take advantage of existing libraries to help them understand programs. For example, the FrTime animation library allows specification of time-varying images (i.e., image behaviors) that respond to events. Since MzTake generates events by tracing program execution, users can visualize program behavior by appropriately connecting these events to the animation library.

An intuitive visual representation can be an effective way of gaining insight into a program's (mis)behavior. Moreover, many programs lend themselves to natural visualizations. For example, we consider the problem of computing the Minimum Spanning Tree (MST) for a collection of points in the plane. (This example is based on the actual experience of one of the authors, in the context of writing a heuristic to solve the traveling-salesman problem.)

A simple greedy algorithm for the MST works by processing the edges in order of increasing length, taking each edge if and only if it does not introduce a cycle. Though the algorithm is straightforward, the programmer might forget to do something important, such as checking for cycles or first sorting the edges by length.

The programmer could write code to isolate the source of such errors, but a simple

visualization of the program's output is much more telling. In Figure 8.12, we show visualizations of three versions of an MST program. On the left, we show the correct MST, in the middle, an edge set computed without cycle detection, and on the right, what happens if we forget to sort the edges.

In Figure 8.13, we show the debugging script that implements this visualization. Its salient elements are:

tree-start-event occurs each time the program begins computing a new MST, yielding an empty edge list

tree-edge-event occurs each time the algorithm takes a new edge, adding the new edge to the list

tree builds a model of the tree by accumulating transformations from these event-streams, starting with an empty tree

display-lines displays the current tree

Though we have not shown the implementation of the MST algorithm, one important characteristic is that it does not maintain the set of edges it has taken: it only accumulates the *cost* of the edges and keeps track of which vertices are reachable from each other. In building an explicit model of the tree, our script highlights an important capability of our debugging system—it can capture information about the program's state that is not available from the program's own data structures. To implement the same functionality without a scriptable debugger, the user would need to amend the program to make it store this extra information.

8.9.2 A Statistical Profiler

Because our scripting language can easily monitor a program's execution, it should be relatively simple to construct a statistical profiler. Such a profiler uses a timer to periodically poll the program. Each time the timer discharges, the profiler records which procedure was executing and then re-starts the timer. The summary of this record provides an indication of the distribution of the program's execution time across the procedures.

MzTake provides a global time-varying value called *where*, which represents the current stack trace of the target process. It is a list of symbolic locations starting with the current line and ending with the location of the *main* function. The value of *where* is updated any time the execution of the target is suspended, either by **trace** or by *set-running-e!*.[6]

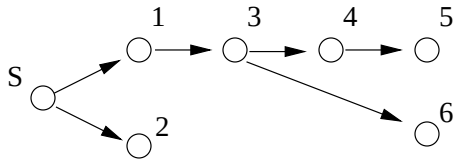
Figure 8.14 uses *where* to implement a statistical profiler that records the top two stack frames at each poll. First, we instantiate a hash table to map stack contexts to their count. Next, each time the *where* behavior changes, we capture the current context and pattern-match on it using *match-lambda*. If the context contains at least a line, a function, and a caller function, we trim the context down to the function name and its caller and increment the count in the hash table. Then we bind *ticks* to a stream that sends an event every 50 milliseconds. Finally, we use *set-running-e!* to suspend the target at each tick. We want to resume the target soon after a pause, but how soon is soon enough? We want to leave just enough time so that the evaluation engine correctly updates the hash table before resuming the target, but no more. Recall that *set-running-e!* synchronizes with the evaluation of the script, so that it waits until all dependencies are fully recomputed before consuming the next event on its input stream. With that in mind, we use *merge-e* to create a stream containing two nearly-simultaneous events: the **false** tick is followed by a **true** tick immediately afterwards. The synchronization ensures that *set-running-e!* will not consume the **true** tick until the data flow consequences of the **false** ticks are completely computed.

This code only gathers profiling information. The script needs to eventually report this information to the user. There are two options: to wait until the program terminates (which the debugger indicates using an event), or to report it periodically based on clock ticks or some other condition. (The latter is especially useful when profiling a reactive program that does not terminate.) Both of these are easy to implement using FrTime's time-sensitive constructs.

[6]We also have another behavior *where/ss* (for *where with single stepping*) which updates at every step of the execution. This is useful for scripts that want to process the entire trace of the target. However, *where/ss* is disabled by default, for performance reasons.

```
class DijkstraSolver {

  public HashMap backtrace = new HashMap();
  private PriorityQueue q = new PriorityQueue();

  public DijkstraSolver(DirectedGraph graph,
                        Node source) {
    source.weight = 0.0;
    q.addAll(graph.getNodes());

    while(!q.isEmpty()) {
      Node node = (Node)q.extractMin();
      List successors = graph.getSuccsOf(node);
      for(Iterator succIt = successors.iterator();
          succIt.hasNext(); )
        relax(node, (Node)succIt.next());
    }
    System.out.println("Result backtrace:\n" +
                       backtrace.keySet());
  }

  public void relax(Node origin, Node dest) {
    double candidateWeight =
      origin.weight + origin.distanceTo(dest);
    if (candidateWeight < dest.weight) {
      dest.weight = candidateWeight;
      backtrace.put(dest, origin);
    }
  }
}
```

Figure 8.1: Implementation of Dijkstra's Algorithm




```
Result backtrace:
[[node 1 : x 150 y 100 weight 55],
 [node 2 : x 150 y 150 weight 55],
 [node 3 : x 200 y 100 weight 105]]
```

Figure 8.2: Sample Input and Output

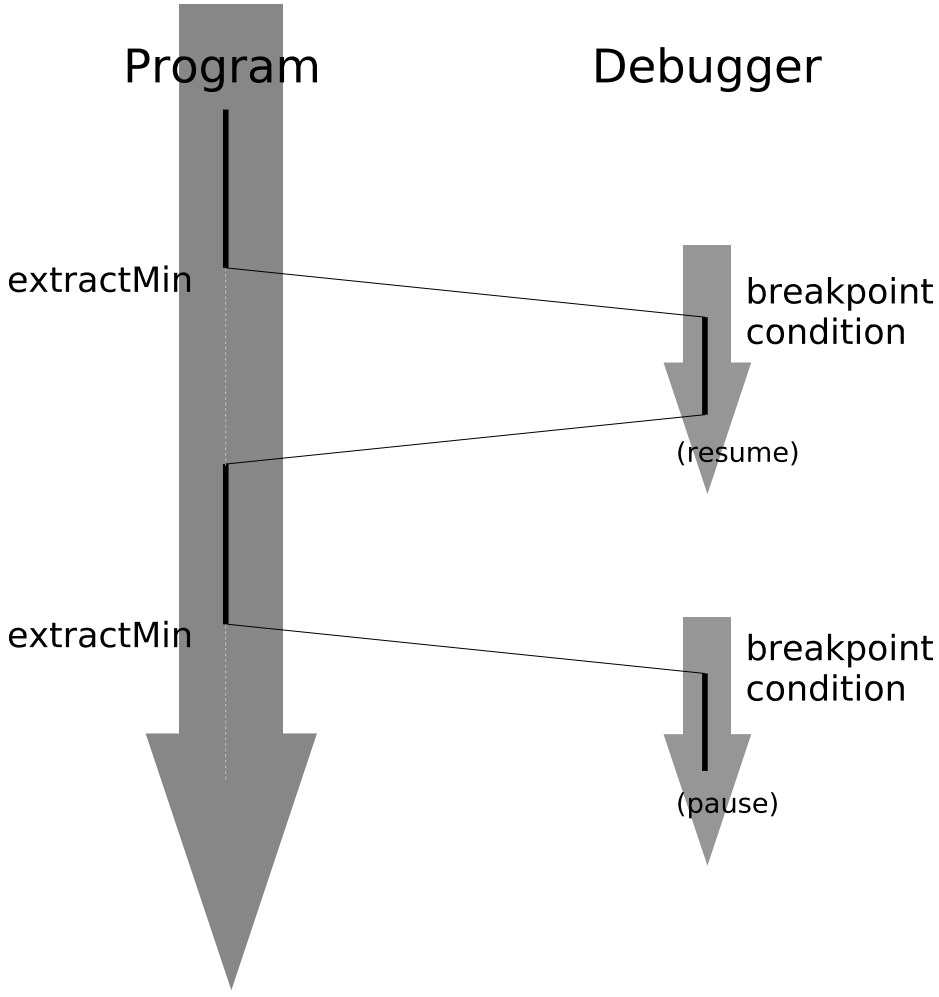


Figure 8.3: Control Flow of Program and Script

```
(define c (start-vm "DijkstraTest"))
(define queue (jclass c PriorityQueue))

(define inserts
  (trace (queue.add entry)
         (bind (item) item.weight)))
(define removes
  (trace (queue.extractMin exit)
         (bind (result) result.weight)))

(define violations
  (not-in-order (merge-e removes (inserts . -=> . 'reset))))
(define latest-violation (hold violations false))
(define (nv)
  (set-running-e! (violations . -=> . false)))
```

Figure 8.4: Monitoring the Priority Queue

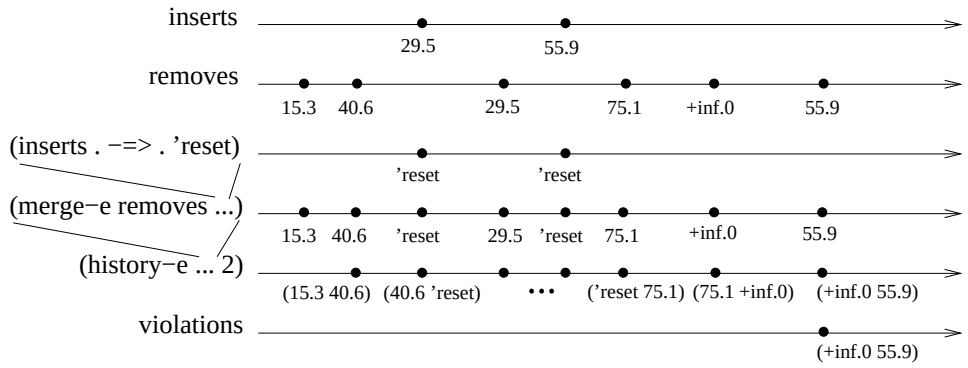


Figure 8.5: Event Streams

```
(define (not-in-order e)
  (filter-e
   (match-lambda
     ['reset _) false]
     [(_ 'reset) false]
     [(previous current) (> previous current)])
   (history-e e 2)))
```

Figure 8.6: The Monitoring Primitive

```
(define inserters
  (inserts . ==> . insert-in-model))
(define removers
  (removes . ==> . remove-from-model))

(define model
  (accum-b (merge-e inserters removers)
           (convert-queue-to-list (bind (q) q))))
```

Figure 8.7: The Redundant Model

```
<debug-expr>   ::=  (bind (<var> ...) <expr> ...)
                |   (trace <expr> <expr>)
                |   (set-running-e! <expr>)

<inspect-expr> ::=  (start-vm <expr>)
                |   (jclass <expr> <name>)

<loc-expr>     ::=  <number> | entry | exit

<frtime-expr>  ::=  (map-e <expr> <expr>)
                |   (filter-e <expr> <expr>)
                |   (merge-e <expr> ...)
                |   (accum-b <expr> <expr>)
                |   (changes <expr>)
                |   (hold <expr> <expr>)
                |   (value-now <expr>)
                |   seconds
                |   key-strokes
                |   (λ (<var> ...) <expr> ...)
                |   (<expr> ...)
                |   (if <expr> <expr> <expr>)
                |   ... ; other Scheme expressions

<expr>         ::=  <debug-expr>
                |   <inspect-expr>
                |   <frtime-expr>
```

Figure 8.8: MzTake Grammar

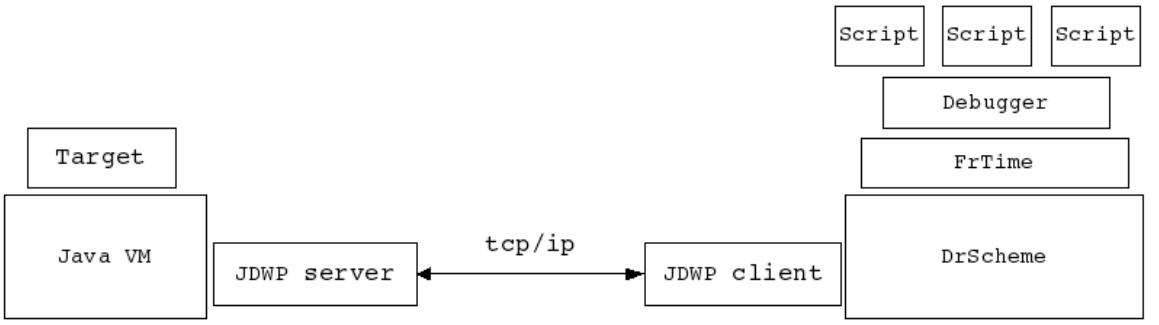


Figure 8.9: MzTake Architecture for Debugging Java

```
(define breakpoints (make-hash-table 'equal))

(define (break location callback)
  (let ([prev-breakpoint
         (if (hash-table-contains? breakpoints location)
             (hash-table-get breakpoints location)
             (trace location true))])

    (hash-table-put! breakpoints location
      (prev-breakpoint
       . ==> .
       (λ (i) (if i (callback) false))))))

(define (resume)
  (set-running-e!
    (apply merge-e (hash-table-values breakpoints))))
```

Figure 8.10: A Typical Start-Stop Policy

```
(define breakpoints empty)

(define (break location callback)
  (set! breakpoints
    (cons (trace location (callback))
          breakpoints)))

(define (resume)
  (set-running-e!
    (apply merge-e breakpoints)))
```

Figure 8.11: A Different Start-Stop Policy

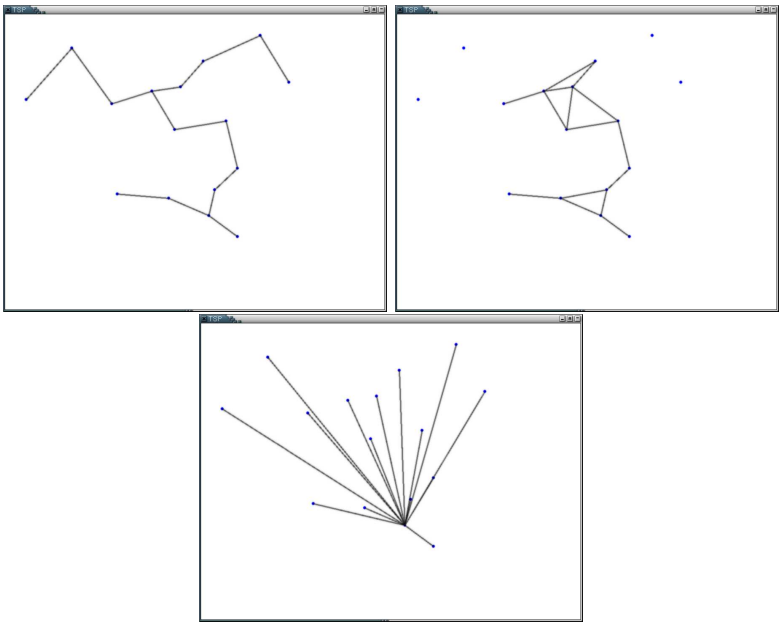


Figure 8.12: Spanning trees computed correctly (left), without detecting cycles (middle), and without sorting edges (right)

```
(define tree-start-event
  (trace ((tsp . jdot . mst) . jloc . entry)
         (bind () (λ (prev) empty))))
(define tree-edge-event
  (trace ((tsp . jdot . mst) . jloc . 80)
         (bind (e)
           (λ (prev)
             (cons (make-edge (e . jdot . v1)
                              (e . jdot . v2))
                   prev)))))
(define tree
  (accum-b (merge-e tree-start-event
                    tree-edge-event)
           empty))
(display-lines tree)
```

Figure 8.13: Recording MST Edges

```
(define pings (make-hash-table 'equal))

((changes where)
 . ==> . (match-lambda [(line function context rest ...)
                        (hash-table-increment! pings (list function context))]
                       [_ (void)]))

(define ticks (changes (quotient milliseconds 50)))

(set-running-e! (merge-e (ticks . -=> . false)
                         (ticks . -=> . true)))
```

Figure 8.14: A Statistical Profiler

Chapter 9

Related Work

Research on dataflow languages began in the early 1970s, and there has been a large body of work since then. An early language was Lucid [98], a pure, first-order dataflow language based on synchronous streams.

Lustre [20] is a synchronous dataflow language similar to Lucid. Programs in Lustre consist of *nodes*, which process streams of values that are computed in synchrony with various user-specified clocks. Variable definitions in Lustre are purely functional but may refer to previous values of themselves and other variables. Thus, its streams are essentially timed versions of the lazy lists found in many functional languages. They also capture the essence of FrTime's behaviors and events.

Lustre is designed to support development of reactive systems that satisfy real-time performance constraints, as well as static verification of safety properties. To that end, programs written in Lustre must be compilable to finite automata. To ensure compilability, Lustre deliberately omits features commonly found in general-purpose languages, such as object-orientation, higher-order functions, dynamic recursion, and recursive data structures, all of which FrTime does support. More fundamentally, Lustre differs from FrTime because it is a self-contained language, whereas FrTime is an embedding of dataflow evaluation within an existing call-by-value language.

Signal [7] is similar to Lustre, but it is based on relations rather than functions, so its underlying model is non-deterministic. FrTime adopts the basic spirit of dataflow programming but embeds it within the dynamic and higher-order context of an existing call-by-value functional language.

Lucid Synchrone [21] implements a Lustre-like notion of dataflow evaluation in a language resembling (purely functional) Caml [2]. Its syntax is very similar to that of Caml, so it approaches the notion of transparent reactivity provided by FrTime. However, it works by whole-program compilation to sequential Caml code (instead of using call-by-value evaluation to construct a dataflow graph), so it does not permit the free interleaving of dataflow and call-by-value evaluation, nor does it support live, incremental development in the style of FrTime.

Esterel [9] is another synchronous language designed for writing real-time reactive systems. Like Lustre, Esterel programs compile to finite automata and support reasoning about safety properties and real-time performance. However, unlike Lustre, which has a functional dataflow semantics, Esterel is imperative. An Esterel program consists of tasks that run in lockstep over a number of time steps, emitting, reading, and waiting for various signals that are either present or absent in any given timestep. Esterel shares with Lustre many of its differences from FrTime, such as a lack of higher-order procedures, dynamic recursion, and recursive data structures, all of which must be sacrificed in order to guarantee compilation to finite automate. However, because of its imperative programming style, Esterel is even more different from FrTime than Lustre.

FairThreads [11] is a framework for synchronous reactive programming, similar in style to Esterel. It allows a programmer to express computations in terms of many lightweight, cooperative threads, each of which gets a chance to run in each logical time step. The threads can communicate via broadcast signals (as in Esterel) as well as shared data. The FairThreads model offers the expressive power of fine-grained concurrency without the complexity and nondeterminism that arise from pre-emptive threads. Implementations of FairThreads exist for C, Java, and Scheme [86].

Functional reactive programming (FRP) [39, 74, 80, 102, 103, 104] merges the synchronous dataflow model 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 (continuously-valued) *behaviors* and (discretely-occurring) *events*. FrTime is inspired and informed by this line of research, borrowing the basic notions of behaviors and events.

Several Haskell FRP implementations are based on lazy stream abstractions, with which it is relatively straightforward to implement a notion of synchronous dataflow computation

based on polling. The dataflow abstractions in these systems are essentially the same as in FrTime, but FrTime also supports imperative features in the interfaces with the outside world, as well as interactive development in the REPL. Its update model is based on push instead of pull, so its performance characteristics are somewhat different; in particular, using push seems to incur significant overhead, but it has the advantage of only recomputing a signal if it depends on something that has changed.

There has been a good deal of work on implementation of FRP. Real-time FRP [103] and event-driven FRP [104] are first-order languages that have more in common with classical synchronous dataflow languages, where the focus is on bounding resource consumption. Parallel FRP adds a notion of non-determinism and explores compilation of FRP programs to parallel code. Elliott discusses several functional implementation strategies for general-purpose FRP systems [38], which suffer from various practical problems such as time- and space-leaks. A newer version, Yampa [74], fixes these problems at the expense of some expressive power: while Fran [39] aimed to extend Haskell with first-class signals, the Yampa programmer builds a network of *signal functions* through a set of *arrow* combinators [51].

FrTime's linguistic goal is to integrate signals with the Scheme language in as seamless a manner as possible. Importantly, because Scheme is eager, the programmer 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. A revised implementation of Fran called NewFran resolves many of the issues from the original Fran, using techniques very similar to those in FrTime. However, FrTime goes a step further by integrating with an interactive programming environment that supports incremental program development, various program-analysis tools, and a rich set of libraries.

Lula [89], a stage lighting system written in Scheme, contains a stream-based implementation of FRP that departs from the Haskell systems in a number of ways. It makes heavy use of Scheme's object-oriented features, modeling different varieties of signals through a class hierarchy. Like FrTime, it implements a set of functional reactive adapters for the MrEd toolkit, although it does not attempt to translate all of their imperative operations to dataflow abstractions. Also like FrTime, it takes selective advantage of certain impure features; for example, it uses threads to merge event streams without synchronous polling. However, unlike FrTime, its notion of reactivity is not transparent, and it has not

been tightly integrated with the DrScheme environment.

Frappé [29] 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. A more important difference from FrTime is that Frappé does not extend Java syntactically, so its reactivity is not transparent.

FRP has been applied to a number of domains, including animation [39], stage lighting [89], user interfaces [30, 85], robotics [77, 78], and computer vision [79]. We have explored animation and user-interface programming with FrTime and have also applied it to scriptable debugging [63]. This dissertation explores its use in more conventional desktop applications, including a graphical spreadsheet and an extension of Findler and Flatt's functional presentation system, Slideshow [43].

A technique similar to that employed by FrTime has been used to implement a form of dataflow for slot-based object systems like CLOS [33]. The basic idea is to extend slot accessor and mutator methods with code to implement dataflow updates. In particular, when an accessor is invoked from a signal-defining context, it records a dependency as well as returning a value. Likewise, when a mutator is invoked, it iterates through its list of dependents and re-evaluates them. This strategy was used to build the one-way constraint systems in Garnet [70] and Amulet [71] and has more recently been used in the Cells [94] library. None of these systems appears to support higher-order reactivity or to address glitches. Rather, they employ a depth-first update algorithm and avoid infinite loops in cycles by recomputing any given value at most once in a given update.

Dataflow-like features are increasingly finding their way into mainstream languages. For example, SuperGlue [65] is a linking language based on notions of behaviors and events. It is used to specify relationships between components in event-driven Java programs. Data dependencies trigger re-evaluation of component code, using a dataflow graph in much the same way as FrTime does. SuperGlue's ties to object-oriented programming are much stronger than those of FrTime, with direct language support for such features as objects, traits, and inheritance. While it does not provide a general notion of higher-order reactivity, it offers convenient abstractions for the common special case of collections. These allow a limited form of automatic generation of dynamic connections between objects.

The JavaFX [91] language supports a notion of *triggers* that execute when changes occur to specific variables. It also provides a more declarative mechanism for *bind*ing variables to expressions, the result being something akin to behaviors; that is, whenever the value of a referenced variable changes, the whole expression is re-evaluated and the result assigned to the bound variable. While the syntax of JavaFX is similar to that of Java, there are a number of differences, and special keywords are required to introduce the dataflow behavior, so its notion of reactivity is not quite *transparent* in the sense of FrTime. Also, JavaFX does not appear to support higher-order reactivity.

Petri nets [81] are, like dataflow, a graph-based model of computation. In a petri net, there are places, transitions, and arcs. When there are tokens at all of a transition's input places, the transition can fire, consuming the tokens and placing a specified number of tokens on the output places. Petri nets can model a wide range of computational patterns, including the update schedule of a synchronous dataflow program. However, they only model a program's control flow and not its the production of values. Moreover, its conjunctive firing rule cannot easily express FrTime's topological update algorithm.

The Aurora [19] and Borealis [22] systems are designed to perform efficient query-processing on time-varying data streams. Their evaluation model and some of their target applications are similar to those of FrTime. They are also linguistically similar to FrTime, as they explore the extension of database query languages with dataflow evaluation. FrTime, however, explores this extension in the context of a higher-order call-by-value language.

LabView [72] is a graphical dataflow language designed for processing signals from scientific instruments. It supports a limited notion of switching, in which fragments of a program's dataflow graph may be enabled and disabled according to time-varying conditions. Its notion of dataflow is, however, fundamentally first-order, and its designers were not constrained by a need to interact seamlessly with an existing host language. Simulink [92] is another commercial dataflow language. It is closely integrated with the MATLAB programming language and is designed for modeling and simulation of dynamic systems rather than expressing general software applications.

Click [58] is a system for programming network routers. The programmer defines a set of packet processors and constructs a network from them. Individual processors can be configured to push or pull; the runtime system, essentially a dataflow machine, inserts

queues between processors as necessary and automatically schedules the packet processors for execution. As it can support both push- and pull-based scheduling, it is in some sense more general than FrTime. However, its notion of dataflow is first-order and its application domain is significantly narrower than FrTime's. Moreover, like many of the languages with which I'm comparing, the dataflow language is a self-contained artifact rather than an embedding into a pre-existing language.

Ptolemy [16] is a framework for implementing and studying models of computation that involve communicating processes. The user defines rules by which processes change state and communicate with each other. He can then observe the execution of a single model or even combine several models and observe how they interact. The framework is general enough to model formalisms like Petri nets, communicating sequential processes, the π-calculus, statecharts, and both synchronous and asynchronous dataflow networks. While Ptolemy is certainly more general than FrTime, the system itself does not directly address the issues involved in integrating reactive programming models with more traditional call-by-value languages. Rather, it largely ignores such practical linguistic concerns as syntax, library support, and programming environments.

Oz [69, 88] is a multi-paradigm programming model that includes concurrency, higher-order functions, logic programming, and a form of dataflow. Oz's notion of dataflow is what is called *monotonic*; in this style of dataflow, a variable may initially be unbound (as in a logic program), then acquire a value at some intermediate stage of a computation, at which point it propagates to other parts of the program that refer to it, triggering new computation. In constrast, FrTime embodies a more traditional notion of dataflow, in which the values of variables may change repeatedly over the course of a program's execution.

The E [68] language is another multi-paradigm programming system, combining concurrency, distribution, and a capability-based security model. E also supports a notion of first-order behaviors, which may be distributed across multiple hosts, communicating via an adaptive push/pull mechanism that interacts smoothly with the language's distributed garbage collector. By comparison, FrTime supports higher-order dataflow and uses a purely push-based update mechanism, but its behaviors cannot be distributed easily or in a space-safe manner.

TBAG [40] is a C++ library for expressing animations in a declarative style. Linguistically, it has much in common with FrTime. For example, it uses static overloading to

define lifted versions of many built-in C++ operators. It thus achieves, to a degree, the same transparent reactivity exhibited by FrTime, though it does not carry the extensions to the level of syntactic forms (e.g., conditionals). TBAG does not provide the concept of switching that distinguishes FRP systems. However, it does support a more general bidirectional constraints constraint mechanism than the simple dataflow in FrTime and the other FRP systems.

An earlier object-oriented constraint-programming language is ThingLab [10], which was designed for expressing and running simulations. Like FrTime, ThingLab maintains dependencies between objects and automatically propagates updates when values change. It also supports interactive development and experimentation with systems, through both a REPL and a graphical interface. However, its constraint language is very general, and solving a system of constraints requires a relatively complex search algorithm, which in general may not succeed. In contrast, the unidirectional equality constraints in a dataflow language can be satisfied, by design, through simple functional evaluation.

Adaptive functional programming (AFP) [3] 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.

Open Laszlo [93] is a language designed for writing interactive applications that run in a Web browser. It is similar to HTML but has a different syntax and provides additional high-level features. The programmer specifies the structure and layout of a user interface in an XML document, which may contain script code written in JavaScript. In addition to a standard imperative callback-based approach to interaction, the language supports a feature its creators call *data-binding*, by which the contents and properties of user interface elements may be bound to mutable *data sources*. Whenever the data change, its consumers are updated automatically, as in a dataflow language. However, the language does not have a general notion of signals, or signal functions, that can be composed arbitrarily, and there is no declarative event-handling mechanism of the sort provided by FRP languages.

FOCUS [14] is a design methodology for specifying reactive systems. The basic approach is to model systems as collections of communicating stream processors whose behaviors are constrained by relations over their inputs and outputs. A system may be modeled at several levels of detail, each model a refinement of the previous one, and various properties of a model may be expressed and proven in a temporal logic framework. Since FrTime programs can also be viewed as communicating stream processors, it could serve as a target language for programs generated from FOCUS specifications. Alternatively, one could start from a FrTime program and apply the reasoning techniques from FOCUS to establish properties of it.

Integration with Object-Oriented Toolkits

The Citrus system [56] consists of a language and toolkit for creating editors for structured data. It provides several dataflow-like features, including automatic synchronization of models and views and the ability to define constraints on values. Citrus's constraints may refer to arbitrary program values, and they are automatically re-evaluated when such values change, using a graph to track dependencies in a manner similar to that of FrTime. Unlike FrTime, Citrus is designed specifically to simplify the construction of structured editors and, as such, does not seem well suited for general-purpose application development. Moreover, while particular features exhibit dataflow-based evaluation, it does not integrate the notion of dataflow with the language as a whole.

The FranTk [85] system adapted the Tk toolkit to a programmer interface based on the notions of behaviors and events in Fran [39]. However, FranTk still had a somewhat imperative feel, especially with regard to creation of cyclic signal networks, which required the use of mutation in the application program. Fruit [30] explored the idea of purely functional user interfaces, implementing a signal-based programming interface atop the Swing [37] toolkit.

All of this previous work is concerned with the problem of designing the dataflow interface for the toolkit, and the emphasis is on the experience for the application programmer. We consider this to be fairly well understood. However, the problem of actually implementing such an interface is less well understood. Though all of these earlier systems have included a working implementation, we understand that their development has been ad hoc,

and the subtle interaction between imperative toolkits and declarative dataflow systems has not been explained in the literature.

Optimization

Deforestation [100] and listlessness [99] are optimization techniques that eliminate intermediate data structures from functional programs. Their purpose is analogous to that of lowering, which eliminates intermediate nodes from a dataflow graph. Although the mechanics of these transformations are quite different from those of lowering, for stream-based FRP implementations [39, 50], we imagine that deforestation and listlessness could have an effect similar to lowering: namely, the weaving of multiple stream iterators into a single processing loop. FrTime, however, seems to require more special techniques because of its imperative implementation.

Most other FRP implementations [29, 74, 102, 103] do not provide the same level of transparency that FrTime offers. They implicitly lift a large number of common operations, but for some this is not possible, and syntactic constructs for features like conditional evaluation and recursive binding have not been extended to handle signals.

Yampa [74] implements a dynamic optimization that achieves essentially the same effect as lowering. When it evaluates a composition of pure signal functions, it replaces them with a single signal function that computes the composition of the original functions. In FrTime, such a dynamic optimization would be difficult to implement without loss of sharing. Specifically, without examining the program's syntactic structure, we cannot determine which intermediate signals can escape their context of creation, in which case they must exist as separate nodes.

Nilsson [73] explores the use of generalized abstract data types (GADTs) to support optimizations in Yampa [74]. The idea is to use GADTs to define special cases of signal processors, such as constants and the identity function, and implement special, optimized logic for them in the evaluator. In particular, Nilsson's implementation performs constant-propagation and automatically eliminates calls to the identity function, yielding measurable improvement in various benchmarks. Moreover, the GADT-based optimizations can be applied to networks of stateful signal processors, which our approach cannot handle.

Real-time FRP (RT-FRP) [103] is an implementation of FRP that shares certain similarities with FrTime, such as the explicit connection to an underlying host language with a collection of base primitives. The goal of RT-FRP is not to produce highly efficient code so much as to establish provable bounds on the time and space required by each round of execution. The language achieves these bounds through a conservative static analysis, but it does not perform any optimizing program transformations.

Event-driven FRP (E-FRP) [104] is a modification of RT-FRP designed to support compilation to efficient imperative code. E-FRP adds some crucial restrictions to RT-FRP that make such compilation possible. Primarily, it takes away the ability to perform dynamic switching, thereby making the program's data dependencies static. It also requires that only one external event can stimulate the system in any given update cycle. As in RT-FRP, the language performs no optimizing program transformations; rather, it uses a syntactic restriction to guarantee limits on the program's resource requirements. In forbidding dynamic switching, E-FRP more closely resembles traditional synchronous dataflow languages, such as Lustre [20], Esterel [9], and Signal [7]. These languages have a common goal of compiling to efficient straightline code, which they achieve by design. This is in contrast to FrTime, whose primary goal is to provide expressive power, often at the expense of performance.

Languages for Scriptable Debugging

There are two main branches of research that relate to MzTake and which have helped inspire it: first, programmable debugging, and second, program monitoring and instrumentation.

Dalek [76] is a scripted debugger built atop `gdb` that generates events corresponding to points in the program's execution. Each event is associated with a callback procedure that can, in turn, generate other events, thus simulating a dataflow style of evaluation. When the propagation stabilizes, Dalek resumes program execution.

MzTake has several important features not present in Dalek. A key difference that a user would notice is that we rely on FrTime to automatically construct the graph of dataflow dependencies, whereas in Dalek, the programmer must construct this manually. Dalek's events are not first-class values, so programmers must hard-wire events to scripts, and

therefore cannot easily create reusable debugging operations such as *not-in-order*.

In Dalek, each event handler can suspend or resume the execution of the target program, but these can contradict each other. Dalek applies a fixed rule to arbitrate these conflicts, in contrast with the variety of start-stop rules discussed in section 8.8. Indeed, using a stream as the guard expression highlights the power of using FrTime as the base language for the debugger, since a few lines of FrTime code can reconstruct Dalek's policy in MzTake: the code shown in figure 8.10 is in fact Dalek's policy. This design addresses an important concern raised in an analysis of Dalek by Crawford, et al. [31].

The Acid debugger [105] provides the ability to respond to breakpoint commands and step commands with small programs written in a debugging script language very close to C. Deet [48] provides a scripting language based on Tcl/Tk along with a variety of the graphical facilities. Dispel [54] defines its own ad-hoc language. Generalized path expressions [15] specify break conditions as regular repressions applied to event traces. The regular expressions are augmented with predicates that can check for base-value relations. In these projects, the programmer must respond to events through callbacks, and there is no notion of a dataflow evaluation mechanism. Each retains the inspection and control mechanism of command-prompt debuggers.

DUEL [47] extends `gdb` with an interpreter for a language intended to be a superset of C. It provides several constructs, such as list comprehensions and generators, for inspecting large data structures interactively. However, it does not address how to control the target program or how to respond to events generated during the execution.

The Coca debugger by Ducassé [35] offers a conditional breakpoint language based on Prolog. Coca uses the backtracking evaluation mechanism of Prolog to identify potentially problematic control and data configurations during the execution, and brings these to the user's attention. As such, Prolog predicates serve as both the conditional breakpoint language and the data-matching language. However, since each predicate application happens in isolation from the other, there is no way to accumulate a model of the execution as it happens through time, such as constructing a trace history or building an explicit representation of an MST (as we have done in this paper).

Like Coca, on-the-fly query-based debugging [61, 62] enables users to interactively select heap objects. The objects are specified using a SQL-like language evaluated using an efficient on-line algorithm. It does not offer a sophisticated scripting mechanism. Like

Coca, this approach does not support relating data between points in time.

Parasight [4] allows users to insert C code at tracepoint locations. The C code is compiled and inserted into the running target program's process in a way that has minimal performance impact. The inserted code must, however, adopt a callback-style to respond to events. While adapting the running program has performance benefits, it also complicates the process of using more expressive languages to perform monitoring and debugging (and indeed, Parasight does not tackle this issue at all, using the same language for both the target program and the scripts).

Alamo [53], like Parasight, instruments binary objects with in-process C code. While the scripts do not take the shape of callbacks, they must manually implement a programming pattern that simulates a coroutine (which is handled automatically in FrTime by the evaluation mechanism). The UFO debugger [6] extends Alamo with a rich pattern-matching syntax over events in terms of the target language's grammar. While MzTake offers a rich, general-purpose language for processing event-streams, UFO efficiently handles the special case of list comprehension followed by folding.

There are several projects for monitoring program execution, as Dias and Richardson's taxonomy describes [32]. Monitors differ from debuggers by virtue of not being interactive, and most do not provide scripting facilities. Instead, many of these systems have better explored the trade-offs between expressiveness, conciseness and efficiency in the specification of interesting events. MzTake simply relies on the powerful abstractions of FrTime to filter events, but at the cost of efficiency.

MzTake supports the notion that debugging code should remain outside a program's source code, to avoid complicating maintenance and introducing time- and space-complexity penalties. A debugging script is thus a classic "concern" that warrants separation from the core program. Aspect-like mechanisms [5] offer one way to express this separation. However, using them for MzTake would not be straightforward; most implementations of aspect mechanisms rely on static compilation, which makes it impossible to change the set of debugging tasks on-the-fly. More importantly, most of them force the debugging script and main program to be in the same language, making it difficult to use more expressive languages for scripting. These mechanisms are therefore orthogonal to MzTake and are possible routes for implementing its scripting language.

Smith [87] proposes a declarative language for expressing equality constraints between

the programmer's model and the execution trace. This can be seen as an aspect-like system in which the aspects are not restricted to the original target language. Smith's language relies on a compiler to generate an instrumented program that maintains the model incrementally. Unfortunately, the compiler has not been implemented and, as the paper acknowledges, developing an implementation would not be easy.

Contracts [67] also capture invariants, but they too suffer from the need for static compilation. In addition, data structures sometimes obey a stronger contract in a specific context than they do normally. For instance, priority heaps permit keys to change, which means there is no *a priori* order on a key's values. However, Dijkstra's algorithm initializes keys to ∞ and decreases them monotonically, and failure to do so indicates an error. The topicality of the contract means it should not be associated with the priority heap in general.

DTrace [18] is a system for dynamically instrumenting all layers of production systems. It supports a variety of instrumentation providers, which are capable of creating and enabling *probes* that fire when specific events occur. A tracing script, written in a custom domain-specific language called D (a variant of C with features akin to Awk) defines a set of probes, as well as consumers that execute and process the data the probes produce. Consumers are invoked implicitly when the associated events occur, though unlike in MzTake, there is no dataflow mechanism with which to build higher-level event-based abstractions. On the other hand, DTrace supports instrumentation at the machine level, using binary rewriting techniques to prevent overhead when tracing is disabled. Its primary application seems to be determining the root causes of performances problems, even when the symptoms are several levels removed from these causes.

Tool Reuse

There is a significant body of work concerned with the mechanical generation of tools like the error-tracer and profiler we have described. For example, Dinesh and Tip [34] show how to derive *animators* and *error reporters* automatically from algebraic specifications of programming language interpreters and type checkers. The tools rely heavily on the technique of origin-tracking [97] for first-order term rewriting systems. The authors note the discovery of critical restrictions and limitations of first-order rewriting, and Van

Deursen and Dinesh [96] subsequently developed an origin-tracking algorithm for higher-order rewriting systems.

The ASF+SDF meta-environment [95] supports automatic generation of interactive systems for creating language definitions and generating tools for them. Programmers write algebraic language definitions in the ASF+SDF language [8], which allows the specification of conditional term-rewriting rules. The system, which compiles specifications to C code, has been used to create several domain-specific languages, along with a number of language-processing tools.

The meta-environment also supports a framework for *generic debugging* [75], where a single debugger supports interaction with a wide variety of programming languages. The implementor of each language defines an *adaptor* that provides a standard interface to the language's control and data abstractions and through which the generic debugger interacts with the language's runtime. This also allows for so-called *multi-level debugging* [28], where a user can debug a program and its implementation language simultaneously.

Compared to these systems, our approach is more specialized but considerably lighter-weight. It does not require that the developer write formal algebraic specifications, and it can track dependencies even when control flows into runtime support libraries. Also, our approach avoids the need for source code manipulation, as demonstrated by our reuse of the host language's generic annotator and tools. This reuse is important for domain-specific languages, where minimizing development costs is critical. We also implicitly reuse the macro-tracking facility [36, 59] in the PLT Scheme system, which allows correlation of expressions in macro-expanded programs with their original source.

Hudak [49] describes a methodology for building embedded domain-specific languages in Haskell, citing various forms of reuse as the main motivation for the technique. Several functional reactive programming systems [39, 74] have been implemented as Haskell embeddings. These systems of course have less need for runtime debugging tools, since the static type checker catches most errors before the program runs. However, if one were to attempt to use a profiler or error-tracer with a Haskell embedding, it would suffer from the same problem that I have needed to address in FrTime. The same is true for other possible host languages. For example, Frappé [29] is a Java implementation of FRP that in many ways resembles FrTime and suffers from the same sorts of tool-reuse problems that FrTime experienced prior to the modifications described in this paper.

Chapter 10

Conclusions and Future Work

I have developed a strategy for embedding a notion of dataflow evaluation within call-by-value languages. The strategy is based on lightweight syntactic extensions and reuse of the base language's evaluator. It emphasizes reuse of as much as possible of the base language, including ideally its libraries and tools. This strategy is embodied within the new language FrTime, which builds on PLT MzScheme.

Much of the value of this work derives from the fact that the embedding strategy applies to other call-by-value languages. To validate this claim, two ports of its evaluation model are currently under development:

10.1 Flapjax

Flapjax[1] is a language designed to support development of interactive Web applications. It extends JavaScript—the most common scripting language for Web browsers—with several features, including dynamic dataflow evaluation in the spirit of FrTime. Like FrTime, Flapjax is a conservative extension of its host language. This means that it is straightforward to integrate Flapjax code with existing JavaScript, or to incrementally migrate JavaScript programs to Flapjax.

Since JavaScript lacks a module system, lifting cannot be implemented with the same sort of linguistic mechanism as in FrTime. Instead, a lightweight compiler expands Flapjax code into ordinary JavaScript with explicit calls to the *lift* library procedure. Just as

[1] *F*unctional *l*anguage for *a*pplication *p*rogramming with A*JAX*.

in FrTime, the resulting code generates a dataflow graph when executed, and a dataflow engine employs essentially the same algorithm to keep the graph up-to-date. The graph-construction primitives are also available in the form of a pure library, so users who prefer not to depend on the compiler can still take advantage of the dataflow capabilities.

JavaScript programs interact with Web pages through a data structure called a *document object model*, or (DOM). The DOM represents the tree structure of the HTML page that the browser renders; if the script modifies the DOM, then the browser updates its display. Most elements of the DOM support user interaction, at least in the form of low-level mouse events; form elements like buttons, check boxes, and text entry fields provide higher-level interfaces. Like most other user-interface toolkits, the JavaScript interface to the DOM is based on callbacks. Thus one of the most important aspects of Flapjax is a signal-based interface to the DOM that models both input and output.

In addition to the dataflow evaluation model, Flapjax provides facilities for communicating with the Flapjax server, which authenticates users and provides access-controlled sharing of a persistent store for time-varying JavaScript objects. The interface to the server is based on signals, so it is easy to make a program's data transparently persistent.

Although the Flapjax system is still maturing, the results so far are encouraging. At least one non-trivial application—a WYSIWYG Wiki engine—has been developed using it. Several hundred users have created accounts on the Flapjax server, while hundreds more have visited the site to experiment with various demos and read the documentation.

10.2 FrC++

FrC++ is a port of FrTime's dataflow evaluation model to C++. Its design, like that of FrTime and Flapjax, is influenced by the particular features of its host language. For example, the static overloading mechanism in C++ allows primitive operators to be extended to handle new datatypes. FrC++ uses this capability to define lifted versions of operators like + and * that can operate on numeric behaviors. Thus FrC++ supports—to a degree—the notion of transparent reactivity that FrTime provides. Moreover, because the overloading is resolved statically, the implementation is more efficient than in FrTime, where a runtime check is performed each time the program applies a lifted operator. However, because the types of procedures must be explicitly declared, FrC++ cannot support transparent code

reuse to the extent FrTime does.

FrC++ makes use of the FC++ [66] library for functional programming in C++, which uses objects to implement first-class procedures, complete with static type-checking and parametric polymorphism. Signals are instances of class FrSignal, which is parameterized over the type of value it carries. The type parameters are checked statically, so a significant class of errors is caught before the program runs. The use of typed signals complicates the implementation of the dataflow engine, which needs to be able to operate on a single datatype. To facilitate this, all signals derive from a single non-parametric base class that provides a nullary method called *update*, which returns a boolean value indicating whether the signal's value has changed. The dataflow engine refers to all signals through the base class, which is sufficient for moving signals in and out of the priority queue and invoking the *update* method.

There are bindings for FC++ to the Gtk user interface toolkit, so the language can be used to write graphical applications. The adaptation follows essentially the same techniques as FrTime does for MrEd. Anecdotal evidence indicates the FrC++ far exceeds both FrTime and Flapjax in terms of execution performance. This is unsurprising given the amount of effort that has been put into the construction of efficient C++ compilers. Since FrC++ is a pure library implementation, employing something like the lowering optimization from Chapter 4 would seem to require the development of significant infrastructure. (Fortunately, the high performance of the host language has thus far made such a endeavor unnecessary.) Moreover, while there are a variety of tools for C++, they necessarily exhibit the same sorts of problems mentioned in Chapter 7.

Bibliography

[1] The Ruby JDWP project. `http://rubyforge.org/projects/rubyjdwp/`.

[2] The Caml language. `http://caml.inria.fr`.

[3] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002.

[4] Z. Aral and I. Gertner. High-level debugging in Parasight. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 151–162. ACM Press, 1988.

[5] Aspect oriented programming (article series). *Communications of the ACM*, 44(10), Oct. 2001.

[6] M. Auguston, C. Jeffery, and S. Underwood. A framework for automatic debugging. In *Automated Software Engineering*, pages 217–222, 2002.

[7] 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.

[8] J. A. Bergstra, J. Heering, and P. Klint. *Algebraic specification*. ACM Press, 1989.

[9] G. Berry. *The Foundations of Esterel*. MIT Press, 1998.

[10] A. H. Borning. The programming language aspects of thinglab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, 1981.

[11] F. Boussinot. Fairthreads: mixing cooperative and preemptive threads in c: Research articles, 2006.

[12] J. P. Bowen and M. J. C. Gordon. A shallow embedding of Z in HOL. *Information and Software Technology*, 37(5-6):269–276, May–June 1995.

[13] G. Bracha and W. Cook. Mixin-based inheritance. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, pages 303–311, 1990.

[14] M. Broy and K. Stølen. *Specification and Development of Interactive Systems: FOCUS on Streams, Interfaces, and Refinement*. Springer, 2001.

[15] B. Bruegge and P. Hibbard. Generalized path expressions: A high level debugging mechanism. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-level Debugging*, pages 34–44, 1983.

[16] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulation (special issue on Simulation Software Development)*, 4:155–182, April 1994.

[17] K. Burchett, G. H. Cooper, and S. Krishnamurthi. Lowering: A static optimization for transparent functional reactivity. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 2007.

[18] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *USENIX Annual Technial Conference*, pages 15–28, 2004.

[19] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebreaker, N. Tatbul, and S. Zdonik. Monitoring streams — a new class of data management applications. In *International Conference on Very Large Databases*, pages 215–226, 2002.

[20] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 178–188, 1987.

[21] P. Caspi and M. Pouzet. Lucid Synchrone, a functional extension of Lustre, 2000.

[22] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *First Biennial Conference on Innovative Data Systems Research*, 2003.

[23] J. Clements and M. Felleisen. A tail-recursive machine with stack inspection. *ACM Transactions on Programming Languages and Systems*, 26(6):1029–1052, 2004.

[24] J. Clements and M. Felleisen. A tail-recursive machine with stack inspection. *ACM Transactions on Programming Languages and Systems*, 26(6):1029–1052, 2004.

[25] J. Clements, M. Flatt, and M. Felleisen. Modeling an algebraic stepper. In *European Symposium on Programming*, 2001.

[26] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *European Symposium on Programming*, 2006.

[27] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1997.

[28] B. Cornelissen. Using TIDE to debug ASF+SDF on multiple levels. Master's thesis, Centrum voor Wiskunde en Informatica, 2004.

[29] A. Courtney. Frappé: Functional reactive programming in Java. In *Practical Aspects of Declarative Languages*. Springer-Verlag, March 2001.

[30] A. Courtney and C. Elliott. Genuinely functional user interfaces. In *Haskell Workshop*, 2001.

[31] R. H. Crawford, R. A. Olsson, W. W. Ho, and C. E. Wee. Semantic issues in the design of languages for debugging. In *Proceedings of the International Conference on Computer Languages*, pages 252–261, 1992.

[32] M. de Sousa Dias and D. J. Richardson. Issues on software monitoring. Technical report, ICS, 2002.

[33] L. G. DeMichiel and R. P. Gabriel. The Common Lisp Object System: An overview. In *European Conference on Object-Oriented Programming*, 1987.

[34] T. B. Dinesh and F. Tip. Animators and error-reporters for generated programming environments. Technical Report CS-R9253, Centrum voor Wiskunde en Informatica, 1992.

[35] M. Ducassé. Coca: an automated debugger for C. In *Proceedings of the 21st International Conference on Software Engineering*, pages 504–513, 1999.

[36] R. K. Dybvig, R. Hieb, and C. Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, Dec. 1993.

[37] R. Eckstein, M. Loy, and D. Wood. *Java Swing*. O'Reilly, 1997.

[38] C. Elliott. Functional implementations of continuous modeled animation. In *International Symposium on Programming Languages: Implementations, Logics, and Programs*. Springer-Verlag, 1998.

[39] C. Elliott and P. Hudak. Functional reactive animation. In *ACM SIGPLAN International Conference on Functional Programming*, pages 263–277, 1997.

[40] C. Elliott, G. Schechter, R. Yeung, and S. Abi-Ezzi. TBAG: A high level framework for interactive, animated 3D graphics applications. In *ACM International Conference on Computer Graphics*, pages 421–434, 1994.

[41] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 102:235–271, 1992.

[42] 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.

[43] R. B. Findler and M. Flatt. Slideshow: Functional presentations. In *ACM SIGPLAN International Conference on Functional Programming*, 2004.

[44] M. Flatt. Composable and compilable macros. In *ACM SIGPLAN International Conference on Functional Programming*, pages 72–83. ACM Press, 2002.

[45] 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.

[46] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183, 1998.

[47] M. Golan and D. R. Hanson. DUEL - a very high-level debugging language. In *Proceedings of the USENIX Annual Technical Conference*, pages 107–118, Winter 1993.

[48] D. R. Hanson and J. L. Kom. A simple and extensible graphical debugger. In *Proceedings of the USENIX Annual Technical Conference*, pages 183–174, 1997.

[49] P. Hudak. Modular domain specific languages and tools. In *International Conference on Software Reuse*, 1998.

[50] P. Hudak. *The Haskell school of expression: learning functional programming through multimedia*. Cambridge, 2000.

[51] J. Hughes. Generalizing monads to arrows. *Science of Computer Programming*, 37(1-3), May 2000.

[52] D. Ignatoff, G. H. Cooper, and S. Krishnamurthi. Crossing state lines: Adapting object-oriented frameworks to functional reactive languages. In *Symposium on Functional and Logic Programming*, 2006.

[53] C. Jeffery, W. Zhou, K. Templer, and M. Brazell. A lightweight architecture for program execution monitoring. In *SIGPLAN Notices*, volume 33, pages 67–74, 1998.

[54] M. S. Johnson. Dispel: A run-time debugging language. *Computer Languages*, 6:79–94, 1981.

[55] R. Kelsey, W. Clinger, and J. Rees. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9), Oct. 1998.

[56] A. J. Ko and B. A. Myers. Citrus: A language and toolkit for simplifying creation of structured editors for code and data. In *UIST*, 2005.

[57] E. E. Kohlbecker Jr. *Syntactic Extensions in the Programming Language Lisp*. PhD thesis, Indiana University, 1986.

[58] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[59] S. Krishnamurthi, M. Felleisen, and B. F. Duba. From macros to reusable generative programming. In *Generative and Component-Based Software Engineering*, 1999.

[60] J. L. Lawall and D. P. Friedman. Towards leakage containment. Technical Report 346, Indiana University, 1992.

[61] R. Lencevicius. On-the-fly query-based debugging with examples. In *Proceedings of the Fourth International Workshop on Automated Debugging*, 2000.

[62] R. Lencevicius, U. Hölzle, and A. K. Singh. Dynamic query-based debugging of object-oriented programs. *Automated Software Engineering*, 10(1):39–74, 2003.

[63] 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*, 2004.

[64] G. Marceau, G. H. Cooper, J. P. Spiro, S. Krishnamurthi, and S. P. Reiss. The design and implementation of a dataflow language for scriptable debugging. *Automated Software Engineering Journal*, 2006.

[65] S. McDirmid and W. C. Hsieh. Component programming with object-oriented signals. In *European Conference on Object-Oriented Programming*, pages 206–229, 2006.

[66] B. McNamara and Y. Smaragdakis. Functional programming in C++. In *ACM SIGPLAN International Conference on Functional Programming*, pages 118–129, 2000.

[67] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[68] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, The Johns Hopkins University, 2006.

[69] M. Müller, T. Müller, and P. V. Roy. Multi-paradigm programming in Oz. In D. Smith, O. Ridoux, and P. Van Roy, editors, *Visions for the Future of Logic Programming: Laying the Foundations for a Modern successor of Prolog*, Portland, Oregon, 7 Dec. 1995. A Workshop in Association with ILPS'95.

[70] B. A. Myers, D. A. Giuse, R. B. Dannenberg, D. S. Kosbie, E. Pervin, A. Mickish, B. V. Zanden, and P. Marchal. Garnet: Comprehensive support for graphical, highly interactive user interfaces. *Computer*, 23(11):71–85, 1990.

[71] B. A. Myers, R. G. McDaniel, R. C. Miller, A. S. Ferrency, A. Faulring, B. D. Kyle, A. Mickish, A. Klimovitski, and P. Doane. The Amulet environment: New models for effective user interface software development. *IEEE Transactions on Software Engineering*, 23(6):347–365, 1997.

[72] National Instruments, Inc. Labview (software system). `http://www.ni.com/labview`.

[73] H. Nilsson. Dynamic optimization for functional reactive programming using generalized abstract data types. In *ACM SIGPLAN International Conference on Functional Programming*, 2005.

[74] H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *ACM SIGPLAN Workshop on Haskell*, pages 51–64, 2002.

[75] P. Olivier. *A Framework for Debugging Heterogeneous Applications*. PhD thesis, Centrum voor Wiskunde en Informatica, 2000.

[76] R. A. Olsson, R. H. Crawford, and W. W. Ho. Dalek: A GNU, improved programmable debugger. In *Proceedings of the Usenix Technical Conference*, pages 221–232, 1990.

[77] J. Peterson and G. Hager. Monadic robotics. In *Domain-Specific Languages*, pages 95–108, 1999.

[78] J. Peterson, P. Hudak, and C. Elliott. Lambda in motion: Controlling robots with Haskell. *Lecture Notes in Computer Science*, 1551:91–105, 1999.

[79] J. Peterson, P. Hudak, A. Reid, and G. Hager. FVision: A declarative language for visual tracking. *Lecture Notes in Computer Science*, 1990:304–321, 2001.

[80] J. Peterson, V. Trifonov, and A. Serjantov. Parallel functional reactive programming. In *Practical Aspects of Declarative Languages*, volume 1753, 2000.

[81] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, University of Bonn, 1962.

[82] G. Pettyjohn, J. Clements, J. Marshall, S. Krishnamurthi, and M. Felleisen. Continuations from lightweight stack inspection. In *ACM SIGPLAN International Conference on Functional Programming*, 2005.

[83] S. L. Peyton Jones. Compiling Haskell by transformation: a report from the trenches. In *European Symposium on Programming*, pages 18–44, 1996.

[84] G. Plotkin. Call-by-name, call-by-value, and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[85] M. Sage. FranTk: A declarative GUI language for Haskell. In *ACM SIGPLAN International Conference on Functional Programming*, pages 106–117, 2000.

[86] M. Serrano, F. Boussinot, and B. Serpette. Scheme fairthreads. In *2th International Lisp Conference*, October 2002.

[87] D. R. Smith. A generative approach to aspect-oriented programming. In *International Conference on Generative Programming and Component Engineering*, volume 3286, pages 39–54, 2004.

[88] G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.

[89] M. Sperber. Developing a stage lighting system from scratch. In *ACM SIGPLAN International Conference on Functional Programming*, pages 122–133, 2001.

[90] R. M. Stallman. *GDB Manual (The GNU Source-Level Debugger)*. Free Software Foundation, Cambridge, MA, third edition, January 1989.

[91] Sun Microsystems. JavaFX. `http://www.sun.com/software/javafx/`.

[92] The MathWorks, Inc. Simulink - simulation and model-based design. `http://www.mathworks.com/products/simulink/`.

[93] The Open Laszlo Project. Open laszlo. `http://www.openlaszlo.org/`.

[94] K. Tilton. Cells. `http://www.tilton-technology.com/cells\_top.html`.

[95] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF meta-environment: A component-based language development environment. In *Compiler Construction*, pages 365–370, 2001.

[96] A. van Duersen and T. B. Dinesh. Origin tracking for higher-order rewriting systems. Technical Report CS-R9425, Centrum voor Wiskunde en Informatica, 1994.

[97] A. van Duersen, P. Klint, and F. Tip. Origin tracking. Technical Report CS-R9230, Centrum voor Wiskunde en Informatica, 1992.

[98] W. W. Wadge and E. A. Ashcroft. *Lucid, the dataflow programming language*. Academic Press U.K., 1985.

[99] P. Wadler. Listlessness is better than laziness. In *ACM Symposium on Lisp and Functional Programming*, pages 45–52, 1986.

[100] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

[101] P. Wadler. The essence of functional programming. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, January 1992.

[102] Z. Wan and P. Hudak. Functional reactive programming from first principles. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–252, 2000.

[103] Z. Wan, W. Taha, and P. Hudak. Real-time FRP. In *ACM SIGPLAN International Conference on Functional Programming*, pages 146–156, 2001.

[104] Z. Wan, W. Taha, and P. Hudak. Event-driven FRP. In *Practical Aspects of Declarative Languages*, 2002.

[105] P. Winterbottom. Acid, a debugger built from a language. In *Proceedings of the USENIX Annual Technical Conference*, pages 211–222, January 1994.