Crossing State Lines: Adapting Object-Oriented Frameworks to Functional Reactive Languages*

Daniel Ignatoff, Gregory H. Cooper, and Shriram Krishnamurthi

Computer Science Department, Brown University
{dignatof, greg, sk}@cs.brown.edu

Abstract. Functional reactive programming integrates dynamic dataflow with functional programming to offer an elegant and powerful model for expressing computations over time-varying values. Developing realistic applications, however, requires access to libraries, such as those for GUIs, that are written in mainstream object-oriented languages. Previous work has developed functional reactive interfaces for GUI toolkits but has not provided an account of the principles underlying the implementation strategy.

In this paper, we investigate this problem by studying the adaptation of the object-oriented toolkit MrEd to the functional reactive language FrTime. The heart of this problem is how to communicate state changes between the application and the toolkit's widget objects. After presenting a basic strategy for adaptation, we discuss abstraction techniques based on mixins and macros that allow us to adapt numerous properties in many widget classes with minimal code duplication. This results in a wrapper for the entire MrEd toolkit in only a few hundred lines of code. We also briefly discuss a spreadsheet developed with the resulting toolkit.

1 Introduction

Functional reactive programming (FRP) extends a general-purpose functional language with abstractions for expressing values that change over time. By combining the features of dataflow and higher-order functional programming, it supports concise, declarative descriptions of reactive and interactive systems. This paper specifically uses the language FrTime [3] (pronounced "father time"), an embedding of FRP in the DrScheme [7] programming environment. FrTime pursues a push-driven evaluation strategy that permits incremental program development (e.g., in a read-eval-print loop) formulated atop the eager semantics of Scheme.

While FRP provides an elegant notation for specifying the computational core of systems, application developers need more: they also must be able to use standard libraries for graphics, user interfaces, networking, and so on. These libraries have several important 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.

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

The integration process must therefore handle this style, and ideally exploit it. An important subtlety is that OO and FRP languages have different notions of 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 representations of state must be reconciled.

We have made considerable progress on this integration problem for the specific case of GUIs. The DrScheme environment provides a large and robust GUI library called MrEd [8], 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. We have discovered several useful abstractions based on *mixins* [2] (classes parameterized over their super-classes) that enable a seamless integration. We have further found that there are patterns to these mixins and abstracted over them using *macros* [10]. As a consequence, the adapter for MrEd is under 400 lines of code.

This paper may appear on the surface to describe work similar to the FranTk [14] and Fruit [4] projects, and in fact the interface we develop for MrEd is similar in spirit to the ones developed for those systems. However, those other systems focus on the design of a programmer's interface for building GUI applications in a functional reactive language. In contrast, our work addresses the lower-level, largely orthogonal issue of importing legacy object-oriented frameworks into an FRP system. Our primary example also happens to be GUI libraries, as these have direct practical applicability and are sufficiently complex to make an interesting case study. However, the ideas we present are not specific to GUI libraries. If anything, adapting other kinds of libraries should be even easier, given the highly imperative nature of GUIs.

This paper is organized as follows. First, we present the implementation of a small GUI application in MrEd, which we use to illustrate some of the difficulties posed by the standard object-oriented GUI programming model. Next we provide a brief overview of FrTime, whose notion of *signals* offers a more natural, declarative mechanism for modeling state. We then discuss the design philosophy that governs our adaptation of MrEd to a signal-based programming interface. The heart of the paper is a description of our implementation of this interface and of the abstractions that capture the essence of the adaptation. We talk briefly about a spreadsheet application built with the adapted toolkit, then discuss related work and provide concluding remarks.

2 GUI Construction with an Object-Oriented Toolkit

We walk through the process of writing a small GUI application in MrEd, the standard GUI library included with the extended object-oriented language supported by DrScheme. MrEd resembles the OO GUI libraries of languages like C++ and Java.

The application we develop is a simple timer; it counts seconds for a user-specified time, displaying the elapsed time both graphically and textually. Figure 1 shows the entire code, which we explain incrementally.

First, we create a window. In MrEd, a normal top-level window is called a *frame%*.[1] A **new** expression constructs an object of a given type with a set of named parameters,

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

```
(define frame (new frame% [label "Timer"] [height 80] [width 300] [alignment '(left top)]))
(define gauge (new gauge% [parent frame] [label "Elapsed Time"] [range 60]))
(define message (new message% [parent frame] [label "0 s"] [stretchable-width true]))
(define slider
  (new slider% [parent frame] [label "Duration (s)"]
               [min-value 30] [max-value 120] [init-value 60]
               [callback (λ (s e) (send gauge set-range (send slider get-value)))]))
(define button (new button% [parent frame] [label "Reset"]
                            [callback (λ (b e) (set! elapsed 0))]))
(define elapsed 0)
(define (loop)
  (when (< elapsed (send slider get-value))
    (send gauge set-value elapsed)
    (send message set-label (format "~a s" elapsed))
    (sleep/yield 1)
    (set! elapsed (add1 elapsed))
    (loop)))
(send frame show #t)
(loop)
```

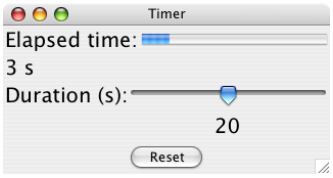

Fig. 1. A simple timer application in Scheme with MrEd, with a screenshot at the lower right

in this case the label, dimensions, and alignment. We next create gauge and message widgets as children of the window. The gauge is to display the elapsed time as a colored bar, and the message to present the same information textually.

We then add a slider that lets the user adjust the timer's duration. The slider needs a parent and a label like the other controls, along with a range. Whenever the user adjusts the duration through the slider, we need to update the gauge's range accordingly. The toolkit provides a *callback* for this purpose, which lets the application register a procedure to be executed whenever the user interacts with a widget.

The last widget is a reset button, which also takes a callback procedure. In this case, the callback simply resets the elapsed time to 0. After creating the widgets, we tell the window to display itself (and its contents) by invoking the *show* method. We then write a loop to count out the duration and keep the gauge and message widgets up to date.

Evaluating the GUI Coding Style

This simple example gives a sense of the nature of GUI programming. Even in a mostly functional language like Scheme, the programming style is very imperative. In particular, the need to handle values that change over time forces the programmer to use mutation and other side-effecting operations. The application needs to know when its values change so it can update the properties of widgets, and it needs to register callbacks so it can find out when widget properties change and accordingly update its internal state.

All these callbacks and imperative operations tend to invert and obscure the system's structure and data dependencies. For example, the contents of the gauge and message depend on the variable *elapsed*, but this relationship is not apparent from the widgets'

definitions. Instead, the loop body is responsible for updating the widgets, so we need to examine *it* to understand the behavior of all the widgets. Similarly, the range of the gauge depends on the value of the slider, but gauge's definition does not express this relationship; instead, the slider's callback invokes methods on the gauge to keep it up-to-date. In general, when an object's state depends on external mutation, reasoning about its behavior requires awareness of all invocations that target the object. This *structural inversion* is a serious impediment to understanding and maintaining the code. For example, if a developer adds, modifies, or removes a widget, then he must be sure to identify and properly update all of its referents.

3 FrTime

Much of the complexity of GUI programming arises from the lack of linguistic support for modeling values that change over time. This is what necessitates the use of imperative state, with the resulting inversion of program structure and increase in complexity.

The goal of dataflow programming is to support the modeling of change. Dataflow languages introduce a concept of *signals*, or time-varying values. This idea has been revived in a recent line of work called functional reactive programming (FRP) [6, 13], which merges dataflow with higher-order functional programming. We have developed an implementation of FRP for DrScheme [7] called FrTime [3].

FrTime publishes a signal called *seconds*, which represents the current time as an integral number of seconds. We can project its value at any moment by asking for its **value-now**. This returns the current constant integer value of the signal. We can also use *seconds* to build new signals; for example, (*even? seconds*) alternates between **true** and **false** every second.

We can model the elapsed time in our application by computing the difference between the current value of *seconds* and its value when the count started. We express this in code with (− *seconds* (**value-now** *seconds*)), where (**value-now** *seconds*) returns a constant, and subtracting it from *seconds* yields a new signal that starts at 0 and increments every second. Because we use signals, the language automatically keeps them up-to-date. Otherwise we would need to keep track of the passage of time (e.g., with a timer or *sleep* command) and manually update all the variables that (transitively) depend on it. These tasks are tedious to perform manually and prone to errors.

The signals we've described so far are all examples of *behaviors*, which mean they have a value at every point in time after their creation. Behaviors correspond naturally to the values of many GUI widgets. For example, a gauge renders a time-varying integer, and a message displays a time-varying string. Likewise, a slider lets the user manipulate a time-varying integer, and a text field lets him edit a time-varying string.

Signals may also take the form of *event sources*, which carry streams of discrete values called *occurrences*. For example, FrTime's animation library provides event sources called *mouse-clicks* and *key-strokes*, which carry the mouse clicks and key strokes captured by a given window. FrTime also provides a collection of event-processing combinators that are analogous to list-processing routines. For example, *filter-e* removes unwanted occurrences from an event stream, while *map-e* transforms each value by applying a given function.

```
(define frame (new ft-frame% [label "Timer"] [width 200] [height 80] [visible true]))
(define slider (new ft-slider% [label "Duration"] [min-value 15] [max-value 60])
(define button (new ft-button% [label "Reset"])
(define duration (send slider get-value-b) )
(define last-click-time ;; initially holds application's start time
  (hold (map-e (send button get-clicks) (λ (_) (value-now seconds)))
        (value-now seconds)))
(define elapsed (min duration (− seconds last-click-time)))
(define gauge (new ft-gauge% [label "Elapsed time:"] [range duration]
                             [parent frame] [value elapsed] ))
(define message (new ft-message% [label (format "~a s" elapsed)]
                                 [parent frame] [min-width 50]))
```

Fig. 2. Implementation of the timer in FrTime

FrTime provides several primitives for converting between behaviors and event streams. One is *changes*, which consumes a behavior and returns an event source that emits the new value each time the behavior changes. Conversely, *hold* consumes an event source and returns a behavior that reflects the last event occurrence value; *hold* also takes an optional initial value to use until the first event occurs. For example, if a program applies *hold* to *key-strokes*, the result is a behavior whose value indicates the last key pressed.

Widgets like buttons and menu items support interaction through discrete events rather than manipulation of continuous values. Thus they correspond naturally to FrTime event streams instead of behaviors. We can use standard FrTime operators to construct behaviors from these event streams. For example, from a stream of button clicks, we can define a behavior that reflects the time of the last click. We express this by mapping a procedure that projects the current time over the stream of clicks (ignoring the click event's *void* value); *hold*ing the resulting stream yields the time of the last click. We provide *hold* with the program's start time, which is the value until the first click. The code is as follows:

```
(define last-click-time
  (hold (map-e (send button get-clicks) (λ (_) (value-now seconds)))
        (value-now seconds)))
```

This definition plays an important role in the program shown in Fig. 2, which presents a FrTime implementation of the timer using the ideas of this paper. The new version is free of callbacks and imperative method invocations. Instead, input widgets like the slider and button provide behaviors and events that reflect the user's interactions, and output widgets like the gauge and message allow the application to provide behaviors that specify property values for their entire lifespan. We draw a box around code that participates in the interface between signals and widgets.

4 Adapting MrEd to FrTime

In Sect. 2 we introduced MrEd, an object-oriented toolkit for building GUIs, and presented a simple example to illustrate some of the difficulties imposed by the standard GUI programming model. In Sect. 3 we presented FrTime, a language that extends DrScheme with support for first-class signals, and we showed how this new feature provides a suitable abstraction for modeling change, which is an important problem in interactive GUI applications. In this section, we put the pieces together and show how to adapt MrEd so that its interface is based on FrTime's behaviors and events.

Recall that we are trying to import a large legacy class framework in a manner consistent with the goals set forth in the Introduction. We wish to reuse the existing 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 main problem we must address is how to communicate state changes 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.

4.1 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. Most importantly, there is at most one signal associated with any given widget property, so there is no 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, then this would seem to be the "safe" order in which to do things, assuming that the application's data dependencies reflect similar dependencies in the toolkit.

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.

4.2 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 $send\text{-}event_e$.

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)))
...
```

4.3 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.

4.4 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.

5 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.

5.1 Parameterized Class Extensions

In Sect. 4 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* [2, 9], 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.

5.2 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. 4, 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. 4 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* [10] 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.

5.3 Language Dependencies

The abstractions we have presented depend heavily on two features of DrScheme: mixins and macros. These features have analogs in other languages. For instance, Smalltalk has recently seen a series of work on *traits* [15], which are similar to mixins and sufficient for our needs. Similarly, C++ programmers have long used the template facility to parameterize classes [17]. Likewise, hygienic macro systems have been defined for a variety of languages including C [18], Java [1] and Haskell [16]. In addition, the use of macros here could potentially also be simulated using features such as meta-classes.

6 A Spreadsheet Application

To evaluate our adapted version of MrEd, we have applied it to a realistic spreadsheet application. The major challenges in building a spreadsheet, in our experience, are implementing a language with its dataflow semantics and managing a large array of cells. Fortunately, FrTime makes the linguistic problem relatively straightforward, since we can reuse its dataflow mechanism to implement update propagation. This leaves the representation and display of the cell grid.

The core of our spreadsheet user interface is an extension of the MrEd *canvas* widget (which we have not discussed so far). A canvas is a region in which the application can listen to key and mouse events and perform arbitrary drawing operations. We render the cell content into a canvas and process 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 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, we would need to manage all of the recomputation 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. As a result, the code is largely a functional specification of the model of the spreadsheet; this is absent in a traditional object-oriented implementation because of the structural inversion introduced by the use of callbacks.

As an illustrative example, consider the following snippet of code, which shows the definition of a text field into which the user can type 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.

7 Related Work

The use of dataflow in a GUI toolkit has been well-studied. The Garnet [11] and Amulet [12] projects were two early C++ toolkits that included a notion of dataflow. More recently, the FranTk [14] system adapted the Tk toolkit to a programmer interface based on the notions of behaviors and events in Fran [6]. 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 [4] explored the idea of purely functional user interfaces, implementing a signal-based programming interface atop the Swing [5] toolkit.

All of the 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. Thus, to the best of our knowledge, ours is the first work to address this problem.

8 Conclusions and Future Work

We have explored the problem of adapting a legacy object-oriented GUI toolkit to an interface based on the concepts of behaviors and events from functional reactive programming. The key to this adaptation is understanding the direction in which various state changes flow: from the application to the toolkit, the toolkit to the application, or both ways. This depends upon the particular widget property that we are adapting.

Since there are many widget properties, many of which are common to many widgets, the implementation would ordinarily require a large amount of code duplication. However, in Scheme, we are able to distill the adaptation to its most abstract essence. We express this as a set of three macros, which are parameterized over the names of the

fields and methods that implement the various properties. We instantiate these macros to produce a collection of mixins—class fragments parameterized over their superclasses. By applying these to the base widget classes, we implement the full interface adaptation to our functional reactive language.

There are two main directions for future work, which complement each other. First, we plan to continue developing the spreadsheet beyond its current research-prototype stage and also to pursue different kinds of applications. This will help us to evaluate the FrTime language and our adaptation of the MrEd GUI toolkit. Second, new applications are likely to require the importation of other legacy frameworks, which will serve to validate the techniques presented in this paper and also to suggest refinements to them. As we co-opt more libraries, we expect FrTime to become an increasingly powerful platform for application development.

References

1. J. Bachrach and K. Playford. The Java syntactic extender. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, pages 31–42, 2001.
2. G. Bracha and W. Cook. Mixin-based inheritance. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, pages 303–311, 1990.
3. G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *European Symposium on Programming*, 2006.
4. A. Courtney and C. Elliott. Genuinely functional user interfaces. In *Haskell Workshop*, 2001.
5. R. Eckstein, M. Loy, and D. Wood. *Java Swing*. O'Reilly, 1997.
6. C. Elliott and P. Hudak. Functional reactive animation. In *ACM SIGPLAN International Conference on Functional Programming*, pages 263–277, 1997.
7. 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.
8. 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.
9. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183, 1998.
10. E. E. Kohlbecker Jr. *Syntactic Extensions in the Programming Language Lisp*. PhD thesis, Indiana University, 1986.
11. 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.
12. 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.
13. H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *ACM SIGPLAN Workshop on Haskell*, pages 51–64, 2002.
14. M. Sage. FranTk: A declarative GUI language for Haskell. In *ACM SIGPLAN International Conference on Functional Programming*, pages 106–117, 2000.
15. N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behavior. In *European Conference on Object-Oriented Programming*, pages 248–274, 2003.

16. T. Sheard and S. P. Jones. Template meta-programming for haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 1–16, 2002.
17. M. VanHilst and D. Notkin. Using C++ templates to implement role-based designs. In *International Symposium on Object Technologies for Advanced Software*, pages 22–37, 1996.
18. D. Weise and R. Crew. Programmable syntax macros. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 156–165, 1993.

Appendix: 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.