Genuinely Functional User Interfaces

Antony Courtney [1,2]

Dept. of Computer Science
Yale University
New Haven, CT 06520

Conal Elliott [3]

Microsoft Research
One Microsoft Way
Redmond, WA 98052

Abstract

Fruit is a new graphical user interface library for Haskell based on a formal model of user interfaces. The model identifies *signals* (continuous time-varying values) and *signal transformers* (pure functions mapping signals to signals) as core abstractions, and defines GUIs compositionally as signal transformers. In this paper, we describe why we think a formal denotational model of user interfaces is useful, present our model and prototype library implementation, and show some example programs that demonstrate novel features of our library.

1 Introduction

Over the years, there have been numerous Graphical User Interface (GUI) libraries for Haskell, presenting a broad range of different programming interfaces. Some libraries, such as TkGofer [26], provide direct access to GUI facilities through the `IO` monad, and therefore have a rather imperative feel. Others, such as Fudgets [4] and FranTk [23], present qualitatively more high-level programming interfaces, so have a more declarative, functional feel.

[1] This material is based upon work supported in part by a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation.

[2] Email: `antony@apocalypse.org`
[3] Email: `conal@microsoft.com`

But what does it mean for one library to be more "high level" or "low level" or "functional" than another? On what basis should we make such comparisons? A pithy answer is given by Perlis [18]:

> A programming language is low level when its programs require attention to the irrelevant.
> –*Alan Perlis*

But how should we decide what is relevant?

Within the functional programming community, there is a strong historical connection between functional programming and *formal modeling* [1,24,25,12]. Many authors have expressed the view that functional programming languages are "high level" because they allow programs to be written in terms of an *abstract conceptual model* of the problem domain, without undue concern for implementation details.

Of course, although functional languages *can* be used in this "high level" way, this is neither a requirement nor a guarantee. It is very easy to write programs and libraries in pure Haskell that are littered with implementation details and bear little or no resemblance to any abstract conceptual model of the problem they were written to solve.

So if we wish to design a high-level interface for implementing GUIs in Haskell, it seems clear that we must first ask:

> What is an abstract conceptual model of a graphical user interface?

and then embed this model in Haskell, so that there is a direct mapping from the types and functions in our library to their counterparts in the conceptual model. As far as we are aware, all previous GUI libraries for Haskell define the conceptual model of a GUI only informally, or defer to some external system (such as X Windows, Tk, Gtk, etc.) for many of the details.

In this paper, we present *Fruit*, a Functional Reactive User Interface Toolkit, based on a formal model of graphical user interfaces. The Fruit model is is based on *AFRP*, an adaptation of ideas from Functional Reactive Animation (Fran) [9,7] and Functional Reactive Programming (FRP) [14,27] to the *arrows* framework recently proposed by Hughes [15]. AFRP is based on two ideas: *signals*, which are functions from real-valued time to values, and *signal transformers*, which are functions from signals to signals. Using only the AFRP model and simple *mouse*, *keyboard* and *picture* types, we define GUIs compositionally as signal transformers.

We believe that developing a simple, precise denotational model of graphical user interfaces is valuable for a number of reasons:

- It provides a starting point for proving properties of programs with graphical interfaces, and for developing related notions of *program equivalence.*
- We can reformulate the question of whether one library is more "high level" than another in precise, objective terms, by comparing the semantic models of the libraries, and asking whether one semantic model is *more abstract*

than another [22].

- By clarifying our understanding of the abstract conceptual model of graphical interfaces, we may gain fresh insight into how to extend the model to new interaction paradigms or see systematic solutions to problems that were previously solved in an ad hoc manner.

We present examples of this final point later in the paper, when we show how continuous spatial scaling (zooming) and multiple views can be accomodated within the Fruit model.

The remainder of this paper is organized as follows. In section 2 we formally define the AFRP programming model, show how the model is embedded in Haskell, and give simple but precise definitions for some useful combinators and primitives. In section 3 we define GUIs within the AFRP model. In section 4, we develop a basic application in Fruit, and show two examples (adding continuous spatial scaling and multiple views) that demonstrate the benefits of our approach. Section 5 discusses related work. Sections 6 and 7 summarize the status of the implementation and present our conclusions.

2 AFRP Programming Model

Like Fran and FRP, the AFRP programming model is implemented as a domain-specific language embedded in Haskell [13]. In order to focus on our new language constructs, we simply assume the existence of a denotational semantics for Haskell in which Haskell functions denote (partial) functions. We define our language extensions by giving denotational definitions for our language constructs that extend this (hypothetical) Haskell semantics.

2.1 Concepts

The Fruit programming model is built around two central concepts: *signals* and *signal transformers*.

Signals

A *signal* is a function from *time* to a value:

$$\textbf{Signal}\ \alpha \quad = \quad \mathit{Time} \rightarrow \alpha$$

We represent *Time* as a non-negative real number. An example of a signal is the mouse's current (x, y) position. If *Point* is the type of two-dimensional points, we can model the time-varying mouse position as a *Signal Point*.

Signal Transformers

A *signal transformer* is a function from *Signal* to *Signal*:

$$\textbf{ST}\ \alpha\ \beta \quad = \quad \mathit{Signal}\ \alpha\ \rightarrow\ \mathit{Signal}\ \beta$$

Informally, we can think of a signal transformer as a box with an "input port" and an "output port". If we connect the box's input port to a *Signal* α value, we can observe a *Signal* β value on the output port.

A simple example of a signal transformer is the identity signal transformer. At every point in time, the identity signal transformer's output signal has the same value as its input signal. Slightly more interesting examples of signal transformers are *lifted functions* (the output signal is the point-wise application of f to the input signal), and `integral` (the output signal is the integration of the input signal over time). Note that the identity signal transformer can be defined as a lifted function, where f is the function *id*.

2.2 Abstract Types

Conceptually, signals are functions of continuous time, and signal transformers are functions from signals to signals. As has been argued elsewhere [9,7], a *continuous* model can be simpler and more natural than a discrete one when modeling animation or user interaction. However, in order to guarantee an efficient implementation on a discrete computer, signal transformers are not written directly as Haskell functions. Instead, we introduce an *abstract type constructor*, `ST`. A value of type `ST a b` denotes a signal transformer:

```
newtype ST a b = ...
```

The implementation provides a number of *primitive* signal transformers, and a set of combinators (the arrow combinators) for assembling new signal transformers from existing ones. Internally, the implementation uses discrete sampling and synchronous streams to approximate the continuous time model. It has been shown that, as the time between samples approaches zero, the discrete implementation converges to the continuous semantics in the limit [27].

Since `ST` is a Haskell type constructor, signal transformers are first-class values: we can pass them as arguments, return them as results, store them in data structures or variables, etc. In contrast, signals are not first-class values. This marks a significant departure from Fran's programming model. Fran's `Behavior` type denotes a signal in the Fruit model, and Fran uses Haskell functions to obtain the equivalent of our signal transformers.

We outlaw signals as first-class values for two reasons. First, signals alone are inherently non-modular: while we can apply point-wise transformations to the observable *output* of a signal, first-class signal values do not have an input signal. In contrast, signal transformers have both an input signal and an output signal, thus enabling us to transform both aspects of an `ST` value. In other words, only providing `ST` as first-class values guarantees that every signal in the program is always *relative* to some input signal.

Second, experience implementing Fran [6] and FRP [14] taught us that allowing signals as first class values inevitably leads to "space-time leaks" [6] in the implementation. A "space-time leak" occurs when the implementation needs the complete time-history of a signal to compute one sample value.

Defining `ST` as a `newtype` and only providing a fixed set of primitives and combinators allows us to prove that the implementation is free of space-time leaks by simple structural induction on the `ST` type.

2.3 Arrows

Recently, Hughes proposed *arrows* as a basis for building combinator libraries [15]. Concretely, *Arrow* is a Haskell type class that denotes a *computation* from some input type to some result type. In his introduction to arrows [15], Hughes presents a number of examples of arrow instances, including Haskell's built-in function type constructor `(->)` and stream processors, and gives many examples that demonstrate the utility of arrows for organizing combinator libraries.

The Arrow type class is defined as:

```
class Arrow a where
  arr :: (b -> c) -> a b c
  (>>>) :: a b c -> a c d -> a b d
  first :: a b c -> a (b,d) (c,d)
```

In the remainder of this section, we give both informal and formal definitions of each of the arrow operators for the `ST` type constructor in terms of our model.

Lifting

One of the most common and useful kinds of signal transformers is a *lifted* function, produced by the `arr` operator. The `arr` operator for signal transformers has type:

```
arr :: (b -> c) -> ST b c
```

Given any Haskell function `f` of type `(b -> c)` (i.e. a function mapping b values to c values), `arr f` denotes a signal transformer that maps a *Signal* b to a *Signal* c by applying `f` point-wise to the input signal.

For example, given the function

```
sin :: Floating a => a -> a
```

from the standard Prelude, `arr sin` is a signal transformer whose output signal at time t is `sin` applied to the signal transformer's input signal at time t.

Formally, we define `arr f` for signal transformers as follows:

$$[\![\texttt{arr f}]\!] \;=\; \lambda s : \textbf{Signal}\ \alpha\ .\ \lambda t : \mathit{Time}\ .\ [\![\texttt{f}]\!](s(t))$$

Serial Composition

The arrow infix operator `(>>>)` composes two arrows. For signal transformers, if we have:

```
fa :: ST b c
```

```
ga :: ST c d
```

then `fa >>> ga` has type `ST b d` and denotes the signal transformer that feeds its input signal to `fa`, uses `fa`'s output signal as `ga`'s input signal, and uses `ga`'s output signal as the resulting output signal.

Formally, we define serial composition for signal transformers as reverse function composition:

$$[\![\texttt{fa >>> ga}]\!] = ([\![\texttt{ga}]\!] \circ [\![\texttt{fa}]\!])$$

Widening

Given an arrow from `b` to `c`, the `first` operator widens it to be an arrow from `(b,d)` to `(c,d)`, for all types `d`. For signal transformers, the `first` operator has type:

```
first :: ST b c -> ST (b,d) (c,d)
```

Informally, `(first fa)` denotes a signal transformer that feeds the first half of its input signal (a signal of `b` values) to `fa` to produce a signal of `c` values, and pairs this with the second half of the original input signal (a signal of `d` values) to produce the output signal.

Formally, we can define `first` as:

$$[\![\texttt{first fa}]\!] =$$

$$\lambda s : \mathit{Signal}(\beta \times \gamma)\ .\ \texttt{pairZ}\ ([\![\texttt{fa}]\!]\ (\texttt{fstZ}\ s))\ \ (\texttt{sndZ}\ s)$$

where `fstZ`, `sndZ` and `pairZ` are the obvious projection and pairing functions for signals of pairs.

ArrowLoop

The names “signal” and “signal transformer” in our model suggest analogies to analog and digital signal processing and computer hardware. In those domains, *feedback cycles* are used in conjunction with the inherent propagation delay of wires to implement many interesting circuits such as flip-flops or latches. In a feedback cycle, some portion of the output signal is *fed back* as an input signal. Feedback cycles are also useful in Fruit, and are defined using the `loop` combinator [16]. The `loop` combinator is defined in the `ArrowLoop` type class:

```
class Arrow a => ArrowLoop a where
          loop :: a (b,d) (c,d) -> a b c
```

For signal transformers, if `fa` has type `ST (b,d) (c,d)`, then `loop fa` denotes a signal transformer that instantiates `fa`, and pairs the second half of `fa`'s output signal with an external input signal to form `fa`'s input signal.

Formally, we define `loop` for signal transformers as:

$$[\![\texttt{loop fa}]\!] \ =$$

$$\lambda s : \textbf{Signal}\ \beta.\ \texttt{fstZ}(\mathbf{Y}(\lambda r.[\![\texttt{fa}]\!](\texttt{pairZ}\ s\ (\texttt{sndZ}\ r))))$$

where **Y** is the standard least fixed point operator.

Discrete Events

In modeling reactive systems in general (and user interfaces in particular), we often need to model *event sources* that produce *event occurrences* at discrete points in time. For example, the left mouse button being pressed is naturally modeled as an event that occurs at some point in time. For now, we define event sources in our conceptual model as signals of `Maybe` values: [4]

$$\begin{aligned}\textbf{EventSource}\ \alpha &= \textbf{Signal}\ (\texttt{Maybe}\ \alpha)\\ &= \mathit{Time}\ \rightarrow\ (\texttt{Maybe}\ \alpha)\end{aligned}$$

If the value of an event source at time t is `Nothing`, then we say that the event source does not occur at time t. Conversely, if the value of an event source at time t is `Just v`, then we say that the event source has an occurrence at time t that carries value `v`.

As with *Signal*, *EventSource* lives in the conceptual model, and does not appear directly in our API. However, `Maybe` types appear as arguments to the `ST` type constructor when event sources are needed. We will see an example of this shortly.

2.4 Primitive Signal Transformers

Fruit defines a number of primitive signal transformers. Each such primitive has a denotational definition in terms of our formal model. The denotational definitions of these primitives are derived directly from their counterparts in Fran and FRP. We define a couple of these primitives here to give a taste of the semantics. The interested reader is referred to the denotational definitions of Fran and FRP semantics for a more complete account [9,7,27].

Piecewise Constant Signals

Given an event source, it is often useful to derive a continuous signal whose value is constant between event occurrences. This is sometimes called a "sample and hold" or "zero-order hold" circuit in the signal processing literature. The primitive `stepper` is provided for this purpose:

```
stepper :: a -> ST (Maybe a) a
```

Informally, `stepper x0` denotes a signal transformer that transforms an *EventSource* α to a *Signal* α. Initially, the output signal of `stepper x0` has value `x0`. When an event carrying value `x1` occurs on its input signal,

[4] This representation of events as continuous signals of `Maybe` values raises some thorny theoretical issues because it allows for *dense* event sources (ones that have infinitely many occurrences in a finite interval of time). We have explored some possible solutions to this problem [27], but a detailed exploration of this issue is outside the scope of this paper.

the value of the stepper's output signal becomes `x1`. The value of the output signal remains `x1` until the next event occurrence (carrying, say, `x2`), at which point the value of the output signal becomes `x2`, and so on.

Formally, we define `stepper` as follows:

$$[\![\texttt{stepper x}]\!] = \lambda s.\lambda t. \begin{cases} \texttt{a} \ \exists \texttt{a}.\exists t_a \in (0,t).((s\ t_a) = \texttt{Just a}) \\ \qquad \wedge\ \nexists t_b \in (t_a,t).((s\ t_b) = \texttt{Just b}) \\ \\ \texttt{x}\ \mathit{otherwise} \end{cases}$$

Integration

The primitive signal transformer `integral` has type:

```
integral :: Floating a => ST a a
```

The output signal of `integral` is the integration of its input signal over time. Formally:

$$[\![\texttt{integral}]\!] \ = \ \lambda s.\lambda t. \int_0^t s(t)dt$$

3 Fruit: A Compositional User Interface Library

We define an interactive graphical user interface (`GUI`) as:

```
type GUI a b = ST (GUIInput, a) (Picture, b)
```

A `GUI a b` is a signal transformer that takes a graphical input signal (`GUIInput`) paired with an auxiliary semantic input signal (`a`) and produces a graphical output signal (`Picture`) paired with an auxiliary semantic output signal (`b`).

In the Fruit model, every interactive component is a value of type `GUI a b` (for some types `a` and `b`). This differs from conventional toolkits in which there are distinct types for "applications", "containers" and "components". We consider this flat type structure a feature, as it leads to a *compositional* model of user interfaces. Any two GUI values can be composed using our layout combinators to form a composite GUI in which the two child GUIs appear adjacent to one another. The result returned from the layout combinator is itself a GUI, and so can also be used in a layout combinator, displayed in a top-level window, etc.

The `GUIInput` and `Picture` signals allow the application to feed time-varying keyboard and mouse information into the GUI, and get back time-varying visual information (pictures to display). The auxiliary input and output signals allow a GUI to observe and emit extra (time-varying) information for use in the rest of the program. For example, a GUI representing a button component might provide an event source output signal that has occurrences when the button is pressed.

The `Picture` type is an abstract type that denotes a static picture that can be rendered on screen. Our prototype implementation uses a scalable vector graphics library based loosely on the graphics library defined in "The Haskell School of Expression" (SOE) [14], but any picture type that supports basic geometric primitives, bounds calculations and affine transforms would work.

3.1 The GUIInput Type

The `GUIInput` type represents the part of the input to a GUI specifically related to its visual interactive characteristics. `GUIInput` is essentially just a pair of records:

```
data Mouse = {mpos :: Point,
              lbDown :: Bool,
              rbDown :: Bool }
data Kbd = { keyDown :: [Char] }

type GUIInput = (Maybe Kbd, Maybe Mouse)
```

The `Kbd` and `Mouse` types are wrapped in `Maybe` types to account for the *focus model*. In modern window systems, there is always a foreground application that receives the keyboard and mouse input from the window system to the exclusion of all other applications running in the background. The window system typically provides a lightweight gesture (such as mouse-over or click-to-type) that allows the user to shift the focus to another application. This concept of focus model is equally applicable within a window, as we can view moving the mouse between two different visible components of a window as shifting the mouse focus from one component to the other. Keyboard focus traversal within a window (using the TAB key, for example) can be modeled analogously.

Each of the `Maybe` values in the `GUIInput` signal to a GUI are `Nothing` when the GUI does not have focus, and `Just` x (for some x) when the component has the focus. Note that, although the types of these signals are the same as a discrete event source, they are, conceptually, *not* discrete event sources. As it turns out, however, many of the event source combinators also have useful interpretations for such continuous `Maybe` signals. We will see several examples of this.

3.2 Composing GUIs

These definitions, combined with the arrow combinators and primitive behaviors from the previous section, form the basis of our GUI library. Even without any additional definitions, we can define many useful and interesting richly interactive user interfaces.

For example, we can define `mouseST` as a signal transformer that takes a `GUIInput` input signal and produces a `Point` output signal that is the mouse's current position if the GUI has the focus, or "remembers" the last position that had focus otherwise:

```
mouseST :: ST GUIInput G.Point
mouseST = arr snd >>> arr (fmap mpos)
                  >>> stepper G.origin2
```

The `G` refers to the qualified import of the graphics library; `G.origin2` is the Cartesian origin. Note that we are feeding the `Maybe Mouse` signal to the `stepper` event source combinator. The result is a continuous signal that maintains the last position of the mouse when the GUI loses mouse focus.

Using this definition, here is a definition for a GUI that draws a red ball that follows the mouse:

```
-- from the graphics library:
move :: Picture -> Point -> Picture

ballPic :: Picture
ballPic = (circle `withColor` red)

ballGUI :: GUI () ()
ballGUI = first (mouseST >>> arr (move ballPic))
```

In the above definition, `ballGUI` is given type `GUI () ()` because it neither observes nor produces any semantic signals other than its `GUIInput` input signal and its `Picture` output signal. The subexpression `(mouseST >>> arr (move ballPic))` has type `ST GUIInput Picture`. By using the `first` operator, we widen this `ST` value to one of type `ST (GUIInput,a) (Picture,a)` for all types a, and this generalized type is of course equivalent to `GUI a a`.

3.3 Running a GUI

A GUI is brought to life with the `runGUI` action, which runs a GUI in a top-level window: [5]

```
runGUI :: Unit a => GUI a b -> IO ()
```

The implementation of `runGUI` handles all low-level (imperative) communication with the graphics library to read primitive window events and draw pictures on the screen. The window displayed by the action `runGUI ballGUI`

[5] For convenience, we define a type class `Unit` with an instance for `()` and all products of `Unit`, such as `()`, `((),())`, etc. This will be convenient for simple `GUIs` composed with layout combinators, as we shall see later.

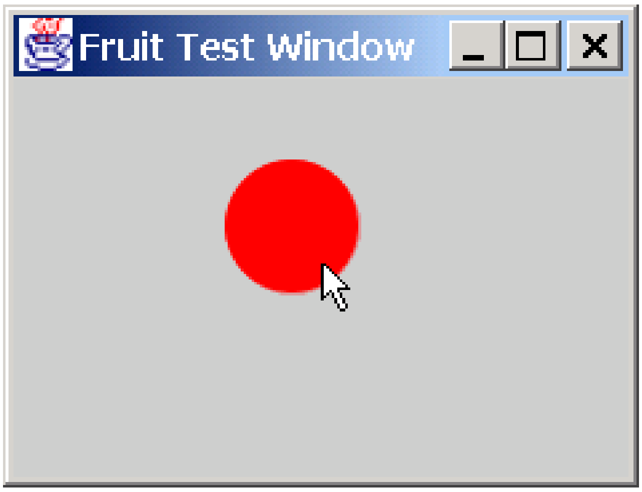

Fig. 1. Running ballGUI

is shown in figure 1.

3.4 Brief Aside: Arrows Syntactic Sugar

When defining signal transformers using the arrow combinators, we must write definitions in a *point-free* style. In the context of Fruit, this means that the names in our program refer to *signal transformers*, but we cannot name *signals* explicitly.

The arrows syntactic sugar is a proposal by Ross Paterson [16] with an existing implementation as a preprocessor. The arrows syntactic sugar allows arrows to be defined using a new syntactic form, introduced by the keyword `proc`. The `proc` form acts as a kind of *abstraction* for arrows, analogous to Haskell's built-in lambda abstraction. Within the body of a `proc`, the arrows syntactic sugar allows us to explicitly name the *signals*, and specify how the signals are connected within a signal transformer.

As with lambda abstraction, `proc` takes a *pattern* that will be matched *point-wise* over the points of the arrow. The identifiers used in the pattern may then be used within the body of the arrow. For example, we could have defined `mouseST` using the arrows syntactic sugar as:

```
mouseST :: ST GUIInput G.Point
mouseST = proc (_, mbm) -> do
  ... -- not shown (yet)
```

In this definition, the pattern `(_,mbm)` is matched against the *input type* (`GUIInput` here).

Informally, the body of an arrow definition consists of a sequence of *arrow applications* of the form:

$$
\begin{array}{rcl}
pat_1 & \xleftarrow{st_1}\!\!\prec & exp_1 \\
pat_2 & \xleftarrow{st_2}\!\!\prec & exp_2 \\
 & \cdots & \\
pat_{n-1} & \xleftarrow{st_{n-1}}\!\!\prec & exp_{n-1} \\
 & \xrightarrow{st_n}\!\!\prec & exp_n
\end{array}
$$

Each such arrow application feeds the signal described by exp_i to the signal transformer st_i. Each pat_i is matched against the output type of st_i, and introduces new *arrow-bound variables* for use in the arrow applications that follow. The final such application (which does not include a pattern) defines the output signal of the entire `proc`. Note that, in the conversion to the ASCII character set, $\xleftarrow{st}\!\!\prec$ is written as `<- st -<`. Our complete definitions for `mouseST` and `ballGUI` using the arrows syntactic sugar are thus:

```
mouseST = proc (_, mbm) -> do
  stepper G.origin2 -< fmap mpos mbm

ballGUI :: GUI () ()
ballGUI = proc (gin,_) -> do
  mouse <- mouseST -< gin
  returnA -< (move ballPic mouse,())
```

where `returnA` is defined as `arr id` in the arrows library.

The subset of the arrows syntactic sugar used in this paper is defined formally by the following translation:

```
proc p -> do { e1 -< e2 } =
  arr (\p -> e2) >>> e1

proc p -> do { p' <- e1 -< e2; A } =
  returnA &&& arr (\p -> e2) >>> second e1 >>>
  proc (p,p') -> do {A}

proc p -> do { let p' = e; A } =
  returnA &&& arr (\p -> e) >>> proc (p,p') -> do {A}

proc p -> do { rec {A}; B } =
  returnA &&& loop proc (p,pA) -> do
    { A; returnA -< (pB,pA)} >>> proc (p,pB) -> do { B }
```

3.5 Simple Components

A GUI's *auxiliary semantic* input and output signals convey semantic signals to and from the GUI not directly related to the GUI's visual behavior, and enable the GUI to be connected to the rest of the program. The Fruit library defines a number of GUI components that use these auxiliary signals.

Labels

The simplest components are labels, defined as:

```
flabel :: GUI LabelConf ()

ltext :: String -> LabelConf
```

A label is a GUI whose picture displays a text string from its auxiliary input signal, and produces no semantic output signal.

We use a trick from Fudgets [4] to specify configuration options. `LabelConf`, `ButtonConf`, etc. are simple *State* → *State* functions. These functions are very similar to the update functions generated by using Haskell's labeled field syntax, in that they will update one component of the state, but leave all others unchanged. This gives us a simple mechanism for composing property definitions (using the function composition operator '`.`') and for assigning default values for component properties. We will see an example of this shortly.

Buttons

A Fruit button (`fbutton`) is a GUI that implements a standard button control. The declaration of `fbutton` is:

```
fbutton :: GUI ButtonConf (Maybe ())

btext :: String -> ButtonConf
enabled :: Bool -> ButtonConf
```

This declares `fbutton` as a GUI that, in addition to its visual input and output signals, takes an input signal of configuration options specifying properties of the button such as the label to display in the button, whether the button is enabled, etc. The button produces an output event source that has an occurence when the button is pressed by the user. Each event occurence on the output signal carries no information other than the fact of its occurence, hence the type `Maybe ()`. Here is an example of a GUI that uses an `fbutton`:

```
butTest :: GUI () (Maybe ())
butTest = proc (inpS,_) -> do
  fbutton -< (inpS,btext "press me!")
```

The display produced by `runGUI butTest` is shown in figure 2. Note that the above example did not need to explicitly specify the button's enabled property (which is `True` by default).

Fig. 2. A simple button

Fig. 3. Using besideGUI

3.6 Basic Layout Combinators

To be able to build more interesting interfaces, we need a mechanism to compose multiple GUIs into a larger GUI. We provide two basic *layout combinators* for this purpose:

```
aboveGUI :: GUI b c -> GUI d e -> GUI (b,d) (c,e)
besideGUI :: GUI b c -> GUI d e -> GUI (b,d) (c,e)
```

The layout combinators produce a combined GUI that behaves as the two child GUIs arranged adjacent to one another. Here is a small example that illustrates the use of `besideGUI`:

```
hello :: GUI () (Maybe (),())
hello = proc (inpS,_) -> do
  (fbutton `besideGUI` flabel) -<
        (inpS, (btext "press me",
                ltext " PLEASE! "))
```

The result of running this GUI in a top-level window with `runGUI` is shown in figure 3. A *translation transformation* has been applied to the second argument GUI to position it beside the first argument. The implementation of spatial transformation for GUIs will be described in detail in section 4.4.1.

In addition to transforming the second argument, the layout combinators must *demultiplex* the input signal into two disjoint signals to be passed to each child. This is achieved by *clipping* the `GUIInput` signal based on the mouse position: The GUI under the mouse receives the (appropriately transformed) keyboard and mouse signals, while its sibling receives `Nothing` values for the keyboard and mouse to indicate that it does not have focus. [6]

As this example illustrates, the composed GUI has auxiliary semantic input and output signals whose types are the product of the corresponding types

[6] Our current implementation of focus is based solely on mouse position. This is slightly simplistic, as modern user interface guidelines stipulate a keyboard focus cycle that is independent of the mouse focus. Extending our implementation to support such a split focus model is straightforward.

from the child GUIs. This has substantial syntactic consequenses. Programs can become complicated rather quickly, because the types of the composed GUI grow in proportion to the nesting depth of the layout. We have written numerous small example programs using our layout combinators without the arrows syntactic sugar, and have found the resulting programs to be a hopelessly unreadable mess of lifted tupling and untupling. We were exploring possible GUI-specific syntactic extensions when we encountered the arrows syntactic sugar proposal. We have been pleasantly surprised by just how well the syntactic sugar works for a specific problem domain (GUIs) for which it wasn't specifically designed.

4 Composing Applications

In this section, we demonstrate Fruit by developing a basic application. The application (a "Paddleball" game with a button for restarting the game) is small enough to allow us to study it in detail, but substantial enough to capture some of the essential issues that arise in building larger applications.

4.1 Paddleball as a GUI

Hudak [14] develops an implementation of a simple Fran-like reactive animation language, and implements "Paddleball in Twenty Lines" as a demonstration of the power and elegance of functional reactive programming. Since the Fruit model subsumes the functional reactive model on which which it is based, it was a simple matter to re-implement paddleball as a `GUI`. The complete source code is shown in figure 4.

A couple of combinators used in `pball` that we have not yet explained are:

```
-- An accumulating stepper: On every event occurence,
-- function carried with the occurence is applied to
-- the state.
stepAccum :: a -> ST (Maybe (a -> a)) a

-- replaces the value in an event occurence with
-- a new value:
ebind :: a -> Maybe b -> Maybe a
ebind = fmap . const
```

Essentially, the code for `pball` does the following:

- Sets up the ball. The ball's x and y position (`xpos` and `ypos`) are defined as the integral of velocity (`xvel` and `yvel`, respectively). The velocities are defined as piece-wise constant signals using `stepAccum`; both start at `vel` (the game velocity given as an argument to `pball`), but flip sign (`negate`) when a bounce event occurs. Note that these definitions are mutually recursive: `xpos` is defined in terms of `xvel`, which is in turn defined in terms

```
paddle :: Double -> G.Rectangle2DDouble
paddle xpos =  G.rectangle (xpos - 25) 200 50 10

-- The paddleball game, capable of playing one game
-- Output signal is an Event Source that occurs
-- when the game ends.
pball :: Double -> GUI () (Maybe ())
pball vel = proc (inpS,_) -> do
  rec xi <- integral -< xvel
      let xpos = 30 + xi
      yi <- integral -< yvel
      let ypos = 30 + yi
      let ballS = ell (xpos-12.5) (ypos-12.5) 25 25
      let ballPicS = G.shapePic ballS `G.withColor`
                              G.yellow
      xbounce <- when -< ((xpos > 175) || (xpos < 30))
      ybounce <- when -< ((ypos < 30) || hitPaddle)
      let hitPaddle = intersects ballS paddleS
      xvel <- stepAccum vel -< ebind negate xbounce
      yvel <- stepAccum vel -< ebind negate ybounce
      mpos <- mouseST -< inpS
      let paddleS = paddle (G.pointX mpos)
      let paddlePicS = G.shapePic paddleS `G.withColor`
                              G.green
      gameOver <- when -< ypos > 200
      let gamePic = G.box
            (walls `G.over` paddlePicS `G.over`
                ballPicS) gameBox
  returnA -< (gamePic,gameOver)
```

Fig. 4. Paddleball GUI source code

of `xbounce`, defined in terms of `xpos`. The `do rec ...` form of the arrows syntactic sugar takes care of setting up the appropriate connections by using the `loop` combinator (from `ArrowLoop`), and the fact that the `integral` of a signal at time t depends on the values of the input signal up to (but not including) time t ensures that the feedback cycle is well-defined.

- Sets up the Paddle: This is just a rectangle shape (`paddle`), whose x position is determined by the mouse position.
- Performs Collision Detection: This is handled by the definitions of `xbounce`, `ybounce` and `hitPaddle`. `hitPaddle` is defined by a call to the graphics library to check for the intersection of the ball (`ballS`) with the paddle (`paddleS`). `xbounce` and `ybounce` are defined using the `when` combinator:

```
when :: ST Bool (Maybe ())
```

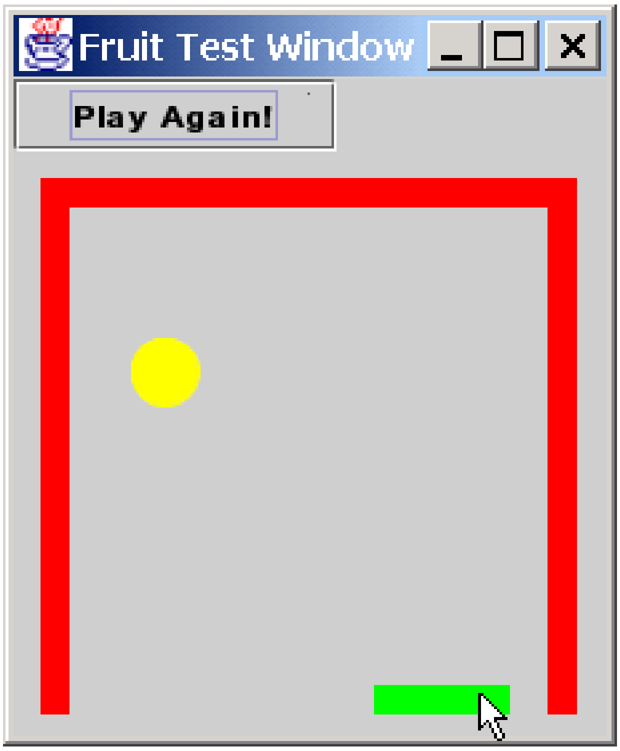

Fig. 5. Paddleball with a Restart Button

> The `when` combinator converts a continuous Boolean signal to an event source. The output event occurs when a *rising edge* is observed on the input signal.

By implementing the Paddleball game as a GUI we obtain spatial modularity (relative to Fran and FRP), and this, in turn, enables reuse: we can use the layout combinators to compose `pball` with other GUIs to form more interesting composite GUIs, and we can have as many Paddleball games active in our GUI as we wish.

4.2 Adding a "Start Over" Button.

Paddleball is a fun game, but, as defined here, it only plays one game. Our first refinement to the game is to add a restart button that allows us to play again, as show in figure 5. We define `rpball0` ("restartable" paddle ball) as follows: [7]

```
-- pbgame is a version of pball that
-- restarts the game when its input
```

[7] We have omitted the code for `pbgame` here to save space. It is easily derived from `pball` using the FRP `switch` combinator, which in `AFRP` has the signature:

```
switch ::  ST b c -> ST (b,Maybe (ST b c)) c
```

See [27] for a detailed discussion of the semantics of `switch`.

```
-- event source has an occurrence:
pbgame :: Double -> GUI (Maybe ()) (Maybe ())

-- paddle ball with a reset button:
rpball0 :: Double -> GUI () ()
rpball0 vel = proc (inpS,_) -> do
  rec (picS,(pressES,_)) <-
         (fbutton `aboveGUI` (pbgame vel)) -<
           (inpS,(btext "Play Again!", pressES))
  returnA -< (picS,())
```

The definition of `rpball0` uses the `do rec...` form of the arrows syntactic sugar to feed the output event source from the reset button (`pressES`) back as an input event source to `pbgame`.

4.3 Selectively Enabling the Reset Button

When implementing GUIs, we frequently need to dynamically disable certain components of the user interface based on the program's state. Components that are disabled are typically rendered with a "grayed out" appearance to give the user a visual cue that the corresponding action is invalid.

We demonstrate this kind of programming in Fruit by disabling the reset button while a game is in progress:

```
-- like rpball0, but selectively disable
-- the restart button:
rpball1 :: Double -> GUI () ()
rpball1 vel = proc (inpS,_) -> do
  rec (picS,(restart,gameEnds)) <-
         (fbutton `aboveGUI` (pbgame vel)) -<
              (inpS,(bprops,restart))
      let bprops = (btext "Play Again!"
                 . enabled allowRestart)
      let gameDone = (ebind True gameEnds)
                 `emerge` (ebind False restart)
      allowRestart <- stepper False -< gameDone
  returnA -< (picS,())
```

The `enabled` property of the button is controlled by the `allowRestart` signal, which is `False` initially, set to `True` when a game ends, and set to `False` again when the `restart` button is pressed. As mentioned in section 3.5, the function composition operator `(.)` is used to combine button properties in the definition of `bprops`.

4.4 Exploring Modularity

Thus far our discussion has simply explored how we can implement user interfaces in a purely functional way. This is certainly an interesting academic exercise, and carries with it (we hope) the benefits of increased reasoning power that we expect from purely functional programming models. But we might be tempted to ask if there are any other purely pragmatic advantages to a purely functional approach? In this section, we explore two such advantages: *continuous spatial scaling* and *multiple views*.

4.4.1 Transforming GUIs

One difference between Fruit and every other production user interface toolkit we are aware of (for either imperative or functional languages) is that Fruit provides a uniform model and programming interface for both "low-level" interactive graphics and "high level" user interface components such as buttons. Moreover, since GUIs are first class values that denote *pure functions*, we can use higher-order operators to manipulate GUIs in useful ways.

One of the most basic higher-order functions is the function composition operator `(.)`; we use `>>>` instead, but the denotation is equivalent. (Recall that `(>>>)` is *reverse* composition, so `f >>> g = g . f` for the function space arrow.) Armed with just this operator, we can define *spatial transformation* of a `GUI`. We will define a generalized `transformGUI` operator that applies an (affine) spatial transform to a GUI to produce a new GUI:

```
transformGUI :: G.Transform ->
                    GUI b c -> GUI b c
```

Assuming that we have a basic understanding of spatial transformation for pictures, how shall we define spatial transformation of a `GUI`?

First, let's quickly review spatial transform for pictures. When we apply a spatial transform to a picture, it changes the size, position, or orientation of the picture. Consider translation of a picture by a displacement vector $(\Delta x, \Delta y)$. In general, this translation maps every (x, y) position in the original image to an (x', y') position in the new image by:

$$(x', y') \;=\; (x \;+\; \Delta x, y \;+\; \Delta y)$$

or, more generally, if *tf* represents the transformation, and `(%$)` is the *apply-transform* operator:

$$(x', y') \;=\; \mathit{tf} \;\texttt{\%\$}\; (x, y)$$

Note that `%$` is defined as part of the *Transformable* type class, so instance declarations may be given for any type that supports spatial transformation.

Since a `GUI`'s visual output is a signal of `Picture`, and our graphics library supports applying affine transforms to `Picture` values, we can transform a GUI's output by point-wise application of the transform to the picture output signal. But what about input?

If `g` is a `GUI`, point-wise transformation of `g`'s picture signal will map every

(x, y) position in g's coordinate system to (x', y'). In order to give an accurate input signal to g, transformGUI must map every (x', y') mouse position back to its corresponding (x, y) position in g. This suggests a general principle for transforming functions: *To transform a function (in time or in space), apply the transform point-wise to the output, and apply the inverse transform point-wise to the input.* This idea corresponds exactly to Pan's spatial "hyper-filters" [8], i.e., spatial transformations of $Image \to Image$ functions.

The implementation of transformGUI is then simply:

```
transformGUI tf g = proc (inp, b) ->
  (pic, c) <- g -< (inverse tf %$ inp, b)
  returnA -< (tf %$ pic, c)
```

This model for transforming GUIs is used in the implementation of the layout combinators to reposition their second argument GUI. The transform to apply to the second argument is determined dynamically by applying a bounds operation point-wise to the Picture signal produced by the first argument GUI.

4.4.2 Spatial Scalability

While our basic layout combinators only use basic horizontal and vertical translations, the transformGUI operator can apply *any* affine transform to a GUI. For example, here is a version of Paddleball that runs in a window 1/2 the size of the original:

```
-- uniform scaling transform (from Graphics library):
uscale :: Double -> Transform

minipb :: Double -> GUI () (Maybe ())
minipb vel =
    transformGUI (uscale 0.5) (pball vel)
```

When run, minipb displays a fully functional version of Paddleball shrunk down to postage stamp size. This type of zooming capability is obviously extremely useful for implementing vector or bitmap graphics editors, document previewers, etc. where zooming is a natural operation. But recent work in the Human/Computer Interaction (HCI) community has proposed continuous zooming can be a useful abstraction in its own right for many applications [17] [2]. Providing continuous zoom allows graphical interfaces to be designed so that users can "zoom out" for an overview of the data and "zoom in" for more detail. Pad [17] and Jazz [2] are two recent research projects that augment the widget set of a traditional imperative GUI toolkit with the abstraction of a continuously zoomable drawing surface.

The starting points for Pad and Jazz were the toolkits Tk and Swing, respectively. Because the Tk and Swing programming interfaces hide their connection with the graphics subsystem, Pad and Jazz are essentially new GUI toolkits, and require that existing applications be rewritten from scratch to take advantage of the zooming capabilities. In contrast, Fruit makes the connection to the interactive graphics subsystem seamless and explicit in the type of `GUI`. As `minipb` demonstrates, this explicit connection to interactive graphics allows us to incorporate novel ideas (such as continuous zooming) without a major restructuring of the library or completely rewriting applications.

4.5 Adding Multiple Views

Many GUI-based applications need multiple *views* on to the same underlying data set. For example, an icon editor might allow the user to open two views of the icon, one showing an editable, highly zoomed-in view where each pixel is a large square, and the other showing a preview of the icon at normal size. As the icon is edited in the zoomed view, the preview view should be updated.

We can really distinguish two kinds of views: *passive* and *active*. A *passive* view observes the underlying data set, but does not provide any means for interacting directly with the data set. The preview window of the icon editor just described is an example of such a passive view. In contrast, an *active* view is interactive: user actions in either view are reflected in other views and the underlying data set.

The requirement for multiple views is so common in user interfaces that the Model-View-Controller (MVC) design pattern has emerged as a way to structure imperative object-oriented programs to support multiple views when using imperative GUI toolkits [11].

For many practical applications (such as icon editors, illustration programs, etc.), the multiple views provided by the application are views of the same underlying (time-varying) picture, with different affine transformations applied to produce the view. For example, a zoomed-in view of an icon is a view of the same picture as a zoomed-out view; the pictures differ only by a scaling transformation. For simple cases such as this, multiple views may be added in Fruit to *any* GUI, without any pre-meditation on the part of the original GUI programmer.

Passive Views

A view can be though of as "a `GUI` with no mind of its own", as shown in figure 6. A view obtains its `Picture` signal from some external source and delivers its `GUIInput` signal to some external source. Concretely, a `view` is a `GUI` that takes a `Picture` signal as its auxiliary semantic signal, and uses this signal as its own `Picture` signal. Similarly, it delivers its `GUIInput` signal as its auxiliary output signal. This describes a simple crossover configuration that leads to the following definition:

```
view :: GUI G.Picture GUIInput
view = arr swap
```

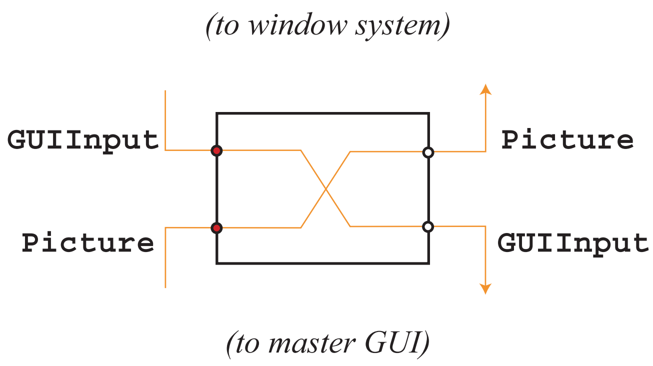

Fig. 6. Implementation of a view

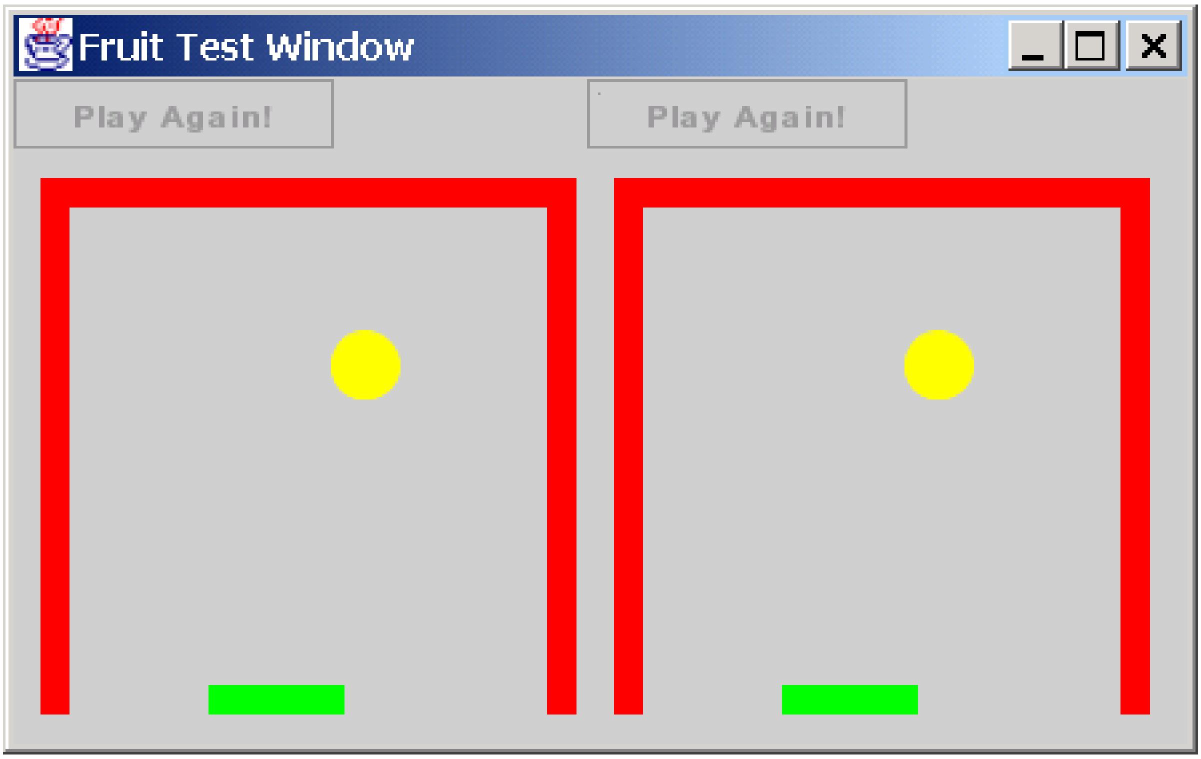

Fig. 7. Multiple Views

Given this definition, we can implement a version of Paddleball that has two views next to each other, as shown in figure 7:

```
pbview :: Double -> GUI () ()
pbview vel = proc (inpS,_) -> do
  rec (picS,(activeIn,_)) <-
        (view `besideGUI` view) -<
          (inpS,(gamePic,gamePic))
      (gamePic,_) <- (rpball1 vel) -<
          (activeIn,Nothing)
```

```
  returnA -< (picS,())
```

In this implementation, there are two views adjacent to each other. The view on the left is an *active* view, as its auxiliary input signal (`activeIn`) is fed as the input signal to the actual `rpball1` GUI. The view on the right is *passive*: Its picture signal is the same (`gamePic`) signal as the active view on the left, but its `GUIInput` signal is not connected to anything.

Multiple Active Views

Adding passive views to a GUI is certainly useful for many applications. But it is much more interesting, useful and symmetric to provide multiple *active* views, so that the user can interact with any view.

Recall from section 3.1 that we defined `GUIInput` to account for a *focus model*: At every point in time, the visual input signal to a GUI is either `(Nothing, Nothing)` (when the component does not have focus), or `(Just kbd,Just mouse)` when the GUI has mouse focus. Further, as described in section 4.4.1, the layout combinators perform *clipping* as well as transformation to ensure that only the GUI under the mouse receives (a transformed view of) the `GUIInput` signal. Recall, too, that our programming model includes a set of *event source combinators* that operate on signals of `Maybe` values.

Armed with this knowledge, we can now consider how to implement active views. In the implementation of `pbview`, each `view` is passed to the `besideGUI` layout combinator. The `besideGUI` combinator uses clipping and transformation to *demultiplex* its input signal into two signals, one for each child. At every point in time, one child's input signal is `(Just kbd, Just mouse)` while the other's is `(Nothing, Nothing)`). Regardless of which `GUI` has focus, the input signal will be transformed into the child's local coordinate system. Given this knowledge, it is a simple matter to define a `mergeGUIInput` combinator that will merge two disjoint `GUIInput` signals back in to a single signal by favoring the `Just` values and discarding the `Nothing` values. We define `mvpball` ("multi-view paddleball") as:

```
-- event merge, left-biased (from AFRP library):
emerge :: Maybe a -> Maybe a -> Maybe a
emerge mbeL mbeR = maybe mbeR id mbeL

mergeGUIInput :: GUIInput -> GUIInput ->
                     GUIInput
mergeGUIInput (mbkA,mbmA) (mbkB,mbmB) =
  (mbkA `emerge` mbkB,
   mbmA `emerge` mbmB)

mvpball :: Double -> GUI () ()
mvpball vel = proc (inpS,_) -> do
  rec (combinedPic,(leftIn,rightIn)) <-
```

Fig. 8. Dynamic Labels Example

```
        (view `besideGUI` view) -<
           (inpS,(masterPic,masterPic))
      let mergedIn = mergeGUIInput leftIn rightIn
      (masterPic,_) <- (rpball1 vel) -<
           (mergedIn,())
  returnA -< (combinedPic,())
```

In this version of Paddleball, both views are treated symmetrically: The user can play or press the restart button in either view, and the action is reflected in both views.

Fruit is the only toolkit we are aware of that provides multiple active views "for free", without requiring any extra forethought or planning by the programmer of the original GUI.

4.6 Dynamic Interfaces

Thus far, all of the GUIs we have defined have been essentially *static* in the sense that the set of interface components visible on screen is fixed over the lifetime of the program. To support realistic user interfaces, it must be possible to dynamically add or remove components from the interface at runtime.

To demonstrate Fruit's support for dynamic interfaces, we implement an application (*dynLabels*) that dynamically adds new labels to an interface in response to a button press. The application is shown in figure 8. Every time the button is pressed, a new label is added to the interface (at the right edge of the current GUI) that display a count of how many times the button has been pressed in the program thus far. The screenshot shows the program after the button has been pressed six times.[8]

Since `GUIs` are first class values, we can maintain the current `GUI` that appears on-screen (using, say, `stepAccum` or some other accumulating signal transformer), and add to this `GUI` by using a layout combinator. Using such an accumulator in conjunction with `switch` allows us to switch from displaying one GUI to displaying the updated GUI. This pattern is so common and useful that we provide an `accumST` combinator to support it:

```
accumST :: (ST b c -> d -> ST b c)
```

[8] This example only adds components to the interface and does not remove them. Extending to allow removal as well as addition is straightforward.

```
            -> ST b c -> ST (b,Maybe d) c
```

The `accumST f st0 -< (iS,eS)` will behave initially as `(st0 -< iS)`. When an event occurs on `eS`, `accumST` passes the current signal transformer and the event occurrence value to `f` to obtain a new signal transformer. The `accumST` will then `switch` in to the signal transformer returned, which becomes the "current" signal transformer. The code for `dynLabels` is:

```
-- A set of counting labels:
countLabels :: GUI (Maybe ()) ()
countLabels =
  let addLabel :: GUI () () ->
                  Int -> GUI () ()
      addLabel labels n = labels
                  `besideGUI_` (mkLabel n)
  in proc (inpS,es) -> do
     lblNumE <- countE -< es
     (picS,_) <- accumST addLabel (mkLabel 0)
                     -< ((inpS,()),lblNumE)
     returnA -< (picS, ())

dynLabels :: GUI () ()
dynLabels = proc (inpS,_) -> do
  rec (picS,(pressES,_)) <-
        (fbutton `besideGUI` countLabels) -<
          (inpS,(btext "press me!", pressES))
  returnA -< (picS,())
```

5 Related Work

There have been many, many GUI toolkits implemented for Haskell, including Haggis [10], TkGofer [5], FranTk [23], and Fudgets [4]. These toolkits cover the spectrum from the mostly imperative (Haggis) to the mostly functional (Fudgets).

FranTk is similar to Fruit in the sense that it too uses the Fran reactive programming model (and its combinators) to specify the connections between user interface components. However, FranTk uses an imperative model for creating widgets, maintaining program state (with mutable variables or "MVars"), and wiring of cyclic connections (which occur in most GUIs, including the examples in this paper).

The closest relative to our work is Fudgets. Fudgets are implemented as *stream processors*, where each Fudget has *high level* and *low level* input and output streams. The high-level streams in Fudgets serve a role similar to the auxiliary semantic signals in our `GUI` type. The programming interface to Fudgets is very similar to that of Fruit, although Fudgets is based on discrete, asynchronous *streams*, whereas Fruit is based on continuous, synchronous *signals*.

Another difference is that Fruit is based on an abstract conceptual model of `GUIs`, whereas Fudgets is based on augmenting Haskell's stream-based I/O system with request and response types for the X Window system. Since we have not seen a formal definition of X windows, it is not clear to us what the denotational model of a Fudget is, beyond saying that it is a stream processor that emits and consumes X protocol requests. We believe that Fruit's model enables more precise reasoning about Fruit programs.

However, the Fruit programming interface is not without cost. Because any Fudget can emit an I/O request, a Fudget to perform file or network I/O can be added to a Fudget program just as easily as adding a graphical Fudget. In contrast, adding such features to Fruit would require explicit threading of the I/O actions through the Fruit program.

Finally, our work is similar to (and partially inspired by) Pike's pioneering work on Mux [20] [19], implemented in the language Newsqueak [21], a successor to Cardelli and Pike's language Squeak [3]. In Mux, every application is a *process* that communicates with the window system using CSP-style synchronous channels. The interface to each process has two input channels for keyboard and mouse input, and an output channel for producing pictures. The Mux window system itself is such a process that does simple *multiplexing* and *demultiplexing* to route messages between its input and output channels and those of its children. Thinking about composition of independent windows as multiplexing and demultiplexing is similar to our layout combinators.

The Fruit programming model owes much to its ancestors Fran and FRP. The most recent implementation of SOE FRP includes input types in the definition of `Behavior`, and an `Arrow` instance declaration for `Behavior`. The SOE FRP combinators are defined as ordinary Haskell functions, and the interface includes a primitive combinator, `runningIn`, that enables a signal to masquerade as a `Behavior`. In contrast, our interface defines every combinator as a signal transformer whose inputs are specified explicitly in its input type, and we use the arrow combinators for composition and application. Our programming interface thus gains modularity (as we can interpose functions such as spatial transformation on an `ST`'s input signal), and emphasizes the distinction between *signal transformers* and *signals*. However, we depend on the arrows syntactic sugar to make our model viable for writing real programs.

6 Current Status

We have implemented a working prototype of Fruit that is capable of running all of the examples presented here. The prototype includes a basic subset of the FRP combinators (implemented as synchronized stream processors). For visual display, Fruit uses a new vector graphics library, *Haven*, that we developed for this project. The interface to Haven is purely functional and implementation-independent, but our reference implementation uses the Java2D rendering engine. The low-level calls to Java2D are handled using another tool, *Elijah*, that provides a connection to the Java Native Interface via GreenCard, also implemented as a side project specifically for use in Fruit. We plan to release both Haven and Elijah as independent projects.

We refer to Fruit as a “prototype” only because it does not yet include a complete set of user interface components. Our focus thus far has been on figuring out the right abstract conceptual model and demonstrating that this model is viable and practical.

7 Conclusions and Future Work

In this paper, we presented a GUI toolkit for Haskell based on a formal model of graphical user interfaces. We showed how this model could be embedded in Haskell, and how the library could be used to construct a plausible example application. We also demonstrated some of the benefits of our approach, by showing how continuous spatial scaling and multiple views could be easily accomodated within the model.

Our results so far are very preliminary but encouraging. Building a library based on a formal model appears to be practical and provides some useful additional benefits, but we need to explore both of these areas in more depth.

In the short term, we plan to replace our low-level stream-based FRP implementation with a much more efficient data-driven implementation, add a complete and realistic set of widgets, and add support for efficient dynamic collections. In addition to this implementation work, we plan to further explore how we can incorporate modern user interface techniques into the model, as suggested in Section 4.4.1. And, of course, we plan to implement some real applications in Fruit, to further explore the benefits and limitations of our approach.

8 Acknowledgements

Special thanks are due to Henrik Nilsson for many patient, thoughtful discussions on the issues presented in this paper. We are also very grateful to members of the FRP research group at Yale (Paul Hudak, Liwen Huang, John Peterson, Walid Taha, Valery Trifonov and Zhanyong Wan) for their work on SOE FRP, many constructive discussions on semantics and implementation, and for reviewing earlier drafts of this paper. We would also like to thank to Magnus Carlsson, Christopher League, Ross Paterson and anonymous reviewers who read an earlier draft of this paper and provided very constructive feedback. Finally, thanks to John Hughes and Ross Paterson for their work on arrows and the arrows syntactic sugar. Without that framework, Fruit would be an interesting theoretical model with no viable implementation.

References

[1] Backus, J., *Can programming be liberated from the von Neumann style? A functional style and its algebra of programs*, Communications of the ACM **21** (1978), pp. 613–641.

[2] Bederson, B., J. Meyer and L. Good, *Jazz: An extensible zoomable user interface graphics toolkit in java*, in: *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology (UIST)*, ACM, 2000, pp. 171–180.

[3] Cardelli, L. and R. Pike, *Squeak: A language for communicating with mice*, in: B. A. Barsky, editor, *Computer Graphics (SIGGRAPH '85 Proceedings)*, 1985, pp. 199–204.

[4] Carlsson, M. and T. Hallgren, "Fudgets - Purely Functional Processes with applications to Graphical User Interfaces," Ph.D. thesis, Chalmers University of Technology (1998).

[5] Claessen, K., T. Vullinghs and E. Meijer, *Structuring graphical paradigms in TkGofer*, in: *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, 1997, pp. 251–262.
URL `citeseer.nj.nec.com/claessen97structuring.html`

[6] Elliott, C., *Functional implementations of continuous modelled animation*, in: *Proceedings of PLILP/ALP '98* (1998).

[7] Elliott, C., *An embedded modeling language approach to interactive 3D and multimedia animation*, IEEE Transactions on Software Engineering **25** (1999), pp. 291–308, special Section: Domain-Specific Languages (DSL).

[8] Elliott, C., *Functional images*, (to appear) Journal of Functional Programming (JFP) (2001).
URL `http://www.research.microsoft.com/~conal/papers/functional-images/`

[9] Elliott, C. and P. Hudak, *Functional reactive animation*, in: *International Conference on Functional Programming*, 1997, pp. 163–173.

[10] Finne, S. and S. P. Jones, *Composing the user interface with Haggis*, Lecture Notes in Computer Science **1129** (1996).
URL `http://citeseer.nj.nec.com/finne96composing.html`

[11] Gamma, E., R. Helm, R. Johnson and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software," Addison Wesley, Massachusetts, 1994.

[12] Henderson, P., *Functional programming, formal specification and rapid prototyping*, IEEE Transactions on Software Engineering **12** (1986), pp. 241–250.

[13] Hudak, P., *Modular domain specific languages and tools*, in: *Proceedings of Fifth International Conference on Software Reuse*, 1998, pp. 134–142.

[14] Hudak, P., "The Haskell School of Expression – Learning Functional Programming through Multimedia," Cambridge University Press, Cambridge, UK, 2000.

[15] Hughes, J., *Generalising monads to arrows*, Science of Computer Programming (2000), pp. 67–111.

[16] Paterson, R., *A new notation for arrows*, in: *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2001)*, 2001.

[17] Perlin, K. and D. Fox, *Pad: An alternative approach to the computer interface*, Computer Graphics **27** (1993), pp. 57–72.

[18] Perlis, A., *Epigrams on programming*, ACM SIGPLAN Notices **17** (1982).
URL `http://www.cs.yale.edu/homes/perlis-alan/quotes.html`

[19] Pike, R., *Window systems should be transparent*, Computing Systems **1** (1988), pp. 279–296.
URL `http://citeseer.nj.nec.com/pike88window.html`

[20] Pike, R., *A concurrent window system*, Computing Systems **2** (1989), pp. 133–153.
URL `http://citeseer.nj.nec.com/pike89concurrent.html`

[21] Pike, R., *Newsqueak: A language for communicating with mice* (1989).

[22] Reynolds, J., "Theories of Programming Languages," Cambridge University Press, 1998.

[23] Sage, M., *Frantk: A declarative gui system for haskell*, in: *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2000)*, 2000.
URL `http://www.haskell.org/FranTk/userman.pdf`

[24] Stoy, J. E., *Some mathematical aspects of functional programming*, in: J. Darlington, P. Henderson and D. A. Turner, editors, *Functional Programming and its Applications*, Cambridge University Press, 1982 pp. 217–252.

[25] Turner, D. A., *Functional programs as executable specifications*, Philosophical Transactions of the Royal Society of London **A312** (1984), pp. 363–388.

[26] Vullinghs, T., D. Tuinman and W. Schulte, *Lightweight GUIs for functional programming*, in: *PLILP*, 1995, pp. 341–356.
URL `citeseer.nj.nec.com/vullinghs95lightweight.html`

[27] Wan, Z. and P. Hudak, *Functional reactive programming from first principles*, in: *Proc. ACM SIGPLAN'00 Conference on Programming Language Design and Implementation (PLDI'00)*, 2000.