Can GUI Programming Be Liberated From The IO Monad[*]

John Peterson
Department of Computer Science
Yale University
New Haven, CT 06520-8285
peterson-john@cs.yale.edu

Antony Courtney
Department of Computer Science
Galois Connections
Beaverton, OR
antony@galois.com

Bart Robinson
Department of Computer Science
Yale University
New Haven, CT 06520-8285
bartholomew.robinson@yale.edu

Abstract

GUI programming in Haskell has developed along two lines: a purely functional one that includes Fudgets, FranTk, and, most recently, Fruit while a more conventional field of research has let to wxHaskell, a powerful portable GUI library in the imperative, object oriented style.

This research began with a project to combine Fruit with wxHaskell. While this effort demonstrated that it was possible to place a functional veneer over a portion of wxHaskell, it also revealed a large semantic gap between the purely functional GUI approach and the semantics of wxHaskell. This gap is apparent in the difficulty of bringing the full functionality of wxHaskell into the purely functional setting. Another problem is that some constructions expressed simply and directly in wxHaskell become convoluted and opaque when transferred to the functional style.

Rather than continue work on the bridge between wxHaskell and Fruit, we elected to explore a design point in the space between the completely functional Fruit GUI style and wxHaskell's imperative foundation. The result of this is wxFRoot, a language that combines the functional reactive programming (FRP) style of Fruit with some of the O-O aspects of wxHaskell. Our aim is to create a system with all of the capabilities of wxHaskell without resorting explicitly to the IO monad for callbacks, object allocation and deallocation, or attribute management. This has resulted in a system with one aspect of O-O programming, object identity, while using the FRP style to control object lifetime and interconnection.

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

1 Introduction

It is widely recognized that programs with Graphical User Interfaces (GUIs) are difficult to design and implement [2, 8, 7]. Myers [8] enumerated several reasons why this is the case, addressing both high-level software engineering issues (such as the need for prototyping and iterative design) and low-level programming problems (such as concurrency). While many of these issues are clearly endemic to GUI development, the subjective experiences of many practitioners is that even with the help of state-of-the-art toolkits, GUI programming still seems extremely complicated and difficult relative to many other programming tasks.

The Haskell community has struggled with GUI development over the years, resulting in many different libraries such as Fudgets [1], FranTk [10], HtK, and, most recently, wxHaskell. These efforts fall into two categories: those seeking to import and Haskellize standard object-oriented GUI libraries and attempts to find a more declarative programming paradigm to replace IO monad which pervades the first approach. This declarative approach is based on replacing the actions (callbacks, widget creation functions, object mutation) with a value oriented using streams or signals. Hardware design is a source of useful metaphors in the domain. Unfortunately, these declarative GUI toolkits have not achieved the popularity, portability, or breadth of functionality needed for serious application development.

These two approaches share a common goal: to make existing GUI practice available to the Haskell programmer. That is, the vocabulary of the modern GUI toolkit is (more or less) agreed upon: frames, windows, buttons, menus, and such. The open question is how integrate these into the Haskell programming style. Since wxHaskell represents the current state of the art in Haskell GUI programming (it is is portable, well-designed, and expressive) it makes sense to start from there to build a high level GUI system sufficiently complete and powerful to entice programmers to abandon the IO monad for GUI applications.

The seeds of this research come from student project, wxFruit, to integrate Fruit, a high level FRP based GUI system, with wxHaskell. This system retains the basic GUI semantics of Fruit while replacing the low level image and event stream plumbing with a request / response layer on top of wxHaskell. While wxFruit is able to handle a number of simple FRP programs (paddleball, for example), it is restricted to a very small portion of the wxHaskell functionality. Designing wxFruit revealed a large semantic gap between the purely functional GUI approach and the wxHaskell which was not easily bridged. Many of the harder problems were avoided by focusing on such a limited subset of wxHaskell.

This led to the question: is the Fruit (and Fudgets, which is similar) model really the only high level abstraction layer possible for a GUI toolkit? We decided to explore designs that were closer to the object-oriented style of wxHaskell without completely embracing the IO monad. Our wxFroot (Functionally Reactive and Object Oriented Too) is close enough to wxHaskell that we are able to encapsulate nearly all of the wxHaskell capabilities in a more functional style. By providing a complete, well featured tool kit we hope that Haskell programmers will be able to evaluate the FRP style of writing GUIs in real applications rather than toy examples.

The wxFroot system is still work in progress. We hope to have more realistic examples in the final version of this paper and a releasable system for the rest of the community to use soon. Even without completed software, though, we feel that our explorations in the GUI design space are of interest.

The rest of the paper is organized as follows: Section 2 covers wxHaskell. In Section 3, we review Fruit and wxFruit. Section 4 deals with the GUI design space and various problems with existing GUI designs. Section 5 discusses "high level wxFroot", a simplified version of wxFroot that captures its basic semantics. Section 6 deals with the low level issues involved in the full interface to the wxHaskell system. Section 7 concludes.

2 About wxHaskell

The wxHaskell library is built on wxWindows, a popular portable library that serves as a common front end to the native window system on a variety of operating systems. Here we will address the fundamentals of the wxHaskell programming style.

The core features of wxHaskell include:

- A type system that permits O-O inheritance. This is orthogonal to reactive system design and remains relatively unchanged in wxFroot.
- Objects which are explicitly created or disposed of in the IO monad. These objects are represented by *handles* at the Haskell level.
- Mutable object attributes changed through IO actions.
- Callbacks that allow objects to invoke IO actions.

Code in wxHaskell is remarkably easy to read. Here is a simple "Hello World" example:

```
hello :: IO ()
hello
    = do f ← frame [text := "Hello!"]
      quit ← button f [text := "Quit",
         on command := close f]
      set f [layout := widget quit]
```

Of particular interest:

- The "frame" and "button" functions return handles that give an identity to a specific frame or button.
- Recursion of the frame and button (the frame contains the button and the button closes the frame) is handled through assignment and mutability.
- Lifetime of the program is implicit; closing *f* terminates the program since no other frames are on the screen. The button is discarded when the frame closes.
- The button must be initialized in the context of a frame, thus the frame must be created before the button.
- Event handlers allow widgets such as a button to run general IO actions.

Another program to consider is the "bouncing balls" program, of which this is a fragment:

```
ballsFrame
    = do vballs ← varCreate [ ]
      f ← frameFixed [text := "Bouncing balls"]
      p ← panel f [on paint := paintBalls vballs]
      t ← timer f [interval := 20,
         on command := nextBalls vballs p]
      set p [on click :=              -- add a ball
            dropBall vballs p
         , on (charKey 'p') :=       -- pause
            set t [enable :~ ¬]
         , on (charKey '-') :=       -- slower
            set t [interval :~ λi → i ∗ 2]
      set f [layout := widget p]
dropBall vballs p pt
    = do varUpdate vballs (bouncing pt:)
      repaint p
```

Again, note that:

- A mutable variable, *vballs*, is needed to communicate among the event handlers.
- An explicit update (via *repaint*) is needed to synchronize display with internal structures.
- Object attributes serve as imperative variables. For example, the interval of the timer is modified by various callbacks.
- The program proceeds mostly via actions, the creation and modification of mutable variables.

3 Previous Work on Functional GUIs

3.1 Fudgets and FranTk

Fudgets is a functional GUI toolkit for Haskell based on stream processors. Fudgets extends the stream-based I/O system of older versions of Haskell with request and response types for the X Window System. The programming model of Fudgets is very similar to FRP, although Fudgets is based on discrete, asynchronous *streams*, whereas FRP is based on continuous, synchronous *signals*.

FranTk uses the Fran [5, 4] reactive programming model to specify the connections among user interface components. GUI operations are handled in a monad, *UI*, which controls object creation. FranTk succeeded in getting a reasonably complete set of GUI objects into a functional setting but was hampered by problems in the underlying Fran implementation and an occasionally cryptic way of combining Fran abstractions with the *UI* monad.

While both of these systems demonstrated the feasibility of a completely functional style of GUI programming, neither caught on in the general Haskell community. Fudgets suffers from being programmed at the raw combinator level: programs can be quite hard to read, especially when streams are mutually recursive. FranTk also has problems with recursion as well as some obscure issues with the underlying Fran system.

3.2 FRP and Yampa

Yampa is a a language embedded in Haskell for describing reactive systems. Yampa is based on ideas from Fran and FRP [11]. Yampa is closely related to Fudgets and FranTk but has the advantage of using the arrow combinators [6] and notation to avoid the difficulty of raw combinator level programming. Since Yampa serves as the basis for wxFRoot, we will describe it in greater detail.

Yampa is based on two central concepts: *signals* and *signal functions*. A signal is a function from time to a value:

$$Signal\,\alpha \;=\; Time \to \alpha$$

Time is continuous, and is represented as a non-negative real number. The type parameter α specifies the type of values carried by the signal. For example, if *Point* is the type of a 2-dimensional point, then the time-varying mouse position might be represented with a value of type *Signal Point*.

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

$$SF\,\alpha\beta \;=\; Signal\,\alpha \to Signal\,\beta$$

When a value of type $SF\,\alpha\beta$ is applied to an input signal of type $Signal\,\alpha$, it produces an output signal of type $Signal\,\beta$.

We can think of signals and signal functions using a simple flow chart analogy. Line segments (or "wires") represent signals, with arrowheads indicating the direction of flow. Boxes (or "components") represent signal functions, with one signal flowing into the box's input port and another signal flowing out of the box's output port. In Yampa, signal functions are provided as an abstract type (*SF a b*), but *signals* are never exposed directly to the programmer.

Programming in Yampa consists of defining signal functions compositionally using Yampa's library of primitive signal functions and a set of combinators. Yampa's signal functions are an instance of the *Arrow* class and hence all of the combinators from the arrow framework may be used to define signal functions. As a concrete example, consider the following signal function defined using the arrow notation:

```
sf0 = proc (a,b) → do rec
  c1 ← sf1−≺ a
  c2 ← sf2−≺ b
  c ← sf3−≺ (c1,c2)
  d ← sf4−≺ (b,c,d)
  returnA−≺ (d,c)
```

As the `SF` arrow is in *ArrowLoop*, we will always use the `rec` keyword to indicate recursive scope of signal values whether or not recursion is being used. Here we have bound the resulting signal function to the variable *sf0*, allowing it to be referred by name. Note the use of the tuple pattern for splitting *sf0*'s input into two "named signals", *a* and *b*. Also note the use of tuple expressions for pairing signals, for example for feeding the pair of signals *c1* and *c2* to the signal function *sf3*.

While some signals model (conceptually) continuous values such as the mouse position, we must also deal with *discrete events*, as expressed by the *Event* type:

data *Event a* = *NoEvent* | *Event a*

The structure of a Yampa system may evolve over time via *mode switches*. This is accomplished through a family of *switching* primitives that use events to trigger changes in the connectivity of a system. The simplest such primitive is *switch*:

```
switch ::
  SF a (b,Event c) → (c → SF a b) → SF a b
```

switch replaces one signal function by another when a switching event occurs. The first argument to *switch* is termed a *task*: a signal paired with a terminating event. This is captured by the following type synonym:

type *SFT i o t* = *SF i* (*o*,*Event t*)

A more complex form of switching handles dynamic collections of signals, as discussed later.

3.3 Fruit

Fruit is a modest library of types and functions for specifying graphical user interfaces using Yampa. GUIs are defined on top of the Yampa signal function type:

type *SimpleGUI* = *SF GUIInput Picture*

The *GUIInput* type represents an instantaneous snapshot of the keyboard and mouse state (formally just a tuple or record) while the *Picture* type denotes a single, static visual image. With an appropriate *reactimate* function (a function similar to *start* in wxHaskell), this type can be used to describe a general GUI system.

While the top level component of a GUI system is concerned with only the display and input, internal components are likely to observe and produce values in addition of the picture and input. The following type definition augments the input and output of a general GUI device:

type *GUI a b* = *SF* (*GUIInput*,*a*) (*Picture*,*b*)

Using this simple base it is possible to construct arbitrary GUI devices from scratch; see [3] for full details.

GUI can be used compositionally: functions such as *aboveGUI*

$$aboveGUI :: GUI\ b\ c \to GUI\ d\ e \to GUI\ (b,d)\ (c,e)$$

serve to create composite GUI devices. Note, however, that the link between *GUI* devices and their connections is somewhat obscured by the notation. For example, in the following:

```
(a,b) ← dev1 `aboveGUI` dev2 −≺ (c,d)
```

the inputs and outputs of the two devices are mingled.

Another abstraction layer on top of *GUI* is used to avoid this problem:

```
newtype GA b c = GA (SF (GUIInput, b) (Picture, c))
newtype Box b c = Box (LayoutF → GA b c)
```

The unique aspect of this is that these types are used to defined new arrows different from the original *SF* arrow in Yampa. The *Box* arrow allows constructions like this:

```
vbox $ proc p → do
  a ← boxSF dev1 −≺ c
  b ← boxSF dev2 −≺ d
```

This is the same as the previous example: two devices, one above the other. The *boxSF* function lifts a signal to a boxed signal. Here the arrangement of the devices has been factored out of the arrow and appears before the **proc**. This makes the program more readable since the input / output signals attach directly to their associated devices.

The *Box* arrow has considerably different properties than *SF*. In *SF*, arrows commute (ordering doesn't matter) while in *Box* the ordering is observable. In *SF*, functions are solely dependent on their input signal, so something like this:

```
b1 ← button ← i1
b2 ← button ← i1
```

would not make sense. Signals *b1* and *b2* would be identical. However, in the *Box* arrow this would create two different buttons.

While Fruit demonstrates that Yampa can serve as a basis on which to build GUI objects from first principles, it is a proof of concept system with only a minimal set of widgets available. The amount of effort to create a large variety of widgets from scratch and the potential performance problems of this technique make it impossible to scale Fruit up to a level where it would be useful to application programmers.

3.4 wxFruit

This project was intended to get just enough wxHaskell functionality into Yampa to run the traditional paddleball program found in the Haskell School of Expression. On the surface, the language was identical to Fruit. Inside the *GUI* type, however, the *GUIInput* and *Picture* types are replaced by request / response types that connect to wxHaskell:

```
type Widget a b =
  SF (WXInput, Event WXResp, a) (Event WXReq, b)
```

The *WXInput* type is used to push global GUI info, in this case the mouse coordinates, into the system. As for the request / response types, the following example shows the constructors associated with the button widget:

```
data WXWidgetReq = WXWBCreateReq WXBState
                 | WXWBSetReq (WX.Button ()) WXBState
data WXWidgetResp = WXWBCreateResp (WX.Button ())
                  | WXWBClick
```

The *WXReq* and *WXResp* types are a union of all possible widget requests and responses, not just those for buttons. The *WXBState* type is of particular interest: it represents the FRP level view of the attributes of the underlying wxHaskell button. The only attributes used in this experiment were the label and the enabled flag:

```
data WXBState = WXBState{
  bsLabel :: String,
  bsEnabled :: Bool}
```

Internally, a button at the FRP level maintains its state (a value of type *WXBState*) and sends a *WXBSetReg* only when the state changes. Attribute values are cached in a Yampa level object to avoid sending new state values to every button at every time step.

This design was unsatisfactory in a number of ways:

- As more widgets are added to the system, the request and response data types become unwieldy.
- These is an unfortunate delay between when a request goes out to *reactimate* (the connection between FRP and the outside world) and when the associated response returns. This requires a "warming up" state for objects such as the button during which they ignore their configuration settings.
- Only a single request / response are be exchanged at each time step. If two buttons were initialized at the same time, one would take longer to warm up than the other.
- The routing of request / response pairs is done at the *Box* level. Objects are identified by the path from the root of the frame through the *vbox* and *hbox* constructions to the actual object.

In spite of these problems, this project demonstrated the feasibility of combining Yampa with wxHaskell and served to inspire the design of wxFroot.

4 The Functional GUI Design Space

Before turning to the particulars of wxFroot, we examine some of the critical design issues in functional GUI programming.

4.1 Object Identity

The wxFroot system has identity (a name supply) wired into the primitive signal function. This example illustrates the advantages of having object identity available.

Consider a GUI consisting of three buttons, *b1*, *b2*, and *b3*, one above and two side by side below. The buttons a wired in a circular fashion so that pressing one sends an event to the next one. We won't worry about what happens when a button sees its neighbor pressed, but instead we will look at how to both wire the buttons and specify their layout on the screen.

In Fruit, the *aboveGUI* and *besideGUI* combinators can be used as follows:

```
(a,(b,c)) ← b1 ‘aboveGUI‘ (b2 ‘besideGUI‘ b3)
   −≺ (b,(c,a))
```

This suffices but becomes notationally awkward as more and more object are composed. An alternative style uses the *Box* arrow to convey the layout of the GUI components. Unfortunately, when different layouts (horizontal and vertical) are combined two different procs are needed:

```
vert $ proc _ → do rec
      a ← b1−≺ b
      (b,c) ← b2b3−≺ (c,a)
   where
      b2b3 = horiz $ proc (c′,a′) →
         b′ ← b2−≺ c′
         c′ ← b3−≺ a′
         returnA−≺ (b′,c′)
```

In contrast, wxHaskell has no problem expressing this (this code is slightly simplified):

```
do b3h ← b3
   b2h ← b2 [on command := doSomething b3h]
   b1h ← b1 [on command := doSomething b1h]
   set b3h [on command := doSomething b2h]
   set f    [layout := above (widget b1)
                (beside (widget b2) (widget b3))]
```

Setting aside the minor awkwardness of the extra *set* (because of the dependency loop) and the "push" rather than "pull" style, this expresses the situation quite nicely. Note that there are two different variables for each button: one that defines its appearance, *b1*, and another that serves as the handle, *b1h*.

The wxHaskell approach can be mirrored in the FRP setting if handles are similarly available. As above, layout is specified separately from connection definitions:

```
proc _ → do rec
   b1h ← b1−≺ buttonClick b2h
   b2h ← b2−≺ buttonClick b3h
   b3h ← b3−≺ buttonClick b1h
   f ← frame−≺ [layout := (above (widget b1h)
      (beside (widget b2h) (widget b3h)))]
```

The handles require an extra level of indirection to get the button click event (via the *buttonClick* function). They also provide a value which the *widget* function can use as a reference. These handles are not the actual wxHaskell handles: they are not use to mutate the buttons or other imperative sorts of things, rather they simply provide identity. Unfortunately the arrow in the previous example is not the well behaved arrow of the Yampa *SF* type but rather similar to the *Box* arrow in that something extra is happening in the background.

While this demonstrates the utility of naming, it doesn't necessarily prove that object identity should be a pervasive feature of the GUI system. Names could also be handed out explicitly to avoid wiring this in at the lowest level:

```
proc (ns,i) → do rec
   (b1h,ns1) ← button−≺ (i,ns)
   (b2h,ns2) ← button−≺ (i,ns1)
   returnA ← (ns2,b1h,b2h)
```

This, though, adds an extra level of plumbing throughout the program and does not guarantee that names are unique since the name supply may be used incorrectly. The FranTk system uses a monad separate from the FRP part of the system to generate names but this also involves a certain amount of awkwardness.

This brings up an important issue: should the GUI system use multiple arrows, as Fruit does, or try and package everything in one arrow? The consequences of a single uniform arrow include:

- The meaning of the arrow syntax is uniform throughout the program, making programs a bit easier to comprehend.
- Lifting between different arrows is never needed.
- The formal properties of the general arrow may not be as strong as those for some individual arrows. This smacks of "throwing everything in the IO monad just in case".
- Extra mechanisms in the one global arrow are not needed in all contexts, slowing down the program.
- Typing will be less precise.

Of course, GUI programmers are happy with just one monad (IO), so a single arrow should be more familiar. There in no way evaluate this design choice in the absence of practical experience with large scale GUI applications. We have opted for a single arrow, confusingly named *SF* even though it differs from *SF* in traditional Yampa, to yield the simplest possible code.

4.2 Contextual Objects and Dynamic Collections

Objects in a GUI are generally part of some larger context. Frames are part of the screen. Buttons are part of a frame. There are two ideas at work here: containment and collections. Each has consequences at the FRP level.

One way of dealing with containment is to stratify the *ST* type in some way so that only the appropriate sort of objects can appear. A signal not contained in a frame would thus not be able to use the *button* function. Unfortunately this does not seem to fit well into the arrow notation. All sub-arrows in a **proc** must be of an identical type so that an arrow of type *SF Button a b* and another of type *SF Frame a b* could not participate in the same mutually recursive arrow proc. A weaker way of expressing this dependency is to mirror the behavior of wxHaskell and pass the handle as an input signal. This yields signatures similar to the following:

```
frame :: SF FrameC FrameH
button :: SF (FrameH, ButtonC) ButtonH
```

The type *FrameC* is a frame configuration; *FrameH* is a frame handle. The config types (borrowed from wxHaskell) carry attribute values to the button or frame. We have chosen this design for its simplicity; later this leads to some unfortunate initialization order issues, as addressed in the next sections.

Dynamic collections can occur in just about any composite object. We address these using techniques developed in Yampa [9]. This dynamic switching function demonstrates our basic intent:

```
multiTask :: Event [SFT a b ()] → SF a [b]
```

The argument to *multiTask* is the task spawner: it continuously feeds in new tasks to the pool of executing tasks, possibly more than one at a time. At each step, *multiTask* returns a list of all output values in currently running tasks. This function is only necessary in contexts in which the contents of an object varies. While our wxHaskell based implementation of dynamic collections differs from the original Yampa-based one, dynamic GUIs can be constructed in a similar manner to [3]. This provides an alternative to the use of mutable variables in wxHaskell to express dynamically varying content.

5 Basic wxFroot

Here we present the design of wxFroot and a simple implementation.

5.1 Hello, wxFroot

We start with the infamous "Hello World" program as implemented in wxFroot:

```
hello :: SFT () () ()
hello = proc _ → do rec
  fr ← frame −≺ [layout := widget b,
                 text := "Hello World"]
  b  ← button −≺ (fr, [text := "Quit"])
  returnA     −≺ ((), buttonPress b)
```

The signature of `hello` resembles `IO ()`. The three parameters to `SFT` are the dynamic input type, the dynamic output type, and the static output type of the terminating event.

This program builds two objects: a frame and a button. The configuration of these objects is expressed as in wxHaskell. The button takes two inputs: a frame (the context) and configuration information.

For each type of reactive object in the underlying toolkit we define a *proxy* that will serve as a handle to the underlying object and as a receptacle for the dynamic attributes of the object. This program requires two proxy objects: one for the button and another for the form. The types of these proxy objects are capture all user-visible attributes as well as two additional fields, one that determines identity and the other containing the current system stimulus. The types *FrameC* and *ButtonC* contain the configuration specific to these objects while *Proxy* augments these with an identity and input value. Type declarations for these proxy objects (minus other wxHaskell attributes not used here) follow:

```
type ObjID   = Integer
data Proxy a = Proxy{ objID :: ObjID,
  curInput                  :: SFInput,
  theObj                    :: a }
data ButtonC = ButtonC{ buttonText    :: String,
                        buttonEnabled :: Bool }
data FrameC = FrameC{ frameText    :: String,
                      frameLayout  :: Layout }
type ButtonH = Proxy ButtonC
type FrameH = Proxy FrameC
```

Other wxHaskell data types are also lifted into wxFroot, such as these:

```
data Layout = LayoutWidget WidgetP
    | LayoutBeside Layout Layout
data WidgetH = WidgetButton ButtonH
```

The types of *button* and *frame* are:

```
button :: SF (FrameH, ButtonC) ButtonH
frame :: SF FrameC FrameH
```

The *FrameH* argument to *button* expresses containment, as discussed in Section 4.2. This is somewhat confusing at first in that these functions generate what appears to be a constant output signal: the "handle" after all never changes once the button or frame is initialized. However, these proxies represent all observable attributes of the object, quantities which do vary in time.

In general, wxFroot requires a parallel type declaration for every wxHaskell type. Reactive objects such as buttons, sliders, or frames are mapped into proxy objects while stateless objects such as layout definitions are used unchanged except that wxHaskell object handles become proxies and all such types must be in *Eq*.

We now turn to semantics: what is the meaning of this program? A wxFroot program describes, at this level, a varying collection of proxy objects (this example doesn't vary but switching, for example, may add or remove proxies). These object descriptions serve as a high level program semantics and allow reasoning about potential execution paths. Each proxy is associated with a set of potential callbacks; execution consists of transitions between sets of proxy objects as determined by the callbacks to the active objects. At a more concrete level, the appearance of the GUI is defined at the wxHaskell level by mapping the proxy objects onto actual wxHaskell objects.

This construction pushes much of the complexity of the system into *reactimate*, where sets of wxFroot proxy objects must be transferred to real wxHaskell objects. This transfer is by no means simple; it must account for object creation when a previously unseen proxy appears, object removal, callback definition, and attribute updates. The order in which individual actions are taken is very important: for example, in the above example the frame must be created before the button since the wxHaskell *button* function takes the parent frame as a parameter.

5.2 Yampa Implementation

Before addressing the implementation of wxFroot we need to look inside the original Yampa system. While Fruit and wxFruit use Yampa "off the shelf", in wxFroot we alter the Yampa *SF* arrow. Although this changes the general properties of Yampa signal functions, it does not interfere with basic Yampa primitives. Thus the underlying Yampa vocabulary will look (and act) the same.

Yampa signals are implemented using continuations. Given an input value, a signal computes the current output value and a residual signal for the next sample. The signal input is augmented by a special type, *SFInput*, that contains information common to all signals. In the original Yampa system this additional input is a time interval (in seconds) that indicates the length of the current time step. A (simplified) definition of *SF* is:

```
data SF a b = SF ((a, SFInput) → (b, SF a b))
```

The *Arrow* and *ArrowLoop* instances for this signal type are easily constructed.

5.3 Implementing Pure wxFroot

In contrast to Fruit and the wxFruit, GUI support is built into the *SF* type, eliminating the need for separate *GUI* or *Widget* types. A single modification to *SF* suffices: we add a state argument (state monad) inside *SF*. This state will handle three things: propagation of stimulating events to widgets, supplying names to newly created widgets, and collecting the set of active widgets. The definition of *SF* is as follows:

```
data SF a b = SF ((a, SFState) → (b, SF a b, SFState))
```

The standard Yampa arrow instances can be changed trivially to accommodate the extra state parameter. The *ArrowLoop* instance does not need to perform recursion on the state, just the signal values.

The state type is as follows:

```
data SFState = SFState{sfObj :: ObjId,
   sfProxies :: [ProxyObj],
   sfInput              :: SFInput}}
data ProxyObj = ButtonProxy ButtonP
   | FrameProxy FrameP
```

The *ProxyObj* type is a union of all proxy object types. The ordering of the proxy list is irrelevant.

The three state-related operators are:

```
nextObjectID :: SF a ObjID
nextObjectID = SF (λ(_, st) → let objID = sfObj st in
                        (objID, c objID,
                           st{sfObj st = 1 + sfObj st}))
                        where c s = SF{λ(_, st) → (s, c s, st))
addProxy :: SF Proxy ()
addProxy = SF (λ(p, _, st) →
   ((), addProxy, st{sfProxies = p : sfProxies st}))
currentInput :: SF a SFInput
currentInput = SF (λ(_, st) → (inp, currentInput, st))
```

The only notable thing about these functions is that *nextObjectID* returns the same new object ID repeatedly once initialized rather than generate a new ID at each time step. The *reactimate* function must propagate the object available object ID from one time step to the next.

We now implement the arrow for the button type:

```
button :: SF (FrameH, ButtonC) ButtonH
button = proc (_, c) → do rec
   myID ← nextObjectID   −≺ ()
   inp    ← currentInput    −≺ ()
   myProxy ← arr mkProxy−≺ (myId, inp, prop)
   ()       ← addProxy        −≺ ButtonProxy myProxy
   returnA                    −≺ myProxy
where mkProxy (theId, inp, settings) =
   Proxy theId inp settings
```

We ignore the frame handle input to the button since this version isn't yet attached to wxHaskell.

The last detail is the *SFInput* type which carries callback information from *reactimate* into the signal functions. We assume that only one event is handled at each time step: two callbacks cannot happen at the same time. The input to the system consists of a delta time (as in the standard Yampa library) and an event containing callback information.

```
data SFInput = SFInput{
   sfDeltaTime :: Double,
   sfEvent :: Event Stimulus}
data Stimulus = ButtonPress ObjID
            | ...
```

Given this type, we write functions which extract specific signals from a proxy object:

```
buttonPress :: ButtonH → Event ()
buttonPress b = case buttonInput b of
   Event (ButtonPress i)
      | i ≡ buttonID b → Event ()
      _ → NoEvent
```

This completes the infrastructure needed to support GUI programming in Yampa. Running the Hello World example yields the same object set at each time step (suppressing the *SFInput* component of the proxy):

```
Proxy 2 _ (ButtonS "Quit" True),
Proxy 1 _ (FrameS "Hello World"
   (LayoutWidget (Proxy 1...)))]
```

The input signal will change at each time step but the object IDs will not. If the stimulus *ButtonPress* 2 is given to the system, the *reactimate* function will get a task termination event and exit the program.

The use of proxy objects achieves a synchrony that is difficult to express at the wxHaskell level. Yampa effectively updates all signals in tandem; the actual evaluation ordering of signals is not observable while in wxHaskell object attributes are updated sequentially and may be subject to various imperative interactions.

6 Signal Functions with IO

The "collection of proxies" design shown previously for basic wxFroot is a good semantic basis for GUI design but impractical to implement. Here, we demonstrate an equivalent design that avoids pushing all IO into *reactimate* and instead allows objects to perform their own IO operations directly. Our goal is to develop a simple and intuitive "lift" that moves widgets from wxHaskell (or other O-O libraries) into FRP. This is not an easy task - each wxHaskell widget has a relatively large vocabulary of operations that must be integrated into the FRP world.

6.1 The Life of a Widget

As before, our plan is to go into the *SF* arrow and add more functionality. Before we address the arrow design, however, we consider how individual objects interact with FRP. The lifetime of an object consists of the following:

- Initialization: the object is supplied any necessary static parameters to create a handle on the object. Handlers must be attached to callback events.
- Sustainment: dynamic parameters must be updated as they change.
- Callback: the system must respond to events coming from the object.
- Finalization: it may be necessary to explicitly dispose of the object when it is no longer part of the computation.

Our goal is to integrate these actions directly into the definition of *SF* rather than trying to deal with everything directly in *reactimate*.

Initialization and finalization are linked to the the *switch* and *multiTask* operators. When switching into a new signal function the old one must be disconnected and disposed of while the new one must be initialized. All components in the signal function share the same lifetime. If a *multiTask* is terminated, all constituent signal functions must also be terminated. As a signal function may define many objects, all such objects must be initialized or terminated at the same time.

The goal of sustainment is to keep objects in synchrony with their proxies. This is done by watching for changes in these settings performing IO operations only when settings change. This is related to more general work by Conal Elliot in the NewFran system in which signals are represented by change events rather than values. Here we only attempt to do this "signal differentiation" at the proxy level, making IO calls only for parameters that change from one time step to the next.

Another aspect of sustainment is that GUI systems use barrier synchronization to minimize work. That is, rather than propagate all changes immediately to the display there is some sort of "redraw" command that indicates that all changes have been made and the system is in a quiescent state. Getting these synchronization barriers to work correctly is a significant challenge.

Callback is relatively easy to deal with. As objects are initialized, their callbacks are directed to a central *reactimate* entry point. We "tie the knot" by passing this through the *SF* arrow in the state. Object identities, as in the pure implementation, connect the source of the callback with the proper destination.

6.2 Another SF Arrow

We now wish to fold IO actions that were previously performed in *reactimate* directly into the *SF* arrow. Our goal is not to allow general IO to pollute signal functions but rather to break apart the IO operations that would normally be part of *reactimate* and spread them into the stepping and switching operations of the arrow. If done correctly, these IO operations will yield the same wxHaskell state as the simpler construction in the previous section.

The design of this *SF* arrow is considerably more complex when imperative actions are incorporated. The *SF* type is used to initialize a signal function and produce a running signal function of type *RSF*, a initial output value, and a finalizer:

```
data SF a b = SF (a → SFState →
  IO (b, RSF a b, IO (), SFState))
```

A running signal function is the same as the signal function in the basic wxFroot design except that the stepping function is in the IO monad:

```
data RSF a b = RSF (a → SFState → IO (RSF a b, b, SFState))
```

The arrow instances become somewhat less obvious. While *arr* and *first* remain relatively unchanged, composition (>>>) becomes trickier:

```
composeSF :: SF a b → SF b c → SF a c
composeSF (SF f1) (SF f2) =
  SF (λi s →
    do (r1, rsf1, final1, s1) ← f1 i s
       (r2, rsf2, final2, s2) ← f2 r1 s1
       let rsf (RSF f1) (RSF f2) =
         RSF (λi s →
           do (rsf1a, r1a, s1a) ← f1 i s
              (rsf2a, r2a, s2a) ← f2 r1a s1a
              return (rsf rsf1a rsf2a, r2a, s2a))
         return (rsf rsf1 rsf2, r2, final2 ≫ final1, s2))
```

Note the use of ≫ to combine the finalizers from each constituent signal. Also note that *f1* and *f2* are initialized in a well defined order. This ordering occasionally leaks out and places order requirements on the elements of a proc. For example, in this code

```
b ← button −≺ (f, i1)
f ← frame −≺ i2
```

the button will be initialized before the frame, resulting in an error since the frame handle is not yet available. In general, dependent objects defined by a wxFroot arrow should appear in the same order they would in wxHaskell. This applies only to initialization dependencies, not signal dependencies. Alternatively, initializers could be deferred until the necessary input values are available. If initialization ordering becomes a serious problem such a capability could be added relatively easily.

The presence of *IO* requires a significant change in the *ArrowLoop* instance for *SF*: a *fixIO* is needed to make feedback work at the signal level.

The implementation of switching is somewhat tedious. As would be expected, when the switching event occurs, the current signal function is terminated using its finalizer and the new signal function is initialized. The one wrinkle is that a switching event changes the finalizer of the overall switch function. Since by design *SF* computes a finalizer just once rather than at each time step we use an *IORef* to redirect the finalizer to to either the initial or switched signal function.

6.3 Generalized Widget Lifting

We now have the tools to define a general widget lifter. Given the actions associated with each phase of the widget's life, we package up the entire functionality into a single signal function which will serve as a"widget factory".

The signature of the widget maker is:

$$\begin{aligned}
&\mathit{makeWidget} :: (a \rightarrow \mathit{SFState} \rightarrow \\
&\qquad \mathit{IO}\ (b, \mathit{SFState}, \mathit{localState})) \\
&\quad \rightarrow (a \rightarrow \mathit{SFState} \rightarrow \mathit{localState} \rightarrow \\
&\qquad \mathit{IO}\ (b, \mathit{SFState}, \mathit{localState})) \\
&\quad \rightarrow (b \rightarrow \mathit{IO}\ ()) \rightarrow \\
&\quad \mathit{SF}\ a\ b
\end{aligned}$$

The type variable *localState* corresponds to an arbitrary piece of hidden state that the sustainer uses. This is often the same as *a*, the input type, but may be more complex. This saved state allows the sustainer to detect changes in object parameters from one step to the next. The initial output value, typically the newly created handle, is passed to the finalizer.

The sustainer also has access to the *SFState*, allowing it to communicate with *reactimate* or other signals. The only reason this is used at present is to delay IO actions that need to wait until all attributes have been updated.

Getting these widget initialization functions correct is tedious but of course quite necessary. Fortunately there are relatively few different reactive types in wxHaskell. Many of the functions are stateless can be lifted in a much more straightforward manner.

6.4 Semantic Issues

The correctness of the system depends on getting all of the various imperative actions required at each time step to occur in the correct order. The fact that many of these actions are independent of each other helps considerably. Actions that simply transfer attributes to wxHaskell objects can usually be performed at any point in the time step. Sustainers generally do not read out attribute values. So long as such values are not affected by updates this is not a problem. Potential read/write sustainment conflicts can be resolved by deferring writes if necessary.

Of more concern is the treatment of barrier synchronization. We handle this by a field in the arrow state that allows a sustainer to defer actions. All deferred actions are saved until all updating actions are complete, allowing actions such as repainting a graphics pane to take place after all attributes have been changed.

It is important that the *SF* arrows commutes (modulo initialization order problems) so that the operation of the system does not depend on the ordering of arrows in the various procs throughout the program. In the basic wxFroot system this is strongly guaranteed by the semantics of signal functions; no *IO* actions can accidentally interfere with each other.

This problem cries out for a formal model that would take into account the partial ordering of actions required by wxHaskell and the various permutations of action ordering that would arise from different orderings of the signal functions in the arrows, showing that the overall meaning of the system is correct in all cases.

We are well aware that within wxHaskell there will be surprises related to the ordering of actions that may require more complex mechanisms to resolve. Only a few widgets have yet been tested and we expect that as more parts of wxHaskell are addressed unexpected orderings or interactions may show up. We are comforted, though, by the basic wxFroot operational model presented in the previous section which provides firm semantic ground on which to stand. The idea of congruence between a set of actual wxHaskell objects and the corresponding set of proxy objects is a powerful one and will keep wxFroot from straying into the semantic wilderness.

7 Conclusions

Functional programming is the search for functional abstractions in real-world problem domains that provide structure and understanding. Monads, for example, are a way to explain and reason about constructs that were once solely a part of the imperative world. This research shows that an existing language of reactive systems, FRP, can be used to address a fully featured GUI toolkit, wxHaskell, and provide a functional veneer to what had previously been an imperative, object oriented computational style.

The main contribution of this paper is in bridging between a large, object oriented toolkit, wxHaskell, and the Yampa FRP system. In addition, we have developed a framework for a general connection between the FRP programming style and object oriented toolkits that could be extended to other application domains.

There is ample room in this design space for other approaches. This effort differs from the original Fruit system in an important way: arrows have identity, eliminating some of the global signal function properties that are present in the Yampa system. Whether this is a useful convenience or a serious mistake is debatable. The advantage of our approach is that it brings the semantics close enough to wxHaskell that the gap is bridgeable in a relatively straightforward manner.

The ultimate goal of this effort is give programmers the opportunity to build real-world applications in the FRP style. We are reluctant to claim that this is superior to the more imperative style of wxHaskell until evidence beyond the level of Hello World applications exists. wxFruit has not yielded programs that are significantly shorter that the corresponding wxHaskell programs - rather it has eliminated some of the imperative idioms that wxHaskell programmers must use and allowed a more data-centered style of programming. We believe that this will yield programs that are simpler and compositional but this is by no means proven at this point.

Finally, we conclude that Daan needs to go to Salt Lake in September and go rock climbing or kayaking with the authors.

References

[1] Magnus Carlsson and Thomas Hallgren. *Fudgets - Purely Functional Processes with applications to Graphical User Interfaces*. PhD thesis, Chalmers University of Technology, March 1998.

[2] Antony Courtney. Engineering insights from an interactive imaging application. *The X Resource: A Practical Journal of the X Window System*, Fall 1991.

[3] Antony Courtney. *Modeling User Interfaces in a Functional Language*. Phd thesis, Yale, 2004.

[4] Conal Elliott. An embedded modeling language approach to interactive 3D and multimedia animation. *IEEE Transactions on Software Engineering*, 25(3):291–308, May/June 1999. Special Section: Domain-Specific Languages (DSL).

[5] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 163–173, June 1997.

[6] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.

[7] Brad Myers. Separating application code from toolkits: Eliminating the spaghetti of call-backs. In *Proceedings of the Fourth Annual ACM SIGGRAPH Symposium on User Interface Software and Technology (UIST)*, November 1991.

[8] Brad A. Myers. Why are human-computer interfaces difficult to design and implement? Technical Report CMU-CS-93-183, Computer Science Department, Carnegie-Mellon University, July 1993.

[9] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02)*, pages 51–64, Pittsburgh, Pennsylvania, USA, October 2002. ACM Press.

[10] Meurig Sage. Frantk: A declarative gui system for haskell. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2000)*, September 2000.

[11] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *Proceedings of PLDI'01: Symposium on Programming Language Design and Implementation*, pages 242–252, June 2000. `http://haskell.org/frp/publication.html#frp-1st`