

Can GUI Programming Be Liberated From The IO Monad

John Peterson

Yale University
peterson-john@cs.yale.edu

Antony Courtney

Goldman Sachs
antony@aya.yale.edu

Henrik Nilsson

University of Nottingham
nhn@cs.nott.ac.uk

Abstract

GUI programming in purely functional programming languages has developed along two lines. The purely functional approach, avoiding the imperative IO monad, includes Fudgets, FranTk, and, most recently, Fruit. Fruit is a GUI library constructed using the Yampa implementation of Functional Reactive Programming (FRP). A more conventional approach, one which embraces the IO monad, has led to a long series of GUI libraries that mirror the traditional object-oriented programming style. Currently, wxHaskell is the most sophisticated and widely-used Haskell GUI package.

This research began with a project to combine Fruit with wxHaskell. While this effort demonstrated the possibility of placing a functional veneer over a portion of wxHaskell, it also revealed a semantic gap between the purely functional GUI approach of Fruit and the object oriented foundation of wxHaskell. It was difficult to lift objects from the imperative wxHaskell library into the purely functional FRP domain. Also, some constructions expressed simply and directly in wxHaskell become convoluted and opaque in wxFruit.

Rather than continue to bridge between wxHaskell and Fruit, we elected to modify the semantic underpinnings of Fruit and explore a different functional foundation which would lessen the gap between the object-oriented programming style and FRP. The result of this is wxFRoot, a language that retains the basic functional style of Fruit while incorporating some idioms of O-O programming into a new FRP implementation. In doing this, we clarified the relationship between FRP and the object-oriented style of programming. This new version of FRP, OFRP, is able to smoothly interoperate with object-oriented libraries and incorporates some useful O-O idioms into the FRP world without resorting to the fully imperative IO monad. This provides a more declarative style of programming than wxHaskell and shows how FRP can be easily layered on top of a comprehensive object-oriented code base. This paper includes examples of GUI programming in wxFroot and a semantic basis for the extension of FRP to OFRP.

1. Introduction

It is widely recognized that programs with Graphical User Interfaces (GUIs) are difficult to design and implement [2, 9, 8]. Myers [9] 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 [11], 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 the IO monad which pervades the first approach. This declarative approach is based on replacing the actions (callbacks, widget creation functions, object mutation) with values (streams or signals), in a manner resembling that used in hardware design. Unfortunately, these declarative GUI toolkits have not achieved the popularity of their imperative counterparts, whether because of the unfamiliar programming style or deficiencies in the toolkit. Fudgets, in particular, demonstrates that a purely functional GUI toolkit can be used in serious applications.

These two approaches share a common goal: to make existing GUI practice available in a high level, declarative manner within the context of a purely functional language. 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 functional programming style. Since wxHaskell represents the current state of the art in Haskell GUI programming (it is portable, well-designed, and expressive) it makes sense to start from there to build a higher level GUI system that will be sufficiently complete and powerful to entice programmers to abandon the IO monad in their GUI applications.

The seeds of this research arose from student project called wxFruit, at attempt to replace the low level graphics and interaction layer of Fruit with wxHaskell. The original implementation of Fruit was designed to run on the “bare metal” of the user interface: a single graphics panel upon which Fruit would draw GUI devices such as push buttons, text, and graphics. Since building a usable, fully featured GUI system at this level would be extremely time consuming we elected to use an existing high-level GUI library as the backend to wxFruit. The wxFruit system retained the GUI semantics of Fruit by adding a low level event stream con-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Haskell'05, Estonia

Copyright © 2005 ACM [to be supplied]...\$5.00.

struct that connected GUI object requests and responses with a “GUI engine” inside the FRP `reactimate` function. In a FRP implementation, `reactimate` mediates between the outside world and the operation of the FRP program. While `wxFruit` is able to handle a number of simple FRP programs (`paddleball`, for example), it was restricted to a very small portion of the `wxHaskell` functionality. Designing `wxFruit` revealed a semantic gap between the FRP approach and the `wxHaskell` layer which was not easily bridged.

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 slightly closer to the object-oriented style of `wxHaskell` without completely embracing the IO monad. This led to the development of a new FRP engine, `OFRP`, and `wxFRoot` (Functionally Reactive and Object Oriented Too), a GUI system that encompasses nearly the entire functionality of `wxHaskell` in a declarative manner.

2. Basic wxFroot

In this section, we give examples of the `wxFroot` programming style and contrast them with the `wxHaskell`.

2.1 Hello, World

We start with the infamous “Hello World” program as implemented in `wxFroot`:

```
hello :: SFT () () ()
hello = proc () -> do rec
  f <- frame    <-> [layout := widget b,
                    text := "Hello World"]
  b <- button   <-> (f, [text := "Quit"])
  returnA      <-> ((), buttonPress b)
```

A component, in this case named `hello`, is a *signal function* with an input (the `()`) following `proc`, an output (the `((), buttonPress b)`) that follows the `returnA`, and a set of internal components. The input and output are time-varying entities, or *signals*. Signal functions are the basic processing element of FRP: they represent components which interact with other signal functions via an input signal and an output signal. Signal functions expressed using *arrows* [7], a high level control abstraction with an associated syntax layered on top of `haskell`. The `SFT` type indicates that this signal function is a *task*, a special sort of signal function that returns two signals in a tuple: an output signal and a termination event. It is this event (here defined as pressing `b1`) that indicates the completion of the program.

Signal functions are generally composed by wiring up a set of subordinate signal functions, in this case the `wxFRoot` primitives `frame` and `button`. Both of these define GUI components: each such component has an output that yields a handle to the object and an input containing configuration and initialization information. Note that the output and input again are signals (time varying values), although the inputs in this case happens to be constant. See below for an example where appearance of a GUI element change over time by virtue of the configuration information being a signal as opposed to a static value.

Here, the handle associated with the `frame` is `r` and the handle associated with the `button` is `b`. The `frame` is configured with a `layout` (arrangement of contained objects) and a `title` (“Hello World”). Since buttons must exist within a `frame`, the enclosing `frame`, `f`, must be specified among the inputs to the `button`. Finally, the output of this component

is a 2-tuple containing an empty value, `()`, and the termination event, the `button press`. The signature of `hello` resembles `IO ()` except that there are signal types (the first two `()` types) in addition to the result type.

Some notable aspects of this program include:

- Mutual recursion allows signal variables to be used freely in sub-components either before or after their definition. That is, each `<->` clause resembles a definition in a Haskell `let` statement which scopes identifiers over all other clauses in the `let`.
- An attribution language using `:=`, adapted from `wxHaskell`, provides a simple syntax for expressing the properties of a component. Attributes used here include `text` and `layout`.
- Handles are used in a manner similar to the object-oriented programming style. These handles provide unique names for components and encapsulate all signals coming out of a component in a convenient manner. Handles also identify components to each other, as in `layout`.

While the arrow syntax is a little noisy in this application (the `rec` and `returnA` are always used in this manner and could be made implicit) it does a good job of expressing the interconnections among the elements of a component.

This program stands in contrast to the original `wxHaskell` version:

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

The lack of a recursive scope requires a `set` to patch up the attributes to the `frame`. The completion of the program is implicit in the closing action; the lifetime of this program is bound up in the operation of the `close` rather than stated in a more declarative manner. What this program does not demonstrate is the dynamic nature of signals: the configuration of the `button` and the `frame` never changes. The following adds a timer to the program which will place the ever changing elapsed time on the `button label`:

```
hello :: SFT () () ()
hello = proc () -> do rec
  f <- frame    <-> [layout := widget b,
                    text := "Hello World"]
  b <- button   <-> (f, [text :=
                      ("Quit " ++ show t)])
  t <- localTime <-> ()
  returnA      <-> ((), buttonPress b)
```

The `localTime` signal function is part of the FRP library; `t` is time since the signal function was invoked. While the `wxHaskell` program would require some sort of looping to continuously change the label on the `button`, here the use of signals hides the mechanism by which time advances.

2.2 A Button Cycle

We next define GUI with a more complex interconnection pattern. This is a window containing three buttons. Only one button at a time is enabled and pressing this button enables the next one in the chain.

```
buttonCycle = proc () -> do rec
  f <- frame    <-> [layout :=
```

```

        row $
        map widget [b1, b2, b3]]
b1 <- button -< (f, [text := "1",
                    enabled := st == 1])
b2 <- button -< (f, [text := "2",
                    enabled := st == 2])
b3 <- button -< (f, [text := "3",
                    enabled := st == 3])
st <- hold 1 -< buttonPress b1 `tag` 2 .|.
        buttonPress b2 `tag` 3 .|.
        buttonPress b3 `tag` 1
returnA -< ((), windowClose f)

```

This GUI demonstrates the use of built-in FRP constructs such as the `hold` signal function and `tag` primitive. The types of these are as follows:

```

tag :: a -> Event b -> Event a
(.|. ) :: Event a -> Event a -> Event a
hold :: a -> SF (Event a) a

```

The `Event` type is used to describe signals which occur at discrete times rather than continuously. The `tag` function replaces the value carried by a signal with new one; `.|.` merges event streams, and `hold` retains the most recent event value in an event signal.

The arrangement of the buttons (as expressed in the layout) is separate from their configuration and functionality. Rather than give this application a separate close button we use the `windowClose` event tied to the close button on the frame.

A more interesting way to specify this GUI demonstrates the ability of `wxFroot` to express wiring patterns. In this example, the set of buttons is generated dynamically using `>>>` (arrow composition) to connect the buttons:

```

buttonCycleN n = proc () -> do rec
  f <- frame -< [layout := row $
                map widget bs]
  (_, _, b) <- buttons -< (f, st, [])
  st <- hold 1 -< presses b
  returnA -< ((), windowClose f)
  where
    buttons = foldr1 (>>>) (map makeb [1..n])
    makeb i =
      proc (f, st, bs) -> do rec
        b <- button -< (f, [text := show i,
                            enabled := i == st])
        returnA -< (f, st, (b:bs))
    presses b = foldr1 (.|. )
      (zipWith
       (\(i,b) ->
        buttonPress b `tag` i)
       ([2..n] ++ [1]) b)

```

Each button is defined by a signal function that operates on a triple containing the enclosing frame, the current state, and a list of button handles. This signal is threaded through the buttons to yield the final list of button handles used in layout and the state computation.

In this example, `wxFroot` demonstrates a mechanism for dynamically gluing together GUI components, thus expressing component level abstractions as well as static component interconnection.

2.3 Tasks and Switching

So far, we have dealt only with static GUIs. A family of FRP *switching* primitives uses events to trigger changes in the connectivity of a system. The `switch` function:

```

switch :: SFT a b c -> (c -> SFT a b d) ->
        SFT a b d

```

replaces one task by another. This is the basis for a task monad, allowing a more natural notation for sequencing.

To demonstrate the use of switching, consider a GUI that assembles an object state `S` using a series of screens with various input devices such as text boxes, buttons, and sliders to fill in the components of the state. We start with a type to represent a screen of information:

```

type Screen = S -> SFT Frame Layout S

```

That is, a screen takes a state and returns a GUI which runs in a given frame and returns the current frame layout. When the task is complete a new `S` value is produced. Since `SFT` is defined as a monad, we do notation to compose signal functions for each screenful of information. This may seem a bit confusing to have both an arrow (`SF`) and a monad (`SFT`) mixed together but they represent two very different aspects of the system. The arrow notation is used to interconnect components; the ordering of the components is arbitrary (just as the ordering of definitions in a `let`) and all of the components coexist in time. In the monad world in which the system switches from task to task, the sequential ordering of tasks in the `do` is significant: it denotes a progression in time as the system moves from one arrangement of components to the next. This monad is used to hide the use of the `switch` primitive under a more readable syntax.

Here is a task to enter all elements of the state and a typical `Screen` subtask where `s0` is the initial state and `finalScreen` is a screen that uses the final state:

```

enterAll :: Screen
enterAll s = do s1 <- enterName s
                s2 <- enterAddress s1
                s3 <- enterItem s2
                return s3
main = proc () -> do rec
  f <- frame -< [layout := 1,
                text := "Entry Form"]
  l <- enterAll s0 >> finalScreen -< f
  returnA -< ((), closeWindow f)

enterName s = proc f -> do rec
  te <- textEntry -< (f, [text := "Enter name",
                        initialState :=
                          stateName s])
  returnA -< (widget te,
             map
              (\t ->
               s { stateName = t}))
             (textEntered te)

```

The `textEntry` control generates a string event when `enter` is pressed and the string is placed into the `stateName` field of the result. A more complex screen may have multiple text entry fields and a submit button. The `map` modifies an event's value.

A more complex version of this example allows the user to move back and forth among input screens. We can accommodate this by wiring more inputs to the `Screen` type:

```

data Screen = S ->

```

```
SFT (Frame, Event (), Event (), Event ())
  Layout (S, Screen)
```

The extra input events are a forward button press, a back button press, and a button that indicates all values are now entered.

```
enterName =
  Screen (\s -> proc (f, fwd, bk, done) ->
    do rec
      let navigation =
          map (const (s, enterAddress)) fwd .|.
          map (const (s, finalScreen)) done)
      te <- textEntry -<
          (f, [text := "Enter name",
              initialText := stateName s])
      returnA -< (widget te,
                 map (\t ->
                     (s { stateName = t},
                      enterAddress))
                 (textEntered te) .|.
                 navigation)

main = proc () -> do rec
  f <- frame -<
    [layout := above (row [nxt, prev, done] l,
                        text := "Entry Form")]
  nxt <- button -< [text := "Next"]
  prev <- button -< [text := "Previous"]
  done <- button -< [text := "Next"]
  l <- sw enterName s0 -<
    (f, buttonPress nxt, buttonPress prev,
     buttonPress done)
  returnA -< ((), closeWindow f)
  where
    sw (Screen sc) s =
      switch sc (\(s', sc') -> sw sc' s'
```

Since there is no screen before this one it does not respond to the back button. This can be read as a component which exits in one of three ways, each defined by a different event. Further code is needed to invoke the continuation screen returned by a screen function.

The role of switching in object creation and disposal is crucial: switching to a new screen disposes of any controls on the previous screen and creates new controls for the current screen. This happens implicitly in the switching operation that underlies task sequencing rather than through explicit calls to construct or dispose of these objects. Objects of different lifetimes (for example, the buttons and text entry) co-exist easily. Unlike a more imperative version of this program, the interfaces between different program components are explicit; the type of `Screen` indicates precisely which inputs each screen sees.

2.4 Bouncing Balls

As a final demonstration of `wxFroot`, we use *dynamic collections* to express a more irregular form of switching. This example is simplified from the bouncing balls demo of `wxHaskell`.

An extended discussion of dynamic collections appears in [10]. Here we will use the `multiTask` primitive to capture this notion:

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

The event argument to `multiTask` generates new signal function tasks to join the collection. As more than one new

task may appear on an event these events carry a list (set) of new tasks. These tasks run to completion and are then removed from the collection. At any given instant in time, the output of `multiTask` is a set of the the task values (of type `b`) for all currently active tasks.

Consider a simplified version of the bouncing balls which uses two mouse buttons, one to launch a ball at the current mouse position and another to delete the balls under the mouse. Once launched, a ball disappears after 5 seconds. These balls exist with a graphics panel `pn`. The computation of ball trajectories is left unspecified here.

```
launchBall :: Point2 ->
  SFT (Event Point2) Point2 ()
launchBall pnt0 = proc killPnt -> do rec
  ball <- motion pnt0 -< ()
  fuse <- localTime -< ()
  returnA -< (ball,
             (killer `filterE`
              (\click -> closeTo ball click)
              `tag` ())
             .|. boolToEvent (fuse > 5))

main = proc () -> do rec
  pan <- panel <- ...
  balls <- multiTask -< (map (\p ->
                             [launchBall p])
                        (leftClick pan),
                        rightClick pan)
```

While space does not permit us to explain all of the functions used here, note that the call to `multiTask` takes two signals: one launching new balls and another that is seen by all existing balls indicating that balls near a certain coordinate must be killed. And in `launchBall` the task generates a ball coordinate and ends when either the ball is clicked on or 5 seconds have elapsed.

Compared to the `wxHaskell` demo, this program is again more specific regarding the interfaces between components of the program. The signal that the balls view appears explicitly in the types while in `wxHaskell` any function can potentially respond to any stimulus in the system or alter any other object.

In the big picture, these small examples demonstrate that it is possible to express dynamic signals, component interconnection, general component design patterns, and evolving component collections without resorting to an imperative style of programming. Each of these programs is succinct, semantically unambiguous, and explicit in communication patterns.

3. FRP and Object Oriented Programming

Having presented examples of `wxFroot` programming, we turn to the design of `wxFroot`. This work started with `wxFruit`, a student project to combine `wxHaskell` and `Fruit`. This system was able to handle some small `Fruit` programs such as the paddleball program found in Hudak's textbook[6]. Internally, a GUI was a Yampa signal function that connected to `wxHaskell` objects to the outside world using a request / response style. Unfortunately, it was necessary to create request and response types with a constructor for every possible control message to every possible GUI device, not a particularly modular design. More fundamentally, we had to decide which specific device each request or response was associated with. This proved to be quite difficult in the `Fruit` programming style and led to a number of more general questions: what is the relationship between FRP and objects?

Is this a gap between implementation techniques or does it reveal an aspect of the Yampa FRP system that makes it difficult to express GUI programs cleanly? In this section we will demonstrate that a relatively small change to the Yampa FRP system narrows this gap and allows the O-O programming style to be integrated into the functional world in an elegant and useful manner. We call this new system OFRP to distinguish it from the Yampa version of FRP.

Before diving into the details of OFRP, we should consider the essential problem of attaching an FRP program to the outside world. This is accomplished by a function `reactimate`, a function name that is used in the Haskell School of Expression. This function mediates between the IO monad and the user-written signal function defining the system. Unfortunately, the `reactimate` function has proven to be a very brittle part of the FRP system architecture: it requires that FRP programs run in a single, fixed context IO context. The types of stimulating signal and output signal cannot be changed without changing `reactimate`. For applications such as interactive animation this works well in that the input (mouse and keyboard) and output (graphics panel) need not be changed. This situation is similar to that in the original Haskell IO system: a Haskell program could interact only through a fixed `Request` and `Response` type. The goal of OFRP is to extend the original FRP by making `reactimate` more modular. Instead of placing the program in a single context, we want to be able to vary the context by allowing new outside widgets such as buttons, panels, or menus to be introduced dynamically as a program runs. This is accomplished by viewing the “outside world” as simply a collection of objects. The new `reactimate` will encompass the behaviors and events which configure these objects and allow them to stimulate the program as well as the process of creating and destroying these objects as the program runs. OFRP is thus a bridge between FRP and the object-oriented programming style. It embraces the basic ideas of O-O programming while also providing a value oriented, synchronized programming style.

The following sections discuss the specific facets of the O-O style that must be incorporated into OFRP.

3.1 Object Identity

An essential difference between the O-O programming style and the purely functional world is the idea of object identity. For example, in `wxHaskell`, a program that puts two buttons side by side in a frame looks like this:

```
do f <- frame
    b1 <- button f
    b2 <- button f
    set f [layout :=
          row 0 [widget b1, widget b2]]
```

The names `b1`, `b2`, and `f` all serve as handles (names) by which an object is referred to. These handles are used by the `set` construct to alter the state of an object and in `widget`, where they represent the appearance of the object on the screen.

Handles are not difficult to express in a functional setting but there is no systematic, simple way of generating unique handles without placing a burden on the programmer to thread a name supply through the program or to invent unique names on the fly. Thus we have chosen to give all signal functions in OFRP access to a name supply. This is in contrast with the design of `Fruit`: there, signal functions are pure: when identical signal functions are attached to the

same input signal they always generate the same output. The presence of a name supply removes this level of signal purity.

3.2 Object Lifetime

We have used the FRP switching constructs to make object lifetime implicit in a `wxFroot` program. That is, there is no code which explicitly disposes of an object when it is no longer part of the computation or initializes object when switching into a new signal function. This style of programming avoids the need to write separate bits of code to handle initialization or finalization; these become implicit in the FRP control structures. The set of objects which must be initialized or finalized becomes implicit in the definition of a signal function, allowing a single `switch` or `multiTask` to correctly manage large collections of GUI objects.

3.3 Object Containers

Objects in a GUI are generally part of some larger context. Frames are part of the screen. Buttons are part of a frame. We address containment by including a handle to the containing object in the input signal to a sub-object. For example, passing the parent frame into a button. This is not a particularly pleasing design; it would be more elegant to make this connection implicit rather than explicit. At present, however, we have chosen to avoid the implementation complexity of implicit containment.

4. Semantic Issues

While presenting examples of `wxFroot` is one goal of this paper, we are also interested in the basic semantic issues underlying objects and FRP. Here we will construct a simple FRP implementation of OFRP and discuss the meaning of an OFRP program.

An OFRP program models the underlying object system as a succession of world states. Each world state is a set of object configurations. An object configuration is a pair: an object identity and values of each object parameter. For example, in `wxHaskell`, object identity is represented by an object handle and configuration is typically specified using `:=` and `set`. A object such as a button is configured by the attributes such as the button text or enabled flag. The object update process is hidden from the user by OFRP. It defines how the world appears but does not elaborate how to achieve this configuration. The succession of configurations is controlled by callbacks which stimulate the system. Any object in the configuration may advance the program to the next state by sending a stimulating value to the OFRP engine. These values become output signals within the FRP program.

In the remainder of this section we will elaborate the design of OFRP and make these concepts concrete. This is accomplished by changing the basic `SF` arrow in FRP to manage object identity and collect the current set of object definitions implicitly. We hide the actual objects behind proxy objects - ordinary FRP values that represent an object configuration separately from the actual object.

To create OFRP, we need only make one basic change to the Yampa FRP system: we add a state to the `SF` arrow for the following purposes:

- As a supply of unique names for newly created objects,
- To gather the set of all active objects, and
- To move input (callbacks) from objects into the FRP world.

Since adding state to an existing arrow is a relatively trivial process, we need not show the arrow instances of this new SF type. The state type is as follows:

```
data SFState = SFState
  {sfObj      :: ObjID,
   sfProxies  :: [ProxyObj],
   sfInput    :: SFInput}}
```

The `ObjID` type serves to identify objects. For efficiency, we use an `Int`. Of more interest is the use of *proxy objects*. A proxy object contains all of the fields that define the configuration of an underlying object type. For example, a simple button is controlled by the following attributes:

```
data ButtonC = ButtonC
  {buttonText  :: String,
   buttonEnabled :: Bool}
```

We must extend this with fields common all proxy objects, as captured in this type:

```
data Proxy a = Proxy
  {objID      :: ObjID,
   proxyInp   :: SFInput,
   theObj     :: a}
```

That is, every proxy contains a unique ID and an object-specific configuration. Thus a button is described by the type `Proxy ButtonC`.

The `ProxyObj` type is a union of all possible proxy types; this allows the system to collect the description of all active objects into a single structure. The `sfProxies` field of `SFState` serves to accumulate the entire set of active objects.

The underlying state is accessed by the following primitive arrows:

```
nextObjectID :: SF () ObjID
currentInput  :: SF () SFInput
addProxy     :: ST ProxyObj ()
```

The `nextObjectID` arrow generates a single unique name on initialization and returns this ID at all subsequent time steps. The `currentInput` function views the current system stimulus and `addProxy` adds to the set of active objects. Using these primitives, a button is defined as so:

```
button :: SF ButtonC (Proxy ButtonC)
button = proc c -> do rec
  myID    <- nextObjectID -< ()
  inp     <- currentInput  -< ()
  myProxy <- arr mkProxy   -< (myId, inp, prop)
  ()      <- addProxy      -< ButtonProxy myProxy
  returnA <- myProxy
  where mkProxy (i, inp, c) =
    Proxy {objID = theId,
           proxyInp = inp,
           theObj = c}
```

This generates a single new unique identifier when switched into and adds the proxy to the set of active objects until the button is switched out of. The handle carries the current input value with it, allowing any stimulus currently associated with an object to be extracted from the handle.

The `buttonPress` function is an example of a function which extracts a signal from a handle:

```
buttonPress :: Proxy ButtonC -> Event ()
buttonPress b = case proxyInp b of
  Event (ButtonPress i)
    | i == objID b -> Event ()
```

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. These object descriptions serve as a high level program semantics and allow reasoning about potential execution paths. Execution proceeds as follows:

- **Synchronization:** The current configuration is obtained by sampling the SF arrow, applied to the current time and system stimulus, to yield a set of proxy objects. The actual set of objects associated with the OFRP program must then be synchronized with this proxy set, changing configuration parameters, deleting unreferenced objects, and creating objects when a new object ID appears.
- **Callback:** the system waits for a callback from one of the active objects. The value associated with the callback is placed in the `SFInput` type along with the object ID. This becomes the new system stimulus and the process is repeated.

This construction pushes much of the complexity of the system into the synchronization process. Proxy object configurations should be transferred to the actual objects only as necessary; when attributes do not change then need not be conveyed from the proxy to the real object. This transfer must also account for object creation object removal, and callback management. Care must be taken to handle initialization and update in the correct order. However complex this process is, though, it significantly simplifies programming at the user level. Instead of requiring that a user carefully update GUI objects in a specific order, we allow the system to infer this order from a description of the requested state.

5. Implementation Details

The “collection of proxies” semantics of OFRP is a good basis for semantic understanding but impractical to implement directly. Here, we discuss the implementation of OFRP and the interface between OFRP and object-oriented libraries.

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 proxy 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.

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.

5.1 Generalized Widget Lifting

To make it practical to import a large object-oriented library such as wxHaskell into FRP, we need to be able to express a general lifting process which creates a signal function from a set of imperative object operations.

Given the actions associated with each phase of the object's life, we encapsulate the entire functionality of the object into a single signal function which will serve as a "widget factory". The signature of the widget maker is:

```
makeWidget :: (a -> SFState ->
              IO (b, SFState, localState))
-> (a -> SFState -> localState ->
   IO (b, SFState, localState))
-> (b -> IO ()) ->
   SF a b
```

The type variable `localState` corresponds to an arbitrary piece of private state that the object uses. This is often the same as `a`, the input type, but may be more complex. This local 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 through the `SF` arrow. The only reason this is used at present is to delay IO actions that need to wait until all attributes have been updated.

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. 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.

This style of widget lifting is unique among the various FRP implementations. While previous FRP implementations are highly composable at the signal function level, they are not well factored at the `reactimate` level. Here, we supply an elegant decomposition of `reactimate` that conforms to the structure of the object library.

6. Related Work

6.1 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 implementa-

tion 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.

6.2 Fruit

Yampa serves as the current standard FRP implementation [12]. *Yampa* is closely related to *Fudgets* and *FranTk* but has the advantage of using the arrow combinators and notation to avoid the difficulty of raw combinator level programming.

Fruit is a modest library of types and functions for specifying graphical user interfaces using *Yampa*. *Fruit* defines GUIs from first principles, using the following type definition:

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

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

Functions such as `aboveGUI` and `besideGUI` arrange GUI elements on the screen

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

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. Assembling a GUI through such combinators is tedious and brittle: a small change may require extensive changes to the patterns which serve to associate signals with GUI devices.

Unlike wxFroot, *Fruit* does not address unique naming in the `SF` arrow.. It does, however, provide a second arrow in addition to `SF` called `Box`. This does give identity to signal functions but unfortunately it also uses the ordering of signals definitions to determine the layout of the component. This is much less flexible than the use of handles in wxFroot and makes it difficult to factor out the appearance from the function of a GUI. It is also somewhat confusing to use two different arrows with completely different formal properties. In wxFroot, we use only one arrow to avoid this sort of confusion.

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.

7. Conclusions

Declarative programming techniques provide structure and understanding in real-world problem domains. This research demonstrates that an declarative language of reactive systems, OFRP, can be used to address domains which have traditionally been the province of object-oriented design styles.

By augmenting our existing FRP implementation to incorporate object identity we were able to retain the declarative structure of FRP while, at a lower level, remaining close enough to O-O programming that it was possible to integrate a complex existing GUI library, wxHaskell, into the OFRP framework with relative ease. While this specific application

deals with GUI programming, this general technique can be applied to other O-O domains.

Our result is not just a functional rehashing of O-O programming; it presents significant advantages over the original O-O library:

- **Synchrony:** just as clocks are a tool by which hardware designers can simplify the basic semantics of combinational circuitry, the implicit clock of FRP simplifies the semantics of an object oriented system. Instead of many small state transitions which are difficult to reason about or understand, the entire set of objects is updated in tandem as the clock advances. Stated another way, we eliminate anomalies which arise from message ordering. In standard O-O programming, the order in which method calls (message sends) are performed is observable in the system semantics. We are able to send multiple messages in the same logical clock cycle, simplifying the system semantics and making our programs more composable.
- **Declarative lifetime management:** we replace imperative object creation and disposal (explicit calls to object constructors and destructors) by FRP switching constructs. This makes object lifetime management more structured: the set of objects that are active within a signal is managed implicitly; when a signal is disconnected from the computation the entire set of objects living within the signal are de-activated in tandem. Signal functions encompass groups of objects, eliminating the need to manage objects one by one. Furthermore, mutual recursion among objects during initialization (a frequent occurrence) is handled in a more natural style using an equational style. Dynamic collections provide semantic clarity and compositional style to dealing with more irregular patterns of object creation and destruction.
- **Explicit state management:** FRP provides a rich set of stateful signal functions such as integral, sample and hold, and state accumulation functions. These have strong algebraic properties and serve as “lightweight” objects that can be used to assemble complex systems in a systematic manner. These signal functions replace undisciplined use of mutable variables and simplify system design.

Ultimately, our approach places the details of the underlying object system in the background and avoids unnecessary distinctions between objects and other software components.

This effort differs from the original Fruit system in the use of OFRP rather than the Yampa implementation of FRP. This eliminates a global signal function property, purity, that is present in the original Yampa system but retains commutativity of sub-components, an essential compositional property. We also believe that the programming style of wxFroot is appropriate to the underlying domain: programs can be expressed in a natural manner and admit to useful formal reasoning.

The ultimate goal of this effort is give programmers the opportunity to build real-world GUI applications in the FRP style. We are reluctant to claim that this is superior to the more imperative style of wxHaskell until more complex applications have been assembled but we feel that this paper presents evidence that there is a deep connection between OFRP and the O-O programming style and that one of the great conveniences of O-O programming, object identity, need not be sacrificed when moving to a more declarative programming style.

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] Paul Hudak. *The Haskell School of Expression*. Cambridge University Press, 2000.
- [7] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.
- [8] 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.
- [9] 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.
- [10] 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.
- [11] Meurig Sage. Frantk: A declarative gui system for haskell. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2000)*, September 2000.
- [12] 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>