Practical Functional Reactive Programming

John Peterson[1], Ken Roe[2], and Alan Cleary[3]

[1] Western State Colorado University, jpeterson@western.edu
[2] The John Hopkins University, roe@cs.jhu.edu
[3] Montana State University, alan.cleary@cs.montana.edu

Abstract. We present our experiences integrating Functional Reactive Programming (FRP) into a new host language, Python, and a variety of computational contexts: a game engine, a GUI system, and an embedded controller. We demonstrate FRP principles extended to a dynamic environment and the integration of object-oriented libraries into the reactive environment. A number of FRP semantic issues are addressed in a straight-forward way to produce an elegant combination of FRP and object-oriented programming. In particular, *reactive proxies* integrate traditional objects with the reactive world and event-triggered *reactors* allow reconfiguration of the network of reactive proxies and interaction with the outside world. This work demonstrates that FRP can serve as a unifying framework that simplifies programming and allows novices to build reactive systems. This system has been used to develop a wide variety of game programs and reactive animations in a summer camp which introduces programming concepts to teenagers.

1 Introduction

The reactive programming style provides for the implicit flow of information among program components. Rather than connecting these with explicit logic to propagate changing values among them the programmer creates virtual "wires" that constantly move information. Spreadsheets work this way - any change in one cell propagates changes throughout. A cell reference is not a one-time transfer of information but rather a continuous connection between formulas. Reactive programming has become increasingly popular[1] and many languages now have libraries which support it.

Functional Reactive Programming (FRP) is a refinement of the reactive paradigm - it captures the notion of values that change over time, enhanced by a set of powerful compositional reactive operators and support for synchrony. These changing values are termed *signals*. Signals are further categorized as either *behaviors*, which are defined over continuous time, and *events*, which have values (messages) at specific moments in time. Synchrony allows multiple signals to update at the same logical time. FRP was originally developed by Conal Elliott and Paul Hudak[7] and has since inspired a wide variety of reactive systems and libraries.

The expressiveness of FRP comes from a number of features:

- Stateful reactive operators. For example, `accumulate` encapsulates a state which is updated in response to an event generated update function; the output reflects the history of the input signal.
- Continuous time. Operators like `integral` treat behaviors as functions over continuous time. Behaviors can be sampled, combined, and monitored.
- Purity - the value of a signal is determined solely by its inputs. This supports a compositional programming style and a variety of optimizations.

While FRP has prompted considerable development in the functional programming community it has had less impact in more traditional settings. Our work serves as a bridge between the purely functional world of FRP and the object oriented infrastructure used to create interactive systems. Our system, which we will refer to as PFRP (the P can be Practical or Python), retains the original simplicity of FRP augmented by an easy interface to a dynamic, object-based external world. This supports the creation of reactive toolkits in a wide variety of existing interactive software libraries.

PFRP has been developed for and used extensively at a summer camp for teaching programming concepts to teenagers. Many games have been written in it and the details of our implementation have been informed by extensive use by novices. More recently we have extended our work to GUI programming and embedded systems and have found that this same system is well suited to these new application domains.

2 Functional Reactive Programming

In this section we present the basics of FRP and introduce our Python-based system. The examples in this section are taken from the Reactive Panda system, an embedding of the Panda3d game engine[8] in FRP. When describing FRP functions we use Haskell type signatures; when introducing our code we will switch to Python.

We start with the lifting operators. Lifting is used to move from the non-reactive world to FRP behaviors. Lifting a value creates a constant behavior while lifting a function applies the function continuously to the argument behaviors. There is no similar lifting operation for events - an event would need an occurrence time as well as a value.

In Haskell, there are a family of lifts which take functions of various arities into functions that operate in the behavioral world:

```
lift0 :: a -> Behavior a
lift1 :: (a -> b) -> Behavior a -> Behavior b
lift2 :: (a -> b -> c) -> Behavior a -> Behavior b -> Behavior c
```

All of these operators are *stateless* - the output at any time is determined only by the input values at that same time. Overloading allows Haskell data operators to be used in a reactive context. This is also true in Python: all of the basic arithmetic operators can be used on behaviors. Unlike Haskell, in Python the

conversion from non-reactive values to reactive ones (the `lift0` operator) is implicit. The simplest non-constant behavior is `time`:

```
time :: Behavior Double
```

This expresses the current time (measured from the start of the program) as a behavior. A "hello world" program in our system would be:

```
text(time)
```

The `text` function is not part of the traditional FRP vocabulary - it is a constructor for an on-screen text label. This is an example of a *reactive proxy* object, to be described later. This program displays the current time flying by as it continuously updates. Lifted math operators allow arithmetic expressions such as `time+1` to be treated as reactive values; a reactive `+` is just a lifted non-reactive `+`.

Next, we demonstrate stateful FRP operators that create an object in the game engine and move it around on the screen. For this we use *hold* and *integral*.

```
s = spaceship()
s.position = integral(hold(p3(0,0,0),
                      key("leftArrow", p3(-1, 0, 0)) +
                      key("rightArrow", p3(1, 0, 0))))
```

The `spaceship` function is a constructor that places a model in the game engine world. The second line shows the use of *reactive attributes*. The spaceship object is placed in variable `s` and its `position` attribute is set to a reactive value. This uses a velocity controller - in this coordinate system x decreases to the left and increases to the right so the arrow keys define appropriate motion vectors (the `p3` constructor). The `+` in this case is not addition but instead it represents the FRP event merge operator:

```
(.|.) :: Event a -> Event a -> Event a
```

The `key` function is the connection to the keyboard - the second argument is the value that is returned when the key is pressed (the `==>` notation is not available in python). This holds the value associated with the most recently pressed key and converts it to a position using `integral`.

An essential idea in FRP is that all signals advance in synchrony. For example, if two different velocity controllers are using the arrow keys, their velocities will effectively change at the same time - the order in which they change will not be observable.

2.1 Start Times and Observation

This section addresses a somewhat obscure semantic issue and is not essential to later discussion. The meaning of a reactive value that includes stateful operators depends on when it is started. For example, the expression `integral(1)` is meaningless without knowing when the integral starts. In the original FRP

system, the start time was determined by signal initialization at the start of execution or in the `switch` operator. In general, we must take care to distinguish an initializing reference to a signal from an observing reference. For example, consider two separate definitions:

```
x = integral(1)

y = x + 1
```

If the definition of `y` occurs at a different time than the definition of `x`, should `y` see an integral that starts at its definition or at the time the previous definition? In the original FRP, there is no notion of assignment; both would start at the same time. Other implementations such as Father Time[3] treat this differently: the first integral starts when `x` is assigned and the second is an *observer*. This situation has been the source of much confusion, especially when FRP is embedded in an imperative language. This issue was first addressed by the Yampa system [4], which uses the arrow abstraction to make initialization explicit.

3 Reactive Proxy Objects

The primary innovation in this work is the augmentation of FRP with *reactive proxy* objects. Reactive proxies mediate between the traditional OOP toolkits of interactive systems and the reactive engine. These proxies serve a number of purposes:

- They have reactive attributes which are continuously updated by the reactive engine.
- They can be associated with *reactors* which can change the system configuration.
- Object components are observable through external references to the object.

Reactive proxies are created through an *object lifting* process, similar to the lifting of functions in FRP. This facilitates construction of new reactive libraries and allows the end user to perceive the library as if it was implemented in a reactive fashion.

The set of reactive proxies under control of the reactive engine is termed the *world*. The original FRP implementations were based on a fixed world, usually a graphics screen, keyboard, and mouse. Here, the world is dynamic - proxies come and go from the world as needed.

3.1 Reactive Attributes

Each proxy object has an associated set of attributes which are continuously being synchronized with the contained object. For example, consider creating a new model in a game engine:

```
spaceship(position = p3(time, 0, 0), size = 2*time)
```

This creates a new reactive object, a spaceship, which moves and gets bigger as governed by the `position` and `size` attributes. We support two different styles of object construction - one in which the reactive components are passed as named parameters to a constructor and one in which reactive components are assigned later using the dot operator. This is strictly notational - both do the same thing.

Some attributes are used to bring signals into the reactive engine:

```
s = slider(min = 0, max = 10)
text(s.value)
```

This creates a slider that ranges from 0 to 10 and displays the value as a string in text label. Every attribute can be read but some, such as `value`, are defined externally and cannot be assigned to.

3.2 Reactors and Object Update

In FRP the `switch` function is used to change a signal in response to an event. Our system provides an alternate way to shape the system behavior: proxy object reactions or *reactors* for short. Each reactor is associated with a reactive proxy and a triggering event. These are similar to event handlers in event-driven programming except that the triggering event is defined by an arbitrary FRP event and the response to the event is more constrained.

All reactors are parameterized over the reacting proxy object and the value of the triggering event. This allows reactions to be reused among different reactive objects and provides a communication channel between the event and its handler.

Reactors have the following capabilities:

- Add or remove reactive proxies from the world.
- Update proxie attributes.
- Sample the current value of reactive attributes.
- Add new reactors to an proxy.

Although PFRP doesn't use monads, it is useful to characterize the capabilities of reaction functions as a Haskell monad named `Reactor`. We will ignore the connection between attributes and type `a` associated with a particular attribute to get the following type signatures:

```
addObject      :: ROSpec -> Reactor RP
updateAtt      :: RP -> Att a -> Signal a -> Reactor ()
removeObject   :: RP -> Reactor ()
now            :: RP -> Att a -> Reactor a
react          :: RP -> Event e -> (RP -> e -> Reactor ()) -> Reactor ()
```

The `RP` type denotes a reactive proxy and `ROSpec` contains the information needed to create a new `RP`. There is a different `ROSpec` for each specific type of proxy object.

Here is an example of a reactor function:

```
def fire(ship, speed):
   p = now(ship.position)
   missle(position = p + integral(p3(0, speed, 0)))
   ship.position = p

s = spaceship()
react(s, key("upArrow", 1) + key("downArrow", -1), fire)
```

The `fire` function is a reactor - it takes two parameters: the object associated with the reaction and the event value. Here, the object is the spaceship firing the missile and the event value determines the y component of the missile velocity, allowing the reactor to fire both up and down. The `react` function and the spaceship constructor are part of the main program which is an implicit reactor triggered at the start of the program. The `addObject` functionality is implicit in the object constructors, `spaceship` and `missle`. The `now` function returns a static (non-reactive) value, a snapshot of the ship's position when the `fire` reactor is invoked. The `updateAtt` functionality is implicit in the assignment of a new `position` attribute to the ship. This will cause the ship to freeze at the spot where the missile is fired. Also note the use of event merging to trigger the reaction on either the up or down arrow.

3.3 Observation

The attributes of a reactive object are observable by other reactive objects. Since attributes may be updated by reaction functions, it is important to determine whether a reference returns just the current signal or if it always tracks the attribute even when changed. We implement the latter: `o.x` refers to the current value of attribute `x` in object `o`, that is, observation always goes through the object.

For example, in

```
ship1.position = ship2.position + p3(1,0,0)
```

the position of `ship1` tracks the other ship at all times - even if the `position` attribute of the ship is changed by another reactor.

One interesting anomaly in our engine is that you can get into trouble by setting an attribute to a function of itself. Consider this statement:

```
ship1.position = ship1.position+1
```

This has the same sort of meaning as `x=x+1` in Haskell - it is a circular reference that leads to an evaluation loop. Using the `now` function breaks this recursion.

3.4 Lifting Proxy Objects

One goal of our work is to make the object lifting process simple enough that adding reactivity to an existing library is relatively pain free. Lifting objects is

more complex than lifting functions since an object may contain an arbitrary set of attributes, some of which may be observations of the underlying object state. Furthermore, the system needs to be able to dispose of the underlying object if it is removed from the world. Thus the proxy lifting code needs to specify the following:

- Object creation code. This may need static parameters in addition to the reactive ones.
- A set of attribute descriptions. Each attribute has a name, a type (used for error messages), a default value, and a reader or writer, depending on whether this attribute takes its value from the underlying object (for example, a slider value) or is used to update this object (for example, the spaceship position).
- Object destruction code.

While this may seem like a lot of work, proxy lifting is done by the library designer, not by the ordinary user. For example, in the panda3d library we lifted about 10 object classes to create the library used by the computer camp students.

4 Implementation

In this section we discuss the implementation of our system and some of the semantic issues involved. Much has been written about the implementation of signals and their update strategies in FRP [1, 3, 7, 6]. Our system uses a simple pull-style update strategy. This eliminates problems with glitches (computations in which some values are out of date) but is generally less efficient than a push-style propagation style. In our experience, the update speed is not a limiting factor in our applications.

The reactive engine, or *reactimate*, implements the overall execution strategy of the system. The clocking strategy determines the heartbeat of the system. Different application domains use different clocking strategies. For the game engine, we clock at frame rate. In the GUI code, we clock based on external events. That is, the system sleeps until an event from the GUI wakes it up. In the game engine, events other than frame update are queued until the next heartbeat update.

Each heartbeat requires the following steps:

- Each reactive object in the world undergoes attribute update. This evaluates each input reactive attribute and then pushes these changes to the object under the proxy. These reactive values are cached so that an attribute is computed only once even when it has multiple observers. Output attributes are polled and stored in the proxy.
- The reaction events of each object are evaluated to collect a set of reactors to be fired. All reactive values are updated before any reaction function is fired. At this point, any residual updates (discussed later) are performed to make sure that all signals move forward during the heartbeat.

- Reaction functions are executed, possibly updating attributes, creating or removing reactive objects from the world, and perhaps adding new reactions to proxy objects. Attribute update is delayed until after all reactors complete.
- Attributes are updated.

A top level loop iterates through each of the proxies to update their values. Signals on which the proxy depends are updated through recursive calls. When a shared signal is encountered the current signal value is cached to avoid recomputation.

This evaluation strategy ensures that the changes made by reaction functions are not observable until the next heartbeat. Thus all signals are *single valued* - their value cannot change during a heartbeat. This strategy need not be applied to signal level switching (the original `switch` function, which is also part of our system). Switching can also be performed immediately - the Yampa language provides both immediate and delayed switching.

Our goal is to ensure that the effects of the reactors executed within a heartbeat are deterministic - the order in which reactors are invoked should not affect system behavior. Looking at the different aspects of reactors,

- Adding and removing reactive objects from the world does not cause problems since the effects are not seen until the next heartbeat.
- Adding new reaction functions to objects will likewise not cause a problem
- Observations are always consistent since all observation is done before any object is updated.

Unfortunately attribute update is not so well behaved. It is possible that two reactors can clash by attempting to update the same attribute. We will temporarily defer consideration of this problem.

Another implementation detail is the implementation of laziness. The `integral` operator doesn't actually need to evaluate the input to produce its present value. In fact, any integral, when initially sampled, returns 0 regardless on the value of the input. The input is only needed to update the state of the integrator so that the value at the next heartbeat is correct. To avoid evaluation loops caused by mutually recursive integrals (a common thing in physical systems), integrators return their value immediately and put the evaluation of their input on a deferred computation list, as described in the heartbeat loop. This is not something that has to be done in a Haskell FRP implementation - laziness accomplishes the same thing.

Another other interesting issue is what happens when a reactive proxy is deleted. Suppose that a signal containing `o.x`, a reference to the `x` attribute of signal `o`, is running at the time tha `o` is removed from the system by a reactor. This is a typical dangling pointer problem. We elect to address this by retaining the proxy object but not updating it in future heartbeats. The attributes will thus remain unchanging after the real object vanishes.

4.1 Signal Initiation

Signal values start their computation at the time they are assigned to a reactive proxy. Consider the following program.

```
x = integral(1)
a.react(ea, lambda m,v: a.position = x)
b.react(eb, lambda m,v: b.position = x)
```

In this program, `a.position` is assigned `x` at time `ea` and `b.position` is assigned `x` at time `eb`, which is presumably different from `ea`. It turns out that the starting time for the `integral` operation is different for the two objects `a` and `b`. When `x` is assigned the reactive value `integral(1)` it is assigned a *frozen* value. The `reactivate` computations start at times `ea` and eb respectively. Put another way, executing reactive expressions like `integral` doesn't do anything - something only happens when the expression is attached to a proxy.

4.2 Optimizations

There are a number of important optimizations that are needed to ensure adequate performance of the system. When reaction functions initialize signals they are compared with other signals initialized at the same heartbeat. This is a reactive "common sub-expression elimination" process that creates explicit sharing in the implementation so that the shared signal is evaluated only once. Furthermore, signals which contain no stateful operators can be shared across heartbeats.

Another important optimization is *chunking* - combining smaller lifted functions into larger ones. In Haskell, this can be expressed as

```
lift f . lift g = lift (f . g)
```

Chunks of mergable code are detected at signal initiation using a greedy algorithm to create optimized signals.

The `once` function creates an event stream that, having delivered one event, goes silent. This is a common idiom when creating reaction events - often only the first occurrence of the event is significant. During the action selection phase of the heartbeat any action connected to a silent event is discarded.

4.3 A Distressing Example

In this section we explore a problematic semantic issue. It is probably of little interest to casual readers and can easily be skipped.

Here is our example:

```
s = ball()
s.vx = 1
s.vy = 1
s.x = integral(s.vx)
s.y = integral(s.vy)
```

```
def reversex(o, v):
    o.vx = -0.9*now(o.vx)
    o.vy =  0.9*now(o.vy)

def reversey(o, v):
    o.vx =  0.9*now(o.vx)
    o.vy =  -0.9*now(o.vy)

b = ball()
when(b, abs(getX(b)) >= 10) reversex)
when(b, abs(getY(b)) >= 10, reversey)
```

It demonstrates a ball object that starts moving with a velocity of (1,1) and that bounces off walls of a square box. The `when` statements specify when the ball should bounce and the `reversex` and `reversey` reactors perform the bounce. The `when` function is a version of `react` that is triggered by a boolean behavior instead of an event. In the reactors, the use of `now` is crucial since it avoids a circular definition of the attribute as mentioned in section 3.3.

Each heartbeat will move `x` and `y` in tandem since they have the same velocities. Eventually, both `b.x` and `b.y` work their way up to 10. At this point both `when` conditions fire. The two reactors will execute in an unspecified order. Assuming the first one goes first, `reversex` does the expected assignment of -0.9 and 0.9 to `b.vx` and `b.vy` respectively. These assignments however are delayed until the second reactor runs. When second reactor `reversey` executes it gets the old values (not the new values assigned by `reversex`) (which is good!) but it creates a conflicting pending assignment of 0.9 and -0.9 to `b.vx` and `b.vy` respectively. Our reactive engine then prints a warning and performs only the first assignment. Thus we have non-deterministic behavior.

This problem stems from the possibility of simultanous event occurance. In the original FRP system, this was manifested as clashing events in the `.|.` operator. One of these events, always the rightmost, would be discarded. Thus if `.|.` is used to select among different behaviors to `switch`, if two occur at once one of those behaviors is lost. This is the same problem encountered in our system - we manifest this as conflicting attribute updates while the original FRP system manifests this as a missing event. In either case, the programmer has to take care to account for the possibility of clashing events. A similar problem occurs in any event driven system in which the system behavior is influenced by an unspecified order in which events are forwarded to listeners.

At this point we have to concede that it is up to the user to be aware of this and program accordingly. At least we print a message to warn the user of this unfortunate circumstance so it doesn't happen silently. We have found that in many cases this non-determinism is acceptable. Here, for example, we can't determine which reactor will win but whichever one loses will fire on the next heartbeat and the ball trajectory will be almost the same either way.

5 Examples

The best way to illustrate the power of our approach is to present examples of reactive programs. PFRP has been integrated into three different interactive environments: the panda3d game engine, Python's tkinter GUI toolkit, and an embedded controller running on a Raspberry Pi. All of these programs use the same reactive vocabulary but have their own set of objects and primitive data types.

These examples are all targeted at novice programmers. A significant advantage of our reactive library is ease of programming. The game engine has been used by kids as young as 12 while the embedded controller is used by art students with no background in computing.

Programs built using PFRP tend to be more compact than those using more traditional approaches. Notice how the game avoids the need for a "main loop" which walks through and updates all the objects. In the GUI example, dynamically changing outputs are specified by a single reactive expression that transforms the input event streams rather than using event handlers. In the embedded example, a main event loop is avoided by use of reactive expressions that connect input and output devices properly.

5.1 Game Engine Programming

The game engine is the most mature of our reactive libraries. It has been in use since 2006 and inspired the development of this style of programming. The following is an example project completed by one of the camp participants.

The student created a dodge game consisting of a character controlled by a player that would avoid objects flying across the screen. If the player's character is hit by an object the game ends and the amount of time elapsed becomes the score. This code also demonstrates a number of design elements such as lighting and texturing that are not part of the game play but add considerable interest to the project.

The program begins by creating the objects - models - and variables needed

```
rectangle(P3(-4,0,-3), P3(4,0,-3), P3(-4,0,3),  # Background
          texture = "Clouds2.jpg")
text(time, size = 1.7, color = white)  # Score
al = ambientLight(color = color(.5,.5,.5)) # Lighting
dl = directionalLight(hpr = hpr(0,-1,0))
p = panda(hpr = hpr(pi*.5,0,0), size = .4,
      color = color(.5,.5,sin(time)))
```

`rectangle`, `text`, and `panda` create reactive proxy objects. `P3`, `hpr`, and `color` are constructors used by the game engine lifted into the reactive vocabulary. Hpr is an orientation from a heading, pitch, and roll. Note that the color of the panda is time-varying.

Once the panda is created, it is moved the arrow keys in a manner similar to earlier examples:

```
v = hold(P3(0,0,0), key("UpArrow", P3(0,0,1.5))+key("DownArrow",
    P3(0,0,-1.5))+key("LeftArrow", P3(-1.5,0,0))+key("RightArrow",
    P3(1.5,0,0)) + happen(getX(p.position) < -3, P3(1,0, 0))
        + happen(getX(p.position) > 3, P3(-1, 0, 0))
        + happen(getZ(p.position)> 2.4, P3(0,0,-1))
        + happen(getZ(p.position)<-2.4, P3(0,0,1)))
p.position = P3(0,-.8,0)+integral(v)
```

The only new things here is the `happen` function - our name for the FRP `predicate` function - it signals an event when a boolean behavior becomes true.

Here are two reactors:

```
def endGame(ball, v):
    exit(m)  # This removes proxies from the world
    exit(m)
    text(format("Fly free! Your score: %i seconds", now(time)),
        size = 3, position = P3(0,0,0), color = blue)

def randomBall(m, v):
    s = sphere(position = P3(4, -.8, randomRange(-2.5,2.5)) +
                          integral(-1, 0, 0),
        size = randomRange(.05,.2), duration = 8, color =
        color(randomRange(.8,1),randomRange(.8,1),randomRange(.8,1)))
    react(s, hit(s,p), endGame)
```

The `react` function triggers the `endGame` reactor when the sphere hits the panda. The `hit` event is computed in the game engine.

Finally recurring alarm event (a clock) that goes off every 1/2 second is used to trigger the generation of a random ball in the `randomBall` reactor.

```
a = alarm(step = .5)
react(a, randomBall)
```

5.2 GUI Programming

As a first step towards developing a framework for prototyping and developing apps on mobile develop devices, we have integrated PFRP with Python's tkinter package. Below, we show a piece of code which puts up a window with a few widgets that are connected through various constructs in our reactive engine. FigureřefGUI shows the functionality of our widgets. This code is very preliminary - the addition of better constructors, removal of boilerplate code, and better access to internal objects should make future versions much more compact. However, the interaction between the reactive values an the toolkit objects is true to our design.

This example demonstrates that either traditional FRP style programming or our more imperative style can be used. The Quit button functionality is defined by a `react` statement which executes an action to exit the GUI. The value displayed in `self.label.text` is defined by adding the value in `self.inputValue`

Fig. 1. Screen shots from the execution of our GUI code. We start by entering the number 1 in the text box. It is reflected in the label below. The second picture shows what happens after "2" is pressed. "12" is now in the text box and the label below. The third picture shows what happens after increment is pressed twice. This label is now two greater than what is in the text box. The fourth picture shows what happens when we change the value in the text box. The label is still two greater. Finally, the last box shows what happens after pressing the reset button. Now the value in the label is the same as what is in the text box.

to the value of a behavior defined in terms of reactive operators that manipulate the stream of events from the increment and reset buttons. Notice how the streams of events are first merged, then mapped to functions that either increment a value or reset it to zero using `mape` (this is the `==>` operator in FRP). Finally, `accum` is used to accumulate the function applications into a behavior signal. Even though at first appearance, this computation seems complex, consider that in more traditional programming, one would have to add two event handlers for the buttons that update the value displayed in the label. It is also likely that a global variable would have to be declared to store this value as an integer to make computation of the updated value simpler.

```
class Application(tk.Frame):
    def __init__(self, master=None):
        tk.Frame.__init__(self, master)
        self.grid()
        self.createWidgets()
    def createWidgets(self):
        self.inputValue = ReactiveEntry(self)
        self.inputValue.text='0'
        self.inputValue.entry.grid(row=1)
        self.label = ReactiveLabel(self)
        self.label.label.grid(row=2)
        self.incrementButton = ReactiveButton(self,'Increment')
        self.incrementButton.button.grid(row=3)
        self.resetButton = ReactiveButton(self,'Reset')
        self.resetButton.button.grid(row=4)
        self.incAmt = accum(0,merge(
                mape(self.incrementButton.press,
                     (lambda x: (lambda v: v+1))),
                mape(self.resetButton.press,
                     (lambda x: (lambda v: 0)))))
        self.label.text =
             ReactiveInt(self.inputValue.text)+self.incAmt
        self.quitButton = ReactiveButton(self,'Quit')
        self.quitButton.button.grid(row=5)
```

```
        self.quitButton.react(self.quitButton.press,
              (lambda x : self.quit()))
        gui.engine.performCycle()
app = Application()
app.master.title('Sample application')
app.mainloop()
```

5.3 Embedded Systems

This system has been developed for an interdisciplinary course in kinetic sculpture[2]. The program below is used in a simple sculpture which responds to the presence of a viewer by slowly closing the petals on a sculpted flower. The reactive engine operates in a fixed environment - the hardware attached to the controller, a Raspberry Pi, doesn't change. The objects referenced correspond to specific hardware drivers attached to the processor. This system was originally programmed in C and required many lines of code. Here, the system can be expressed much more succinctly.

```
dist = distanceSensor(port = 1)
motor = stepperMotor(port = 2)
motor.shaftAngle = a0 + (min(dist.distance, dMax)) * a1
```

The logic of this controller is very simple: the angle is high (the flower is open) when the sensor measures a distance greater than dMax. As the viewer approaches the angle lowers - the constants a0 and a1 determine the range of angles that the shaft will use. This program demonstrates a useful system programmed in a very minimal way that can easily be understood by non-programmers.

6 Related Work

Our system is a synthesis of ideas which have been percolating in the FRP world since its introduction. While much of this system has been anticipated in past efforts, we have managed to find a "sweet spot" in the FRP implementation space that is useful in a wide range of applications.

6.1 Comparison With Classic FRP

We have adopted a style similar to classic FRP and share the same signal-level semantics. Our reactive proxies lead to a slightly different programming style with respect to switching. Users of our system tend to place switches at the object level rather than the signal level, a more familiar programming style. The `snapshot` function is mostly replaced by the use of `now` in reaction functions. The use of observation semantics for object attributes makes it easier to mix signals that are started at different times. The treatment of reactive expressions is similar - signals are only initiated by switching. Unlike classic FRP, our system has undefined behavior when an attribute is updated simultanously by multiple reaction functions but, conversely, we seldom need the sort of event merging that can lead to unanticipated event loss.

6.2 Father Time

Much of this work was presaged in the Scheme community by Greg Cooper's Father Time system (frTime)[3]. In frTime, as in our work, FRP coexists with an underlying imperative language. FrTime provides a fixed set of reactive proxy objects as part of a small GUI kit but does not have a user-level object lifter. It also uses the original FRP style switching instead of reaction functions which makes some styles of programming more difficult.

FrTime implements all signals in an observational manner, starting them when first evaluated. This makes it harder to abstract over signal definitions. For example, in our system the definition

```
localTime = integral(1)
```

would create an uninitialized signal that is initialized in the referring context while in frTime, this integral would start at the time of definition.

The basic differences between our work and frTime are exemplified by the Tetris game example in their distribution. In it, reactivity is used to manipulate as single big state value which is then used to draw the game and is transformed by the game logic. In our system, the state can be broken down into small pieces (individual reactive blocks) which are programmed individually rather than as a group.

6.3 Modern FRP Implementations

The Haskell-based implementation of FRP has gone in a number of directions. Conal Elliott has continued to refine Fran style FRP, with the reactive library and a foray into tangible values[5], which addresses the connection between used-visible artifacts and reactive programming. These efforts have an admirable semantic purity and are deeply embedded in the Haskell type system. Bringing this style into an untyped language would be daunting at the very least. It also requires a level of sophistication that is probably beyond the users of our system.

Yampa[4] remains under active development. The use of arrow notation address many of the same issues that we have (dynamic contexts, observation, and initialization) in a pleasing way. However, we believe that this style doesn't move easily to languages without type systems or arrow notation and lacks the simplicity of our approach.

Finally, the *reactive banana* library[9] has become the library of choice for the casual Haskell developer. It has managed to preserve the basic feel of classic FRP while allowing reactive programming to exist in a dynamic context. This system uses explicit calls to `reactimate` to integrate the OO and reactive components of the program. Our work makes the boundaries between reactive code and the rest of the application more implicit and encourages a more functional connection between these worlds.

6.4 Other Reactive Systems

There are a number of other systems which simplify interactive programming. In Python, the Trellis system improves callback management and simplifies event-driven programming but does not have a high level object abstraction or deal with continuous behaviors. Yoopf is a simple reactive system but without stateful signals, making it similar to spreadsheets.

7 Conclusions

We have built a system which comfortably integrates Functional Reactive Programming into general interactive toolkits. Using reactive proxy objects, entities defined by the underlying system become first-class citizens of the reactive world, allowing their attributes to implicitly change and propagate over time. Our system also demonstrates that a declarative framework such as FRP can fit comfortably into conventional programming languages.

We have extended FRP in a number of ways: we are able to lift object classes into a reactive context and have simplified the use of the reactive language by introducing reactors and attribute observation. Furthermore, these reactors support dynamic reconfiguration of the application and support a programming style that is intuitive and well suited for novice programmers. This system has been implemented and shown to be practical in multiple contexts: game engine programming, GUI construction, and embedded control.

References

1. Engineer Bainomugisha, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. A survey on reactive programming. *ACM Computing Surveys*, 2012.
2. Eric Brunvand and Paul Stout. Kinetic art and embedded systems: a natural collaboration. In *Proceedings of the 42nd ACM technical symposium on Computer Science education*, pages 323–328. ACM, 2011.
3. Gregory H Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *Programming Languages and Systems*, pages 294–308. Springer, 2006.
4. Antony Courtney, Henrik Nilsson, and John Peterson. The yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 7–18. ACM, 2003.
5. Conal Elliott. Tangible functional programming. In *International Conference on Functional Programming*, 2007.
6. Conal Elliott. Push-pull functional reactive programming. In *Haskell Symposium*, 2009.
7. Conal Elliott and Paul Hudak. Functional reactive animation. *Internation Conference on Functional Programming*, 1997.
8. M. Goslin and M.R. Mine. The Panda3D graphics engine. *Computer*, 37(10):112–114, 2004.
9. HaskellWiki. Reactive banana. website, 2013.