The Yampa Arcade[*]

Antony Courtney
Department of Computer Science
Yale University
New Haven, CT 06520-8285
courtney@cs.yale.edu

ABSTRACT

Simulated worlds are a common (and highly lucrative) application domain that stretches from detailed simulation of physical systems to elaborate video game fantasies. We believe that Functional Reactive Programming (FRP) provides just the right level of functionality to develop simulated worlds in a concise, clear and *modular* way. We demonstrate the use of FRP in this domain by presenting an implementation of the classic “Space Invaders” game in Yampa, our most recent Haskell-embedded incarnation of FRP.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*functional languages, dataflow languages*; I.6.2 [**Simulation And Modeling**]: Simulation Languages; I.6.8 [**Simulation And Modeling**]: Types of Simulation—*continuous, discrete event*

General Terms

Languages

Keywords

FRP, Haskell, functional programming, synchronous dataflow languages, modeling languages, hybrid modeling

1. INTRODUCTION

*This material is based upon work supported in part by a National Science Foundation (NSF) Graduate Research Fellowship. We also gratefully acknowledge the support from the Defense Advanced Research Projects Agency (DARPA) MARS program. 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

or DARPA.

Henrik Nilsson
Department of Computer Science
Yale University
New Haven, CT 06520-8285
nilsson@cs.yale.edu

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

Functional Reactive Programming (FRP) integrates the idea of *time flow* into the purely functional programming style. By handling time flow in a uniform and pervasive manner an application gains clarity and reliability. Just as lazy evaluation can eliminate the need for complex control structures, a uniform notion of time flow supports a more declarative programming style that hides a complex underlying mechanism.

This paper was inspired by some gentle taunting on the

Haskell GUI list by George Russell [15]:

> I have to say I'm very sceptical about things like Fruit[1] which rely on reactive animation, ever since I set our students an exercise implementing a simple space-invaders game in such a system, and had no end of a job producing an example solution. Things like getting an alien spaceship to move slowly downward, moving randomly to the left and right, and bouncing off the walls, turned out to be a major headache. Also I think I had to use "error" to get the message out to the outside world that the aliens had won. My suspicion is that reactive animation works very nicely for the examples constructed by reactive animation folk, but not for my examples.

We believe two basic problems have led to the current state of affairs. First, the original reactive animation systems such as Fran [7, 6] and FAL [8] lacked features that were essential to this application domain. As FRP has evolved we have added a number of important capabilities, most notably switching over dynamic collections of reactive entities, that make this example much more manageable. Also, the use of arrows [10] and the arrow notation [14] makes it easier to write and understand FRP programs.

The second problem is one that we address in this paper: the lack of good, practical examples of FRP-based code for examples that are more complex than a simple paddleball game. In this paper we show that *Yampa*, our current implementation of FRP, has the right functionality for this sort of application by presenting an implementation of Space Invaders that illuminates the use of Yampa in this domain. We are concerned with presenting Yampa from a user's perspective: the design and implementation of Yampa has been discussed elsewhere [9, 13].

After presenting our Space Invaders implementation, we describe some of the advantages of Yampa over other imple-

[1]Fruit is a GUI toolkit based on Yampa [4].

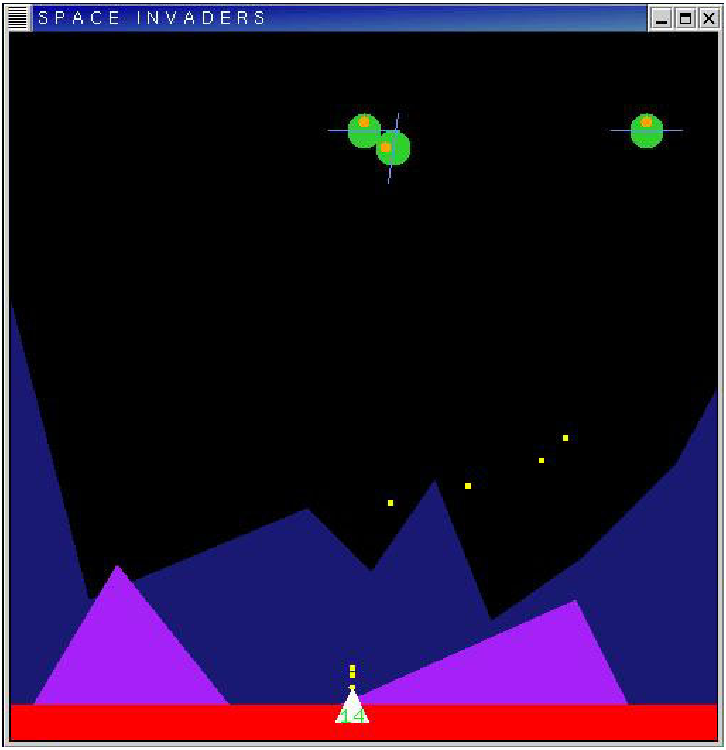


Figure 1: Screen-shot of Space Invaders

mentation alternatives. In particular, the way Yampa allows reactive entities to encapsulate state leads to highly modular and extensible programs. We demonstrate this by showing how features can be added to the game with very minor changes to the source code.

The remainder of this paper is organized as follows: Section 2 describes the Space Invaders game, and serves as an informal requirements specification. Section 3 reviews the essential ideas of Yampa and the arrows syntax. Section 4 shows how reactive game objects can be specified simply and concisely as signal functions in Yampa. Section 5 composes the individual reactive components into a complete game using Yampa's support for switching over dynamic collections. The game is extended in section 6, with hardly any changes required to the game structure. Section 7 then discusses alternative implementation approaches and the advantages of Yampa. Finally, sections 8 and 9 present related work and our conclusions.

2. GAME PLAY

Our version of Space Invaders is based on the classic 2-D arcade game of the same name. This section briefly describes the game, and will serve as a highly informal requirements specification. A screen-shot of our Space Invaders is shown in figure 1.

Aliens are invading a planet in a galaxy far, far away, and the task of the player is to defend against the invasion for as long as possible. The invaders, in flying saucers, enter the game at the top of the screen. Each alien craft has a small engine allowing the ship to maneuver and counter the gravitational pull. The thrust of the engine is directed by turning the entire saucer, i.e. by adjusting its *attitude*. Aliens try to maintain a constant downward landing speed, while maneuvering to horizontal positions that are picked at random every now and again.

The player controls a gun (depicted as a triangle) that can move back and forth horizontally along the bottom of the screen. The gun's horizontal position is controlled (indirectly) by the mouse.

Missiles are fired by pressing the mouse button. The initial position and velocity of the missile is given by the position and velocity of the gun when the gun is fired. The missiles are subject to gravity, like other objects. To avoid self-inflicted damage due to missiles falling back to the planet, missiles self-destruct after a preset amount of time.

There are also two repelling force fields, one at each edge of the screen, that effectively act as invisible walls.[2] Moving objects that happens to "bump" into these fields will experience a fully elastic collision and simply bounce back.

The shields of an alien saucer are depleted when the saucer is hit by a missile. Should the shields become totally depleted, the saucer blows up. Shields are slowly recharged when below their maximal capacity, however. Whenever all aliens in a wave of attack have been eliminated, the distant mother ship will send a new wave of attackers, fiercer and more numerous than the previous one. The game ends when an alien successfully lands on the planet.

3. YAMPA

This section gives a short introduction to Yampa, a language embedded in Haskell for describing reactive systems. Yampa is based on ideas from Fran [7, 6] and FRP [18]. We only give a brief summary of Yampa here; a more detailed account is given in the Yampa programming tutorial [9].

3.1 Concepts

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

$$\texttt{Signal}\,\alpha \approx \texttt{Time} \rightarrow \alpha$$

`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`:

$$\texttt{SF}\,\alpha\,\beta \approx \texttt{Signal}\,\alpha \rightarrow \texttt{Signal}\,\beta$$

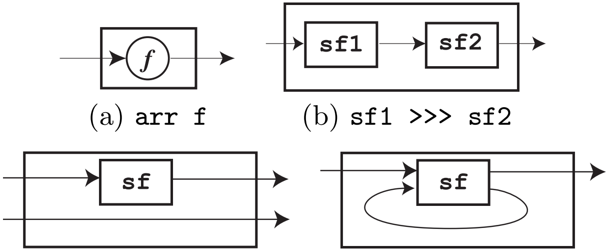
When a value of type $\texttt{SF}\,\alpha\,\beta$ is applied to an input signal of type $\texttt{Signal}\,\alpha$, it produces an output signal of type $\texttt{Signal}\,\beta$.

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 in to the box's input port and another signal flowing out of the box's output port.

3.2 Composing Signal Functions

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 framework proposed by Hughes [10]. Three combinators from that framework are

[2]It is believed that they are remnants of an ancient defense system put in place by the technologically advanced, mythical Predecessors.



(a) `arr f`   (b) `sf1 >>> sf2`
(c) `first sf`   (d) `loop sf`

Figure 2: Some core signal function combinators.

`arr`, which lifts an ordinary function to a stateless signal function, and the two signal function composition combinators `<<<` and `>>>`:

```
arr   :: (a -> b) -> SF a b
(<<<) :: SF b c -> SF a b -> SF a c
(>>>) :: SF a b -> SF b c -> SF a c
```

Yampa also provides a combination, the arrow-compose combinator:

```
(^<<) :: (b -> c) -> SF a b -> SF a c
```

Through the use of these and related plumbing combinators, arbitrary signal function networks can be expressed. Figure 2 illustrates some of these combinators.

The core primitives all have simple, precise denotations in terms of the conceptual model presented in section 3.1. For example, `arr` is (conceptually) pointwise application:

```
arr :: (a -> b) -> SF a b
arr f = \s -> \t -> f (s t)
```

In order to ensure that signal functions are executable, we require them to be *causal*: The output of a signal function at time t is uniquely determined by the input signal on the interval $[0, t]$. All primitive signal functions in Yampa are causal and all combinators preserve causality.

3.3 Arrow Syntax

One benefit of using the arrow framework in Yampa is that it allows us to use Paterson's arrow notation [14]. Paterson's syntax (currently implemented as a preprocessor for Haskell) effectively allows signals to be named, despite signals not being first class values. This eliminates a substantial amount of plumbing resulting in much more legible code. In fact, the plumbing combinators will rarely be used in the examples in this paper. In this syntax, an expression denoting a signal function has the form:

```
proc pat -> do
    pat_1 <- sfexp_1 -< exp_1
    pat_2 <- sfexp_2 -< exp_2
    ...
    pat_n <- sfexp_n -< exp_n
    returnA -< exp
```

Note that this is just *syntactic sugar*: the notation is translated into plain Haskell using the arrow combinators.

The keyword `proc` is analogous to the λ in λ-expressions, *pat* and pat_i are patterns binding signal variables pointwise by matching on instantaneous signal values, *exp* and exp_i are expressions defining instantaneous signal values, and $sfexp_i$ are expressions denoting signal functions. The idea is that the signal being defined pointwise by each exp_i is fed into the corresponding signal function *sfexp*$_i$, whose output is bound pointwise in *pat*$_i$. The overall input to the signal function denoted by the `proc`-expression is bound by *pat*, and its output signal is defined by the expression *exp*. The signal variables bound in the patterns may occur in the signal value expressions, but *not* in the signal function expressions (*sfexp*$_i$). An optional keyword `rec`, applied to a group of definitions, permits signal variables to occur in expressions that textually precede the definition of the variable, allowing recursive definitions (feedback loops). Finally,

```
let pat = exp
```

is shorthand for

```
pat <- identity -< exp
```

where `identity` is the identity signal function, allowing binding of instantaneous values in a straightforward way.

For a concrete example, consider the following:

```
sf = proc (a,b) -> do
    c1 <- sf1 -< a
    c2 <- sf2 -< b
    c  <- sf3 -< (c1,c2)
    rec
      d <- sf4 -< (b,c,d)
    returnA -< (d,c)
```

Here we have bound the resulting signal function to the variable `sf`, allowing it to be referred by name. Note the use of the tuple pattern for splitting `sf`'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`.

3.4 Events and Event Sources

While some aspects of a program (such as the mouse position) are naturally modeled as continuous signals, other aspects (such as the mouse button being pressed) are more naturally modeled as *discrete events*. To model discrete events, we introduce the `Event` type, isomorphic to Haskell's `Maybe` type:

```
data Event a = NoEvent | Event a
```

A signal function whose output signal is of type `(Event T)` for some type `T` is called an *event source*. The value carried with an event occurrence may be used to carry information about the occurrence. The operator `tag` is often used to associate such a value with an occurrence:

```
tag :: Event a -> b -> Event b
```

3.5 Switching

The structure of a Yampa system may evolve over time. These structural changes are known as *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
```

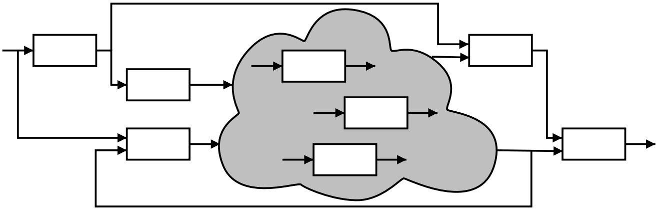


Figure 3: System of interconnected signal functions with varying structure

`switch` switches from one subordinate signal function into another when a switching event occurs. Its first argument is the signal function that initially is active. It outputs a pair of signals. The first defines the overall output while the initial signal function is active. The second signal carries the event that will cause the switch to take place. Once the switching event occurs, `switch` applies its second argument to the value tagged to the event and switches into the resulting signal function.

Yampa also includes *parallel* switching constructs that maintain dynamic collections of signal functions connected in parallel. Signal functions can be added to or removed from such a collection at runtime in response to events; see figure 3. The first class status of signal functions in combination with switching over dynamic collections of signal functions makes Yampa an unusually flexible language for describing hybrid systems [13].

3.6 Animating Signal Functions

To actually execute a Yampa program we need some way to connect the program's input and output signals to the external world. Yampa provides the function `reactimate` (here slightly simplified) for this purpose:

```
reactimate :: IO (DTime,a)      -- sense
              -> (b -> IO ())  -- actuate
              -> SF a b
              -> IO ()
```

The first argument to reactimate (`sense`) is an IO action that will obtain the next input sample along with the amount of time elapsed (or "delta" time, `DTime`) since the previous sample. The second argument (`actuate`) is a function that, given an output sample, produces an IO action that will process the output sample in some way. The third argument is the signal function to be animated. Reactimate approximates the continuous-time model presented here by performing discrete sampling of the signal function, invoking `sense` at the start and `actuate` at the end of each time step.

In the context of our video game, we use `sense` and `actuate` functions which read an input event from the window system and render an image on the screen, respectively.

4. PROGRAMMING REACTIVE GAME OBJECTS

As described in section 2, there are three essential entities or *objects* in the space invaders game: the gun, the invaders and the missiles. They all react to external events and stimuli, such as collisions with other objects and control commands from the player; i.e., they are *reactive*. This section explains how to develop and test such reactive objects in Yampa by presenting an implementation of a somewhat simplified gun. This serves as a good introduction to section 5, where the implementation of the real game is presented in detail. However, this section also exemplifies a useful Yampa development strategy where individual reactive objects are first developed and tested in isolation, and then wired together into more complex systems. The last step may require some further refinement of the individual object implementations, but the required changes are usually very minor.

4.1 Implementing a Gun

Each game object produces time-varying output (such as its current position) and reacts to time-varying input (such as the mouse-controlled pointer position). The gun, for example, has a time-varying position and velocity and will produce an output event to indicate that it has been fired. On the input side, the gun's position is controlled with the mouse, and the gun firing event is emitted in response to the mouse button being pressed. We can thus represent the gun with the following types:

```
data SimpleGunState = SimpleGunState {
  sgsPos :: Position2,
  sgsVel :: Velocity2,
  sgsFired :: Event ()
}

type SimpleGun = SF GameInput SimpleGunState
```

where `GameInput` is an abstract type representing a sample of keyboard and mouse state, and `Position2` and `Velocity2` are type synonyms for 2-dimensional vectors.

A simple physical model and a control system for the gun can be specified in just a few lines of Yampa code:

```
simpleGun :: Position2 -> SimpleGun
simpleGun (Point2 x0 y0) = proc gi -> do
  (Point2 xd _) <- ptrPos -< gi
  rec
    -- Controller
    let ad = 10 * (xd - x) - 5 * v

    -- Physics
    v <- integral -< clampAcc v ad
```

```
         x <- (x0+) ^<< integral -< v

       fire <- leftButtonPress -< gi
       returnA -< SimpleGunState {
                    sgsPos = (Point2 x y0),
                    sgsVel = (vector2 v 0),
                    sgsFired = fire
                  }
```

In the first line in the body of `simpleGun`, the horizontal position of the pointer is extracted from the `GameInput` signal using the `ptrPos` signal function (provided by the bindings to the graphics library). We could, of course, simply define the position of the gun to be that of the pointer. However, to add some degree of physical realism, the model of the gun includes inertia and adds bounds on the acceleration and velocity. The position of the pointer is thus interpreted as the gun's *desired* horizontal position, `xd`, and a control system is then used to compute the *desired* acceleration, `ad`, based on the difference between the current and desired position and the current velocity. The goal of the control system is to make the gun reach the desired position quickly subject to the physical constraints.[3]

The bounds on the acceleration and velocity are imposed through the auxiliary function `clampAcc`:

```
clampAcc v ad =
  let a = symLimit gunAccMax ad
  in if (-gunSpeedMax) <= v && v <= gunSpeedMax
```

```
            || v < (-gunSpeedMax) && a > 0
            || v > gunSpeedMax && a < 0
         then a
         else 0

    limit ll ul x = max ll (min ul x)

    symLimit l = limit (-abs l) (abs l)
```

The equations for the velocity `v` and horizontal position `x` are simply Newton's laws of motion, stated in terms of integration:

```
    integral :: VectorSpace a k => SF a a
```

4.2 Testing the Gun

Individual game objects such as `simpleGun` can be tested in isolation using `reactimate` (described in section 3.6). To test `simpleGun` we simply define an ordinary Haskell function to render a gun state as a visual image, and compose (using `>>>`) `simpleGun` with a lifted (using `arr`) version of this function:

```
    renderGun :: SimpleGunState -> G.Graphic
    renderGun = ...

    gunTest :: IO ()
    gunTest = runGame (simpleGun >>> arr renderGun)
```

`runGame` is defined using `reactimate` and suitable IO actions for reading an input event from the window system and rendering a graphic on the screen. Executing `gunTest` will open a window in which the gun can be positioned using the mouse.

5. THE GAME

5.1 Game Structure

In section 4 we showed that individual game objects could be implemented as signal functions. However, in order to form a complete game, we will need a *collection* of game objects (the gun, aliens and missiles) that are all active simultaneously. We will also need some facility to add or remove objects from the game in response to events such as missiles being fired and missiles hitting targets. Thus the collection has to be *dynamic*. Moreover, the implementation of the gun in section 4 only reacted to external mouse input, whereas in the actual game objects will also need to react to each other.

The addition and deletion of signal functions constitute structural changes of the network of interconnected active signal functions. In Yampa, structural changes are effectuated by means of switching between modes of continuous

[3]The control system coefficients in this example have not been mathematically optimized.

operation. In particular, Yampa provides a family of *par-*

allel switching combinators that allow collections of signal functions to be simultaneously switched in to and out of the network of active signal functions as a group. When switched in, the signal functions in these collections are connected in parallel (hence the name parallel switch). The collections are allowed to change at the points of switching, in effect allowing them to be dynamic. Combining parallel switching with feedback enables game objects to interact in arbitrary ways. Figure 4 shows the resulting overall structure of the game.

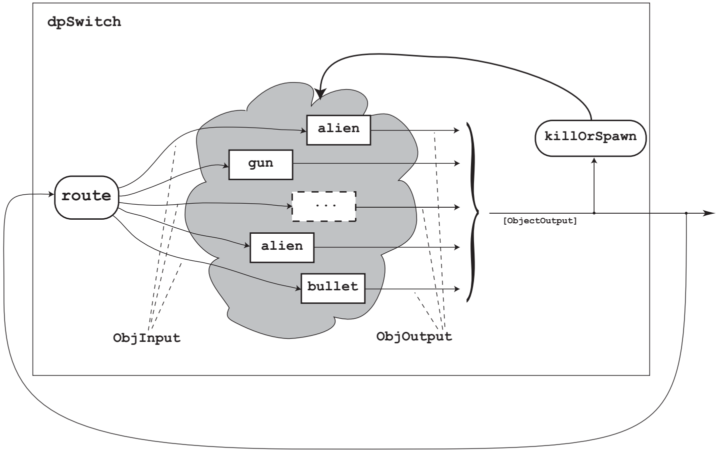


Figure 4: Dynamic collection of game objects maintained by dpSwitch.

The design and implementation of Yampa's parallel switching combinators is described in detail elsewhere [13]. Here we will focus on how to use one particular parallel switching combinator, `dpSwitch`, in the context of the Space Invaders game. As illustrated in figured 4, there are two key aspects of maintaining the dynamic collection which are under control of the user of `dpSwitch`:

- The `route` function, which specifies how the external input signal to the collection is distributed (or "routed") to individual members of the collection.

- The `killOrSpawn` function, which observes the output of the collection (and possibly the external input signal) and determines when signal functions are added to or removed from the collection.

In order to specify these two functions, we must first develop a clear understanding of how the different members of the collection interact with one another and the outside world, and under what circumstances signal functions are added to or removed from the collection. We will do that in the next section.

5.2 The Object Type

The type of `dpSwitch` requires that all signal functions in the dynamic collection to be maintained by `dpSwitch` have a uniform type. For our Space Invaders game, we use the following type for all game objects:

```
type Object = SF ObjInput ObjOutput
```

We must ensure that the types for the input and output signal (`ObjInput` and `ObjOutput`) of this common type are rich enough to allow us to implement all necessary interactions between game objects. Based on a careful reading of the requirements in section 2, a concise specification of the interactions between game objects is as follows:

- A missile is spawned when the gun is fired.
- Missiles are destroyed when they hit an alien craft or when they self-destruct after some finite time.
- An alien craft is destroyed when it has been sufficiently damaged from a number of missile hits.

The analysis makes it clear that game objects react only to collisions with other objects and external device input. Hence we can define the `ObjInput` type as follows:

```
data ObjInput = ObjInput {
    oiHit       :: Event (),
    oiGameInput :: GameInput
}
```

On the output side, we observe that there are really two distinct kinds of outputs from game objects. First, there is the *observable object state*, consisting of things like the position and velocity of each object that must be presented to the user and that must be available to determine object interactions (such as collisions). Second, there are the events that cause addition or removal of signal functions from the dynamic collection, such as the gun being fired or an alien being destroyed. This leads to the following type definition:

```
data ObjOutput = ObjOutput {
    ooObsObjState :: !ObsObjState,
    ooKillReq     :: Event (),
    ooSpawnReq    :: Event [Object]
}
```

The `ooObsObjState` field contains the observable object state of type `ObsObjState`:

```
data ObsObjState =
      OOSGun {
          oosPos    :: !Position2,
          oosVel    :: !Velocity2,
          oosRadius :: !Length,
      }
    | OOSMissile {
          oosPos    :: !Position2,
          oosVel    :: !Velocity2,
          oosRadius :: !Length
      }
    | OOSAlien {
          oosPos    :: !Position2,
          oosHdng   :: !Heading,
          oosVel    :: !Velocity2,
          oosRadius :: !Length
      }
```

Note in the above that the constructors differentiate the different kinds of game objects. However, the `oosPos`, `oosVel` and `oosRadius` fields are common to each alternative. This simplifies the implementation of many functions (such as collision detection), since collision detection can simply apply `oosPos` and `oosRadius` to any observable object state value, without concern for the kind of game object that produced this value.

The `ooKillReq` and `ooSpawnReq` fields are object destruction and creation events, passed to `killOrSpawn` by `dpSwitch`. On an occurrence of `ooKillReq`, the originating object is removed from the collection. On an occurrence of `ooSpawnReq`, the list of objects tagged to the event will be spliced in to the dynamic collection. A list rather than a singleton object is used for the sake of generality. For example, this allows an exploding alien craft to spawn a list of debris.

5.3 Gun Behavior

We now turn to describing the behavior of the game objects. This section covers the gun; the next section deals with the aliens. We leave out the missiles since that code is basically a subset of the code describing the alien behavior. Section 4 explained the basic idea behind modeling game objects using signal functions. With that as a starting point, we have to develop objects that conform to the `GameObject` type and interact properly with the world around them and the object creation and destruction mechanism.

The resulting code for the gun is very similar to what we have already seen:

```
gun :: Position2 -> Object
gun (Point2 x0 y0) = proc oi -> do
  let gi = oiGameInput oi
  (Point2 xd _) <- ptrPos -< gi

  rec
    -- Controller
    let ad = 10 * (xd - x) - 5 * v

    -- Physics
    v <- integral -< clampAcc v ad
    x <- (x0+) ^<< integral -< v

  fire <- leftButtonPress -< gi
  returnA -<
    ObjOutput {
      ooObsObjState = oosGun (Point2 x y0)
                             (vector2 v 0),
      ooKillReq     = noEvent,
      ooSpawnReq    =
        fire `tag`
          [missile
             (Point2 x (y0 + (gunHeight/2)))
```

```
                 (vector2 v missileInitialSpeed)]
        }

    missile :: Position2 -> Velocity2 -> Object
    missile p0 v0 = proc oi -> do ...
```

The only significant change is the interaction with the object creation and destruction mechanism. Note how `noEvent` is used to specify that a gun never removes itself from the dynamic collection, and how a signal function representing a new missile is tagged onto the `fire` event, yielding an object creation event.

5.4 Alien Behavior

This section presents the code describing the behavior of aliens. Like the gun, the description contains a simple physical model along with a control system, the only difference being that aliens move in two dimensions. Unlike the gun, the target for the control system is generated internally, partly through a random process. Also unlike the gun, aliens can be destroyed, which happens when their shields become depleted.

```
    alien :: RandomGen g =>
      g -> Position2 -> Velocity -> Object
    alien g p0 vyd = proc oi -> do
      rec
        -- Pick a desired horizontal position
        rx   <- noiseR (xMin, xMax) g -< ()
        smpl <- occasionally g 5 ()   -< ()
        xd   <- hold (point2X p0) -< smpl `tag` rx

        -- Controller
        let axd = 5 * (xd - point2X p)
                  - 3 * (vector2X v)
            ayd = 20 * (vyd - (vector2Y v))
            ad  = vector2 axd ayd
            h   = vector2Theta ad

        -- Physics
        let a = vector2Polar
                    (min alienAccMax
                         (vector2Rho ad))
                    h
        vp  <- iPre v0    -< v
        ffi <- forceField -< (p, vp)
        v   <- (v0 ^+^) ^<< impulseIntegral
              -< (gravity ^+^ a, ffi)
        p   <- (p0 .+^) ^<< integral -< v

        -- Shields
        sl  <- shield -< oiHit oi
        die <- edge   -< sl <= 0

      returnA -< ObjOutput {
                     ooObsObjState = oosAlien p h v,
                     ooKillReq     = die,
                     ooSpawnReq    = noEvent
                   }
      where
        v0 = zeroVector
```

Picking a desired position is accomplished by occasionally sampling a noise signal. The signal function `noiseR` generates a random signal (noise) in the specified interval. The signal function `occasionally` is an event source for which the *average* event occurrence density can be specified. In this case, events occur on average once every 5 seconds. Thus, on each such occurrence, the noise source is sampled by tagging its value `rx` to the sample event `smpl`. A piecewise continuous signal indicating the desired horizontal position at all points in time is then obtained by feeding the discrete noise samples through the signal function `hold`. The typings for these signal functions are as follows:

```
noiseR :: (RandomGen g, Random a) =>
  (a,a) -> g -> SF () a
occasionally :: RandomGen g =>
  g -> Time -> b -> SF a (Event b)
hold :: a -> SF (Event a) a
```

The output from the control system is the desired acceleration `ad`, a 2-dimensional vector. The horizontal component of this vector `axd` is computed based on the difference between the current horizontal position and desired horizontal position, along with the current velocity. The vertical component `ayd` is given by a simple proportional controller that tries to maintain a constant vertical velocity by comparing the desired vertical speed with the actual vertical speed. The direction of the desired acceleration vector in turn gives the attitude `h` of the alien craft.[4]

In the physical model, the real acceleration `a` is obtained by limiting the desired acceleration according to the capability of the craft. The velocity `v` and position `p` of the craft are then given by integration of the sum of the acceleration from the craft's engines and the gravity in two steps according to Newton's laws of motion. The force field is modeled by the auxiliary signal function `forceField`. It outputs an impulsive force `ffi`, modeled as an event, whenever an alien craft bumps into the force field. The orientation and magnitude of this impulsive force is such that the craft experiences an instantaneous, fully elastic collision. The effect of impulsive forces acting on the craft, a discontinuous change in velocity, is taken into account through the use of the signal function `impulseIntegral` rather than `integral` in the equation for the velocity.[5]

```
impulseIntegral :: VectorSpace a k =>
  SF (a, Event a) a
```

The shield is modeled by the auxiliary signal function `shield`. Its output is the present shield level `sl`. Its input are events indicating that the craft has been hit by a missile, which causes the shield level to drop. Countering this, the shield is recharged at a constant rate, up to a maximal level. The shield level is monitored, and should it drop below zero, an event `die` is generated that indicates that the craft has been destroyed. The `die` event is defined using Yampa's *edge detector* primitive: `edge :: SF Bool (Event ())`.

Finally, the output signal is defined. The position `p`, attitude `h`, and velocity `v` make up the observable object state. The `die` event is made into a kill request, while `noEvent` is used to specify that alien crafts do not spawn any other objects.

5.5 Maintaining Dynamic Collections

We will now put the pieces we have developed thus far together into a complete game, as outlined in figure 4. Thus we have to use `dpSwitch` to maintain a dynamic collection of game objects, we have to design the control mechanism for adding and deleting objects (`killAndSpawn`), and we have to set up the proper interconnection structure by means of the `dpSwitch` routing function and feedback.

`dpSwitch` has the following signature:

```
dpSwitch :: Functor col =>
  (forall sf . (a -> col sf -> col (b, sf)))
  -> col (SF b c)
  -> SF (a, col c) (Event d)
  -> (col (SF b c) -> d -> SF a (col c))
  -> SF a (col c)
```

The first argument is the routing function. Its purpose is to pair up each running signal function in the collection maintained by `dpSwitch` with the input it is going to see

[4]Rotational inertia is not modeled: it is assumed that the alien craft can change direction instantaneously.
[5]The second author has developed a more systematic alternative based on Dirac impulses [12].

at each point in time. The rank-2 universal quantification of `sf` renders the members of the collection opaque to the routing function; all the routing function can do is specify how the input is distributed. The second argument is the initial collection of signal functions. The third argument is a signal function that observes the external input signal and the output signals from the collection in order to produce a switching event. In our case, this is going to be the `killAndSpawn` function alluded to in figure 4. The fourth argument is a function that is invoked when the switching event occurs, yielding a new signal function to switch into based on the collection of signal functions previously running and the value carried by the switching event. This allows the collection to be updated and then switched back in, typically by employing `dpSwitch` again.

The collection argument to the function invoked on the switching event is of particular interest: it captures the *continuations* of the signal functions running in the collection maintained by `dpSwitch` at the time of the switching event, thus making it possible to preserve their state across a switch. Since the continuations are plain, ordinary signal functions, they can be resumed, discarded, stored, or combined with other signal functions.

In order to use dpSwitch, we first need to decide what kind of collection to use. In cases where it is necessary to route specific input to specific signal functions in the collection (as opposed to broadcasting the same input to everyone), it is often a good idea to "name" the signal functions in a way that is invariant with respect to changes in the collection. For our purposes, an association list will do just fine, although we will augment it with a mechanism for generating names automatically. We call this type an *identity list*, and its type declaration along with the signatures of some useful utility functions, whose purpose and implementation should be fairly obvious, are as follows:

```
type ILKey = Int
data IL a = IL { ilNextKey :: ILKey,
                 ilAssocs  :: [(ILKey, a)] }

emptyIL   :: IL a
insertIL_ :: a -> IL a -> IL a
listToIL  :: [a] -> IL a
elemsIL   :: IL a -> [a]
assocsIL  :: IL a -> [(ILKey, a)]
deleteIL  :: ILKey -> IL a -> IL a
mapIL     :: ((ILKey,a)->b) -> IL a -> IL b
```

IL is of course also an instance of Functor. Incidentally, associating some extra state information with a collection, like ilNextKey in this case, is often a quite useful pattern in the context of dpSwitch.

Let us use dpSwitch to implement the core of the game:

```
gameCore :: IL Object
         -> SF (GameInput, IL ObjOutput)
               (IL ObjOutput)
gameCore objs =
  dpSwitch route
           objs
           (arr killOrSpawn >>> notYet)
           (\sfs' f -> gameCore (f sfs'))
```

We will return to the details of the routing function and killOrSpawn below. But the basic idea is that the switching event from killOrSpawn carries a function that when applied to the collection of continuations yields a new signal function collection to switch into. That in turn is achieved by invoking gameCore recursively on the new collection.

killOrSpawn in a plain Haskell function that is lifted to the signal function level using arr. The resulting signal function is composed with notYet :: SF (Event a) (Event a) that suppresses initial event occurrences. Thus the overall result is a source of kill and spawn events that will not have any occurrence at the point in time when it is first activated. This is to prevent gameCore from getting stuck in an infinite loop of switching. The need for this kind of construct typically arises when the source of the switching events simply passes on events received on its input in a recursive setting such as the one above. Since switching takes no time, the new instance of the event source will see the exact same input as the instance of event source that caused the switch, and if that input is the actual switching event, a new switch would be initiated immediately, and so on for ever.

The routing function is straightforward. Its task is to pass on the game input to all game objects, and to detect collisions between any pair of interacting game objects and pass hit events to the objects involved in the collision:

```
route :: (GameInput, IL ObjOutput) -> IL sf
         -> IL (ObjInput, sf)
route (gi,oos) objs = mapIL routeAux objs
  where
    routeAux (k, obj) =
      (ObjInput {oiHit = if k ‘elem‘ hs
                         then Event ()
                         else noEvent,
                 oiGameInput = gi},
       obj)
    hs = hits (assocsIL
                (fmap ooObsObjState oos))
```

`route` invokes the auxiliary function `hits` that computes a list of keys of all objects that are involved in some collision. For all game objects, it then checks if the key of that object is contained in the list of colliding objects. If so, it sends a collision event to the object, otherwise not. `hits` performs its computation based on the fed-back object output. This gives the current position and velocities for all game objects. Two objects are said to collide if they partially overlap and if they are approaching each other. However, alien crafts do not collide in this version of the game.

killOrSpawn traverses the output from the game objects, collecting all kill and spawn events. If any event occurs, a switching event is generated that carries a function to update the signal function collection accordingly:

```
killOrSpawn :: (a, IL ObjOutput)
               ->(Event (IL Object->IL Object))
killOrSpawn (_, oos) =
  foldl (mergeBy (.)) noEvent es
  where
    es :: [Event (IL Object -> IL Object)]
    es = [ mergeBy (.)
                   (ooKillReq oo
                    `tag` (deleteIL k))
                   (fmap (foldl (.) id
                             . map insertIL_)
                          (ooSpawnReq oo))
           | (k,oo) <- assocsIL oos ]
```

A kill event is turned into a function that removes the object that requested to be deleted by partially applying deleteIL to the key of the object to be removed. A spawn event is turned into a function that inserts all objects in the spawn request into the collection using insertIL_. These individual functions are then simply composed into a single collection update function. We have found this approach to collection updating to be quite useful and applicable in a wide range of contexts [13].

5.6 Closing the Feedback Loop

We can now take one more step toward the finished game by closing the feedback loop. We also add features for game initialization and score keeping. The function `game` plays one round of the game. It generates a terminating event carrying the current (and possibly final) score either when the last alien craft in the current wave of attack is destroyed, or when the game is over due to an alien touch down:

```
game :: RandomGen g =>
  g -> Int -> Velocity -> Score ->
  SF GameInput ((Int, [ObsObjState]),
                Event (Either Score Score))
game g nAliens vydAlien score0 = proc gi -> do
  rec
    oos <- gameCore objs0 -< (gi, oos)
  score    <- accumHold score0
                  -< aliensDied oos
  gameOver <- edge -< alienLanded oos
  newRound <- edge -< noAliensLeft oos
  returnA -< ((score,
               map ooObsObjState
                   (elemsIL oos)),
              (newRound ‘tag‘ (Left score))
              ‘lMerge‘ (gameOver
                         ‘tag‘ (Right score)))
  where
```

```
        objs0 =
          listToIL
            (gun (Point2 0 50)
             : mkAliens g (xMin+d) 900 nAliens)
```

The central aspect of this function is the closing of the feedback loop using the recursive arrow syntax. The arguments to `game` are a random number generator (used to seed the signal functions representing the alien crafts), the number of alien crafts in this wave of attack, the desired landing speed of the aliens, and an initial score carried over from any preceding rounds. Score is kept by simply counting kill requests from aliens in the object output, and events signaling a new round and game over is obtained by applying edge detectors to predicates over the object output looking for the absence of alien crafts and the landing of an alien craft, respectively.

An unsatisfying aspect of the current design of the Yampa switching combinators is that the exact choice of combinator in `gameCore` (here `dpSwitch`) is critical to the design of `game`. This point is discussed further in section 5.8.

5.7 Playing Multiple Rounds

Finally, a multi-round game can be built on top of `game`. After each successful defeat of a wave of invaders, `game` is reinvoked with more and faster alien crafts, passing on the current score. Once an alien has landed, the game starts over from the beginning.

```
    multiRoundGame :: RandomGen g =>
    g -> SF GameInput (Int, [ObsObjState])
  multiRoundGame g = rgAux g nAliens0 vydAlien0 0
    where
      nAliens0 = 2
      vydAlien0 = -10

      rgAux g nAliens vydAlien score =
        switch (game g' nAliens vydAlien score)
              $ \status ->
        case status of
          Left score' ->
            rgAux g''
                  (nAliens+1)
                  (vydAlien-10)
                  score'
          Right finalScore ->
            rgAux g'' nAliens0 vydAlien0 0
        where
          (g', g'') = split g
```

All that then remains is to connect the top-level signal function to the outside world. This involves feeding in a signal of mouse positions and button presses, and mapping the output signal pointwise to a suitable graphic representation. The details are specific to the graphics systems that are used. In this particular case we were using HGL, the Haskell Graphics Library.

5.8 Which Switch?

Yampa provides a family of parallel switching combinators. Two members are `pSwitch` and `dpSwitch` that have exactly the same type signature, and as mentioned in section 5.6, which one is chosen can have a significant impact on the design of a program.

The difference between `pSwitch` and `dpSwitch` is that the output from the switcher *at the point of a switching event* in the former case is determined by the signal function being switched into, which in turn usually means from the outputs of the signal functions in a *new*, updated collection, whereas the output is the latter case is given by the output from the signal functions in the *old* collection. This allows `dpSwitch` to be *non-strict* in the switching event; i.e., the output from `dpSwitch` at any point in time can be determined without demanding the switching event at that same point in time. The "d" in the name of `dpSwitch` stands for "delayed", meaning that the effect of a switch cannot be observed immediately. All Yampa switchers have delayed versions, and all those are non-strict in the switching event.

Employing a switcher that is non-strict in the switching event may be enough to make it possible to close a feedback loop without any unit delay on the feedback path. In our case the lazy demand structure is such that this indeed is the case, and hence there is no unit delay (`iPre`) on the feedback path in `game` in section 5.6.

Using `dpSwitch` also means that the requests for removal from the objects are going to be visible outside the switcher. This was exploited for the score keeping mechanism. Had pSwitch been used, we would only be able to observe the output from the objects *remaining* after a switch. This does of course not include the output from objects that just removed themselves by emitting kill requests, and these requests are exactly what is counted for keeping score. A more robust alternative would be to associate extra state information with the collection type (like the counter used for naming in the identity list IL of section 5.5) and use that to keep score.

6. AN EXTENSION

Our current version of the gun is rather unrealistic: the ammunition supply is infinite, and a missile is fired whenever the left mouse button is pressed. To make the game more realistic, we extend the game by modifying the gun to add a *magazine* with bounded capacity. The gun will only fire a missile if there are missiles available in the magazine. We assume that the magazine is reloaded with new missiles (up to the magazine's capacity) at some fixed rate.

As we will see, adding this extension requires only small, localized changes to our existing program. We will revisit this point in the next section, when we consider other approaches to implementing games in Haskell.

```
magazine ::
  Int -> Frequency
      -> SF (Event ()) (Int, Event ())
magazine n f = proc trigger -> do
```

```
    reload <- repeatedly (1/f) () -< ()
    (level,canFire)
        <- accumHold (n,True) -<
               (trigger `tag` dec)
               `lMerge` (reload `tag` inc)
    returnA -< (level,
        trigger `gate` canFire)
    where
      inc :: (Int,Bool) -> (Int, Bool)
      inc (l,_) | l < n = (l + 1, l > 0)
                | otherwise = (l, True)
      dec :: (Int,Bool) -> (Int, Bool)
      dec (l,_) | l > 0 = (l - 1, True)
                | otherwise = (l, False)
```

First, `reload` is bound pointwise to an event signal that will occur repeatedly every `1/f` seconds. The lines that follow implement a simple state machine, where the state consists of `level`, an integer specifying the number of missiles currently in the magazine, and `canFire`, a Boolean value derived from `level` which specifies whether or not the gun will actually fire in response to a trigger event. The state machine is implemented using Yampa's `accumHold` primitive:

```
accumHold :: a -> SF (Event (a -> a)) a
```

The argument to `accumHold` is an initial state and the result is a signal function whose output signal is *piecewise constant*. The signal function maintains a state value internally which is also made available as its output signal. Whenever an event occurs on its input signal, the internal state is updated by applying a state-to-state function carried with the event occurrence. In this case, the input signal to `accumHold` is formed by merging the external event signal `trigger` with the internal event signal `reload`. An occurrence of these events is tagged to carry a function that modifies the state by decreasing or increasing the ammunition level, respectively. Finally, the output signal of `magazine` is simply a tuple consisting of the current ammunition level and the trigger event gated by the internal boolean signal `canFire`: trigger events from the input signal will only be passed to the output signal when `canFire` is `True`.

Of course, we must extend the `gun` implementation from section 5.3 to make use of the magazine. To do this, in the body of `gun` we simply replace the line

```
    fire <- leftButtonPress -< gi
```

with the lines:

```
    trigger <- leftButtonPress -< gi
    (level,fire) <- magazine 20 0.5 -< trigger
```

7. ALTERNATIVE IMPLEMENTATION APPROACHES

Of course, there are any number of other ways one might implement Space Invaders in Haskell without using Yampa's parallel switching framework. To consider just two:

- We could completely forego Yampa's switching primitives and dynamic collections framework. Instead of representing individual game objects as signal functions and composing then into a dynamic collection with `dpSwitch`, we could define a single, monolithic `GameState` type that would encapsulate the complete state of the game (consisting of both the internal and external state of the gun, all of the missiles and aliens). Given such a `GameState` type, the game could be defined entirely as a first-order signal function of type:

  ```
  type Game = SF GameInput GameState
  ```

  This Game type could be implemented using just basic Yampa primitives without the need for any switching constructs, just as we defined the `SimpleGun` itself in section 4.

- Alternatively, we could forego the Yampa framework entirely and implement our own top-level animation framework. This framework would implement a top-level loop to invoke a state transition function at regular intervals to process keyboard and mouse events and render an image computed from this game state. This is exactly the approach taken by Lüth [11] to implement an asteroids game in Haskell using HGL.

We believe that using Yampa and its parallel switchers offers important advantages over the above approaches.

First, developing individual game components as signal functions provides the programmer with a natural way to factor the game development task. Before diving into the larger task of composing the components into a complete game, the programmer can develop and test the individual game components in isolation, just as we did with the `SimpleGun` in section 4. In fact, we took exactly this incremental approach when developing the version of Space Invaders presented here.

Second, an essential feature of signal functions is that they can accumulate internal state that is not directly accessible via their input or output signals, which makes signal functions both *modular* and *extensible*. For example, while individual game objects expose their *velocity* on their output signal, some of the objects also have an internal *acceleration* signal that is hidden from the rest of the game. This feature is also what enabled us to add the auto-refilling ammo magazine to the gun. In the other implementation alternatives discussed above, the internal state of each object would have to be maintained as part of the monolithic `GameState` type. While there are ways of carefully designing a `GameState` type and state update functions so that each game entity can maintain its own localized internal state, signal functions are *always* modular in this sense.

8. RELATED WORK

As discussed in section 6, Christoph Lüth [11] presented an implementation of asteroids in Haskell, used as part of an introductory programming course. His implementation was split into two parts: the top level animation loop (in the IO monad) and a pure function of type `Event -> State -> State` to process events. He provided students with a reference implementation that implemented just a single space ship and basic animation and reactivity, and required the students to implement other game objects and the interaction between components, which they could do by writing their own state transition function. We believe that Yampa could be used in a similar context given a similar level of hand-holding. In fact, the predecessor to Yampa was successfully used in that way in an undergraduate robotics class taught at Yale.

In their joint Ph.D. thesis [2], Hallgren and Carlsson outline an implementation of Space Invaders in Fudgets. Fudgets is based on discrete-time, asynchronous *stream* processors, whereas Yampa is based on continuous-time, synchronous *signal* processors, but the programming styles of both systems are fundamentally similar. The particular Space Invaders implementation in Fudgets outlined by Hallgren and Carlsson is based on a static network of Fudgets, with a fixed number of aliens and missiles. In contrast, our implementation is centered around `dpSwitch`, Yampa's primitive for higher-order dynamic collections of signal functions, and aliens and missiles are added to and removed from our game at runtime in response to events. Fudgets provides similar functionality to `dpSwitch` through its `dynListF` primitive, but `dynListF` only allows one Fudget to be added or removed from the collection at a time, whereas `dpSwitch` provides synchronous bulk updates.

Another, rather different approach to modular decomposition of interactive applications and localized state handling in a purely functional language is Clean's Object I/O System [1]. Clean's object I/O system is the basis for a sophisticated (and highly polished) GUI library, as well as a library for 2-D scrolling "platform" games. A fundamental difference between Clean's objects and Yampa's signal functions is that every Clean object has access to a global, mutable "World" that can be updated in response to events. These updates are specified using callback functions whose type is essentially `s -> World -> (World,s)` for some local state type `s`; Clean's uniqueness type system [17] ensures that the world is used in a single-threaded way. Clean object I/O depends on updates to the world value for inter-object communication and to create and destroy objects dynamically at runtime. In contrast, Yampa eschews all notions of a global, mutable external "world". Yampa requires all component interactions to take place over connections made explicitly with wiring combinators, and uses parallel switching combinators to express dynamic creation and removal of signal functions. As has been argued elsewhere [4, 3] using explicit connections as the sole means of communication between components enables the programmer to reason precisely about inter-component interactions; such reasoning is simply not practical if every component can perform arbitrary side-effects on a shared "world".

In all fairness to the original email message to the Haskell list that prompted this work, there certainly were substantial limitations in previous Functional Reactive Programming systems that would have made it difficult or impossible to implement a version of space invaders as sophisticated as the one presented here. Fran [7, 6], the progenitor of all subsequent FRP systems, provided the user with signals (called "behaviors" in Fran) as first-class values. At first glance, this appears to be considerably more convenient than Yampa's combinator-based design, since wiring in Fran can be expressed directly using Haskell's function application and recursion, and dynamic collections can be formed using higher-order behaviors and ordinary Haskell data types. Unfortunately, the implementation of Fran suffered from a number of serious operational problems, most notably time and space leaks [5]. A subsequent implementation, Hudak and Wan's FRP library [18], attempted to avoid these operational problems by blurring the distinction between signals and signal functions. Unfortunately, this design made it impossible to maintain the internal state of a behavior across a `switch`, which proved a severe limitation when building complete applications. Subsequent attempts to regain Fran's expressive power (by the introduction of a `runningIn` combinator) greatly complicated the library semantics, particularly in the presence of recursion.

Yampa was developed as a successor to these previous systems based on our experience attempting to use Fran and FRP to construct graphical users interfaces and a large, interactive robotics simulator. Yampa maintains a clear distinction between the notions of *signal* and *signal function* and avoids many of the operational issues of earlier FRP systems by only allowing signal functions, not signals, as first class entities. A potential drawback to the Yampa approach is that all wiring must be expressed via combinators. However, we have found that the arrows syntactic sugar largely ameliorates this issue.

Another successor to Fran, FranTk [16], provided functionality equivalent to that of Yampa's `dpSwitch` via what it referred to as "Behavioral collections". However, FranTk depended crucially on an extended IO monad (the GUI monad) for adding or removing behaviors from behavioral collections. As noted in our earlier discussion of Clean's I/O system, an important goal of Yampa is to provide a flexible, higher-order data-flow language without appealing to a global, mutable "World" type, whether provided explicitly (as in Clean), or hidden behind a monadic interface.

9. CONCLUSIONS

FRP addresses application domains that have not been traditionally associated with pure functional programming. While the imperative features of Haskell can be used to implement reactive networks of stateful objects in a traditional style, this approach suffers from many of the same problems that standard object oriented programming would. Using

Yampa, we have accomplished a number of things:

- Interfaces between the system components are fully explicit. That is, every signal function makes it explicit what stimuli it is reacting to and what effects it can have on the outside world. Since the objects in this system do not use the IO monad, they cannot interact with each other in unexpected ways.

- Our system is modular in a way that allows significant changes in the design of the constituent objects to be made without requiring major structural changes to the system.

- Time flow is managed in a uniform and synchronous manner. This synchrony avoids event ordering anomalies that characterize object oriented systems. Mutually recursive signals are represented in a simple and intuitive manner.

- Dynamic collections of signal functions support a changing world in a simple and uniform manner. While `pSwitch` and its siblings may seem daunting, they implement a significant and error prone aspect of the system in a semantically well-defined manner.

Of course, Yampa is not without flaws. For example, choosing the right kind of switching combinator can be tricky, as discussed in section 5.8, and the full implications of the choice is not always readily apparent. And understanding when to use the event suppression combinator `notYet`, or where to employ the unit delay `iPre`, both critical for avoiding infinite loops or black holes in certain cases, requires a firm grasp of rather subtle details of the Yampa semantics.

It could also be argued that the syntax is not as intuitive as it should be. While the arrow notation is a huge improvement over the raw combinator style, it offers no help for the specifics of Yampa like switching. Moreover, the Yampa arrow has properties that would allow a less "linear" syntax than what is required for general arrows, closer to the standard function application notation. Indeed, Fran and earlier versions of FRP *did* use such syntax, as discussed in section 8. For these reasons we are considering designing a Yampa specific syntax, but for now we are content to use an existing pre-processor.

Nevertheless, we feel that the Yampa style of programming is a very good way to motivate students and others familiar with imperative programming to learn basic functional programming techniques and appreciate the expressiveness and generality of the purely functional programming style.

10. REFERENCES

[1] Peter Achten and Rinus Plasmejer. Interactive functional objects in clean. In *Proc. of 9th International Workshop on Implementation of Functional Languages, IFL'97*, volume 1467 of *LNCS*, pages 304–321, September 1997.

[2] Magnus Carlsson and Thomas Hallgren. *Fudgets –*

Purely Functional Processes with Applications to Graphical User Interfaces. PhD thesis, Department of Computing Science, Chalmers University of Technology, 1998.

[3] Antony Courtney. Functionally modeled user interfaces. In *Proceedings DSV-IS 2003, Eurographics Workshop on Design, Specification and Verification of Interactive Systems*, 2003.

[4] Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, Firenze, Italy, September 2001.

[5] Conal Elliott. Functional implementations of continuous modelled animation. In *Proceedings of PLILP/ALP '98*. Springer-Verlag, 1998.

[6] 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).

[7] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of ICFP'97: International Conference on Functional Programming*, pages 163–173, June 1997.

[8] Paul Hudak. *The Haskell School of Expression – Learning Functional Programming through Multimedia*. Cambridge University Press, Cambridge, UK, 2000.

[9] Paul Hudak, Antony Courtney, Henrik Nilsson, and

John Peterson. Arrows, robots, and functional reactive programming. In Johan Jeuring and Simon Peyton Jones, editors, *Advanced Functional Programming, 4th International School 2002*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer-Verlag, 2003.

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

[11] Christoph Lüth. Haskell in space. In Simon Thompson Michael Hanus, Shriram Krishnamurthi, editor, *Functional and Declarative Programming in Education (FDPE 2002)*, pages 67– 74. Technischer Bereicht 0210, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel, September 2002.

[12] Henrik Nilsson. Functional automatic differentiation with dirac impulses. Accepted for publication at ICFP 2003, Uppsala, Sweden, 2003.

[13] 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.

[14] Ross Paterson. A new notation for arrows. In *Proceedings of the 2001 ACM SIGPLAN International Conference on Functional Programming*, pages 229–240, Firenze, Italy, September 2001.

[15] George Russell. Email message, subject: Fruit & co, February 2003. Message posted on the Haskell GUI mailing list, available at `http://www.haskell.org/-pipermail/gui/2003-February/000140.html`

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

[17] S. Smetsers, E. Barendsen, M. v. Eekelen, and R. Plasmeijer. Guaranteeing safe destructive updates through a type system with uniqueness information for graphs. *Lecture Notes in Computer Science*, 776, 1994.

[18] 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.