Yampa

Richard J. Lupton

October 29, 2015

Chapter 1

What is Yampa?

Yampa is a Haskell library for functional reactive programming. Functional reactive programming is a high-level declarative style of programming for systems which must respond to continuous streams of input, which are often time dependant, without undue delay. Example of such systems include video games, robots, animations and simulations. Haskell with its lazy evaluation, and separation of pure and impure code doesn't make it obvious how to work in these kinds of domains, without adopting a procedural style, so Yampa was developed to provide a more idiomatic approach.

1.1 How does Yampa operate?

Functional Reactive Programming is about processing signals. At its core, Yampa takes a varying input signal (in applications this might be, for example, the temperature from a temperature sensor), and processes it in some way, providing a corresponding varying output signal on the other side (continuing our imaginary example, this might include information on whether a heater should be on or not). The upper part of Figure **??** illustrates this idea.

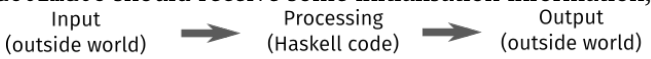
So Yampa is concerned with building objects which can take a continuously varying input and provide a corresponding continuously varying output. If we refer to values which continuously vary as *signals*, then Yampa is a library concerned with building and using *signal functions*.

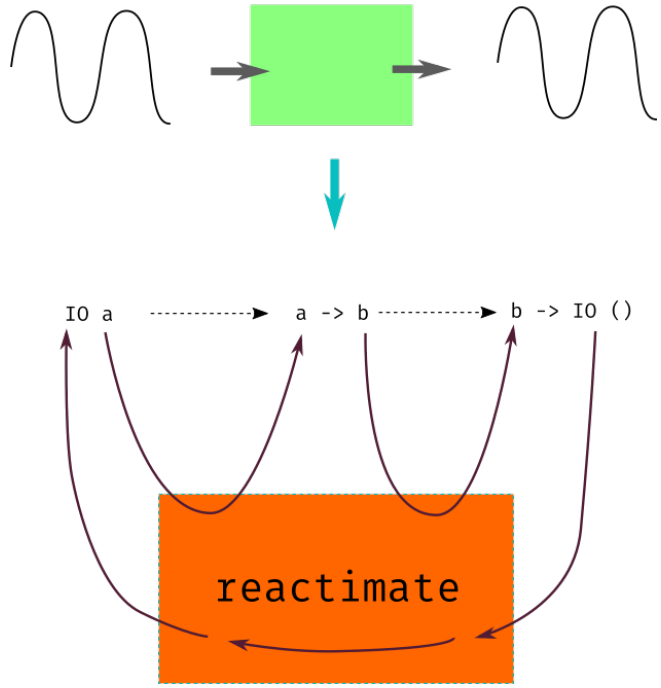
The bulk of Yampa is concerned with building signal functions. However, it is useful to see how a signal function is actually used in real world Haskell code to process signals. The tool that integrates signal functions into normal Haskell code is called `reactimate`. `reactimate` is a sample-process-output loop. The bottom part of Figure **??** illustrates an idealised loop. `reactimate` has three important inputs.

First `reactimate` has an input of type `IO a`, whose role is to take a sample of a signal which takes values of type `a`. It might get the position of a mouse cursor, determine whether a key on the keyboard has been pressed, or get the temperature from a temperature sensor. Secondly, `reactimate` needs a signal function, which is described using Yampa code. After collecting its input sample, of type `a`, `reactimate` processes it with the signal function and obtains an output sample, of type `b`, say. `reactimate` then has an output action, which knows what to do with this value of type `b` in the outside world. For example, our output sample might be a set of coordinates to position an image on the screen, and our output function will take these coordinates and render the image there. The output `IO` action comes back with a value of type `Bool`, which encodes whether reactimate should continue looping or stop. So the input to `reactimate` which describes how our signal should be output to the

word is of type `b -> IO Bool`.

Clearly, `reactimate` should be capable of input and output, and so it should be a function in the `IO` monad, and indeed, `reactimate` returns a value of type `IO ()`. In addition, `reactimate` should receive some initialisation information, again of type `IO a`.





So `reactimate` continuously samples the input, processes it with a signal function, and performs some output, until our output function says to stop. The above captures the essence of what reactimate does, but in reality the type of reactimate is a little more opaque:

```
reactimate :: IO a
    -> (Bool -> IO (DTime, Maybe a))
    -> (Bool -> b -> IO Bool)
    -> SF a b
    -> IO ()
```

The first `IO a` here is the initialisation input, and the value of type `a` is the value of the first sample. The second input, of type `Bool -> IO (DTime, Maybe a)` is our input function. In the definition of `reactimate` the input value of `Bool` is unused, so really one just needs to specify a value of type `IO (DTime, Maybe a)`. In fact to make this simpler, lets define

```
sInput :: IO (DTime, Maybe a) -> Bool -> IO (DTime, Maybe a)
sInput inp _ = inp
```

to convert a value of type `IO (DTime, Maybe a)` in a value of type `Bool -> IO (DTime, Maybe a)`. Now lets examine the values wrapped in the `IO` type. First `DTime`: `reactimate` doesn't have a built in time tracking system, so on each sample of the input signal, one is required to input the elapsed time since the last sample was taken. `DTime` is a synonym for `Double` used to represent this. Later, we define our own reactimate which should work on any POSIX system, and hides this time tracking. Presumably it is kept visible for systems with less uniform or custom time tracking requirements. The second value here, of type

`Maybe a`, is the input sample, wrapped in `Maybe` since there may be no input signal, or our sampling may fail. If a value of `Nothing` is fed in, then the value from the previous sample is used.

The third input to `reactimate` specifies how to deal with output. Again, the first `Bool` in `(Bool -> b -> IO Bool)` is unused, so lets define

```
sOutput :: (b -> IO Bool) -> Bool -> b -> IO Bool
sOutput out _ = out
```

to wrap a value of type `b -> IO Bool` in the type required for `reactimate`. The value of type `b -> IO Bool` works as described above, where a value of `True` from the output function indicates to `reactimate` that it should stop. The fourth value of type `SF a b` is the signal function, processing a signal taking values of type `a` into a signal taking values of type `b`. Yampa is concerned with building these signal functions. This will be the subject of (most of) the remainder of this guide.

Using our simplified input and output wrappers, we define a simplified version of `reactimate`:

```
sReactimate :: IO a -> IO (DTime, Maybe a) -> (b -> IO Bool) -> SF a b -> IO
    ()
sReactimate init inp out sigFun = reactimate init (sInput inp) (sOutput out)
    sigFun
```

which looks a little clearer, and more like what we discussed above. In fact, we will define a function `yampaMain`

```
yampaMain :: IO a -> IO (Maybe a) -> (b -> IO Bool) -> SF a b -> IO ()
```

which also deals with the timing in POSIX compatible systems. We write this in a module `YampaUtils.hs` which we will use in the rest of this guide. Listing 1.1 gives the complete contents of the module. Note `yampaMain` is essentially the same as `sReactimate` but wrapped in a time tracking system.

```
module YampaUtils
( yampaMain, sReactimate ) where

import FRP.Yampa as Y
import Data.Time.Clock.POSIX
import Data.IORef

sReactimate :: IO a -> IO (DTime, Maybe a) -> (b -> IO Bool) -> SF a b -> IO
    ()
sReactimate init inp out sigFun = Y.reactimate init (sInput inp) (sOutput out)
     sigFun
    where sInput inp _ = inp
          sOutput out _ = out

yampaMain :: IO a -> IO (Maybe a) -> (b -> IO Bool) -> SF a b -> IO ()
yampaMain init input output sigFun = do
  t <- getPOSIXTime
  timeRef <- newIORef t
  let timeWrapInput ins = do
        inV <- ins
        t' <- getPOSIXTime
```

```
        t <- readIORef timeRef
        let dt = realToFrac (t' - t)
        writeIORef timeRef t'
        return (dt, inV)
  sReactimate init (timeWrapInput input) output sigFun
```

Listing 1.1: YampaUtils.hs

1.2 The structure of a Yampa program

One can think of Yampa programs as programs of the form illustrated in Listing 1.2. To write a Yampa program, we need to specify an initialization, input and an output function, and also construct a signal function to do the required transformations. We then feed all this in to `yampaMain` which does the processing we require.

```
import FRP.Yampa as Y
import YampaUtils

init :: IO a
-- Do some initialisation

input :: IO (Maybe a)
-- Get some input

output :: b -> IO Bool
-- Do some output

sigFun :: SF a b
-- Do some signal transformations

main :: IO ()
main = yampaMain init input output sigFun
```

Listing 1.2: General form of a Yampa program

Of course, real programs will take many different forms, but the idealised above form illustrates what we need to build in order to have an executable program.

1.3 Signals and signal functions

Yampa is a library for building and using signal functions. We have deliberately avoided making precise what is meant by a signal for two reasons

One can think of a signal taking values of type `a`, or more succintly a signal of type `a`, as a value of type `a` with a context. Indeed, one can really think of it as being a value of type `IO a` (see for instance `getPOSIXTime` and compare it with the output from the `time` signal function later). Like any other value of type `IO a`, extracting values at different times can yield different results.

Chapter 2

Hello, Yampa

Now we've pinned down the mechanics of how we will incorporate our signal functions into real working code, we can write our first example. Save a copy of `YampaUtils.hs` in your working directory, for your use in these examples. Our first example, Listing 2.1, is the Hello, World of Yampa.

```
-- Example 1
-- Hello world, with an intermediate (2 second) pause

import FRP.Yampa as Y
import YampaUtils

initialise :: IO ()
initialise = putStrLn "Hello..."

output :: Bool -> IO Bool
output b = if b then putStrLn "...Yampa!" >> return True else return False

waitTwo :: SF () Bool
waitTwo = time >>> arr (>=2)

main :: IO ()
main = yampaMain initialise (return Nothing) output waitTwo
```

Listing 2.1: HelloYampa.hs

Our discussion in the introduction about `reactimate`, and our tidied version `yampaMain` , should give you a rough idea of what is going on here. `yampaMain` first calls `initialise`, which prints some text to the screen, but provides no input (or, more precisely, provides as input `()`). The input function passed to `yampaMain` is just `return Nothing`, which again provides no input to the system.

`waitTwo` is a signal function, and is the new code in this listing. For the moment, lets just observe that it converts a signal of values of type `()` into a signal of values of type `Bool`, which is what its type signature `SF () Bool` is meant to indicate (" a signal function from signals of type `()` to signals of type `Bool`").

The output function, `output`, takes this boolean value, and if its false, just instructs `yampaMain` to continue by passing out a `False` value in the `IO` monad. However, if the

boolean value is true, the output function first prints a message to the screen, and then returns an IO value of True, which instructs yampaMain to stop executing, and the program finishes.

Precisely when the value True emerges from yampaMain is determined in the signal function waitTwo. So lets take a look at this object.

```
waitTwo :: SF () Bool
waitTwo = time >>> arr (>=2)
```

The specification for waitTwo is that it should take the input signal (which here is constantly taking value ()), and if waitTwo has been transforming input values for less than 2 seconds, it should transform the input into a value False, and otherwise, it should transform the input signal into a value True. This boils down to saying, yampaMain should receive a value True to give to output, precisely when yampaMain has been running for 2 seconds. Compiling and running this code seems to confirm this.

So how has waitTwo been defined? It consists of two parts combined together with the operator >>>. First lets take a look at time.

```
time :: SF a Time
```

time takes a signal of any value, and replaces it with a value of type Time, a synonym for Double. The value is the time the particular instantiation of time has been in use, starting from 0. So in our case, it takes the value (), and replaces it with the time that yampaMain has been using waitTwo, which is the length of time yampaMain has been running.

Secondly, we have arr (>=2). Now inspecting the type of this in ghci reveals arr is a function which makes use of the Arrow type class. SF is an instance of this type class, and specialising its type signature to SF, we have:

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

arr is a way of lifting pure functions into SF, analogous to how return lifts values in a monad, and pure lifts values for applicative functors. More precisely, if f :: a -> b, then arr f takes a signal of values of type a, and transforms it into a signal of values of type b, by applying f to each value of type a. In some sense, it is the continuous signal analogue of fmap for the functor typeclass.

In our example, we have the signal function arr (>=2), which by specialising to types SF again has type:

```
arr (>=2) :: (Num a, Ord a) => SF a Bool
```

arr (>=2) takes a signal with numerical and orderable values, and applies (>=2) to these values. So our transformed signal will output True precisely when the input signal takes on values greater than or equal to 2.

So we have two signal functions, with an operator which combines them (a *combinator*). Lets inspect the type of >>>, specialised to SF again.

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

The type signature (and the fact that the more general version is for objects from the typeclass Category), is a giveaway for what (>>>) does. (>>>) is the SF analogue of composition. Given a way of transforming signals of value type a into signals of value type b, and a way of transforming signals of value type b into signals of value type c, by performing each transformation in order, I can transform signals of value type a into signals of value type c. This is the function of the (>>>) combinator. More precisely, f >>> g first transforms with f, and then with g (recall f . g first applies g *then* applies f).

So now we can analyse what waitTwo actually does. waitTwo takes an input signal of values (), and replaces these values with the time which yampaMain has been running. It then takes this time, and if it is less than 2 seconds, replaces this time value with a False.

Otherwise, it replaces this time value with `True`. In total, `waitTwo` transforms `()` into `False` if `yampaMain` has been running less than 2 seconds, and `True` once its been running at least two seconds.

Observation 1 *Observe that* `arr (f.g) = arr g >>> arr f`. *Observe that* `time` *is not the* `arr` *of a pure function.*

2.1 Building signal functions

Our first example is intentionally not the most intricate or complicated of examples. We build a very simple signal function, with the intention of showing how a Yampa program fits together.

The bulk of the Yampa library is concerned with building signal functions. Rather than building signal functions explicitly, Yampa supplies a basic set of signal functions (we saw for instance, `time`, above), and a set of *combinators* for combining various signal functions (for instance, signal function composition >>>).

2.2 Some basic signal functions and combinators

Lets begin with some of the basic signal functions which Yampa provides. As one might expect from a library based around transformations, one has analogues for the identity map, and constant maps.

```
identity :: SF a a
constant :: b -> SF a b
```

Of course, `identity` preserves the value of a signal, and `constant v` takes a signal, and replaces its value by the value `v`.

Exercise 2 *Write definitions for* `identity` *and* `constant` *in terms of* `arr`.

In our first example, we saw the signal function `time`. `time` takes a signals value, and replaces it with the *local* time, that is the time a signal function has been processing a signal. For the moment, this would appear to be the same time as `yampaMain` has been running, and with what has been covered so far, it is, but later we will see that signal functions can be switched in and out in response to events, at which point the time counter starts over from zero. For the sake of completeness (recall `Time` is a synonym for Double):

```
time :: SF a Time
```

An interesting built in function is `integral`. This performs numerical integratation of a signal over time. Of course this is useful for describing dynamical systems!

```
integral :: SF Double Double
```

`integral` actually has a more general type than the above, but this suffices for now. Yampa provides an similar signal function for derivatives also.

Observation 3 `time = constant 1.0 >>> integral`.

We also have, as we have seen already, a way of lifting pure functions `a -> b` to values of type `SF a b`:

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

`arr f` will take a signal of type `a`, and process its value with `f`, to create a signal of type `b`.

Now that we have a basic collection of arrows, we woud like some ways to combine them, to form more interesting arrows. The type constructor `SF` is an instance of the `Arrow` typeclass, a generalisation of the `Monad` type class, so we have all the combinators from `Arrow` available for use. We have already seen `arr` above, which is part of the `Arrow` interface, and earlier we saw `(>>>)`.

`(>>>)` is the composition operator for signal functions. `(f >>> g)` applies `f` to a signal, then `g` (note the order is reverse of `.`). Similarly, `(<<<)` is the composition operator the other way around, so `(f <<< g)` performs `g` then `f`.

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

Sometimes we want to process a signal in two different ways. The combinator `(&&&)` is designed to allow us to do exactly that.

```
(&&&) :: SF a b -> SF a c -> SF a (b,c)
```

Here `(f &&& g)` is intended to mean take the value of a signal, and apply `f` to it, placing the output in the first component of a tuple. Also apply `g` to the value of the signal, and place the output in the second component of the tuple.

Since we can produce a signal which takes a tuple for values, (which we can think of as being two signals captured in one), we have combinators for processing them.

First we have `(***)`, which allows us to take two signal functions and make each act on a particular element of a tuple.

```
(***) :: SF a1 b1 -> SF a2 b2 -> SF (a1, a2) (b1, b2)
```

`(f *** g)` acts on a signal which takes a tuple for values, by applying `f` to the first component, and `g` to the second component.

Sometimes, we only want to transform one of the elements of a signal of tuple type. While something of the form `(f *** identity)` would do, Yampa defines such combinators for us:

```
first :: SF a b -> SF (a, c) (b, c)
second :: SF b c -> SF (a, b) (a, c)
```

`first f` transforms only the first component of a signal of tuple values with `f`, and `second g` transforms only the second component of a signal with tuple values with `g`.

Exercise 4 *Write `second` in terms of `first` and `arr`.*

Observation 5 *`(f *** g) = (first f) >>> (second g)` .*

Our last combinator is `loop`. `loop` allows us to build recursive signal functions. It has type

```
loop :: SF (a, c) (b, c) -> SF a b
```

`(loop f)` is obtained from `f` by looping its values' second component back in to `f` for the next sample.

2.3 Arrow notation

The above primitives and combinators allow us to form more complex signal functions. In fact, any combination of basic signal functions can be formed. However, using the combinators directly can often lead to code which is difficult to read. For this reason, the arrow syntax was developed. This is the arrow analogue of monadic do notation. It allows us to express more clearly how we wish to process the values which signals take. A preprocessor transforms this notation into a description using the combinators we described above. When using arrow notation in ghci, load ghci with the -XArrows option, e.g. `ghci -XArrows`. Similarly, when compiling with ghc, use the -XArrows option, e.g. `ghc -XArrows main.hs`.

Let us first inspect the general form of a block in arrow syntax. Also, take a look at the example that follows the discussion, since this will make it clearer. The basic form of arrow syntax is given in the following.

```
sigFun :: SF a b
sigFun = proc input -> do
  processedSample1 <- sigFun1 -< sample1
  processedSample2 <- sigFun2 -< sample2
  ...
  returnA -< finalSample
```

`proc` is a keyword to indicate that the following is a block written in arrow syntax. `input` represents a sample of type `a`, and can be pattern matched against. A line of the form

```
processedSample <- sigFun -< sample
```

takes the sample (from a signal) named `sample`, processes it with `sigFun`, and binds the resulting value to `processedSample`. We can pattern match in the position of `processedSample`. The line

```
returnA -< finalSample
```

yields `finalSample` as the final transformed sample. Note there is no binding, just like (and for the same reason as) a monadic `do` block. As with `do` blocks, `let` bindings are also allowed in arrow syntax. As a word of warning, recursive definitions in arrow syntax must be preceeded by the keyword `rec`. We will investigate this fully later.

By way of example, lets describe a signal function which receives a signal of type `Double`, which is intended to represent the acceleration of an object, and is intended to output a signal describing its position (assuming it starts at 0, lets say). Using the normal combinators we could write the following:

```
accToPos :: SF Double Double
accToPos = integral >>> integral
```

which to be fair, is clear enough. In arrow syntax however, we can write this as

```
accToPos :: SF Double Double
accToPos = proc acc -> do
  vel <- integral -< acc
  pos <- integral -< vel
  returnA -< pos
```

So we obtain `vel` (velocity) by integrating the input `acc` (acceleration), and we obtain `pos` (position) by integrating the velocity `vel`. `pos` is the value we want, so we return that. It is not too hard to imagine signal functions where using the arrow syntax can greatly aid clarity. Observe that the wiring of samples through signal functions is represented in an intuitive manner.

The above example would potentially be used in an animation program, or a simulation of some kind. We will build these kinds of programs later, but for the moment it would be nice to have some working code. We will develop our "Hello, Yampa!" program a little further. Here, we start by asking the user to specify a time to delay finishing the greeting, and feed this value in to `yampaMain` via the initialisation value.

```
import FRP.Yampa as Y
import YampaUtils

-- Compile with ghc -XArrows askAndPause.hs

-- Our signal function will be of type SF Double Bool
sigFun :: SF Double Bool
sigFun = proc inp -> do
  t <- time -< inp
  b <- arr (uncurry (>)) -< (t, inp)
  returnA -< b

output :: Bool -> IO Bool
output False = return False -- Continue the reactimate
output True = putStrLn "Done" >> return True

main :: IO ()
main = do
  putStrLn "Enter␣a␣time␣to␣wait:"
  ins <- getLine
  yampaMain (return $ read ins) (return Nothing) output sigFun
```

Listing 2.2: askAndPause.hs

Chapter 3

Laboratory

As exciting as variations of "Hello, World" are, it would be nice to be able to create some more sophisticated examples which illustrate some of the power and expression of Yampa.

To this end, we will develop a small system which we will develop in a variety of ways. The system will make use of Haskell's high level SDL bindings, so we can retrieve live input from the user, and also render some representation of the output signal in real-time to the screen. Make sure Haskell's high-level SDL bindings are installed on your system[1].

```
import Graphics.UI.SDL as SDL
import FRP.Yampa as Y
import YampaUtils

import SDLTools

type Position = (Double, Double)
data Object = Object Position

data ProcessedSample = ProcessedSample [Object] Bool

-- input :: IO (Maybe ..)
-- input = ..

output :: Surface -> [Surface] -> ProcessedSample -> IO Bool
output surface sfs pss = do
    wipe surface
    let (ProcessedSample obs exit) = pss
    sequence $ zipWith (renderTo surface) obs sfs
    SDL.flip surface
    return exit

-- sigFun :: SF ...
```

[1]The package you require is `SDL`, and *not* `sdl2`. `SDL` is the package containing the high-level bindings for SDL, while `sdl2` only contains the basic wrappers for the C library. Make sure the SDL C library is installed on your operating system first (using your package manager), then for the Haskell bindings simply `cabal install SDL`.

```
-- sigFun =

main = do
    SDL.init [InitEverything]
    screen <- setVideoMode 640 480 32 [SWSurface]
    person  <- loadBMP "manSprite.bmp"

    yampaMain (return False) input (output screen [person]) sigFun

    SDL.quit

wasQuitEvent :: IO Bool
wasQuitEvent = do
  events <- pollEvents
  let quitAsked = or $ map (==SDL.Quit) events
  return quitAsked

renderTo :: Surface ->  Object -> Surface -> IO Bool
renderTo surface (Object (x,y)) image =
  blitSurface image Nothing surface (Just $ Rect (round x) (round y) 0 0)
```

Listing 3.1: template.hs

SDLTools.hs is a small module which wraps some common SDL tasks we will use in our programs. The code is not taxing, but its also not necessary to fully understand how each function works. The tasks they perform are largerly intuitive, and will be explained when they are used. It is listed here for completeness.

```
module SDLTools
( wipe
, pollEvents ) where

import Graphics.UI.SDL as SDL

-- wipe clears a surface (paints it black)
wipe :: Surface ->  IO Bool
wipe surface = fillRect surface Nothing (Pixel $ 0)

-- Poll Events gets all the pending events in one go for processing
pollEvents :: IO [SDL.Event]
pollEvents = do
  event <- pollEvent
  if event == SDL.NoEvent then return [] else (fmap (event:)  pollEvents)
```

Listing 3.2: SDLTools.hs

Chapter 4

Transforming discrete signals: Events and switching combinators

One of the nice properties of Yampa is that it can handle continuous and discrete signals (which are realised as a type of continuous signal) in one framework. A discrete signal is thought of as a signal whose value is either "Nothing happened", or "Something happened, and here is some information". More prescisely, a discrete signal is a signal with values of type `Event a` where

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

Examples of discrete events might be mouseclicks or key presses, or when the ambient temperature exceeds a certain level.

We refer to signals of `Event` type as *event streams*. A signal function of type `SF a (Event b)` is called an *event source*.

A sample of an event stream will take two (types of) value: either `NoEvent` or `Event a` for some value of type `a`. However, events should not occur with infinite density on a signal, and indeed, the sampling rate provides an upper bound for how often events can occur on a signal. If this is a problem, one should implement some buffering for events.

4.1 Switching and events

Events are used to initiate changes in signal functions. Yampa provides combinators which allow us to describe switching in one signal function for another when an event occurs. The most basic of these is `switch`:

```
switch :: SF a (b, Event c) -> (c -> SF a b) -> SF a b
```

The first argument of switch is best thought of as taking the form `(sigFun &&& eventSource)`, where `sigFun :: SF a b` and `eventSource :: SF a (Event c)`. The idea is that `switch` says to use the signal function `sigFun`, until `eventSource` supplies an event. The event will be a sample of type `Event c`, and this wrapped value of type `c` is used to determine what the replacement signal function should be — this is what the second input of type `c -> SF a b` does. The result is a signal function of type `SF a b`.

In a more intuitive language, `sigFun` is switched for a different signal function when an event occurs. Let's provide an example. We will rewrite our first hello program to use a switch to output a `True` signal when a time of two seconds elapses.

```
-- Example 3
-- Hello event, with an intermediate (2 second) pause

import FRP.Yampa as Y
import YampaUtils

initialise :: IO ()
initialise = putStrLn "Hello..."

output :: Bool -> IO Bool
output b = if b then putStrLn "...Event!" >> return True else return False

-- twoElapsed generates an event after two seconds
twoElapsed :: SF a (Event ())
twoElapsed = time >>> arr (>=2) >>> edge

-- waitTwo outputs false, but upon receiving an event from
-- twoElapsed, switches to output True
waitTwo :: SF () Bool
waitTwo = ((constant False) &&& twoElapsed) `switch` (\_ -> constant True)

main :: IO ()
main = yampaMain initialise (return Nothing) output waitTwo
```

Listing 4.1: HelloEvent.hs

Our new version is certainly no simpler than the original, but it demonstrates basic use of events. `waitTwo` is still our signal function, but now it makes use of a `switch` to change its output. First `waitTwo` outputs a constant signal false. When an event occurs from `twoElapsed` however, it takes the value from this event (in this case `()`) and uses the passed function to decide what signal function to switch to. In our case, the supplied function has a constant value `constant True`, so upon an event from `twoElapsed`, our signal function changes to take the value `True` constantly.

So now the timing work must be done in `twoElapsed`. This looks much the same as before, except now `time >>> arr (>=2)` is fed into a new signal function `edge`. `edge` has type `SF Bool (Event ())`. `edge` takes an input signal of value type `Bool`, and when the signal value changes (from `True` to `False`, or `False` to `True`), it ouputs an `Event ()` on its output; it otherwise outputs `NoEvent`. In sum then, this means an event is generated when (at least) two seconds have elapsed.

The sum effect is that our program prints its conclusion after waiting for two seconds. Notice that `switch` is written in infix notation. One of the benefits of this is that it the initial signal function occurs on the left of `switch` and the replacement upon an event on the right.

Exercise 6 *When a signal function `sf` is switched for a signal function `sf'`, the time for `sf'` starts again from zero. In other words signal functions have their own local time. Write a small program to demonstrate this.*

4.1.1 Other event generating functions

We saw above how to use `edge` to generate an event when a signal changes from `True` to `False` or `False` to `True`, but Yampa provides a few other functions to generate events also.

```
never :: SF a (Event b)
```

simply never generates an event. More useful perhaps is

```
now :: b -> SF a (Event b)
```

which generates an event (with value) of type `b` immediately. In other words `now v` is a signal function which has value `Event v` immediately, and `NoEvent` afterwards.

```
after :: Time -> b -> SF a (Event b)
```

generates an event after some specified period time, with a value of type `b`. `after t v` will generate a signal of value `Event v` in `t` seconds, with a value of `NoEvent` at all other times.

Exercise 7 *Rewrite listing 4.1 to use `after`.*

For those situations where having an event reoccur at regular intervals, Yampa provides

```
repeatedly :: Time -> b -> SF a (Event b)
```

`repeatedly t v` generates a signal which has value `NoEvent` at all times except whole number multiplicities of `t` seconds, at which point the signal has value `Event v`. Zero time is when `repeatedly` starts running.

Note that `edge` produces an `Event ()` signal. Sometimes we might want to attach or force an event to take on certain value. This is what the function `tag` is for.

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

In fact since `Event` is a functor (it is isomorphic to `Maybe`), `tag` can be defined by

```
tag e v = fmap (const v) e.
```

.

4.1.2 More switches

4.1.3 Other ways of interacting with events

Switches are the main way in which event signals interact with other signals (or discrete signals interact with continuous signals if you prefer), but there are other ways we can interact with events.

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

`hold v` initially takes on the value `v`, but when an event occurs on the input signal, it switches to the value wrapped in the event. In other words `hold v` outputs the value wrapped by the previous event, and if no event has previously occured, it outputs value `v`.

Instead of changing a value, we also have the possibility of transformation.

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

`accum v` receives a stream of events of type `a->a`, and upon receiving an event, takes the wrapped function and applies it to the value of the previous event which was dispatched, and `v` if none exists. Since it is natural to want to generate a signal consisting of just the value and not an event, Yampa also provides `accumHold`:

```
accumHold :: a -> SF (Event (a->a)) a
accumHold v = accum v >>> hold v
```

This is useful for creating counters and other similar devices.