Yampa Book

Gerold Meisinger

Mar 25, 2023

CONTENTS:

Learn yourself a Yampa for great good!

CHAPTER

ONE

INTRODUCTION

Project status: As of 2023-03 there are four chapters with examples but lots of todos and missing prose.

Tip: If you don't care about the motivation, background, history of functional reactive programming and know Haskell already, just skip to *Embedding*.

Functional Reactive Programming (FRP) is an elegant approach for developing interactive programs, such as games, which inherently consist of dynamic structures and operate within a *temporal* context. Rooted in a solid mathematical foundation and using pure functional programming, FRP yields highly reusable and composable components which lead to the creation of dynamic *signal networks*, as illustrated in the following diagram:

In contrast to the *imperative programming* paradigm, where the process resembles a step-by-step execution of actions (e.g. if there is input, move the player, handle collisions, draw the character), the *functional reactive programming* paradim takes a different perspective. It uses a declarative approach, where a picture is displayed at the player's position, which again is derived from input and collision events over time.

This book focuses on the Yampa libary and aims to provide a gentle introduction to game programmers transitioning from an imperative background (e.g. Unity), who posses some knowledge of Haskell but may not be fully versed in Monads and more complex type-classes. Based on the [FrpRefac16] paper, this tutorial breaks down the concepts into manageable steps and how to apply them to real-world game programming scenarios.

1.1 Motivation

1.1.1 Why functional

A lot has been said about "why functional programming matters" [WhyFP90] already. There is no point on iterating about it here much more.

1.1.2 Why reactive

Let's start with a simple uni-directional user interface example of a counter. Let there be three buttons: increase, decrease and reset which change a label text.

```
let count = 0

const buttonIncrease = document.createElement('button')
const buttonDecrease = document.createElement('button')
```

(continues on next page)

(continued from previous page)

```
    const buttonReset    = document.createElement('button')
    const labelCounter   = document.createElement('label')
    const components = [buttonIncrease, buttonDecrease, buttonReset, labelCounter]

    buttonIncrease.innerHTML = "increase"
    buttonDecrease.innerHTML = "decrease"
    buttonReset   .innerHTML = "reset"
    labelCounter  .innerHTML = count.toString()

    buttonIncrease.onclick = () => { count += 1; labelCounter.innerHTML = count.
↪toString() }
    buttonDecrease.onclick = () => { count -= 1; labelCounter.innerHTML = count.
↪toString() }
    buttonReset   .onclick = () => { count  = 0; labelCounter.innerHTML = count.
↪toString() }
```

counter0.html

There is a lot to criticise about this code already. Maybe we should introduce a class Counter which extends (or composes) a Label and make count private to hide it away from all other sites. While this may be better the basic problem remains. After all, there are already three if-conditions which change the counter variable. We just move the problem into Counter and encapsulate it a bit shielding it from outside manipulation. If you are not disciplined enough this might get out of hand again soon, the code might look the same, just within the Counter class.

There is a popular architectural pattern called Model-View-Controller (MVC for short). The model represents some intrinsic state. The view displays the states in multiple forms (text, charts). The controller manipulates the model state. And maybe we should add a general EventManager singleton and call EventManager.onEvent("inc") and let the corresponding components handled it themselves. While this removes the reference to labelCount it introduced another indirection in that we don't know were the event came from.

```
// model states
count = 0

// views representing the model states
labelCount = new Label("0")
labelCount.update() =>
  this.text = count.toString()

buttonInc = new Button("inc")
buttonDec = new Button("dec")
buttonNul = new Button("nul")

// controllers manipulating the model state
countHandler = (type) =>
  if (type == "inc") count += 1
  if (type == "dec") count -= 1
  if (type == "nul") count  = 0
EventManager.addHandler(countHandler)

buttonInc.onClick = () => EventManager.onEvent("inc")
buttonDec.onClick = () => EventManager.onEvent("dec")
buttonNul.onClick = () => EventManager.onEvent("nul")
```

Lets add another uni-directionl event source which acts on the counter. Let the counter be increased or decreased by

keyboard input.

```
InputManager.onKeyPressed = (key) =>
  if (key == "plus" ) EventManager.onEvent("inc")
  if (key == "minus") EventManager.onEvent("dec")
  if (key == "del"  ) EventManager.onEvent("nul")
```

Again, this looks very innocent but in reality we can never tell where an event was fired from, thus loosing all call stack information in debugging. At some cycle the onEvent listener may be called with "add" but we don't know who called it: buttonAdd? onKeyPressed? Did mysteryProcedure add some other event calling sites? We also note a lot of criss- and cross-referencing of variables and objects. labelCount references count. Does something else also reference count? What is mysteryProcedure doing to labelCount?

Let's make it even more weird and look at a bi-directional user interface example. Let there be a number field and a slider while one always shows the value of the other.

```
    const numberInput = document.createElement('input')
    const sliderInput = document.createElement('input')
    sliderInput.type = "range"
    const components = [numberInput, sliderInput]

    numberInput.value = 0
    sliderInput.value = 0

    numberInput.onkeyup     = () => sliderInput.value = numberInput.value
    sliderInput.onmousemove = () => numberInput.value = sliderInput.value
```

`bidirectional0.html`

Okay, but who is in charge of the model state "value" now?

```
    let value = "0"
    let changedBy = null

    const numberInput = document.createElement('input')
    const sliderInput = document.createElement('input')
    const components = [numberInput, sliderInput]
    sliderInput.type = "range"

    numberInput.update = () => {
      if (changedBy != null && changedBy !== "numberInput") {
        numberInput.value = value
        changedBy = null
      }
    }
    sliderInput.update = () => {
      if (changedBy != null && changedBy !== "sliderInput") {
        sliderInput.value = value
        changedBy = null
      }
    }

    numberInput.onkeyup     = () => { value = numberInput.value; changedBy = "numberInput
↪" }
    sliderInput.onmousemove = () => { value = sliderInput.value; changedBy = "sliderInput
```

(continues on next page)

(continued from previous page)

```
↪" }

    const update = () => {
      components.forEach(c => c.update())
      window.requestAnimationFrame(update)
    }

    update()
```

bidirectional1.html

There is a way to make all of this more structured called "immediate mode user interfaces" (IMGUI). We can imagine it like rendering and handling the UI components at every update cycle.

```
value = 0

update() =>
  enteredValue? = numberField(value)
  slidedValue?  = slider(value)

  if (enteredValue != null || slidedValue != null)
    value = merge(enteredValue, slidedValue)

main() =>
  while(true)
    update()
```

This also makes another property very clear: could it be possible that the number field and slider is in some way changed at the same time… like on a multi-touch device? And if so which component should win out? merge could for example bias towards the first parameter.

Now in functional reactive programming we would define value to BE the (merged) input of numberField and slider over time. Just like you would define the cell of a spreadsheet to BE the sum of multiple cells and whenever one of the referenced cells changes the sum changes automatically too. Except that spreadsheets usually don't have a way to handle bi-directional ("cyclic") data flow.

```
 | A | B | C
-+---+---+---
1|123|   |
-+---|---+---
2|234|   |
-+---|---+---
3|345|   |
-+---|---+---
4|=SUM(A1:A3)
-+---|---+---
5|=AVG(A1:A3)
```

Let's imagine a more complex real-time, interactive computer game which uses an user interface with hierarchical component structure, user inputs bubbling down and up the component tree, each component handling the input and/or prevent further bubbling, while the user interface overlays an interactive game scene, were objects can be selected or dragged, UI components can be selected or dragged, depending on some internal operation state. Scene objects overlay scene objects, UI component overlay UI components, UI components overlay scene objects and some of them have a bi-directional dependency on each other to form an internal model state (like a player position). Good luck with that!

User input events will pop up out of nowhere, going anywhere, changing global-ish states, states which may be changed at multiple sites, concerns scattered across multiple locations, impossible to track, debug, test and extend in the long run.

With (functional) reactive programming the data flow is clear.

1.1.3 Why temporal

1.2 History

Todo: Complete history of FRP based on papers

Todo: Timeline of FRP

- Fran allows to describe and compose animations over time (no support for *Events* and dynamic list of *Behaviours*)
- Push and pull discussion
- Optimizing CCAs
- Fruits criticism https://mail.haskell.org/pipermail/gui/2003-February/000140.html "Things like getting an alien spaceship to move slowly downwards, moving randomly to the left and right, and bouncing off the walls, turned out to be a major headache." => no changing behaviours.
- Yampa arcade in [YamCade03]
- Wormholes to route IO into the signal network
- [FrpRefac16] to provide Reader and Writer monads within in the signal network

Perez et al developed *Monadic Stream Functions* and showed that *Yampa* could be described as special case of Reader MSFs which provides time deltas. FRP apparently is the first concept to describe inherently stateful tasks which supposedly required imperative programming (things like simulations, GUIs). With [FrpRefac16] it appears we finally reached a point were most of the issues are out of the way:

> [FrpExt17] 3.3.2: Limitations of FRP and Arrowized FRP [regarding old Yampa]: Fixed Time Domain and Clock, I/O Bottleneck, Explicit Wiring, Referential Transparency Across Executions, Style and code modularity.

Attention: Differenciate between Yampa (old Yampa, Yampa 1), Dunai (MSFs) and BearRiver Yampa (new Yampa, Yampa 2) which is based on Dunai. The function signatures of BearRiver are a little different to the old Yampa (e.g. `embed` is now a Monad).

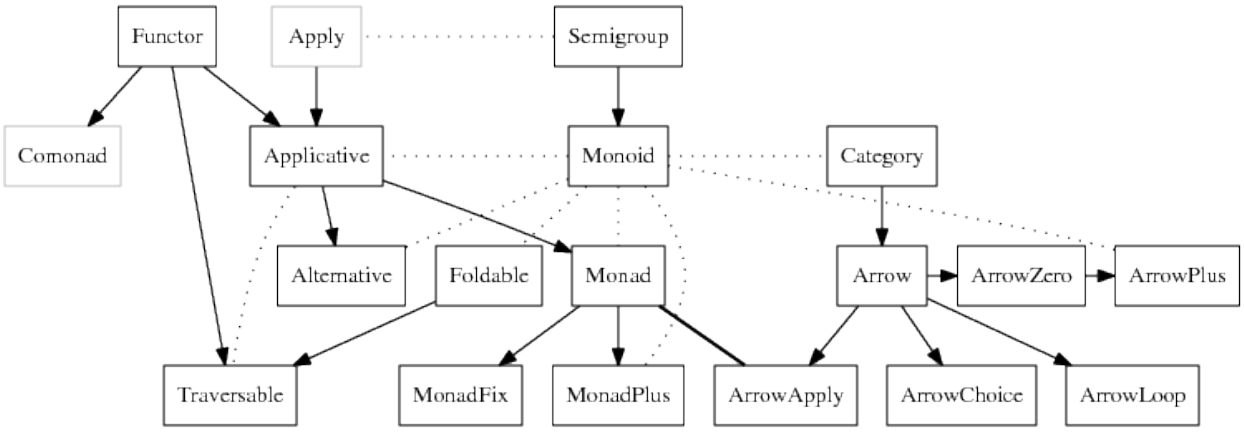
bearriver Hackage "Because dunai is particularly fast, especially with optimizations enabled, this implementation is faster than traditional Yampa for medium-sized and large applications."

Ivan Perez: "There's some fundamental differences, like the fact that, in principle, bearriver signals do not exist at time 0"

Reddit - Ivan Perez on What makes Dunai special?

1.3 Haskell

This books assumes the reader is familiar with progamming interactive applications, basic Haskell and Monads. It will not provide yet another tutorial on Haskell and Monads because there are enough out there already. Some important aspects will be reiterated if need be. If you must learn Haskell first I recommend to start with [LearnGood11]. Monads are on a different difficulty level. Always remember they are an abstract mathematical concept and a lot of smart computer scientists found them useful in many different situations and therefore it's worthwhile to learn them the hard way. Keep away from metaphors. I recommend to start with [MonadFP95] which is still relevant even after 25 years and clearly shows the motivation using simple examples for imperative programmers, how to use them and why. Then take a deep dive with [AllMonads03] and make sure you understand how the abstraction builds up from *Functor*, *Applicative* to *Monad* (*Monad Transformer*) up to *Arrow* according to the Typeclassopedia:



1.4 Arrows

Arrows are an essential building block of Yampa and together with the arrow notation provide a way to write clear and readable signal networks. Similar to Monads, Arrows are a general way to "combine things" hence a "combinator library". Different to Monads however, Arrows allow to specify exactly which input parameter of a tuple is used and how it connects to the output parameters. It is also important to understand how all of this is put together to form Yampa:

- The Arrow type class provides the general combinators. Type classes need concrete instances however.
- There are libraries out there (which one?) which provide general arrow combinations independent of the concrete class instance (i.e. independent of Yampa), only using the combinator functions (>>>, &&& etc.).
- Dunai's MSFs and Yampa's signal functions are an instance of the arrow type class which encapsulate and abstract away the concept of continuous time.
- Dunai provides a general way to implement causality ("step! step! step!").
- BearRiver Yampa is a specific implementation of causality in the concept of continuous time ("tiiick") using MSFs plus events.
- Yampa provides additional functions which are useful within the context of continuous time and events (`integral`, `accumHold` etc.).

See the Arrows homepage for additional information. Here are description excerpts from the arrow papers:

> [GenMonArr00] One of the distinguishing features of functional programming is the widespread use of combinators to construct programs. A combinator is a function which builds program fragments from

program fragments; in a sense the programmer using combinators constructs much of the desired program automatically, rather that writing every detail by hand.

[NewNotatArr01] The categorical notion of monad, used by Moggi to structure denotational descriptions, has proved to be a powerful tool for structuring combinator libraries. Moreover, the monadic programming style provides a convenient syntax for many kinds of computation, so that each library defines a new sublanguage.

[ArrComp03] Many programs and libraries involve components that are "function-like", in that they take inputs and produce outputs, but are not simple functions from inputs to outputs. This chapter explores the features of such "notions of computation", defining a common interface, called "arrows". This allows all these notions of computation to share infrastructure, such as libraries, proofs or language support. Arrows also provide a useful discipline for structuring many programs, and allow one to program at a greater level of generality.

[ProgArr05] We can think of arrows as computations, too. The Arrow class we have defined is clearly analogous to the usual Monad class - we have a way of creating a pure computation without effects (`arr`/`return`), and a way of sequencing computations (`(>>>)`/`(>>=)`). But whereas monadic computations are parameterised over the type of their output, but not their input, arrow computations are parameterised over both.

1.4.1 arr

Lift a function to an arrow.

1.4.2 returnA

The identity arrow, which plays the role of return in arrow notation.

1.4.3 first second

Pass-through component and leave it unchanged.

1.4.4 (>>>) (<<<)

Just feed the output of one arrow as input into the other.

1.4.5 (***)

1.4.6 (&&&)

Called fan-out or widening.

1.4.7 loop

1.4.8 (^>>) (>>^) (<<^) (^<<)

Convenience function if just want to compose with a pure function but don't want to write `arr` all the time.

Todo: add simple arrow combinator examples

1.5 Arrow notation

Introductions from the arrow notation papers:

> [NewNotatArr01] Recently, several workers have proposed a generalization of monads, called variously "arrows" or Freyd-categories. The extra generality promises to increase the power, expressiveness and efficiency of the embedded approach, but does not mesh as well with the native abstraction and application. Definitions are typically given in a point-free style, which is useful for proving general properties, but can be awkward for programming specific instances.
>
> [ArrComp03] With this machinery, we can give a common structure to programs based on different notions of computation. The generality of arrows tends to force one into a point-free style, which is useful for proving general properties. However it is not to everyone's taste, and can be awkward for programming specific instances. The solution is a point-wise notation for arrows, which is automatically translated to the functional language Haskell. Each notion of computation thus defines a special sublanguage of Haskell.

```
myArr static0 = proc (in0, in1, in2) -> do
  x <- anotherArrA -< in0
  let a = x + in0
  y <- anotherArrB -< (in1, in2)
  z <- anotherArrC -< in2
  rec
    r0 <- recursiveArrA -< r1
    r1 <- recursiveArrB -< r0
  returnA -< (x + y + z, 123, "abc")
```

1.6 Installation

If you want to follow along with the examples, which is highly recommended, you need to setup some things first:

```
>>> apt install build-essential curl libffi-dev libffi7 libgmp3-dev libgmp10 libncurses-
↪dev libncurses5 libtinfo5 # using libgmp3-dev instead of libgmp-dev and libffi7␣
↪instead of libffi6
```

- Get and install ghcup

```
>>> ghcup tui
```

Install GHC 9.0.2 (as of 2022-08-21 newer versions are incompatible with Haskell Language Server 1.7)

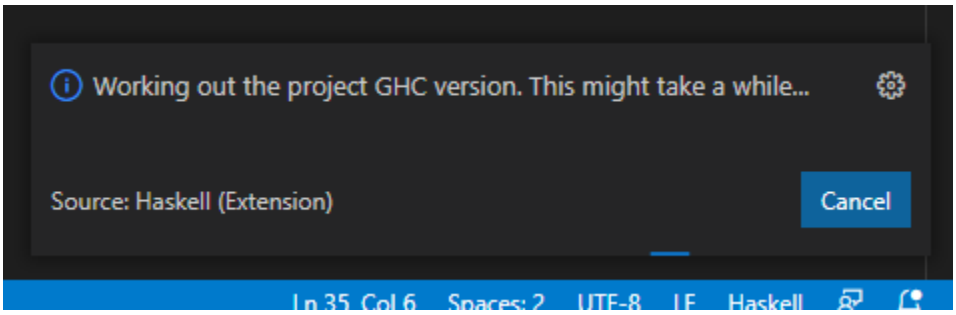
```
>>> cabal update
>>> cabal build
```

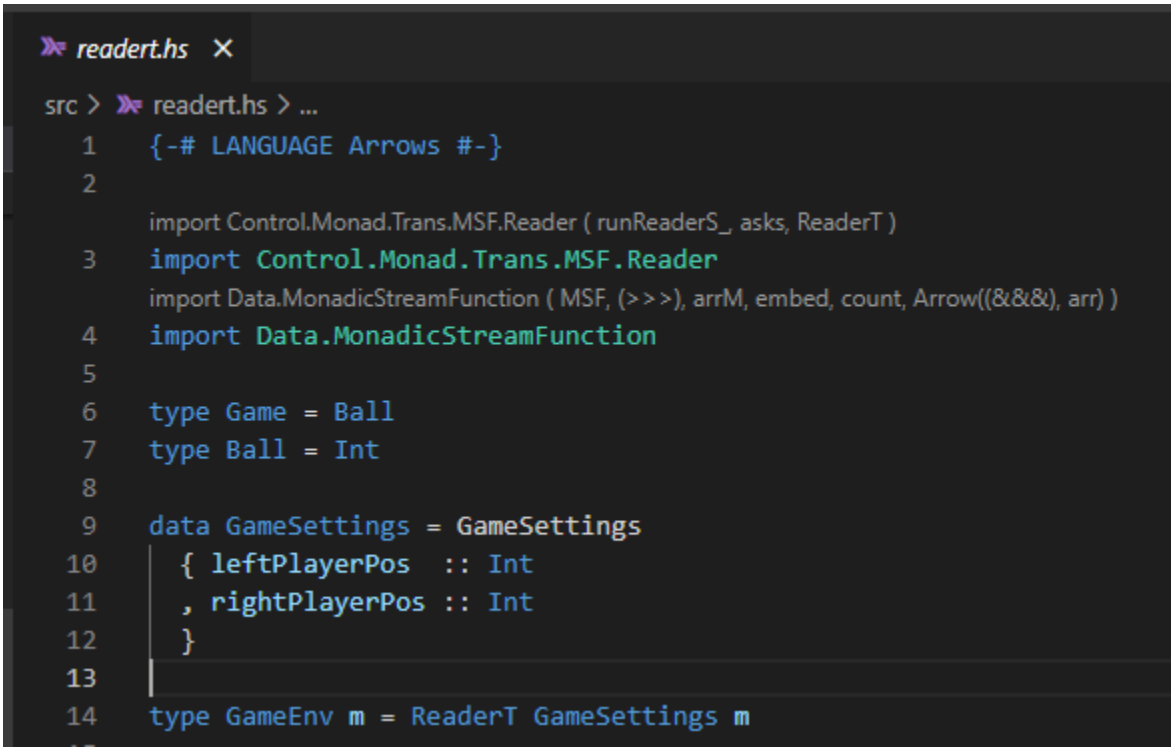
- Get the git repository with

```
>>> git clone https://gitlab.com/gerold.meisinger/yampa-book.git
```

- Optional: Install Visual Studio Code and the Haskell extension.

If everything works you should see a notification in Visual Studio saying:





Visual Studio Code may suggest other extension for Restructedtext, Python etc. Install them if you like.

This is a book, so you might as well just read it.

If you like to contribute to this docs and learn how to build the documentation see *Contributing*. You can add public annotations and highlights on the sidebar right.

1.7 Troubleshooting

For troubleshooting of contributing-related issues see contributing troubleshooting.

```
Visual Studio Code + Haskell Language Server: ghcide not found
```

Restart the `Haskell Language Server` with `Control-Shift-p`

```
error: parse error on input '->'

sf = proc input -> do
  ...
```

Add the following line on top of your file: `{-# LANGUAGE Arrows #-}`

```
<no location info>: error:
  Could not find module `Data.MonadicStreamFunction'
  It is not a module in the current program, or in any known package.
```

Start with `cabal repl` instead of just `ghci` otherwise the modules won't be loaded.

```
I'm getting strange garbled output in ghci on Windows
```

You interrupted the execution with `Control-C` and now need to restart your terminal!. This is a known issue which I still don't know how to resolve.

CHAPTER

TWO

EMBEDDING

2.1 Quickstart

```
:{
do
  let yourMSF = count
  ls <- embed yourMSF ["input0", "input1", "input2"]
  print ls
:}

:{
do
  let yourSF = integral
      dt = 1 / 60
  ls <- runReaderT (embed yourSF $ replicate 10 123.0) dt
  print ls
:}

:{
do
  let yourSF = integral
      dt = 1 / 60
  ls <- embed (runReaderS_ yourSF dt) $ replicate 10 123.0
  print ls
:}
```

2.2 Basics

Read the [YamCade03] and [FrpRefac16] papers and if you don't understand it lets do it step-by-step. If you need a more verbose version of these papers there is [FrpExt17].

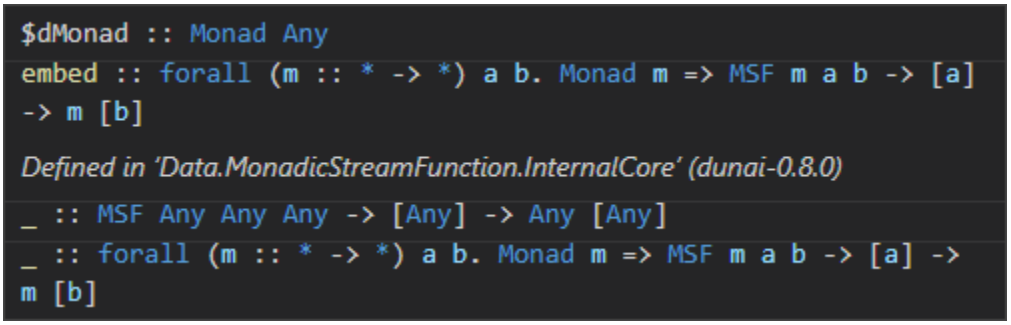
A `MSF` describes a general stepper function which moves a simulation forward in a *causal* manner. In it's most basic for it's just "step step step…", but we can apply Monads to describe more complex forms of what this step is (like time).

In a real program we are going to run MSFs in an endless loop called *reactimate*. With embed we can create deterministic simulations with predefined inputs and them once. This is great for learning and debugging and lets us increase the difficulty gradually.

Here is how you can find out how `embed` is defined:

1. Search with Hoogle.

2. Look up the definition on Hackage.
3. Hover over the function name in Visual Studio Code.



4. Start GHCi and ask for the type:

```
>>> cabal repl
>>> :t Data.MonadicStreamFunction.embed
-- embed :: Monad m => MSF m a b -> [a] -> m [b]
```

Let's use GHCi to start a simple `embed` example:

```
>>> cabal repl
>>> :m Data.MonadicStreamFunction
>>> embed (arr (+1)) [1, 2, 3]
-- [2,3,4]
```

It is important to note that GHCi always provides an `IO` Monad if necessary, which means the free type variable `m` is inferred as `IO`. I think it's a bit confusing however to see what's going on when, sometimes a Monad is provided by GHCi, and sometimes it's not required, but in the end we will write full programs which all run within a `main :: IO ()`. Thus I'm going to avoid the GHC interpreter for now and we load examples directly.

Open a text editor and write the following text into a file named `main.hs`:

```
{-# LANGUAGE Arrows #-}

import Data.MonadicStreamFunction

main :: IO ()
main = do
  putStrLn "Hello world"
```

To start the program:

```
>>> cabal repl helloworld
>>> main
-- Hello world
```

In `main` replace the code with:

```
main = do
  [1,2,3]
-- > ERROR! Couldn't match expected type: IO t0 with actual type: [a0]
```

This doesn't work because main needs to return a Monad. Thus we write:

```
main = do
  pure [1, 2, 3] -- or: return [1, 2, 3]
-- > [1,2,3]
```

Note: Recall that `pure = return` (somtimes called `unit`) but we are going to use `pure` all over this text to avoid confusion with keywords used in imperative languages. `return` in Haskell is an ordinary function and not any special language construct.

`embed` needs a `MSF` as parameter, we can always use an identity Arrow which only passes through values. This is not very exciting but helps us understand how everything is build up.

```
main = do
  embed (arr id) [1, 2, 3] -- or: embed returnA [1, 2, 3]
-- > [1,2,3]
```

With `arr` we can add pure functions into an Arrow network.

```
main = do
  embed (arr (\n -> 1 + n)) [1, 2, 3]
-- > [2,3,4]

-- embed (arr (1 + )) [1, 2, 3] -- suggested by Haskell Language Server
```

Note that `embed` returns a Monad similar to `pure`. We could also bind it to a variable and print it:

```
main = do
  ls <- embed (arr (1 +)) [1, 2, 3]
  print ls
-- > [2,3,4]

-- print =<< embed (arr (1 +)) [1, 2, 3]
```

So far you might be wondering: What's the point? If all we want is a converted list, why don't we just use `fmap`?

We are going to use the recursive arrow count now to build up a state.

```
count :: (Num n, Monad m) => MSF m a n
```

```
main = do
  embed count ["foo", "bar", "baz"]
-- > [1,2,3]
```

`count` doesn't care what it gets. The type variable `a` is free. The function only counts how often the simulation was called. So it's okay to just use unit types or any other type:

```
main = do
  embed count [(), (), ()]
-- > [1,2,3]
```

Internally `count` uses the function sumFrom, which again uses the fundamental function feedback.

```
feedback :: Monad m => c -> MSF m (a, c) (b, c) -> MSF m a b
```

Note: It's interesting to see how some of these functions are implemented in source.

```
-- | Sums the inputs, starting from an initial vector.
sumFrom :: (VectorSpace v s, Monad m) => v -> MSF m v v
sumFrom = accumulateWith (^+^)

-- | Applies a function to the input and an accumulator,
-- outputting the updated accumulator.
-- Equal to @\f s0 -> feedback s0 $ arr (uncurry f >>> dup)@.
accumulateWith :: Monad m => (a -> s -> s) -> s -> MSF m a s
accumulateWith f s0 = feedback s0 $ arr g
  where
    g (a, s) = let s' = f a s in (s', s')

-- | Well-formed looped connection of an output component as a future input.
feedback :: Monad m => c -> MSF m (a, c) (b, c) -> MSF m a b
feedback c sf = MSF $ \a -> do
  ((b', c'), sf') <- unMSF sf (a, c)
  return (b', feedback c' sf')
```

Todo: Add discussion for feedback function

2.3 Reader

Warning: There is an error in refactored 3.2 lifting. There is no `liftS :: (a -> m b) -> MSF m a b` function. Searching Hoogle for `(a -> m b) -> MSF m a b` we get `arrM`. `liftS` should be introduced as an deprecated alias for `arrM`.

```
{-# DEPRECATED liftS "Use arrM - the alias liftS was only used in the refactored
paper." #-} liftS = arrM
```

already mentioned here

Recall that the Reader monad is a readonly context which in our case could be used to inject config files. We are using a very simple config here which just says "how much should be added by each count". The `runXyz` functions of most Monads are used to peel of the monadic context and reveal the value within the Monad, which is usually used at the outermost calling site.

Todo: what's the point? why not just pass the config as a value?

```
runReader :: Reader r a -> r -> a
```

Note how `r` is gone in the final result.

```
countReader :: MSF (Reader Int) () Int
countReader = count >>> arrM (\x -> ask >>= (\e -> pure (x * e)))
```

(continues on next page)

(continued from previous page)

```
main = do
  runReader (embed countReader [(), (), ()]) 5
-- > ERROR: Couldn't match expected type: IO t0 with actual type: [Int]
```

Note that a `MSF` is also a Monad defined by the `m` type variable and thus `countReader` is a `Reader` Monad. If we use `runReader` here, we peel of the `Reader` Monad, we get what's inside, a plain value. Either we bind the variable (with >>=) or just assign and `print` it.

```
main = do
  let ls = runReader (embed countReader [(), (), ()]) 5
  print ls

--print $ runReader (embed countReader [(), (), ()]) 5
```

2.3.1 Refinement

Let's shorten the definition of `countReader` a bit. Recall that <$> is just an alias for `fmap` which is function application within a context (in this case, the `Reader` Monad).

```
 ($)  ::                    (a ->   b) ->   a ->   b -- application operator
(<$>) :: Functor     f =>   (a ->   b) -> f a -> f b -- applying a function within a␣
↪context
(<*>) :: Applicative m => m (a ->   b) -> m a -> m b
(=<<) :: Monad       m =>   (a -> m b) -> m a -> m b -- applying a function within a␣
↪context which produces a new context

(>>=) :: Monad       m =>   m a -> (a -> m b) -> m b
```

With this we can shorten `countReader` to

```
countReader = count >>> arrM (\x -> (\e -> pure (x * e)) =<< ask)
countReader = count >>> arrM (\x -> fmap (x * ) ask)
countReader = count >>> arrM (\x -> (x * ) <$> ask)
```

Instead of `(Reader Int)` we can use an arbitrarly complex data type for our config like.

```
data Config = Config
  { velocity    :: Int
  , soundVolume :: Int
  , isWindowed  :: Bool
  -- ...
  }
```

and then use `asks velocity` instead of `ask`.

2.4 ReaderT

> [FrpRefac16] 4. : Monads and monad transfomers have associated execution functions to run computations and extract results, consuming or trapping effects in less structured environments. For instance, in the monad stack `ReaderT e m` we can eliminate the layer `ReaderT e` with `runReaderT r :: e -> ReaderT e m a -> m a`, obtaining a value in the monad `m`.

If we use a `ReaderT` now, like it's used in the paper, you have to remember that it still a Monad and the `MSF` needs another Monad type.

```
countReaderT :: Monad m => MSF (ReaderT Int m) () Int -- what is m going to be?
countReaderT = count >>> arrM (\x -> (x * ) <$> ask)
```

This is important because if we run it in `main :: IO ()` it will infer the Monad type `IO` into the transformer.

Todo: ? why does that matter? m is a free variable, so we cannot use it for anything specific. we might as well set it to Identity? but I'm getting "Couldn't match expected type: IO t0 with actual type: Identity [Int]"

```
main :: IO ()
main = do
  ls <- runReaderT (embed countReaderT [(), (), ()]) 5
  print ls

-- countReaderT :: MSF (ReaderT Int IO) () Int
```

> [FrpRefac16] 4.1: This execution method, however, is outside the invocation of embed, so we cannot make the game settings vary during runtime. To keep the ReaderT layer local to an MSF, we define a temporal execution function analogous to runReaderT (implemented using an unwrapping mechanism presented in Section 5).
>
> WIWINWLH - Monad Transformers: It's useful to remember that transformers compose *outside-in* but are *unrolled inside out*.

```
import Control.Monad.Trans.MSF.Reader -- note the MSF here, it is not: import Control.
↪Monad.Trans.Reader

main = do
  embed (runReaderS_ countReaderT 3 &&& runReaderS_ countReaderT 5) [(), (), ()]
-- > [(3,5),(6,10),(9,15)]
```

(If you are confused of were the tuple comes from it's the &&& parallel arrow combinator.)

2.5 ReaderT in paper

> **Warning:** there is another error in the paper. we are actually using `runReaderS_` here

From paper:

```
runReaderS  :: Monad m => MSF (ReaderT r m) a b -> r -> MSF m a b
runReaderS_ :: Monad m => MSF (ReaderT r m) a b -> MSF m (r,a) b
runReaderS_ :: Monad m => MSF (ReaderT s m) (s, a) b -> MSF m a b
```

From source:

```
runReaderS  :: Monad m => MSF (ReaderT r m) a b -> MSF m (r, a) b
runReaderS_ :: Monad m => MSF (ReaderT s m) a b -> s -> MSF m a b
readerS     :: Monad m => MSF m (r, a) b -> MSF (ReaderT r m) a b
```

```
import Control.Monad.Trans.MSF.Reader
import Data.MonadicStreamFunction

type Game = Ball
type Ball = Int

data GameSettings = GameSettings
  { leftPlayerPos  :: Int
  , rightPlayerPos :: Int
  }

type GameEnv m = ReaderT GameSettings m

ballToRight :: Monad m => MSF (GameEnv m) () Ball
ballToRight = count >>> arrM (\n -> (n+) <$> asks leftPlayerPos)

hitRight :: Monad m => MSF (GameEnv m) Ball Bool
hitRight = arrM $ \i -> (i >=) <$> asks rightPlayerPos

testMSF = ballToRight >>> (arr id &&& hitRight)

main :: IO [((Ball, Bool), (Ball, Bool))]
main = do
  embed (runReaderS_ testMSF (GameSettings 0 3)) (replicate 5 ())
  -- > [(1,False),(2,False),(3,True),(4,True),(5,True)]
  embed (runReaderS_ testMSF (GameSettings 0 2)) (replicate 5 ())
  -- > [(1,False),(2,True),(3,True),(4,True),(5,True)]
  embed (runReaderS_ testMSF (GameSettings 0 3) &&& runReaderS_ testMSF (GameSettings 0␣
↪2)) (replicate 5 ())
  -- > [((1,False),(1,False)),((2,False),(2,True)),((3,True),(3,True)),((4,True),(4,
↪True)),((5,True),(5,True))]
```

readert.hs

```
>>> cabal repl readert
>>> main
```

2.6 WriterT

```
countReaderT :: Monad m => MSF (ReaderT Int (WriterT String m)) () Int
countReaderT = count >>>
  arrM (\x -> ask >>= (\e -> pure (x * e))) >>>
  arrM (\cv -> lift $ tell (show cv) >> pure cv)

main = do
  embed (runWriterS (runReaderS_ countReaderT 3)) [(), (), ()]
```

2.7 WriterT in paper

From paper:

```
runWriterS :: Monad m => MSF (WriterT r m) a b -> MSF m a (b, r)
```

From source:

```
runWriterS :: Monad m => MSF (WriterT r m) a b -> MSF m a (r, b)
```

```
type GameEnv m = WriterT [String] (ReaderT GameSettings m) -- the other way around?␣
↪otherwise we would have to lift all existing code

ballToRight :: Monad m => MSF (GameEnv m) () Ball
ballToRight =
  count >>> liftS addLeftPlayerPos >>> liftS checkHitR -- liftS = arrM, addLeftPlayerPos␣
↪is not defined (use lambda from previous section)
  where
    checkHitR :: n -> GameEnv m Int -- what is type n ?
    checkHitR n = do
      rp <- asks rightPlayerPos
      lift $ when (rp > n) $ tell ["Ball at " ++ show n] -- lift was missing?
      pure n -- need to return type Int
-- Couldn't match expected type ‘Int’ with actual type ‘n’. ‘n’ is a rigid type variable␣
↪bound by
```

```
import Control.Monad
import Control.Monad.Trans
import Control.Monad.Trans.MSF.Reader
import Control.Monad.Trans.MSF.Writer
import FRP.BearRiver

type Game = Ball
type Ball = Int

data GameSettings = GameSettings
  { leftPlayerPos  :: Int
  , rightPlayerPos :: Int
  }
```

(continues on next page)

(continued from previous page)

```
--type GameEnv m = ReaderT GameSettings m
--type GameEnv m = WriterT [String] (ReaderT GameSettings m)
type GameEnv m = ReaderT GameSettings (WriterT [String] m)

ballToRight :: Monad m => MSF (GameEnv m) () Ball
ballToRight = count >>> arrM (\n -> (n+) <$> asks leftPlayerPos) >>> arrM checkHitR
--ballToRight = count >>> arrM (\n -> (n+) <$> asks leftPlayerPos) >>> withSideEffect␣
↪checkHitR
  where
    checkHitR :: Monad m => Int -> GameEnv m Int -- change Int to () when using␣
↪withSideEffect
    checkHitR n = do
      rp <- asks rightPlayerPos
      lift $ when (rp > n) $ tell ["Ball at " ++ show n]
      pure n -- remove when using withSideEffect

hitRight :: Monad m => MSF (GameEnv m) Ball Bool
hitRight = arrM $ \i -> (i >=) <$> asks rightPlayerPos

testMSF = ballToRight >>> (arr id &&& hitRight)

main = do
  embed (runWriterS (runReaderS_ testMSF (GameSettings 0 3))) (replicate 5 ())
```

writert.hs

```
>>> cabal repl writert
>>> main
```

2.8 Other stuff

Todo: use mtl MonadReader variant on reader and writer examples

```
>>> cabal repl
>>> :m Control.Arrow Control.Monad.Identity Control.Monad.Trans.MSF.Reader Data.
↪MonadicStreamFunction.InternalCore FRP.BearRiver
>>> :t unMSF
-- unMSF :: MSF m a b -> a -> m (b, MSF m a b)
>>> runIdentity (runReaderT (unMSF (constant 1.0) ()) 0.1)
-- error: No instance for (Show (MSF (ClockInfo Identity) () Double)) arising from a use␣
↪of ‘print’
>>> let res = runIdentity (runReaderT (unMSF (constant 1.0) ()) 0.1)
>>> fst res
-- 1.0
```

```
>>> :m Control.Arrow Control.Monad.Identity Control.Monad.Trans.MSF.Reader Data.
↪MonadicStreamFunction.InternalCore FRP.BearRiver
>>> :l src/Main
```

(continues on next page)

(continued from previous page)

```
>>> :set -fbreak-on-error
>>> :trace fst $ runIdentity (runReaderT (unMSF (spring2 1.0 30.0 20.0) ()) 0.1)
```

Todo: incorporate Dunai Issue 245 - using embed with a constant 1.0 MSF to get (printable) results with an example for Yampa

```
main = do
  embed (count >>> arrM print) [(), (), ()]
-- > 1
-- > 2
-- > 3
-- > [(), (), ()]
```

If you only care for the side effects and want to ignore the returned list.

```
import Control.Monad.Trans.MSF.Maybe

main = do
  embed_ (count >>> arrM print) [(), (), ()]
-- > 1
-- > 2
-- > 3
```

CHAPTER

THREE

REACTIMATION

3.1 Basic reactimation in BearRiver Yampa

Type signatures of `reactimate`

```
reactimate
  :: Monad m =>
     m in                                  -- inputInit/senseInit
     -> (Bool -> m (DTime, Maybe in))      -- input/sense
     -> (Bool -> out -> m Bool)            -- output/actuate
     -> SF Identity in out                 -- process/signal function
     -> m ()
```

Note that the input and output procedure also have an `Bool` parameter which is unused if you look at the implementation therefore we can just ignore it.

Here is a minimal implementation which just produces 1.0s time deltas independent of the real world time. So far it's just a deterministic simulation like `embed` and doesn't provide any utility until we get real time deltas.

```
main = do
  reactimate inputInit input output time
    where
      inputInit  = pure ()
      input  _   = pure (1.0, Just ())
      output _ o = print o >> pure False
-- > 1.0
-- > 2.0
-- > 3.0
-- > ...
-- > Interrupted.
```

Type signature of `time`:

```
time :: Monad m => SF m a Time
```

The first question that comes to mind is: How do we quit `reactimate`? Well in this case we don't, so just press `Control-c`. On Windows there is an issue which messes up the console once you press `Control-c` within the app. So you probably have to restart ghci now otherwise you get garbeld output and strange error messages.

Here is a more complete example which implements a full game loop with real time deltas. Note that in a real game loop you wouldn't write it this way because you would like to have a stable frame rate, make the delay dependent of missed time and account for skipped frames. Also I don't know how precise `Data.Time.Clock.getCurrentTime` is.

But let's keep it simple for now. On Windows, make sure you run this example without the `--io-manager=native` option, otherwise `threadDelay` hangs the execution.

```
{-# LANGUAGE Arrows #-}

import Control.Concurrent (threadDelay)
import Data.Functor.Identity
import Data.IORef
import Data.Time.Clock
import FRP.BearRiver

inputInit :: IO String
inputInit = do
  pure "inputInit" -- use this if input doesn't produce anything

input :: IORef UTCTime -> UTCTime -> Bool -> IO (DTime, Maybe String)
input dtRef tInit _ = do
  now <- getCurrentTime
  prev <- readIORef dtRef
  writeIORef dtRef now
  let dt = realToFrac $ diffUTCTime now prev  -- delta time
  let at = realToFrac $ diffUTCTime now tInit -- absolute time
  return (dt, Just (show now)) -- usually you would want to use device inputs or␣
↪resource media here but let's produce some input strings from time

sf :: Monad m => SF m String String
sf = proc nowStr -> do
  t <- integral -< 1.0 :: Double
  returnA -< " integral: " ++ show t ++ " now: " ++ nowStr

output :: Bool -> String -> IO Bool
output _ out = do
  putStrLn out
  threadDelay secs -- DON'T USE IT LIKE THIS IN A REAL GAME LOOP
  return False
  where secs = 1000 * 1000 -- pico seconds

main = do
  putStrLn "Warning: On Windows threadDelay only works without io-manager=native options!
↪"
  putStrLn "Time progress in about 1sec (+ some processor time) steps. Interrupt with␣
↪[Control-C]!"
  t <- getCurrentTime
  dtRef <- newIORef t

  reactimate inputInit (input dtRef t) output sf

  putStrLn "...end"

-- > start...
-- > now: 2021-09-12 09:42:57.1321897 UTC integral: 0.0
-- > now: 2021-09-12 09:42:58.1421375 UTC integral: 1.0099473
-- > now: 2021-09-12 09:42:59.1492352 UTC integral: 2.0170455
-- > Interrupted.
```

reactimate.hs

```
>>> cabal repl reactimate # DON'T USE io-manager=native HERE!
>>> main
```

Press `Control-c` again to quit. On Windows you need to restart GHCI one more time.

Note that `inputInit` is only used as fallback here if input never produces anything. You can see that if you end input with:

```
input dtRef tInit _ = do
  ...
  pure (dt, if at < 3 then Nothing else Just (show now))

-- > start...
-- > now: inputInit integral: 0.0
-- > now: inputInit integral: 1.0099473
-- > now: inputInit integral: 2.0170455
-- > now: 2021-09-12 09:42:59.1492352 UTC integral: 3.0270455
-- > Interrupted.
```

On Windows make sure you use GHC 9 and start with `--io-manager=native -RTS` option, otherwise `NoBuffering` and `getChar` wont work:

```
>>> cabal repl reactimate --ghci-options '+RTS --io-manager=native -RTS'
```

In this example we use a poor man's input handling with `getChar` which just reads one character from the console. Depending on how long the user waits we get the corrsponding time delta. Once you press `Q` the app quits.

```
{-# LANGUAGE Arrows #-}

import Data.Functor.Identity
import Data.IORef
import Data.Time.Clock
import FRP.BearRiver
import Numeric
import System.IO ( stdin, hSetBuffering, BufferMode(NoBuffering) )

inputInit :: IO (Char, String)
inputInit = do
  pure ('\x00', "inputInit")

input :: IORef UTCTime -> UTCTime -> a -> IO (DTime, Maybe (Char, String))
input dtRef tInit _ = do
  c <- getChar

  now <- getCurrentTime
  prev <- readIORef dtRef
  writeIORef dtRef now
  let dt = realToFrac $ diffUTCTime now prev  -- delta time
  let at = realToFrac $ diffUTCTime now tInit -- absolute time
  pure (dt, Just (c, show now))

sf :: SF Identity (Char, String) (Bool, String)
sf = proc (input, nowStr) -> do
```

(continues on next page)

(continued from previous page)

```
  t <- integral -< 1.0 :: Double
  let out = "integral: " ++ showFFloat (Just 2) t "" ++ " now: " ++ nowStr
  returnA -< (input == 'q', out)

output :: a -> (Bool, String) -> IO Bool
output _ (quit, out) = do
  putStrLn out
  pure quit

main = do
  putStrLn "Repeatedly press any key or [Q] to quit!"

  hSetBuffering stdin NoBuffering
  t <- getCurrentTime
  dtRef <- newIORef t

  reactimate inputInit (input dtRef t) output sf

  putStrLn "...end"

-- > start...
-- > Repeatedly press any key or [Q] to quit!
-- > integral: 0.64 now: 2021-09-12 10:09:21.9324477 UTC
-- > integral: 1.80 now: 2021-09-12 10:09:23.0927901 UTC
-- > integral: 2.73 now: 2021-09-12 10:09:24.0177477 UTC
-- > ...end
```

input.hs

```
>>> cabal repl input #--ghci-options '+RTS --io-manager=native -RTS'
>>> main
```

3.2 Reactimate in Dunai

type signatures of reactimate

```
-- BearRiver
reactimate
  :: Monad m =>
     m in                                  -- inputInit/senseInit
     -> (Bool -> m (DTime, Maybe in))      -- input/sense
     -> (Bool -> out -> m Bool)            -- output/actuate
     -> SF Identity in out                 -- process/signal function
     -> m ()
```

```
-- Dunai
reactimate :: Monad m => MSF m () () -> m ()
```

```
main = do
  hSetBuffering stdin LineBuffering -- important if you work on Windows
```

(continues on next page)

(continued from previous page)

```
  putStrLn "Enter some words: "
  Dunai.reactimate (arrM (const getLine) >>> arr reverse >>> arrM putStrLn)
-- > Enter some words:
-- > hello
-- > olleh
-- > world
-- > dlorw
```

Note that the functions `arrM_` and `liftS` mentioned on https://github.com/ivanperez-keera/dunai don't really exist.

From the paper:

```
liftLM :: (Monad m, Monad n) => (forall a . m a -> n a) -> MSF m a b -> MSF n a b

liftST = liftLM . lift
```

From source:

```
-- | Lifts a monadic computation into a Stream.
constM :: Monad m => m b -> MSF m a b
constM = arrM . const

-- | Apply a monadic transformation to every element of the input stream.
-- Generalisation of 'arr' from 'Arrow' to monadic functions.
arrM :: Monad m => (a -> m b) -> MSF m a b
arrM = ...

-- | Apply trans-monadic actions (in an arbitrary way).
-- This is just a convenience function when you have a function to move across
-- monads, because the signature of 'morphGS' is a bit complex.
morphS :: (Monad m1, Monad m2)
      => (forall c . m1 c -> m2 c)
      -> MSF m1 a b
      -> MSF m2 a b
morphS = ...

-- | Lift inner monadic actions in monad stacks.
liftTransS :: (MonadTrans t, Monad m, Monad (t m))
           => MSF m a b
           -> MSF (t m) a b
liftTransS = morphS lift
```

Therefore:

```
arrM_  => constM
liftS  => arrM
liftLM => morphS
liftST => liftTransS
```

3.3 MyReactimate

Let's implement our own `myreactimate`

```
{-# LANGUAGE Arrows #-}
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE KindSignatures #-}

import Control.Monad.Trans.MSF.Reader
import Data.Functor.Identity
import Data.MonadicStreamFunction
import System.IO

type SF m = MSF (ReaderT Double m)

input :: IO (Double, Char)
input = do
  i <- getChar
  pure (dt, i)
  where dt = 1.0

process :: SF Identity Char String
process = proc i -> do                -- input is inserted as an arrow input
  tStr <- constM (show <$> ask) -< () -- whereas the time deltas are provided from the␣
↪reader monad
  returnA -< tStr

output :: String -> IO ()
output o = putStrLn o

myreactimate :: forall a b. IO (Double, a) -> (b -> IO ()) -> SF Identity a b -> IO ()
myreactimate sense actuate sf = reactimate $ senseSF sense >>> sfIO sf >>> actuateSF␣
↪actuate
  where
    senseSF :: forall (m :: * -> *) a. Monad m => m a -> MSF m () a
    senseSF s = constM s
    actuateSF :: forall (m :: * -> *) a b. Monad m => (a -> m b) -> MSF m a b
    actuateSF a = arrM a
    sfIO :: forall (m2 :: * -> *) r a b. Monad m2 => MSF (ReaderT r Identity) a b -> MSF␣
↪m2 (r, a) b
    sfIO s = morphS (pure . runIdentity) (runReaderS s)

main :: IO ()
main = do
  putStrLn "Press any key to get fake delta times and [Control-C] to interrupt!"
  hSetBuffering stdin NoBuffering
  myreactimate input output process
```

`myreactimate0.hs`

```
>>> cabal repl myreactimate0 --ghci-options '+RTS --io-manager=native -RTS'
>>> main
```

Now if want to make the type signatures explicit

```
myreactimate :: IO (Double, a) -> (b -> IO ()) -> SF Identity a b -> IO ()
myreactimate sense actuate sf = reactimate $ senseSF >>> sfIO >>> actuateSF
  where
    senseSF :: MSF IO () (Double, a)
    senseSF = constM sense
    actuateSF :: MSF IO b ()
    actuateSF = arrM actuate
    sfIO :: MSF IO (Double, a) b
    sfIO = morphS (pure . runIdentity) (runReaderS sf)

-- Couldn't match type ‘b1’ with ‘b’
-- Expected: MSF IO b1 ()
-- Actual: MSF IO b ()
-- ‘b1’ is a rigid type variable bound by
-- the type signature for: actuateSF :: forall b1. MSF IO b1 ()
```

Thats because the variable b mentioned in the signature is different from the b mentioned in the signature of `actuateSF` (which is renamed to `b1`). We would have to use `ScopedTypeVariables` and `forall`, which says something like: “for the following block assume the variables to be the same.”

```
{-# LANGUAGE ScopedTypeVariables #-}

myreactimate :: forall a b. IO (Double, a) -> (b -> IO ()) -> SF Identity a b -> IO ()
myreactimate sense actuate sf = reactimate $ senseSF >>> sfIO >>> actuateSF
  where
    senseSF :: MSF IO () (Double, a)
    senseSF = constM sense
    actuateSF :: MSF IO b ()
    actuateSF = arrM actuate
    sfIO :: MSF IO (Double, a) b
    sfIO = morphS (pure . runIdentity) (runReaderS sf)
```

Here is the version that Haskell Language Server suggested

```
{-# LANGUAGE RankNTypes, KindSignatures #-}

myreactimate :: forall a b. IO (Double, a) -> (b -> IO ()) -> SF Identity a b -> IO ()
myreactimate sense actuate sf = reactimate $ senseSF sense >>> sfIO sf >>> actuateSF␣
↪actuate
  where
    senseSF :: forall (m :: * -> *) a. Monad m => m a -> MSF m () a
    senseSF s = constM s
    actuateSF :: forall (m :: * -> *) a b. Monad m => (a -> m b) -> MSF m a b
    actuateSF a = arrM a
    sfIO :: forall (m2 :: * -> *) r a b. Monad m2 => MSF (ReaderT r Identity) a b -> MSF␣
↪m2 (r, a) b
    sfIO s = morphS (pure . runIdentity) (runReaderS s)
```

Can we use Yampa functions within? Turns out: yes

Yampa definitions:

```
type SF m = MSF (ClockInfo m)
type ClockInfo m = ReaderT DTime m
```

(continues on next page)

(continued from previous page)

```
type DTime = Double

SF :: Monad m => MSF (ReaderT Double m) a b
```

```
{-# LANGUAGE Arrows, ScopedTypeVariables #-}

import Control.Monad.Trans.MSF.Reader
import Data.Functor.Identity
import Data.MonadicStreamFunction
import FRP.BearRiver hiding (reactimate)
import System.IO

input :: IO (Double, Char)
input = do
  i <- getChar
  pure (dt, i)
  where dt = 1.0

process :: SF Identity Char String -- this is an BearRiver.SF now, not a Dunai.MSF
process = proc i -> do
  t <- sumS -< 1.0 :: Double -- using BearRiver.sumS here!
  returnA -< show t

output :: String -> IO ()
output o = putStrLn o

myreactimate :: forall a b. IO (Double, a) -> (b -> IO ()) -> SF Identity a b -> IO ()
myreactimate sense actuate sf = reactimate $ senseSF >>> sfIO >>> actuateSF
  where
    senseSF :: MSF IO () (Double, a)
    senseSF = constM sense
    actuateSF :: MSF IO b ()
    actuateSF = arrM actuate
    sfIO :: MSF IO (Double, a) b
    sfIO = morphS (pure . runIdentity) (runReaderS sf)

main :: IO ()
main = do
  putStrLn "Press any key to get custom process counts and [Control-C] to interrupt!"
  hSetBuffering stdin NoBuffering
  myreactimate input output process
```

myreactimate1.hs

```
>>> cabal repl myreactimate1 #--ghci-options '+RTS --io-manager=native -RTS'
>>> main
```

So if you disagree on how Yampa's `reactimate` is implemented, this is your chance. Let's get back to using *BearRiver.reactimate* though because it already works.

3.4 Terminating myreactimate

> [FrpRefac16] 4.3 Exceptions and Control Flow - MSFs can use different monads to define control structures. One common construct is switching, that is, applying a transformation until a certain time, and then applying a different transformation. We can implement an equivalent construct using monads like Either or Maybe. We could define a potentially-terminating MSF as an MSF in a MaybeT m monad. Following the same pattern as before, the associated running function would have type: runMaybeS :: Monad m => MSF (MaybeT m) a b -> MSF m a (Maybe b). Our evaluation function step, for this monad, would have type MSF Maybe a b -> a -> Maybe (b, MSF Maybe a b) indicating that it may produce no continuation. runMaybeS outputs Nothing continuously once the internal MSF produces no result.

Well, that doesn't help us much. Fortunately Dunai provides a `reactimateB` function which uses the `ExceptT` internally and allows us to quit using a `Bool`.

```haskell
{-# LANGUAGE Arrows #-}

import Control.Monad.Trans.MSF.Except (reactimateB)
import Control.Monad.Trans.MSF.Reader
import Data.Functor.Identity
import Data.MonadicStreamFunction
import System.IO

type SF m = MSF (ReaderT Double m)

input :: IO (Double, Char)
input = do
  i <- getChar
  pure (dt, i)
  where dt = 1.0

process :: SF Identity Char (Bool, String)
process = proc i -> do                -- input is inserted as an arrow input
  tStr <- constM (show <$> ask) -< () -- whereas the time deltas are provided from the␣
↪reader monad
  returnA -< (i == 'q', tStr)

output :: (Bool, String) -> IO Bool
output (quit, o) = do
  putStrLn o
  pure quit
  -- we could also move the quit logic from process to output directly

myreactimate :: IO (Double, a) -> (b -> IO Bool) -> SF Identity a b -> IO ()
myreactimate sense actuate sf = reactimateB $ senseSF sense >>> sfIO sf >>> actuateSF␣
↪actuate
  where
    senseSF :: Monad m => m a -> MSF m () a
    senseSF s = constM s
    actuateSF :: Monad m => (a -> m b) -> MSF m a b
    actuateSF a = arrM a
    sfIO :: Monad m2 => MSF (ReaderT r Identity) a b -> MSF m2 (r, a) b
    sfIO s = morphS (pure . runIdentity) (runReaderS s)
```

(continues on next page)

(continued from previous page)

```
main = do
  putStrLn "Press any key to get fake delta times and [Q] to quit!"
  hSetBuffering stdin NoBuffering
  myreactimate input output process
```

myreactimateB.hs

```
>>> cabal repl myreactimateB # --ghci-options '+RTS --io-manager=native -RTS'
>>> main
```

```
{-# LANGUAGE Arrows #-}

import Control.Monad
import Control.Monad.Trans.MSF.Except (reactimateB)
import Control.Monad.Trans.MSF.Reader
import Data.Functor.Identity
import Data.IORef
import Data.MonadicStreamFunction
import System.IO

type SF m = MSF (ReaderT Double m)

input :: IORef Bool -> IO (Double, Char)
input quitRef = do
  i <- getChar
  when (i == 'q') $ writeIORef quitRef True
  pure (dt, i)
  where dt = 1.0

process :: SF Identity Char String
process = proc i -> do                -- input is inserted as an arrow input
  tStr <- constM (show <$> ask) -< () -- whereas the time deltas are provided from the␣
↪reader monad
  returnA -< tStr

output :: IORef Bool -> String -> IO Bool
output quitRef o = do
  putStrLn o
  readIORef quitRef

myreactimate :: IO (Double, a) -> (b -> IO Bool) -> SF Identity a b -> IO ()
myreactimate sense actuate sf = reactimateB $ senseSF sense >>> sfIO sf >>> actuateSF␣
↪actuate
  where
    senseSF :: Monad m => m a -> MSF m () a
    senseSF s = constM s
    actuateSF :: Monad m => (a -> m b) -> MSF m a b
    actuateSF a = arrM a
    sfIO :: Monad m2 => MSF (ReaderT r Identity) a b -> MSF m2 (r, a) b
    sfIO s = morphS (pure . runIdentity) (runReaderS s)

main = do
```

(continues on next page)

(continued from previous page)

```
  putStrLn "Press any key to get fake delta times and [Q] to quit!"
  hSetBuffering stdin NoBuffering
  quitRef <- newIORef False
  myreactimate (input quitRef) (output quitRef) process
```

`myreactimateB1.hs`

> [FrpExt17] 6.4 Experience - Personally, I have found that using monad stacks with multiple transformers makes MSFs hard to work with, in spite of the multiple benefits of being able to describe switching in terms of EitherT or parallelism in terms of ListT. With this in mind, it is perhaps still convenient to add layers of abstraction on top of MSFs to describe Functional Reactive Programming or other mathematical constructs, as opposed to expressing programs directly in terms of Monadic Stream Functions (even if they are still implemented that way). Note, however, that programmers with a more mathematical background may find working with monad stacks more straightforward.

Okay...

CHAPTER

FOUR

YAMPA

4.1 Quickstart

runReader (embed (yourSF "init") [NoEvent, NoEvent, Event ()]) 1.0

4.2 Accumulating state

The point of this example is: time is entered as input, but keyboard presses are also entered as input.

```
{-# LANGUAGE Arrows #-}

import FRP.BearRiver
import Data.IORef
import Data.Time.Clock
import Data.Functor.Identity
import Numeric
import System.IO
import Data.Char

inputInit :: IO Char
inputInit = do
  pure '\x00'

input :: IORef UTCTime -> a -> IO (DTime, Maybe Char)
input dtRef _ = do
  c <- getChar

  now <- getCurrentTime
  prev <- readIORef dtRef
  writeIORef dtRef now
  let dt = realToFrac $ diffUTCTime now prev  -- delta time
  pure (dt, Just c)

process :: SF Identity Char (Bool, String, String)
process = proc i -> do
  t <- time -< ()

  let n = if isDigit i then digitToInt i else 0
  c <- sumS -< fromIntegral n :: Double -- required for VectorSpace
```

(continues on next page)

(continued from previous page)

```
  -- c <- feedback 0 (arr feedbackAdd) -< n
  -- c <- stateful 0 statefulAdd -< n
  -- let nE = if isDigit i then Event (digitToInt i) else NoEvent
  -- c <- accumHoldBy accumAdd 0 -< nE

  returnA -< (i == 'q', showFFloat (Just 2) t "", show c)
    where
      -- feedbackAdd (accu, new) = dup (accu + new)
      -- statefulAdd accu new = accu + new
      -- accumAdd new accu = accu + new
      -- stateful :: Monad m => b -> (a -> b -> b) -> SF m a b
      -- stateful bInit f = proc a -> do
      --   b' <- feedback bInit (arr (\(a, b) -> dup $ f a b)) -< a
      --   returnA -< b'

output :: a -> (Bool, String, String) -> IO Bool
output _ (quit, timeStr, countStr) = do
  putStrLn $ "time: " ++ timeStr ++ " counter: " ++ countStr
  pure quit

main = do
  putStrLn "Enter some chars to tick, digits to add or quit with [Q]!"
  hSetBuffering stdin NoBuffering
  t <- getCurrentTime
  dtRef <- newIORef t

  reactimate inputInit (input dtRef) output process

  putStrLn "...end"
```

state.hs

```
>>> cabal repl state # --ghci-options '+RTS --io-manager=native -RTS'
>>> main
```

4.3 Stateful functions

- constant
- arr sin
- time
- count
- integral
- sumS
- feedback 0 (arr feedbackAdd
- stateful 0 statefulAdd
- hold 0

- accumHoldBy accumAdd 0

4.4 Animation

The point of this example is: it doesn't matter how sophisticated the rendering will be in the end, we can define a basic animation function. It doesn't matter if we are using an ASCII renderer, 2D pixels, 2D vectors or an 3D renderer (which is still further abstracted into OpenGL, Direct3D and Vulkan and makes everything complicated).

```
{-# LANGUAGE Arrows #-}

import Data.Functor.Identity
import Data.IORef
import Data.Time.Clock
import FRP.BearRiver
import System.IO
--import System.Process (system)           -- alternative Windows clearScreen
--import System.Console.ANSI (clearScreen) -- alternative Linux console controls

-- a poor mans' console controls
hideCursor    = putStr  "\ESC[?25l"
clearScreen   = putStr  "\ESC[1J"
setCursor y x = putStr ("\ESC[" ++ show (max (y + 1) 1) ++ ";" ++ show (max (x + 1) 1)␣
↪++ "H")

inputInit :: IO (Maybe Char)
inputInit = do
  pure Nothing

input :: IORef UTCTime -> UTCTime -> a -> IO (DTime, Maybe (Maybe Char))
input dtRef tInit _ = do
  hasInput <- hWaitForInput stdin 100
  mc <- if hasInput then Just <$> getChar else pure Nothing

  now <- getCurrentTime
  prev <- readIORef dtRef
  writeIORef dtRef now
  let dt = realToFrac $ diffUTCTime now prev
  pure (dt, Just mc)

process :: SF Identity (Maybe Char) (Bool, String)
process = proc a -> do
  t <- time -< ()

  frame <- arr (\t -> ball !! (floor t `mod` length ball)) -< t

  let quit = Just 'q' == a
  returnA -< (quit, frame : "")
  where
    ball = "°Oo_oO"

output :: a -> (Bool, String) -> IO Bool
output _ (quit, frame) = do
```

(continues on next page)

(continued from previous page)

```
  clearScreen
  --system "cls" -- alternative Windows clearScreen
  setCursor 0 0
  putStr frame
  pure quit

main = do
  putStrLn "Watch animation and press [Q] to quit!"

  hideCursor
  hSetBuffering stdin NoBuffering
  t <- getCurrentTime
  dtRef <- newIORef t

  reactimate inputInit (input dtRef t) output process

  putStrLn "...end"

-- > °Oo_oO°Oo_oO°Oo_oO (animating one frame per 100ms)
```

animation.hs

```
>>> cabal repl animation # --ghci-options '+RTS --io-manager=native -RTS'
>>> main
```

4.4.1 Refinement

Now we have several options for animations:

Use arrow combinators with time and speed up the animation by 5

```
process = proc a -> do
  frame <- (\t -> ball !! (floor (5.0 * t) `mod` length ball)) ^<< time -< ()
  ...
```

Extract the animation logic into several functions which allows us to reuse it however we want:

```
animate :: [a] -> Double -> SF Identity () a
animate frames speed = constant speed >>> integral >>^ getFrame frames
--animate frames speed = time >>^ (* speed) >>^ getFrame frames

getFrame :: [a] -> Double -> a
getFrame frames t = let n = length frames in frames !! (floor t `mod` n)

process = proc a -> do
  frame <- animate ball 5.0 -< ()
  ...
```

4.5 Movement

```haskell
{-# LANGUAGE Arrows #-}

import FRP.BearRiver
import Data.IORef
import Data.Time.Clock
import Data.Functor.Identity
import System.IO

ball  = "°Oo_oO"

hideCursor    = putStr  "\ESC[?25l"
clearScreen   = putStr  "\ESC[1J"
setCursor y x = putStr ("\ESC[" ++ show (max (y + 1) 1) ++ ";" ++ show (max (x + 1) 1)␣
↪++ "H")

animate :: [a] -> Double -> SF Identity () a
animate frames speed = constant speed >>> integral >>^ getFrame frames

getFrame :: [a] -> Double -> a
getFrame frames t = let n = length frames in frames !! (floor t `mod` n)

inputInit :: IO (Maybe Char)
inputInit = do
  pure Nothing

input :: IORef UTCTime -> UTCTime -> a -> IO (DTime, Maybe (Maybe Char))
input dtRef tInit _ = do
  hasInput <- hWaitForInput stdin 100
  mc <- if hasInput then Just <$> getChar else pure Nothing

  now <- getCurrentTime
  prev <- readIORef dtRef
  writeIORef dtRef now
  let dt = realToFrac $ diffUTCTime now prev
  pure (dt, Just mc)

process :: SF Identity (Maybe Char) (Bool, (Int, Int, String))
process = proc a -> do
  obj <- object -< a
  let quit = Just 'q' == a
  returnA -< (quit, obj)

object :: SF Identity (Maybe Char) (Int, Int, String)
object = proc a -> do
  x <- accumHoldBy (+) 0 -< case a of Just 'd' -> Event 1; Just 'a' -> Event (-1); _ ->␣
↪NoEvent
  y <- accumHoldBy (+) 0 -< case a of Just 's' -> Event 1; Just 'w' -> Event (-1); _ ->␣
↪NoEvent
  frame <- animate ball 5.0 -< ()
  returnA -< (x, y, frame : "")
```

(continues on next page)

(continued from previous page)

```
output :: a -> (Bool, (Int, Int, String)) -> IO Bool
output _ (quit, (x, y, frame)) = do
  clearScreen
  setCursor y x
  putStr frame
  pure quit

main = do
  putStrLn "Press [WASD] to move around and [Q] to quit!"
  hideCursor
  hSetBuffering stdin NoBuffering
  t <- getCurrentTime
  dtRef <- newIORef t
  hasInput <- hWaitForInput stdin 5000

  reactimate inputInit (input dtRef t) output process

  clearScreen
  putStrLn "...end"
```

objmove.hs

```
>>> cabal repl objmove # --ghci-options '+RTS --io-manager=native -RTS'
>>> main
```

4.5.1 Refinement

If you want to add bounds to the position you could write:

```
object = proc a -> do
  x <- accumHoldBy (\old dir -> max 0 . min 30 $ old + dir) 0 -< case a of Just 'd' ->␣
↪Event 1; Just 'a' -> Event (-1); _ -> NoEvent
  y <- accumHoldBy (\old dir -> max 0 . min 10 $ old + dir) 0 -< case a of Just 's' ->␣
↪Event 1; Just 'w' -> Event (-1); _ -> NoEvent
  ...
```

Here are some other animations

```
stick = "-/|\\"
emoji = ""
clock = ""
```

4.6 Recursive states

```haskell
{-# LANGUAGE Arrows #-}

import Control.Monad
import Control.Monad.Fix
import Control.Monad.Trans
import Control.Monad.Trans.MSF.Reader
import Control.Monad.Trans.MSF.Writer
import Data.Functor.Identity
import Data.IORef
import Data.Text.Chart (plot)
import Data.Time.Clock
import FRP.BearRiver
import Numeric
import System.IO

hideCursor    = putStr  "\ESC[?25l"
clearScreen   = putStr  "\ESC[1J"
setCursor y x = putStr ("\ESC[" ++ show (max (y + 1) 1) ++ ";" ++ show (max (x + 1) 1)␣
↪++ "H")

inputInit :: IO (Maybe Char)
inputInit = do
  pure Nothing

input :: IORef UTCTime -> IORef Bool -> a -> IO (DTime, Maybe (Maybe Char))
input dtRef quitRef _ = do
  hasInput <- hWaitForInput stdin 100
  mc <- if hasInput then Just <$> getChar else pure Nothing
  when (mc == Just 'q') $ writeIORef quitRef True

  now <- getCurrentTime
  prev <- readIORef dtRef
  writeIORef dtRef now
  let dt = realToFrac $ diffUTCTime now prev
  pure (dt, Just mc)

output :: IORef Bool -> a -> Double -> IO Bool
output quitRef _ x = do
  clearScreen
  setCursor 0 0
  putStr $ replicate (round x) '~' ++ "O"
  readIORef quitRef

spring :: Monad m => Double -> Double -> Double -> SF m a Double
spring k x0 xdInit = feedback xInit $ proc (_, xt) -> do
  let xd = x0 - xt
  ft  <- integralFrom 0     -< xd
  xt' <- integralFrom xInit -< ft / k
  returnA -< (xt', xt')
  where
```

(continues on next page)

(continued from previous page)

```
    xInit = x0 + xdInit

main = do
  putStrLn "Watch an animated spring and press [Q] to quit!"
  hasInput <- hWaitForInput stdin 5000
  hSetBuffering stdin NoBuffering
  hideCursor
  t <- getCurrentTime
  dtRef <- newIORef t
  quitRef <- newIORef False

  reactimate inputInit (input dtRef quitRef) (output quitRef) (spring k x0 xdInit)

  clearScreen
  setCursor 0 0
  putStrLn "Time-distance diagram (using ASCII plot):"
  ls <- embed (runReaderS_ (spring k x0 xdInit) 0.1) (replicate 120 ())
  plot $ map round ls

  putStrLn "...end"
  where
    k      =  1.0
    x0     = 30.0
    xdInit = 20.0

-- > Watch an animated spring and press [Q] to quit!
-- > ~~~~~~~~~~~~~~~~~~~~O (bouncing back and forth)
-- > Time-distance diagram (using Unicode plot):
-- > 50.00                                                           ───                ␣
↪                                         ────
-- > 47.14 ──                                                    ──   ──                ␣
↪                                    ─    ──
-- > 44.29    ──                                              ──         ─              ␣
↪                                 ──         ─
-- > 41.43       ─                                          ─              ─            ␣
↪                               ─              ─
-- > 38.57                                                ─                  ─          ␣
↪                             ─                  ─
-- > 35.71          ─                                                          ─        ␣
↪                                                  ─
-- > 32.86                                             ─                                ␣
↪                          ─
-- > 30.00                                                                              ␣
↪
-- > 27.14              ─                                                          ─    ␣
↪                                                      ─
-- > 24.29                                         ─                                    ␣
↪                      ─
-- > 21.43                 ─                                                          ─ ␣
↪                                                         ─
-- > 18.57                   ─                  ─                                      ␣
↪─                   ─
```

(continues on next page)

(continued from previous page)

```
-- > 15.71                    –              –                                  ␣
↪ –              –                                    –
-- > 12.86                      –         ——                                   ␣
↪   –          ——                                        –
-- > 10.00                        ——————————                                  ␣
↪       ——————————
-- > ...end
```

spring.hs

```
>>> cabal repl spring # --ghci-options '+RTS --io-manager=native -RTS'
>>> main
```

```haskell
-- | Well-formed looped connection of an output component as a future input.
feedback :: Monad m => c -> MSF m (a, c) (b, c) -> MSF m a b
feedback c sf = MSF $ \a -> do
  ((b', c'), sf') <- unMSF sf (a, c)
  return (b', feedback c' sf')
```

4.7 Switching behaviour

type	immediate	delayed
once	`switch`	`dSwitch`
recurring	`rSwitch`	`drSwitch`
parallel using broadcasting	`pSwitchB rpSwitchB`	`dpSwitchB drpSwitchB`
parallel using routing	`pSwitch rpSwitch`	`dpSwitch drpSwitch`
continuation	`kSwitch`	`kdSwitch`

Fig. 1: switch

Fig. 2: pSwitchB

Fig. 3: pSwitch

Reddit - How to read Yampa diagram?

These diagrams were originally designed by the author of this book. If you understand german you can read a description of `switch`, `pSwitchB` and `pSwitch` here: [GamArchYam10]

Todo: add english translation of switching diagrams from master thesis

CHAPTER

FIVE

USER INTERFACES

5.1 Musings

When we think about a UI element like a button we usually think about something simple like a rectangle which contains and reacts to a left-mouse click when the mouse cursor is within the button and then calls an action. If you take a look at the Unity3D UI ClickEvent however there are much more nuances to that. Let's muse about a button a bit. A button has a position, a rectangular dimension (width, height), text, border-, background- and text-color, text font etc. The mouse cursor has a position, which might have never been changed while the program started. We don't have a history of past positions, so we cannot define a direction over past movements. This might be interesting if we wanted to change the orientation of element to the direction of dragging. Does a drag start after a while or do we have to move. How far has the mouse cursor to move until it's considered a drag as opposed to a click with some jitter. Is the mouse position within the button. Is the click area of a button different than the visual representation so it is easier to click. Is there a difference between the mouse position within the visual representation or the magnetic area. If there is another UI element behind the button, does the click into it's visual area have higher precedence of the magnetic area of the button in front. If the user pressed a mouse button, moves the cursor and releases it at another position, did the click start in another element, in no element or the element we are looking right now. How to visual changes like highlight, hover, focus and press state react to those events. When does a hover start and show a tooltip. When did the click start. Can multiple events occurs at the same time (left down+right up) or can a click occur together with other events (key presses, modifier keys). Do we manage UI elements within a hierarchy and bubble events down and up. If we look at the device levels, what happens if a mouse is disconnected midway or can there even be mulitple mouse devices. With multi-touch this becomes even more interesting because we need to track multiple fingers with only an approximate accuracy of a touch. Are UI elements animated over time and how is this reflected on properties relevant for event handling (like a UI element moving away under a drag event).

Todo: Research UI architecture design considerations

With FRP and combinator type classes like Monads and Arrows we have great tools to separate all these concerns into their most basic and abstract form independent of a concrete input system (SDL, Unity3D, HTML etc.) and visual representation (console, OpenGL, HTML etc.).

5.2 Bi-directional UI elements

See Apfelmus - Three principles for GUI elements with bidirectional data flow. A textfield is a canonical example of a bi-directional UI element. The text can be changed programmatically but also by the user. So who is in charge of the internal text representation and how do we handle changes from each other side?

[FrpRefac16] also provides a good example across over multiple UI elements about who is in charge of the current page number in a text viewer.

> [FrpRefac16] 3.3.2: There are four different ways to move from one page to the next: with the toolbar buttons (top), by dragging the central area with the mouse (centre left), by scrolling down the page (centre right), and with the bottom toolbar controls. Each of these acts both as an input and an output.

Lets define a textfield which represent an initial text, a blinking cursor, allows to enter new character on the cursor position and change the cursor position, or change the text programmatically altogether. To keep the system simple we are going to use the console again, only allow one key at a time. We consider a few keys special for deleting (backspace) and moving left and right. To allow programmatic text changes we bind the num keys to fire setText-events at the length represented by the corresponding number (e.g. 5="XXXXX").

```
-- 3. GUI elements generate events only in response to user input, never in response to␣
↪program output.
textfield :: String -> SF Identity (KeyPressed, Event String) Textfield
textfield textInit = proc (keyPress, setText) -> do
  let
    backE  = filterE (== keyBack ) keyPress
    leftE  = filterE (== keyLeft ) keyPress
    rightE = filterE (== keyRight) keyPress
    charE  = if isNoEvent $ mergeEvents [backE, leftE, rightE] then keyPress else NoEvent

  rec
    let handleBack = backE `tag` (if cursorPosOld > 0 then removeAt textOld cursorPosOld␣
↪else textOld)
        handleChar = charE <&> insertAt textOld cursorPosOld
        limitPos p = min (length textNew) . max 0 $ p
    textNew      <- hold textInit -< mergeEvents [setText, handleBack, handleChar]
    cursorPosNew <- hold posInit  -< mergeEvents
      [ setText `tag`  cursorPosOld
      , backE   `tag` (cursorPosOld - 1)
      , leftE   `tag` (cursorPosOld - 1)
      , rightE  `tag` (cursorPosOld + 1)
      , charE   `tag` (cursorPosOld + 1)
      ] <&> limitPos
    textOld      <- iPre textInit -< textNew
    cursorPosOld <- iPre posInit  -< cursorPosNew

  cursorFrame <- animate cursorFrames 5.0 -< ()
  returnA -< (textNew, cursorPosNew, cursorFrame)
```

textfield1.hs

Warning: There is a bug in Dunai which makes *rec* and *iPre* definitions run into an infinite loop (see MSF arrows aren't associative in terms of evaluation). That's why the Dunai package is defined in *cabal.project*.

CHAPTER

SIX

CHEATSHEET

```
:{
do
  let yourMSF = count
  ls <- embed yourMSF ["input0", "input1", "input2"]
  print ls
:}

:{
do
  let yourSF = integral
      dt = 1 / 60
  ls <- runReaderT (embed yourSF $ replicate 10 123.0) dt
  print ls
:}

:{
do
  let yourSF = integral
      dt = 1 / 60
  ls <- embed (runReaderS_ yourSF dt) $ replicate 10 123.0
  print ls
:}
```

CHAPTER

SEVEN

LINKS

7.1 Main Yampa papers

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.

Functional Reactive Programming (FRP) has come to mean many things. Yet, scratch the surface of the multitude of realisations, and there is great commonality between them. This paper investigates this commonality, turning it into a mathematically coherent andpractical FRP realisation that allows us to express the functionality of many existing FRP systems and beyond by providing a minimal FRP core parametrised on a monad. We give proofs for our theoretical claims and we have verified the practical side by benchmarking a set of existing, non-trivial Yampa applications running on top ofour new system with very good results.

Programming GUI and multimedia in functional languages has been a long-term challenge, and no solution convinces the community at large. Purely functional GUI and multimedia toolkits enable abstract thinking, but have enormous maintenance costs. General solutions like Functional Reactive Programming present a number of limitations. FRP has traditionally resisted efficient implementation, and existing libraries sacrifice determinism and abstraction in the name of performance. FRP also enforces structural constraints that facilitate reasoning, but at the cost of modularity and separation of concerns. This work addresses those limitations with the introduction of Monadic Stream Functions, anextension to FRP parameterised over a monad. I demonstrate that, in spite of being simpler thanother FRP proposals, Monadic Stream Functions subsume and exceed other FRP implementations. Unlike other proposals, Monadic Stream Functions maintain purity at the type level, whichis crucial for testing and debugging. I demonstrate this advantage by introducing FRP testingfacilities based on temporal logics, together with debugging tools specific for FRP. I present two uses cases for Monadic Stream Functions: First, I show how the new constructsimproved the design of game features and non-trivial games. Second, I present Reactive Valuesand Relations, an abstraction for model-view coordination in GUI programs based on a relationallanguage, built on top of Monadic Stream Functions. Comprehensive examples are used to illustrate the benefits of this proposal in terms of clarity, modularity, feature coverage, and its low maintenance costs. The testing facilities mentioned before are used to encode and statically checkdesired interaction properties.

7.2 All Yampa papers

Sorted newest first:

- Wormholes: Introducing Effects to FRP (2012) by Daniel Winograd-Cort, Paul Hudak (video)
- Virtualizing Real-World Objects in FRP (2011) by Daniel Winograd-Cort, Hai Liu, Paul Hudak
- Causal commutative arrows and their optimization (2009) by Hai Liu, Eric Cheng, and Paul Hudak
- Demo-outline: Switched-on Yampa: Programming Modular Synthesizers in Haskell (2007) by George Giorgidze, Henrik Nilsson
- Plugging a space leak with an arrow (2007) by Hai Liu, Paul Hudak
- Dynamic optimization for functional reactive programming using generalized algebraic data types (2005) by Henrik Nilsson

7.3 Non-english Yampa papers

7.3.1 German

7.4 FRP papers

Sorted newest first:

- Back to the Future: Time Travel in FRP (2017) by Ivan Perez
- Rhine - frp with type-level clocks (2016) by Manuel Bärenz
- Higher-order functional reactive programming without spacetime leaks (2013) by N. R. Krishnaswami
- Push-Pull Signal-Function Functional Reactive Programming (2012) by E. Amsden
- Higher-order functional reactive pro- gramming in bounded space (2012) by N. R. Krishnaswami, N. Benton, J. Hoffmann
- Improving Push-based FRP (2008) by W. Jeltsch
- E-FRP with priorities (2007) by R. Kaiabachev, W. Taha, A. Zhu
- Plugging a space leak with an arrow (2007) by Hai Liu, Paul Hudak
- Functional Reactive Programming for Real-Time Reactive Systems (2002) by Wan, Zhanyong
- Event-driven FRP (2002) by Z. Wan, W. Taha, P. Hudak
- Real-time FRP (2001) by Z. Wan, W. Taha, and P. Hudak
- Functional reactive programming from first principles (2000) by Zhanyong Wan, Paul Hudak
- Functional reactive animation (1997) by Conal Elliott, Paul Hudak
- Reification of time in FRP
- Functional reactive programming for real-time reactive systems (2002) by Zhanyong Wan
- Fault tolerant functional reactive programming (2018) by Ivan Perez

Todo: split up technical papers and tutorials

7.5 Other papers

Sorted oldest first:

- Notions of computation and monads (1991) by Moggi
- Comprehending monads (1992) by Philip Wadler

Todo: where is the 2000 version of Generalising monads to arrows online?

7.6 Codedocs

- Hackage Dunai
  - Hackage Dunai - Core
- Hackage BearRiver
- Hackage Yampa (old Yampa!)

7.7 Repos

- https://github.com/ivanperez-keera/Yampa
  - https://github.com/keera-studios/keera-hails/tree/develop/keera-hails-reactive-yampa
- https://github.com/ivanperez-keera/dunai
  - https://github.com/ivanperez-keera/dunai/tree/develop/dunai-frp-bearriver
- https://github.com/turion/rhine

7.8 Examples

- SpaceInvaders
- Haskanoid
- Bearriver Arcade
- Pang-a-Lambda
- e1338
- 2048
- MandelbrotYampa

Todo: include from https://github.com/ivanperez-keera/Yampa/

7.9 Tutorials

- Yampy Cube by Konstantin Zudov at Helsinki FRP Meetup May 6, 2015 (video)
- Brief Introduction to Functional Reactive Programming and Yampa (2007) Henrik Nilsson
- wiki.haskell.org Yampa (old Yampa)
  - wiki.haskell.org Yampa/reactimate (old Yampa)
- The Yampa Arcade - Slides (archived: [1] (old Yampa)
- Yampa, Arrows, and Robots by Paul Hudak (old Yampa)
- Keera Studios Blog
  - Building a reactive calculator in Haskell

7.10 Old Yampa Examples

- https://github.com/pedromartins/cuboid
  - https://www.youtube.com/watch?v=-IpE0CyHK7Q
- https://github.com/helsinki-frp/yampy-cube
- https://github.com/werk/YampaShooter
- Vehicle Platooning Simulations with Functional Reactive Programming
- Stackoverflow - Is there a way to create a Signal Function out of getLine in Yampa using reactimate
- Stackoverflow - N-body with Yampa FRP, haskell

7.11 Libraries

- Euterpea
- reactive-banana
- elera
- elm
- TODO

7.12 Books

7.13 TODO

- Bridging the GUI Gap with Reactive Values and Relations (2005) by Ivan Perez, Henrik Nilsson
- Fault-Tolerant Swarms
- The essence and origins of FRP (2015) by Conal Elliott
- FRP Zoo

- Controlling Time and Space: understanding the many formulations of FRP" (2014) by Evan Czaplicki
- Imperative functional programming?
- https://www.youtube.com/watch?v=1MNTerD8IuI
- Wayward Tide was written in Haskell?
- https://notebook.readthedocs.io/en/latest/haskell/learn-you-a-haskell/chapter-2.html
- B. Victor, "Inventing on Principle." http://vimeo.com/36579366, 2011
- Declarative Game Programming (2014) by Henrik Nilsson, Ivan Perez (slides)
- Roy, Peter Van und Seif Haridi: Concepts, Techniques, and Models of Computer Programming. MIT Press, 2004.
- Asynchronous functional reactive programming for GUIs (2013) by E. Czaplicki and S. Chong
- A survey on reactive programming by E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter
- https://imve.informatik.uni-hamburg.de/files/116-blom_beckhaus_DIVE_VR.pdf
- https://imve.informatik.uni-hamburg.de/files/55-SEARISworkshop_DIVEs-FRVR.pdf
- Dunai Issue 236 - Bearriver: ArrowLoop yampa and bearriver differences
- Dunai Issue 245 - using embed with a constant 1.0 MSF to get (printable) results
- Dunai Issue 174 - Space Leak
- Cogs and Levers - Functional Reactive Programming with Yampa

CHAPTER

EIGHT

FREQUENTLY ASKED QUESTIONS

Q: Are Arrows a bad design choice?

Reddit - apfelmus on Yampa "In my opinion, Yampa's arrow style is very clunky to use, though.", "I don't perceive them as great. The main feature of arrows is that they restrict the flow of information (keyword: no strong monad), which is necessary for an efficient implementation of FRP, but the terrible price is that you have to do explicit plumbing. They don't solve the problem of restricting information flow in the usual applicative style, but that's exactly the problem I want to solve when I embed FRP into ordinary Haskell."

apfelmus is the implementor of reactive-banana and therefore his opinion matters on this topic.

What I Wish I Knew When Learning Haskell - Arrows "In practice this notation is not often used and may become deprecated in the future." (citation required)

CHAPTER

NINE

GLOSSARY

9.1 Haskell Base

Applicative
: just links and papers

Arrow
: short description okay, links and papers

Functor
: just links and papers

lifting
: lifting a function to a Monad, lifting a Monad down the stack, lifting a Monad to MSF

Monad
: just links and papers

Monad Transformer
: just links and papers

Monadic Stream Function
: see *Dunai*, [FrpRefac16]

Monadic Streams
: TODO

MSF
: see *Monadic Stream Function*

widening
: Arrow widening

9.2 Terminology

AFRP
: see *functional reactive programming, arrowized*

Behaviour
: TODO

causal
: TODO

CFRP
see *functional reactive programming, classic*

circuit network
see *signal network*

DCTP
see *denotative continuous time programming*

denotative continuous time programming
Stackoverflow.com - Specification for a Functional Reactive Programming language (2011) by Conal Elliott: "I'm glad you're starting by asking about a specification rather than implementation first. There are a lot of ideas floating around about what FRP is. ..."

domain-specific language
TODO

dynamic structure
it's one thing to run a fixed sized list of signals all behaving the same but another to make it change dynamically and with changing behaviours

Event
TODO

FP
see *functional programming*

FRP
see *functional reactive programming*

functional programming
TODO

functional reactive programming
Stackoverflow.com - What is (functional) reactive programming? (2009) by Conal Elliott

[FrpExt17] 3.3.2 "FRP tries to shift the direction of data-flow from message passing to data dependency. This helps reason about what things are over time, as opposed to how changes propagate."

functional reactive programming, arrowized
TODO: links to papers

functional reactive programming, classic
TODO: links to papers

functional reactive programming, pull-based
TODO: links to papers

functional recative programming, push-based
TODO: links to papers

hybrid system
TODO

imperative programming
TODO

object-oriented programming
TODO

procedural programming
TODO

programming paradigm
Roy, Peter Van und Seif Haridi: Concepts, Techniques, and Models of Computer Programming. MIT Press, 2004.

reactive programming
TODO: links to papers

simulation, deterministic
TODO

simulation, non-deterministic
TODO

space-leak
also see *time-leak*

state
building state up over time, not to be confused with State monad

temporal
TODO

time, continuous
Conal Elliott - Why program with continuous time?

time, discrete
TODO

time, hybrid
TODO

time-leak
also see *space-leak*

9.3 Yampa

embedding
TODO

reactimating
TODO

sense
TODO

actuate
TODO

wormholes
TODO

white hole
TODO

black hole
TODO

Switch
TODO

Signal
TODO

SF
see *Signal Function*

Signal Function
TODO

Signal Network
TODO: image

9.4 Implementations

BearRiver
new Yampa, compare with Dunai and Yampa

Clean
FRP implementation

Dunai
compare with Yampa and BearRiver

E-FRP
eventbased, FRP implementation

Elm
[FrpRefac16] 3.3.2 Limitations of FRP and Arrowized FRP: "Some FRP and FRP-inspired implementations and languages offer mechanisms to work around this problem. Elm [83], for instance, offers handles to push specific changes onto widgets, thus helping to break cycles involving interactive visual elements."

Esterel
[FrpRefac16] 11.1 Related Work - MSFs: "Esterel is a synchronous data-flow programming language with support for concurrency and signal inhibition. Esterel rejects programs that give multiple values to the same signal at the same sampling time. Our framework support classic FRP and, in principle, signals are defined once for all time. Signals in Esterel can be broadcasted across the whole program from any point. This form of broadcasting might be implementable with MSFs by means of a State monad. However, Esterel provides additional static guarantees, for example, that a broadcasted signal only has one value per cycle, and this could only be detectable during runtime with the approach based on MSF's mentioned above."

Euterpea
compare with Yampa

FRPNow!
FRP library

Lucid Synchrone
not to be confused with :hackage:lucid DSL for HTML

[FrpRefac16] 11.1 Related Work - MSFs: "Lucid Synchrone is a programming language inspired by Lustre, which extends it with high-level concepts from functional programming, a richer type language, and a more versatile and usable clock system. Types in Lucid Synchrone are polymorphic, and the type system supports type inference. This approach is closer to our MSFs, due to the richness of the language, although MSFs do not explicitly support clocks or include any kind of clock calculus. Like in the case of Lustre, the language of MSFs is bigger than that of Lucid Synchrone nodes: constructions that would be rejected by Lucid Synchrone's compiler might be accepted by the Haskell compiler if described using MSF. This makes these languages, in principle, safer than full MSFs, unless the"

Lustre

[FrpRefac16] 11.1 Related Work - MSFs: "Lustre is a synchronous data-flow programming language that has been used in aerospace, transportation, nuclear power plants and wind turbines, among many others. Lustre programs are defined in terms of flows (streams) and nodes (causal stream functions). Lustre's type system includes both information about the amount of history examined of each flow examined by each node, and clocking rates at which each flow is being produced. Together with Lustre's clock calculus, this helps guarantee that all flows are well-formed (always defined). MSFs do not have any notion of clocks or clocking rate, making it, in principle, harder to capture those constraints and ensure the same kinds of guarantees. MSFs are defined and combined using a series of combinators that ensure that all values are well-typed. Nevertheless, our language is embedded in Haskell, which allows representing bottoms, so a program written using MSFs may not be guaranteed to be productive (simply because the programmer may have used non-terminating Haskell expressions)."

Netwire

FRP library

http://hub.darcs.net/ertes/netwire.

Reactive Banana

[FrpRefac16] 3.3.2 Limitations of FRP and Arrowized FRP: "Some FRP and FRP-inspired implementations and languages offer mechanisms to work around this problem... Reactive Banana offers sinks for each WX widget property, to which a signal can be attached. These mechanisms are not general solutions applicable to every reactive element, but ad hoc solutions to enable pushing changes to specific kinds of resources."

RT-FRP

realtime, FRP implementation

Unity

Unity3D game engine

Yampa

old Yampa, compare with BearRiver and Dunai

CHAPTER

TEN

CONTRIBUTING

10.1 Comment using issues

You can also add comments with gitlab issues.

10.2 Fork GIT and create pull request

Fork the git repository, change the text and create a pull requests.

```
>>> git clone https://gitlab.com/gerold.meisinger/yampa-book
```

If you want to build the docs to see the final result before sending pull requests you need to install the sphinx documentation generator.

- Install python3

```
>>> apt install python3 python3-pip
```

- Install sphinx

```
>>> pip install sphinx sphinx-rtd-theme sphinx-comments
```

- Build the docs

```
>>> cd docs
>>> make html
```

If you want to edit SVGs they use the Computer Modern Unicode font which is also bundled with some Latex distributions and is used in a lot of scientific papers.

Windows: install manually from link

Linux:

```
>>> apt install cm-super
```

10.2.1 Troubleshooting

```
Could not install docutils: run pip install pip --upgrade manually
```

```
apt install python3-pip
```

10.3 Housekeeping tasks

Some housekeeping task everyone can contribute without knowledge on Haskell and Yampa:

- Correct typos
- Rewrite sentences with proper english (I'm not native speaker after all)
- Research links (papers, tutorial, examples etc.) and add RST markup
- Retrieve archive.org wayback machine versions for important links (papers) to make them permanent
- Fill in the glossary terms with description and citations from papers where the terms are mentioned
- Record screenshots and gif animations of graphical example outputs
- Work on todo list (see below)
- Run und test examples, add comments
- Add or answer *FAQs*
- Answer issues

10.4 Add content

- See roadmap in readme
- Good enough is good, better to have more content
- Keep the examples as simple as possible and self-contained
  - Is it really necessary to start up an SDL or OpenGL context to talk about animation or is the console with ASCII art sufficient
  - Is it really necessary to use a 3 dimensional vector to talk about stateful positions or is 1 dimension sufficient
  - Is it really necessary to use point-free super sections or is a verbose lambda with explicit type signature more comprehendable
- Scope
  - Yampa FRP meaning everything based on Dunai, BearRiver, Yampa (old), Euterpea(?), Rhine(?). It's okay to add comparisons to other libraries like Reactive-Banana, Elm, Elera etc. though.
  - Don't teach fundamentals about Haskell, Monads, computer game programming, computer graphics, physics programming etc.. It's okay to recall concepts required for a specific example.

10.5 Restructured Text

- https://bashtage.github.io/sphinx-material/rst-cheatsheet/rst-cheatsheet.html
- https://ghc.gitlab.haskell.org/ghc/doc/users_guide/editing-guide.html
- https://www.sphinx-doc.org/en/master/usage/restructuredtext/basics.html
- https://docutils.sourceforge.io/docs/ref/rst/directives.html
- https://sublime-and-sphinx-guide.readthedocs.io/en/latest/references.html

Use doctext >>> syntax for bash and ghci examples because it doesn't get marked when copying to clipboard and can be pasted as-is without changes.

doctext:

```
>>> echo hello world
# hello world
```

codeblock:

```
$ echo hello world
# -bash: $: command not found
```

10.6 Todos

Todo: Add discussion for feedback function

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/yampa-book/checkouts/main/docs/embed.rst, line 163.)

Todo: what's the point? why not just pass the config as a value?

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/yampa-book/checkouts/main/docs/embed.rst, line 181.)

Todo: ? why does that matter? m is a free variable, so we cannot use it for anything specific. we might as well set it to Identity? but I'm getting "Couldn't match expected type: IO t0 with actual type: Identity [Int]"

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/yampa-book/checkouts/main/docs/embed.rst, line 259.)

Todo: use mtl MonadReader variant on reader and writer examples

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/yampa-book/checkouts/main/docs/embed.rst, line 371.)

Todo: incorporate Dunai Issue 245 - using embed with a constant 1.0 MSF to get (printable) results with an example for Yampa

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/yampa-book/checkouts/main/docs/embed.rst, line 392.)

Todo: Complete history of FRP based on papers

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/yampa-book/checkouts/main/docs/intro.rst, line 140.)

Todo: Timeline of FRP

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/yampa-book/checkouts/main/docs/intro.rst, line 141.)

Todo: add simple arrow combinator examples

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/yampa-book/checkouts/main/docs/intro.rst, line 264.)

Todo: split up technical papers and tutorials

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/yampa-book/checkouts/main/docs/links.rst, line 91.)

Todo: where is the 2000 version of Generalising monads to arrows online?

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/yampa-book/checkouts/main/docs/links.rst, line 120.)

Todo: include from https://github.com/ivanperez-keera/Yampa/

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/yampa-book/checkouts/main/docs/links.rst, line 156.)

Todo: Research UI architecture design considerations

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/yampa-book/checkouts/main/docs/ui.rst, line 10.)

Todo: add english translation of switching diagrams from master thesis

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/yampa-book/checkouts/main/docs/yampa.rst, line 162.)

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[YamCade03] The Yampa Arcade (2003) by Antony Courtney, Henrik Nilsson, John Peterson (archived)

[FrpRefac16] Functional Reactive Programming, Refactored (2016) by Ivan Perez, Manuel Bärenz, Henrik Nilsson

[FrpExt17] Extensible and Robust Functional Reactive Programming (2017) by Iván Pérez Domínguez, MSc (First Year Report)

[TestDebugFrp17] Testing and Debugging Functional Reactive Programming (2017) by Ivan Perez, Henrik Nilsson

[CcaRevisit16] Causal Commutative Arrows Revisited (2016) by Jeremy Yallop, Hai Liu

[SetNonInfFrp14] Settable and Non-Interfering Signal Functions for FRP (2014) by Daniel Winograd-Cort, Paul Hudak (more)

[SwitchOn08] Switched-On Yampa (2008) by George Giorgidze, Henrik Nilsson

[DynInterVR08] Dynamic, Interactive Virtual Environments (2008) by Kristopher James Blom

[Fp3dGames05] Functional Programming and 3D Games (2005) by Mun Hon Cheong

[ArrRobFrp03] Arrows, robots, and Functional Reactive Programming (2003) by Paul Hudak, Antony Courtney, Henrik Nilsson, John Peterson

[FrpCont02] Functional Reactive Programming, Continued (2002) by Henrik Nilsson, Antony Courtney and John Peterson

[GamArchYam10] Game-Engine-Architektur mit funktional-reaktiver Programmierung in Haskell/Yampa (2010) von Gerold Meisinger

[VisIdeAfrp07] Visuelle Entwicklungsumgebung zur Erzeugung von Haskell AFRP Code (2007) von Piotr Szal

[MathPropMSF16] Mathematical Properties of Monadic Stream Functions (2016) by Manuel Bärenz, Ivan Perez, Henrik Nilsson

[SurvFrp10] A Survey of Functional Reactive Programming (2010) by Edward Amsden (archived)

[PushPullFrp09] Push-pull functional reactive programming (2009) by Conal Elliott (video) (slides)

[SafeFrpDep09] Safe functional reactive programming through dependent types (2009) by Neil Sculthorpe, Henrik Nilsson (video)

[GenuineUI01] Genuinely Functional User Interfaces (2001) by Antony Courtney and Conal Elliott

[WhyFP90] Why functional programming matters (1990) by John Hughes

[MonadFP95] Monads for functional programming (1995) Philip Wadler

[GenMonArr00] Generalising monads to arrows (2000) by John Hughes

[NewNotatArr01] A New Notation for Arrows (2001) by Ross Paterson

[AllMonads03] All About Monads (2003) Jeff Newbern

[ArrComp03] Arrows and Computation (2003) by Ross Paterson

[ProgArr05] Programming with Arrows (2005) by John Hughes

[AppProg08] Applicative Programming with Effects (2008) by Conor McBride, Ross Paterson

[HaskExpr00] The Haskell School of Expression (2000) by Paul Hudak

[HaskHiPerf16] Haskell High Performance Programming - Chapter 13: Functional Reactive Programming (2016) by Samuli Thomasson

[ManningFrp16] Functional Reactive Programming (2016) by Stephen Blackheath, Anthony Jones

[GameProgHask15] Game programming in Haskell (2015) by Elise Huard

[LearnGood11] Learn You A Haskell For Great Good! (2011) by Miran Lipovaca

INDEX

T

U

W

Y