Functional Reactive Programming, Refactored

Ivan Perez
University of Nottingham
ixp@cs.nottingham.ac.uk

Manuel Bärenz
University of Bamberg
manuel.baerenz@uni-bamberg.de

Henrik Nilsson
University of Nottingham
nhn@cs.nottingham.ac.uk

Abstract

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 and practical 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 of our new system with very good results.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features – Control Structures

Keywords functional reactive programming, reactive programming, stream programming, monadic streams, Haskell

1. Introduction

Functional Reactive Programming (FRP) [9, 10, 18] is a declarative approach to implementing reactive applications centred around programming with time-varying values (as opposed to operational descriptions of how to react to individual events). There are a very large number of different implementations of FRP frameworks catering for different settings in terms of platforms and languages, specific application needs, and ideas about how to structure such frameworks in the most appropriate way [5]. Particular differences include whether a discrete or hybrid (mixed discrete and continuous) notion of time is adopted, and how to handle I/O and other effects in a way that is both notationally convenient and scalable.

While this diversity makes for a rich and varied FRP landscape, it does raise practical obstacles and concerns for the FRP user in terms of which system to pick, especially if the needs are diverse or can be anticipated to change. Given the underlying conceptual similarity between the various implementations, this raises the question of whether, through appropriate generalisations, it might be possible to provide a single framework that covers a significantly broader set of applications and needs than what any one FRP implementation up until now has done, and that is easily extensible.

This paper proposes such a unifying framework by adopting stream functions parametrised over monads as the central reactive abstraction. We demonstrate that such a minimalistic framework subsumes and exceeds existing FRP frameworks such as Yampa [8, 18] and Reactive Values [21]. Through composition of standard monads like reader, exception, and state, any desirable combination of time domain, dynamic system structure, flexible handling of I/O and more can be obtained in an open-ended manner.

Specifically, the contributions of this paper are:

- We define a minimal Domain-Specific Language of causal Monadic Stream Functions (MSFs), give them precise meaning and analyse the properties they fulfill.
- We explore the use of different monads in MSFs, how they naturally give rise to known reactive constructs, like termination, switching, higher-order, parallelism and sinks, and how to compose effects at a stream level using monad transformers [15].
- We implement three different FRP variants on top of our framework: 1) Arrowized FRP, 2) Classic FRP and 3) Signal/Sink-based reactivity.
- We demonstrate the practical feasibility of our approach by applying it to real-world games and applications.

2. Background

In the interest of making this paper sufficiently self-contained, we summarize the basics of FRP and Yampa in the following. For further details, see earlier papers on FRP and Arrowized FRP (AFRP) as embodied by Yampa [9, 10, 18]. This presentation draws heavily from the summary in [9].

2.1 Functional Reactive Programming

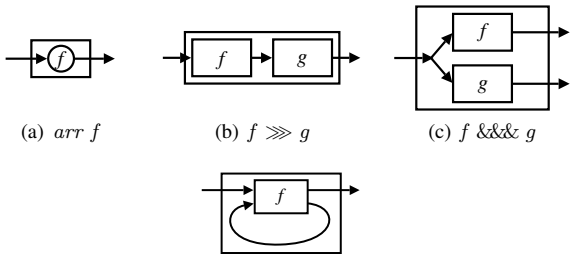
FRP is a programming paradigm to describe hybrid systems that operate on time-varying data. FRP is structured around the concept of *signal*, which conceptually can be seen as a function from time to values of some type:

$$Signal\ \alpha \approx Time \to \alpha$$

$Time$ is continuous, and is represented as a non-negative real number. The type parameter α specifies the type of values carried by the signal. For example, the type of an animation would be $Signal\ Picture$ for some type $Picture$ representing static pictures. Signals can also represent input data, like the position of the mouse on the screen.

Additional constraints are required to make this abstraction executable. First, it is necessary to limit how much of the history of a signal can be examined, to avoid memory leaks. Second, if we are interested in running signals in real time, we require them to be *causal*: they cannot depend on other signals at future times. FRP implementations address these concerns by limiting the ability to sample signals at arbitrary points in time.

The space of FRP frameworks can be subdivided into two main branches, namely Classic FRP [10] and Arrowized FRP [18]. Classic FRP programs are structured around signals or a similar notion representing internal and external time-varying data. In contrast,



(a) $arr\ f$ (b) $f \ggg g$ (c) $f \&\&\& g$

(d) $loop\ f$

Figure 1. Basic signal function combinators.

Arrowized FRP programs are defined using causal functions between signals, or *signal functions*, connected to the outside world only at the top level.

Arrowized FRP renders modular, declarative and efficient code. Pure Arrowized FRP separates IO from the FRP code itself, making the latter referentially transparent *across executions*. This is a crucial debugging aid, since we can run programs multiple times providing the same inputs and they will deliver the exact same outputs. In the following, we turn our attention to Arrowized FRP as embodied by Yampa, and later explain current limitations that our framework addresses.

2.2 Fundamental Concepts

Yampa is based on two concepts: *signals* and *signal functions*. A signal, as we have seen, is a function from time to values of some type, while a *signal function* is a function from $Signal$ to $Signal$:

$$\begin{aligned} &Signal\ \alpha \approx Time \to \alpha \\ &SF\ \alpha\ \beta\ \ \approx Signal\ \alpha \to Signal\ \beta \end{aligned}$$

When a value of type $SF\ \alpha\ \beta$ is applied to an input signal of type $Signal\ \alpha$, it produces an output signal of type $Signal\ \beta$. Signal functions are *first class entities* in Yampa. Signals, however, are not: they only exist indirectly through the notion of signal function.

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

2.3 Composing Signal Functions

Programming in Yampa consists of defining signal functions compositionally using Yampa's library of primitive signal functions and a set of combinators. Yampa's signal functions are an instance of the arrow framework proposed by Hughes [12]. Some central arrow combinators are arr that lifts an ordinary function to a stateless signal function, composition $\ggg$, parallel composition $\&\&\&$, and the fixed point combinator $loop$. In Yampa, they have the following types:

$$\begin{aligned} &arr &&:: (a \to b) \to SF\ a\ b \\ &(\ggg) &&:: SF\ a\ b\ \ \to SF\ b\ c \to SF\ a\ c \\ &(\&\&\&) &&:: SF\ a\ b\ \ \to SF\ a\ c \to SF\ a\ (b, c) \\ &loop &&:: SF\ (a, c)\ (b, c) \to SF\ a\ b \end{aligned}$$

We can think of signals and signal functions using a simple flow chart analogy. Line segments (or "wires") represent signals, with arrowheads indicating the direction of flow. Boxes represent signal functions, with one signal flowing into the box's input port and another signal flowing out of the box's output port. Figure 1 illustrates the aforementioned combinators using this analogy. Through the use of these and related combinators, arbitrary signal function networks can be expressed.

2.4 Time-variant signal functions: integrals and derivatives

Signal Functions must remain causal and leak-free, and therefore Yampa introduces limited ways of depending on past values of other signals. Integrals and derivatives are important for many application domains, like games, multimedia and physical simulations, and they have well-defined continuous-time semantics. Their types in Yampa are as follows:

$$\begin{aligned} &integral &&:: VectorSpace\ v\ s \Rightarrow SF\ v\ v \\ &derivative &&:: VectorSpace\ v\ s \Rightarrow SF\ v\ v \end{aligned}$$

Time-aware primitives like the above make Yampa specifications highly declarative. For example, the position of a falling mass starting from a position $p0$ with initial velocity $v0$ is calculated as:

$$\begin{aligned} &fallingMass :: Double \to Double \to SF\ ()\ Double \\ &fallingMass\ p0\ v0 = arr\ (const\ (-9.8)) \\ &\quad \ggg integral \ggg arr\ (+v0) \\ &\quad \ggg integral \ggg arr\ (+p0) \end{aligned}$$

which resembles well-known physics equations (*i.e. "the position is the integral of the velocity with respect to time"*) even more when expressed using Paterson's Arrow notation [20]:

$$\begin{aligned} &fallingMass :: Double \to Double \to SF\ ()\ Double \\ &fallingMass\ p0\ v0 = \mathbf{proc}\ () \to \mathbf{do} \\ &\quad v \leftarrow arr\ (+v0) \lll integral \multimap (-9.8) \\ &\quad p \leftarrow arr\ (+p0) \lll integral \multimap v \\ &\quad returnA \multimap p \end{aligned}$$

2.5 Events and Event Sources

To model discrete events, we introduce the $Event$ type:

$$\mathbf{data}\ Event\ a = NoEvent \mid Event\ a$$

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

$$tag :: Event\ a \to b \to Event\ b$$

2.6 Switching

The structure of a Yampa system may evolve over time. These structural changes are known as *mode switches*. This is accomplished through a family of *switching* primitives that use events to trigger changes in the connectivity of a system. The simplest such primitive is $switch$:

$$switch :: SF\ a\ (b, Event\ c) \to (c \to SF\ a\ b) \to SF\ a\ b$$

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

Yampa also includes *parallel* switching constructs that maintain *dynamic collections* of signal functions connected in parallel [18]. Signal functions can be added to or removed from such a collection at runtime in response to events, while *preserving* any internal state of all other signal functions in the collection (see Fig. 2). The first class status of signal functions in combination with switching over dynamic collections of signal functions makes Yampa an unusually flexible language for describing hybrid systems.

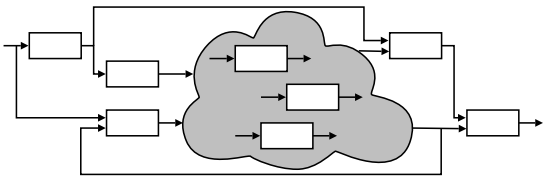


Figure 2. System of interconnected signal functions with varying structure

2.7 Limitations of current AFRP systems

Yampa has a number of limitations, described in the following. Most of these also apply to other current AFRP systems.

Fixed time domain and clock Yampa has a global clock that progresses as driven by an external time-sensing monadic (IO) action. This is a serious limitation, as some games require the use of nested clocks (game clock versus application clock), and others require that time progresses at different speeds or with different precisions for different parts of the game. As a consequence, Continuous Collision Detection in Yampa is very complex.

The time domain in Yampa is fixed to Double, which is not always the most appropriate. Many games run on a discrete clock, while others require a rational clock with arbitrary precision or no clock at all. In such cases, keeping and passing an additional continuous clock becomes an unnecessary nuisance.

I/O bottleneck Yampa's input and output is connected to the external world once at the top level, in the invocation of the function that runs a signal function. This helps keep Signal Functions pure and referentially transparent across executions, but at the expense of having to poll all input data every time and handling more complex data structures in the output.

Explicit wiring Pure implementations of AFRP do not allow communication between signal functions except through explicit input/output signals. All data that a signal function needs must be manually routed down, and outputs manually routed up. In practice we often want to make part of that wiring implicit.

A manifestation of this problem is that it is not possible to debug from within signal functions except by adding explicit output signals carrying debugging information or by using functions like $Debug.Trace.trace$, which output to standard output directly (using $unsafePerformIO$ under the hood). Code that uses $trace$ is not portable, for instance, to platforms like Android, as debug messages must be printed to a special debug log facility.

In the following section we introduce a more fundamental abstraction to specify reactive programs that addresses these concerns, while remaining able to express all the definitions of Yampa.

3. Monadic Stream Functions

Monadic Stream Functions (MSFs) are a minimal abstraction to represent synchronous, effectful, causal functions between streams. MSFs are a generalisation of Yampa's Signal Functions (SFs), with additional combinators to control and stack side effects.

Monadic Stream Functions represent an extensible, minimal core on top of which we can implement other reactive and FRP formulations. In this section we introduce the definitions of MSFs and the basic combinators of our library. In the next sections we will extend our language with combinators to integrate effects using monad stacks, demonstrating with specific monads.

Notation The definitions that follow are simplified for clarity; performance and memory footprint of our implementation are discussed in Section 10. We use the shorter name MSF, instead of $MStreamF$, used in the implementation. The first argument of an MSF is always a monad, and consequently also a functor and an applicative. However, we will usually omit the corresponding monad constraint from type signatures in the following to reduce clutter.

3.1 Definitions

Monadic Stream Functions are defined by a polymorphic type MSF and an evaluation function that applies an MSF to an input and returns, in a monadic context, an output and a continuation:

$$\textbf{newtype}\ MSF\ m\ a\ b$$

$$step :: Monad\ m \Rightarrow MSF\ m\ a\ b \rightarrow a \rightarrow m\ (b, MSF\ m\ a\ b)$$

The type MSF and the $step$ function alone do not represent causal functions on streams. It is only when we successively apply the function to a stream of inputs and consume the side effects that we get the unrolled, streamed version of the function. Causality is given by this progressive application, sample by sample.

For the purposes of exposition we provide functions to express the meaning of applying an MSF to an input and looking only at the output or only at the continuation:

$$\begin{aligned} head_M &:: Monad\ m \Rightarrow MSF\ m\ a\ b \rightarrow a \rightarrow m\ b \\ head_M\ msf\ a &= fst\ \lhd\rhd\ step\ msf\ a \end{aligned}$$

$$\begin{aligned} tail_M &:: Monad\ m \Rightarrow MSF\ m\ a\ b \rightarrow a \rightarrow m\ (MSF\ m\ a\ b) \\ tail_M\ msf\ a &= snd \lhd\rhd step\ msf\ a \end{aligned}$$

We also provide a function to apply an MSF to a *finite list* of inputs, with effects and continuations chained sequentially. This is merely a debugging aid, not how MSFs are actually executed:

$$embed :: Monad\ m \Rightarrow MSF\ m\ a\ b \rightarrow [a] \rightarrow m\ [b]$$

3.2 Lifting

The simplest kind of transformation we can apply to a stream is point-wise to every sample. We provide two functions for this purpose: arr, which produces an output and no side effects, and $liftS$, which applies an effectful function to every input sample.

$$\begin{aligned} arr &:: (a \rightarrow b) \rightarrow MSF\ m\ a\ b \\ liftS &:: (a \rightarrow m\ b) \rightarrow MSF\ m\ a\ b \end{aligned}$$

We describe their meaning by showing that they act point-wise on the head, and that the continuation is the same MSF unchanged:

$$\begin{aligned} head_M\ (arr\ f)\ \ a &\equiv return\ (f\ a) \\ head_M\ (liftS\ f)\ a &\equiv f\ a \end{aligned}$$

$$\begin{aligned} tail_M\ \ (arr\ f)\ \ a &\equiv return\ (arr\ f) \\ tail_M\ \ (liftS\ f)\ a &\equiv f\ a \gg return\ (liftS\ f) \end{aligned}$$

Example One trivial way of using these combinators is the stream function that adds a constant number to the input:

$$\begin{aligned} add\ \ &:: (Num\ n, Monad\ m) \Rightarrow n \rightarrow MSF\ m\ n\ n \\ add\ n0 &= arr\ (\lambda n \rightarrow n + n0) \end{aligned}$$

which we test in a session (in GHC, monad-parametric computations are run in the IO monad and the results printed, if possible):

```
> embed (add 7) [1,2,3]
[8,9,10]
```

3.3 Widening

We can define new MSFs that only affect the first or second components of pairs, passing the other component completely unmodified:

$$\begin{aligned} first\ \ &:: MSF\ m\ a\ b \rightarrow MSF\ m\ (a, c)\ (b, c) \\ second &:: MSF\ m\ b\ c \rightarrow MSF\ m\ (a, b)\ (a, c) \end{aligned}$$

$$\begin{aligned} head_M\ (first\ msf)\ (a, c) &\equiv (\lambda b \rightarrow (b, c)) \lhd\rhd head_M\ msf\ a \\ tail_M\ \ (first\ msf)\ (a, c) &\equiv first\ (tail_M\ msf\ a) \end{aligned}$$

Examples Extending the previous example, we can write:

```
> embed (second (add 7)) [(1, 1), (2, 2), (3, 3)]
[(1, 8), (2, 9), (3, 10)]
```

3.4 Serial and parallel composition

*MSF*s can be composed serially using the combinator (⋙). Composing $f \ggg g$ first applies f to the input producing an output, and applies g to that output, producing a final result. Side effects are sequenced in the same order, which we detail using *step*:

```
(⋙) :: MSF m a b → MSF m b c → MSF m a c
step (f ⋙ g) a ≡ do (b, f') ← (step f) a
                    (c, g') ← (step g) b
                    return (c, f' ⋙ g')
```

MSFs can be composed in parallel with the *Arrow* functions (***), which applies one MSF to each component of a pair, and (&&&), which broadcasts an input to two MSFs applied parallely:

```
(***) :: MSF m a b → MSF m c d → MSF m (a, c) (b, d)
(&&&) :: MSF m a b → MSF m a c → MSF m a (b, c)
```

These combinators are not primitive; their standard definitions are based on combinators defined previously:

```
f *** g = first f ⋙ second g
f &&& g = arr (λx → (x, x)) ⋙ (f *** g)
```

When composed in parallel like above, the effects of the f are produced before those g. This is of prime importance for non-commutative monads. For a discussion, see Section 9.

Example The following example of palindromes demonstrates monadic and composition combinators:

```
testSerial :: MSF IO () ()
testSerial = liftS (λ() → getLine)
           ⋙ (arr id &&& arr reverse) ⋙ liftS print
> embed testSerial (repeat 1 ())
Text
("Text", "txeT")
```

3.5 Depending on the past

We provide a way of keeping state by producing an extra value or *accumulator* accessible in future iterations:

```
feedback :: c → MSF m (a, c) (b, c) → MSF m a b
```

This combinator takes an initial value for the accumulator, runs the MSF, and feeds the new accumulator back for future iterations:

```
step (feedback c msf) a ≡ do
  ((b, c'), msf') ← step msf (a, c)
  return (b, feedback c' msf')
```

Unlike in the definition of *loop* from the *ArrowLoop* type class, the input and the output accumulator are not the same during a simulation step. Conceptually, there is a one step delay in the wire that feeds the output c back as input for the next run.

Example The following calculates the cumulative sum of its inputs, initializing an accumulator and using a feedback loop:

```
sumFrom     :: (Num n, Monad m) ⇒ n → MSF m n n
sumFrom n0 = feedback n0 (arr add2)
  where add2 (n, acc) = let n' = n + acc in (n', n')
```

A counter can now be defined as follows:

```
count :: (Num n, Monad m) ⇒ MSF m () n
count = arr (const 1) ⋙ sumFrom 0
```

3.6 Example: One-dimensional Pong

Before moving on to control structures and monadic MSFs, let us present part of an example of a game of pong in one dimension.

The game state is just the position of the ball at each point. We assume the players sit at fixed positions:

```
type Game = Ball
type Ball = Int

rightPlayerPos = 5
leftPlayerPos  = 0
```

The ball will move in either direction, one step at a time:

```
ballToRight :: Monad m ⇒ MSF m () Ball
ballToRight = count ⋙ arr (λn → leftPlayerPos + n)

ballToLeft  :: Monad m ⇒ MSF m () Ball
ballToLeft  = count ⋙ arr (λn → rightPlayerPos − n)
```

We can detect when the ball should switch direction with:

```
hitRight :: Monad m ⇒ MSF m Ball Bool
hitRight = arr (⩾ rightPlayerPos)

hitLeft  :: Monad m ⇒ MSF m Ball Bool
hitLeft  = arr (⩽ leftPlayerPos)
```

Switching itself will be introduced in Section 4.3, when we talk about Control Flow and Exceptions.

3.7 Mathematical Properties of MSFs

Monadic Stream Functions are Arrows when applied to any monad. A proof of one of the arrow laws, as well as links to proofs of other laws and additional properties, are included in Appendix A.

By constraining the monad we can obtain additional guarantees about MSFs. We have proved that MSFs are *Commutative Arrows* when applied to commutative monads, a result we can exploit for optimisation purposes [16]. For monads that are instances of *MonadFix* we provide a well-behaved MSF instance of *ArrowLoop*. Our library also defines instances that facilitate writing declarative code, like *ArrowChoice* and *ArrowPlus*. MSFs partially applied to inputs are functors and applicative functors.

There are several isomorphisms between Monadic Stream Functions and Monadic Streams defined as $Stream\ m\ a = m\ (a, Stream\ m\ a)$. Some properties are easier to prove or to express when MSFs are seen as streams, something we elaborate on in Section 5. In turn, abstractions defined in terms of streams can be expressed using MSFs, an idea we use in Sections 6 and 7 to obtain implementations of reactive frameworks for free.

4. Monads, modularity and control in MSFs

This section motivates and explores the use of different monads with Monadic Stream Functions. Monads like *Reader* and *State* help modularise and increase the expressiveness of programs. Others like *Maybe*, *Exception* and the list monad give rise to control combinators for termination, higher order and parallelism.

Temporal running functions Monads and monad transfomers [15] 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 $runReaderT :: e \to ReaderT\ e\ m\ a \to m\ a$, obtaining a value in the monad m.

Analogously, MSFs applied to monads have associated *temporal running functions*, which limit effects to part of a reactive network. In this section we introduce MSF lifting/running combinators for concrete monads, exploring how they augment expressivity, help modularise code and give rise to known structural reactive combinators. In Section 5 we present a general way of combining

effects in MSFs. In Section 7 we use these combinators to express FRP abstractions like Arrowized FRP.

4.1 Reader

We now want to make our example MSFs parametric on the player positions. One option is to pass the player position as an argument, as in $ballToRight :: Monad\ m \Rightarrow Int \to MSF\ m\ () \ Ball$. However, this complicates the implementation of every MSF that uses $ballToRight$, which needs to manually pass those settings down the reactive network. We can define such a parametrisation in a modular way using a Reader monad with the game preferences in an environment (we use $ReaderT$ for forward compatibility):

```
type GameEnv = ReaderT GameSettings
data GameSettings = GameSettings
  { leftPlayerPos  :: Int
  , rightPlayerPos :: Int
  }
```

We rewrite the game to pass this environment in the context:

```
ballToRight :: Monad m ⇒ MSF (GameEnv m) () Ball
ballToRight =
  count ⋙ liftS (λn → (n+) <$> asks leftPlayerPos)

hitRight :: Monad m ⇒ MSF (GameEnv m) Ball Bool
hitRight = liftS $ λi → (i >=) <$> asks rightPlayerPos
```

To run a game with a fixed environment we could use $runReaderT :: ReaderT\ r\ m\ a \to r \to m\ a$ as before. We test the expression $testMSF = ballToRight \ggg (arr\ id\ \&\&\&\ hitRight)$ with different settings as follows:

```
> runReaderT (embed testMSF (repeat 5 ()))
             (GameSettings 0 3)
[(1, False), (2, False), (3, True), (4, True), (5, True)]
> runReaderT (embed testMSF (repeat 5 ()))
             (GameSettings 0 2)
[(1, False), (2, True), (3, True), (4, True), (5, True)]
```

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):

```
runReaderS :: MSF (ReaderT r m) a b
           → r
           → MSF m a b
```

Now we can run two games in parallel with different settings:

```
> embed (      runReaderS testMSF (GameSettings 0 3)
        &&& runReaderS testMSF (GameSettings 0 2))
        (repeat 5 ()))
[((1, False), (1, False)), ((2, False), (2, False))
, ((3, False), (3, True)), ((4, True), (4, True))
, ((5, True), (5, True))]
```

We could run the MSF obtaining a new Reader environment from the input signal at every iteration, giving us $runReaderS :: MSF\ (ReaderT\ r\ m)\ a\ b \to MSF\ m\ (r, a)\ b$. In Section 5 we will see that both definitions follow from the type of the run function of the Reader monad, and there is a systematic way of defining various temporal monadic execution functions.

4.2 Writer

We can use a similar approach to introduce monads like $Writer$ or $State$, for instance, to log debug messages from MSFs.

We first extend our environment with a $WriterT$ wrapper:

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

We now modify $ballToRight$ to print a message when the position is past the right player (indicating a goal):

```
ballToRight :: Monad m ⇒ MSF (GameEnv m) () Ball
ballToRight =
  count ⋙ liftS addLeftPlayerPos ⋙ liftS checkHitR
  where checkHitR   :: n → GameEnv m Int
        checkHitR n = do
          rp ← asks rightPlayerPos
          when (rp > n) $ tell ["Ball at " ++ show n]
```

Notice that we have changed the monad and $ballToRight$, but the rest of the game remains unchanged. Having used the transformer $ReaderT$ instead of $Reader$ in the previous step now pays off in the form of added modularity.

Like with the reader monad, we may be interested in consuming the context (for instance, to print accumulated messages and empty the log). We provide the *temporal execution function*:

```
runWriterS :: Monad m
           ⇒ MSF (WriterT r m) a b
           → MSF m a (b, r)
```

We can test this combinator as follows:

```
> embed (runWriterS
          (runReaderS testMSF (GameSettings 0 3)))
        (repeat 5 ()))
[((1, False), []), ((2, False), []), ((3, True), [])
, ((4, True), ["Ball at 4"]), ((5, True), ["Ball at 5"])]
```

Similarly we could have used a $State$ monad to define configurable game settings (for instance, settings that can be adjusted using an options menu, but remain immutable during a game run).

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 $MSF\ Maybe\ a\ b \to a \to Maybe\ (b, MSF\ Maybe\ a\ b)$ indicating that it may produce *no continuation*. $runMaybeS$ outputs $Nothing$ continuously once the internal MSF produces no result. "Recovering" from failure requires an additional continuation:

```
catchM :: Monad m
       ⇒ MSF (MaybeT m) a b
       → MSF m a b
       → MSF m a b
```

We can now make the ball bounce when it hits the right player:

```
ballBounceOnce :: MSF (GameEnv m) () Ball
ballBounceOnce = ballUntilHitRight `catchM` ballLeft

ballUntilRight  :: MSF (MaybeT (GameEnv m)) () Ball
ballUntilRight  = liftST (ballToRight
                  ⋙ (arr id &&& hitRight))
                  ⋙ liftS filterHit
  where
    filterHit (b, c) = MaybeT $ return $
      if c then Nothing else Just b
```

The utility function $liftST$ is defined in Section 5.

We define $ballUntilLeft$ analogously and complete the game:

```
game :: Monad m ⇒ MSF m () Ball
game = ballUntilRight `catchM` ballUntilLeft `catchM` game
```

Let us interpret the game by inserting a list as input stream. The output shows the ball position going up and bouncing back between 10 (the right player's position) and 0 (the left player's position).

```
> embed game $ replicate 23 ()
[1,2,3,4,5,6,7,8,9,10,9,8,7,6,5,4,3,2,1,0,1,2,3]
```

The implementation of switching in our libary is based on a more general monad $ExceptT\ c\ m$, but the key idea is the same.

4.4 Creating and destroying objects with $ListT$

In games it is often necessary to create and destroy "objects". For example, a gun may fire a bullet or a target vanish when hit. Such dynamicity requires specific combinators in other reactive frameworks. In ours, the list monad provides the sought-after behaviour.

A stream function over the list monad can produce zero, one or several output values and continuations. How we continue depends on our interpretation of that list. If we explore all continuations simultaneously, we will be implementing parallel broadcasting.

To produce multiple outputs we provide $zeroA$, which produces no outputs or continuations and $<+>$, which concatenates lists produced by two MSFs in a list monad. (In our library these are available for any instance of $Alternative$ and $MonadPlus$.)

```
zeroA :: Monad m ⇒ MSF (ListT m) a b
(<+>) :: Monad m ⇒ MSF (ListT m) a b
                 → MSF (ListT m) a b
                 → MSF (ListT m) a b
```

We now change the game logic such that, each time the ball starts moving left, it splits into two balls (note the use of $(<+>)$):

```
type GameEnv m = ReaderT GameSettings (ListT m)

ballLeft :: Monad m ⇒ MSF (GameEnv m) () Ball
ballLeft = singleBallLeft <+> singleBallLeft
  where
    singleBallLeft =
      count >>>
        liftS (λn → (λp → p − n) <$> asks rightPlayerPos)
```

We can escape the list monad or the $ListT$ transformer by collecting the outputs from all continuations into a list:

```
runListS :: Monad m ⇒ MSF (ListT m) a b
                    → MSF          m  a [b]
```

Our approach proves to be very modular, and we only need to modify our top function slightly to extract the list effect:

```
mainMSF :: MSF IO () ()
mainMSF =
  runListS   (   runReaderS game (GameSettings 20 17)
             &&& runReaderS game (GameSettings 10 4))
  >>> liftS print
```

Note that the $ReaderT$ layer is inside the $ListT$ layer, and therefore both games are duplicated when either ball starts moving left. Running the above MSF prints the output:

```
[(18, 5)]
[(19, 6)]
[(20, 7)]
[(19, 8), (19, 8)]
[(18, 9), (18, 9)]
[(17, 10), (17, 10)]
[(18, 9), (18, 9), (18, 9), (18, 9)]
...
```

The standard implementation of $ListT$ is only valid for commutative monads. Alternative implementations exist, but the discussion is beyond the scope of this article.

4.5 State

Keeping type signatures parametric, like in the stream function $ballLeft :: Monad\ m \Rightarrow MSF\ (GameEnv\ m)\ ()\ Ball$, renders more reusable definitions, since we can stack more monads on top.

For example, it is easy to introduce a global state variable using the $State$ monad, to implement a counter of the number of rounds that have been played. The counter can be increased with:

```
incOneRound :: Monad m ⇒ StateT Integer m ()
incOneRound = modify (+1)
```

which we use in the game, accounting for the side effect in the type:

```
game :: Monad m
     ⇒ MSF (GameEnv (StateT Integer m)) () Ball
game =     ballToRight `untilM` hitRight
  `catchM` ballToLeft  `untilM` hitLeft
  `catchM` (lift incOneRound `andThen` game)
```

The function $andThen :: Monad\ m \Rightarrow m\ () \rightarrow MSF\ m\ a\ b \rightarrow MSF\ m\ a\ b$ performs the monadic action in the first argument once and immediately carries on processing the MSF in the second argument. The function $lift :: (MonadTrans\ t, Monad\ m) \Rightarrow m\ a \rightarrow t\ m\ a$ from the transformers package lifts the state modification into the $GameEnv$ monad.

When we run the reactive program, we have to pass the initial state, in a similar way to $runStateT :: StateT\ s\ m\ a \rightarrow s \rightarrow m\ (a, s)$, which passes an initial state to a monadic state transformation and extracts the computed value and the final state. The corresponding function for streams has this type:

```
runStateS :: Monad m ⇒ MSF (StateT s m) a     b
          → s        → MSF              m  a (s, b)
```

Using this function in the main loop is a simple change:

```
mainMSF :: MSF IO () ()
mainMSF = runStateS parallelGame 0 >>> liftS print
where
  parallelGame  =  runReaderS game (GameSettings 20 17)
                &&& runReaderS game (GameSettings 10 4)
```

The output of running this MSF, presented in two columns for reasons of space, is:

```
(0, (18, 5))        (1, (19, 8))
(0, (19, 6))        (1, (20, 7))
(0, (20, 7))        (1, (19, 6))
(0, (19, 8))        (1, (18, 5))
(0, (18, 9))        (1, (17, 4))
(0, (17, 10))       (3, (18, 5))
(1, (18, 9))        ...
```

The first value is the counter of total rounds, the other two values are the positions of the ball in the two games, respectively.

We have introduced this change without altering any code that did not use the state variable. In standard Arrowized FRP, we would have had to alter all signal functions and pass the state manually.

5. Monadic lifting/running combinators

In the last section we saw functions to combine MSFs on different monads, all of which conform to one of the following patterns:

- Lifting a purer MSF into an MSF in an impurer monad (e.g. $MSF\ Identity\ a\ b \rightarrow MSF\ IO\ a\ b$).
- Running a more structured computation inside a less structured monad (e.g. $MSF\ (t\ m)\ a\ b \rightarrow MSF\ m\ (I\ a)\ (F\ b)$).

This section explores these transformations and presents a way of thinking about MSFs over different monads and monad stacks.

5.1 Lifting MSFs

Whenever we use *monad transformers* or other effect stacking mechanism, we may be interested in embedding an $MSF\ m\ a\ b$ into a larger $MSF\ (t\ m)\ a\ b$ (where $t\ m$ denotes a monad that encloses the effects of m, such as a transformer t applied to m).

We can lift between any two arbitrary monads, provided we have a monad morphism $(Monad\ m, Monad\ n) \Rightarrow m\ a \to n\ a$.

$$\begin{aligned} &liftLM :: (Monad\ m, Monad\ n) \\ &\quad \Rightarrow (\forall\ a\ .\ m\ a \to n\ a) \\ &\quad \to MSF\ m\ a\ b \to MSF\ n\ a\ b \end{aligned}$$

Users are responsible for providing an actual monad morphism that retains any information in m inside the context in n. For Monad Transformers, with a monad morphism $lift :: m\ a \to t\ m\ a$, we define the convenience lifting function $liftST = liftLM\ lift$.

5.2 Temporal running functions

Monad stacks are a common design pattern. If one introduces a transformer on the stack, one frequently wants to remove it after performing some effects in it, by executing it. An example is the "running function" $runReaderT :: ReaderT\ r\ m\ a \to r \to m\ a$.

In Section 4, we introduced temporal execution functions that remove a transformer from the monad stack *inside* the MSF, such as $runReaderS :: MSF\ (ReaderT\ r\ m)\ a\ b \to r \to MSF\ m\ a\ b$. Similar examples were given for $WriterT$, $MaybeT$, $ListT$ and $StateT$. Essentially, they are implemented by commuting the transformer past MSF and then applying the running function of the transformer. This will be explained in the following.

Monadic Streams and MSFs To simplify the implementation of the temporal running functions, we are going to exploit an isomorphism between Monadic Stream Functions and certain Monadic Streams. MSFs are defined as follows:

$$\begin{aligned} &\mathbf{newtype}\ MSF\ m\ a\ b = MSF \\ &\quad \{step :: a \to m\ (b, MSF\ m\ a\ b)\} \end{aligned}$$

Depending on the monad m, we may have one, none or several outputs and continuations. Moving from a monad like $Maybe$ or $Either\ c$ to another monad requires retaining the possibility of providing no output, or recovering from a *termination* or *exception*.

In turn, Monadic Streams can be defined as:

$$Stream\ m\ a \cong m\ (a, Stream\ m\ a)$$

It is easy to see that our abstraction MSF is isomorphic to a Monadic Stream in a Reader context:

$$MSF\ m\ a\ b \cong Stream\ (ReaderT\ a\ m)\ b$$

Note that $Stream$ is a transformer, with $lift :: m\ a \to Stream\ m\ a$ given by the infinite repetition of the same effect. Because $MSF\ m\ a \cong Stream\ (ReaderT\ a\ m)$, this makes MSF with the second argument a preapplied also a transformer.

Streams are functors and applicatives, but not necessarily monads. As transformers they can still be applied and, with some constraints, commuted, so the lack of a general monad instance does not invalidate our argument.

Implementing temporal running functions Commuting a transformer t past MSF means commuting t past $ReaderT$ and subsequently past $Stream$. We can capture all this in a type class:

$$\begin{aligned} &\mathbf{class}\ MonadTrans\ t \Rightarrow Commutation\ t\ \mathbf{where} \\ &\quad commuteReader :: Monad\ m \\ &\qquad \Rightarrow ReaderT\ r\ (t\ m)\ a \to t\ (ReaderT\ r\ m)\ a \\ &\quad commuteStream :: Monad\ m \\ &\qquad \Rightarrow Stream\ (t\ m)\ a \to t\ (Stream\ m)\ a \\ &\quad preserveMH :: (Monad\ m1, Monad\ m2) \\ &\qquad \Rightarrow (\forall\ a\ .\ m1\ a \to m2\ a) \to t\ m1\ b \to t\ m2\ b \end{aligned}$$

As a technicality, we also have to assume that t preserves the isomorphism $MSF\ m\ a \cong Stream\ (ReaderT\ a\ m)$, but most transformers preserve monad homomorphisms. There is no requirement that the commutations be isomorphisms, and in cases like $ListT$ and $MaybeT$ a change of effects is actually desired.

The above typeclass gives rise to a commutation function:

$$\begin{aligned} &commute :: Commutation\ t \\ &\quad \Rightarrow MSF\ (t\ m)\ a\ b \to t\ (MSF\ m\ a)\ b \end{aligned}$$

Defining a temporal execution function is now as simple as defining an instance of the $Commutation$ type class and composing $commute$ with any running function. To give a few examples:

$$\begin{aligned} &runReaderS \equiv runReaderT \circ commute \\ &runWriterS \equiv runWriterT \circ commute \\ &runStateS \equiv runStateT \circ commute \\ &catchM\ msf\ handler \equiv \\ &\quad fromMaybe \Diamond handler \Diamond runMaybeT\ (commute\ msf) \end{aligned}$$

Transformers commuting with $ReaderT$ are abundant, and our library defines $Commutation$ instances for common transformers.

Running with a changing input Other execution functions are also useful in practice. For example, we may be interested in obtaining an $Reader$ context at every input sample, requiring:

$$runReaderS_- :: MSF\ (ReaderT\ r\ m)\ a\ b \to MSF\ m\ (r, a)\ b$$

This kind of definition is trivial once we observe that Reader commutes with itself, and that the following are isomorphic:

$$ReaderT\ r1\ (ReaderT\ r2\ m)\ a \equiv ReaderT\ (r1, r2)\ m\ a$$

and therefore:

$$\begin{aligned} &MSF\ (ReaderT\ r\ m)\ a\ b \\ &\equiv \{\ \textit{MSFs as Streams}\ \} \\ &\quad MStream\ (ReaderT\ a\ (ReaderT\ r\ m))\ b \\ &\equiv \{\ \textit{Commutativity of reader}\ \} \\ &\quad MStream\ (ReaderT\ r\ (ReaderT\ a\ m))\ b \\ &\equiv \{\ \textit{Reader r1 (Reader r2 a) == Reader (r1,r2) a}\ \} \\ &\quad MStream\ (ReaderT\ (r, a)\ m)\ b \\ &\equiv \{\ \textit{Streams as MSFs}\ \} \\ &\quad MSF\ m\ (r, a)\ b \end{aligned}$$

6. Reactive Programming and Monadic Streams

Reactive Programming is a programming paradigm organised around information producers and consumers, or *streams* and *sinks*. In this section we present definitions of Streams and Sinks based on Monadic Stream Functions. Some properties of monadic streams also apply to stream functions or are easier to prove in that setting. The existing research on streams and causal stream functions makes establishing this relation useful in its own right.

We present definitions of Streams and Sinks based on Monadic Stream Functions, and demonstrate how to do Reactive Programming using the ideas introduced in the previous section. We also present an extension suitable for event-driven settings like GUIs. In Section 7.2 we use similar concepts to implement Classic FRP.

6.1 Streams and sinks

Monadic Stream Functions that do not depend on their input model Monadic Streams. We can capture that idea with:

$$\mathbf{type}\ Stream\ m\ b = MSF\ m\ ()\ b$$

Disregarding bottoms and applying unit as the only possible argument, the above expands to $Stream\ m\ b \cong m\ (b, Stream\ m\ b)$, and $Stream\ Identity$ is isomorphic to standard infinite streams.

If streams can be seen as MSF that do not depend on their inputs, *sinks* can be seen as MSF do not produce any output:

```
type Sink m b = MSF m b ()
```

These represent dead-ends of information flow, useful when we are only interested in the side effects.

Monadic Streams, as defined above, are *Functors* and *Applicatives*. Sinks, in turn, are *contravariant functors*:

```
instance Contravariant (Sink m) where
  contramap :: (a → b) → Sink m b → Sink m a
  contramap f msf = arr f >>> msf
```

Examples These abstractions allow us to write more declarative code. For instance, given $mouseX :: Stream\ IO\ Int$, represeting the changing X coordinate of the mouse position, we can write:

```
mirroredMouseX :: Stream IO Int
mirroredMouseX = (−) <$> 1024 <*> mouseX
```

We can sometimes simplify code further. For example, using the previous instances we can give a Num instance for Num-carrying $Stream$s, and overload the standard numeric operators:

```
mirroredMouseX :: Stream IO Int
mirroredMouseX = 1024 − mouseX
```

Note that, in this new definition, 1024 has type $Stream\ IO\ Int$.

Streams and sinks are MSFs, so we can use MSF combinators to transform them and connect them. The following reactive program chains a stream and a sink to print the mouse position:

```
reactiveProgram = mouseX >>> arr show >>> printSink

printSink :: Sink IO String
printSink = liftS putStrLn
```

6.2 Reactive Values

External mutable entitities can be regarded as both sources and sinks. For instance, in programs with Graphical User Interfaces, text boxes can be seen as String-carrying streams and sinks.

Formulations like Reactive Values [21] and Wormholes [25] are built around external mutable entitites. Introducing this abstraction as a first-class concept minimizes duplication in the presence of multiple circular dependencies, omnipresent in GUI programs.

We could represent reactive entities as pairs of a Stream and a Sink on the same monad and type. However, connecting reactive entities in a circular way can render incorrect results. For example, imagine that we try to synchronize two text boxes, each seen as a pair of a Stream and a Sink, connected to the actual GUI element:

```
(   (txtField1Stream >>> txtField2Sink)
&&& (txtField2Stream >>> txtField1Sink))
```

Because expressions are evaluated in order, the second text field would always be updated with the text from the first text box, regardless of which one originally changed.

To address this problem, we need push-based evaluation, for which we can follow the original design of the Reactive Values and tuple a stream, a sink, and a change event handler installer that executes an arbitrary monadic action whenever the value changes:

```
type ReactiveValueRO m a  = (Stream m a, m () → m ())
type ReactiveValueWO m a = Sink m a
type ReactiveValueRW m a =
  (Stream m a, Sink m a, m () → m ())
```

The only other requirement is to modify $reactimate$, to run only one step of the MSF each time a source stream changes:

```
pushReactimate :: MSF IO () () → IO (IO ())
pushReactimate msf = do
  msfRef ← newIORef msf
  return $ do
    msf' ← readIORef msfRef
    msf'' ← tail_M msf' ()
    writeIORef msfRef msf''
```

With this interface, push-based connections need to be specified at monadic level. Both directional and bi-directional connections can be expressed declaratively using rule-binding combinators:

```
(=:=) :: ReactiveValueRW IO a
      → ReactiveValueRW IO a → IO ()
(sg1, sk1, h1) =:= (sg2, sk2, h2) = do
  (sg1, h1) =:> sk2
  (sg2, h2) =:> sk1

(=:>) :: ReactiveValueRO IO a
      → ReactiveValueWO IO a → IO ()
(sg, h) =:> sk = h =<< pushReactimate (sg >>> sk)
```

Note that, unlike the $(>>>)$ combinator, $(=:>)$ does not actually update the right hand side if the left hand side has not changed. We can now express the example above as:

```
do txtField1Stream =:> txtField2Sink
   txtField2Stream =:> txtField1Sink
```

Expressing RVs in terms of MSFs simplifies the implementation of supporting libraries, as we can rely on the MSF combinators to implement transformations on RVs. At a conceptual level, this also shows that monadic stream functions are a suitable abstraction to reason about discrete bidirectional connections between reactive entities, and can aid in the design of combinators for compositional, bi-directional reactive rules with well-defined semantics.

7. Extensible Functional Reactive Programming

Functional Reactive Programming (FRP) [9, 10, 18] is a paradigm to describe systems that change over time. Time in FRP is explicit and *conceptually continuous*. FRP is structured around a concept of signals, which represent time-varying values:

```
type Signal a  ≃  Time → a
type Time      ≃  ℝ+
```

While this conceptual definition enables giving FRP denotational semantics, execution is still carried out by sampling signals progressively. The ideal semantics are approximated more precisely as the sampling frequency increases [24].

Using Monadic Stream Functions we can add time information to the monadic context. In this section we use this approach to implement Arrowized FRP [9, 18] and Classic FRP [10]. We show that our alternatives remain flexible enough to address the concerns expressed in the introduction. We limit our discourse to the programming abstractions; performance and comparisons to other FRP variants are discussed in Sections 8 and 9.

7.1 Extensible Arrowized FRP

Arrowized FRP (AFRP) is an FRP formulation structured around the concept of signal functions. Signals, in AFRP, are not first class citizens. Conceptually:

```
type SF a b  ≃  Signal a → Signal b
```

Multiple AFRP implementations exist. In the following we implement Yampa [8, 18], used to program multimedia and games. We demonstrate that our implementation can easily adress some of the limitations of Yampa by implementing systems with multiple clocks and a form of Continuous Colision Detection.

7.1.1 Core definitions

Signal Functions (SFs) are executed by successively feeding in a stream of input samples, tagged with their time of occurrence. The time always moves forward, and is expressed as the strictly positive time passed since the occurrence of the previous sample.

Leaving optimisations aside, Yampa's running SFs are defined as $\mathbf{type}\ SF'\ a\ b = DTime \to a \to (b, SF'\ a\ b)$, which we can realise by passing time deltas in a Reader monad environment:

```
type SF a b    = MSF ClockInfo a b
type ClockInfo = Reader DTime
type DTime     = Double
```

Most of Yampa's core primitives [23], like arr, $(\ggg)$ or $switch$, are time-invariant, and the definitions in previous sections implement the same behaviour as Yampa's. We only need to add:

```
integral :: VectorSpace a s ⇒ SF a a
integral = eulerSteps ⋙ sumFrom zeroVector
  where eulerSteps = liftS $ λx → asks (x*)
```

7.1.2 Reactimating Signal Functions

The second part of our Yampa replacement is a *reactimation* or simulation function. The signature of Yampa's top-level simulation function demonstrates the bottleneck effect mentioned earlier[1]:

```
reactimate :: IO (DTime, a) → (b → IO ()) → SF a b
           → IO ()
```

The first argument gathers inputs and delta times, the second consumes outputs, producing side effects. Our MSF $reactimate$ function has a simpler type signature, $MSF\ m\ ()\ ()\ \to\ m\ ()$, meaning that all I/O is done inside and all we care about, in the end, are the effects. We implement Yampa's $reactimate$ as follows:

```
reactimate sense actuate sf =
  MSF.reactimate $ senseSF ⋙ sfIO ⋙ actuateSF
where
  sfIO :: MSF IO a b
  sfIO = liftLM (return ∘ runIdentity) (runReaderS_ sf)

  senseSF :: MSF IO () (DTime, a)
  senseSF = liftS (λ() → sense)

  actuateSF :: MSF IO b ()
  actuateSF = liftS actuate
```

There are two notable aspects reflected by $sfIO$. First, we use $runReaderS\_ :: MSF\ (ReaderT\ s\ m)\ (s, a)\ b \to MSF\ m\ a\ b$ to provide the time deltas in a reader environment for the Yampa monad. Second, after extracting the $ReaderT$ layer, we use $liftLM$ to lift a computation in the $Identity$ monad into the IO monad. The top-level MSF runs in the IO monad and effects are consumed, and presented to the user, progressively.

We have also verified our implementation with multiple Yampa games. This will be discussed in Section 8.

7.1.3 Time, Clocks and Continuous Collision Detection

A limitation of pure Arrowized FRP is that the clock is controlled externally and globally for the whole simulation. In the following we show that our new implementation can accommodate multiple clocks and enable some forms of Continuous Collision Detection.

Using lifting MSF combinators we can embed SFs running on one clock inside others running on different clocks or time domains. For instance, the following runs MSFs at different speeds:

```
game = twiceAsFast fallingBall &&& fallingBall

twiceAsFast :: MSF (ReaderT DTime m) a b
            → MSF (ReaderT DTime m) a b
twiceAsFast = liftLM (withReaderT (*2))

fallingBall = fallingMass 100 10   -- defined in section 2
  -- From ReaderT transformer class
withReaderT :: (r' → r) → ReaderT r' a → ReaderT r a
```

A useful variation would be to sample an MSF with a fixed sampling period, regardless of the external clock.

The function $twiceAsFast$ above runs both clocks with the same precision: both "tick" in synchrony, even if one advances twice as much. Using the low-level API of our library, we can make one clock actually tick twice as many times per sampling period:

```
twiceAsFast msf = MSF $ λa → do
  dt ← ask
  (_, msf1) ← runReaderT (step msf  a) (dt / 2)
  (b, msf2) ← runReaderT (step msf1 a) (dt / 2)
  return (b, twiceAsFast msf2)
```

The introduction of subsampling mechanisms like the latter needs to be addressed with care. Arbitrarily fine subsampling can lead to inherent memory leaks.

Continuous Collision Detection Physics simulations in Yampa are implemented by looking at overlaps between objects, at the sampling time. This leads *tunnelling* or *bullet-through-paper* effects, in which moving objects can pass through other objects if the simulation is not executed at a time when they actually overlap.

With MSFs we can implement some forms of Continuous Collision Detection (CCD), by letting signal functions tell the top level simulation when the next sampling should occur. The top-level $reactimate$ can then decide whether the real application time delta should be used for the simulation, or whether several steps and higher precision are needed for a particular step, as seen before.

We implement a form of CCD with a Writer monad. We use the monoid of time deltas with the minimum and infinity as identity:

```
data FutureTime = AtTime DTime
                | Infinity
  deriving (Eq, Ord)

instance Monoid FutureTime where
  mempty  = Infinity
  mappend = min
```

Signal Functions that try to change the $Writer$ state with a requested sampling time will only do so when such time is smaller than one currently in the writer state. At the end of each simulation iteration, the log will contain the closest suggested sampling time.

Using this approach, we can define a bouncing ball that never goes below the floor. We do so by calculating the expected time of impact ($nextT$) and by recording that time in the Writer context ($liftS \circ lift \circ tell$). For clarity, we use Paterson's arrow notation [20]:

```
type CCDGameMonad m =
  ReaderT DTime (WriterT FutureTime m)

bouncingBall :: Double → Double
             → MSF CCDGameMonad () Double
bouncingBall p0 v0 = switch
  (proc () → do
    (p, v)  ←  fallingBall p0 v0 -≺ ()
    bounce  ←  edge              -≺ (p ⩽ 0 ∧ v < 0)
    let nextT = if v < 0
                  then sqrt (2 * p / gravity)   -- Down
                  else 2 * v0 / gravity         -- Up
    liftS (lift (tell nextT)) -≺ ()   -- Put "next" time
    returnA -≺ ((p, v), bounce `tag` (p, v)))
  (λ(p, v) → bouncingBall p (-v))
```

7.2 Classic FRP

Classic FRP lets users define time-varying values, or *signals*, using combinators and applying functions to other signals. Signals may represent external information sources (e.g. mouse position).

[1] Yampa's $reactimate$ has a more complex signature for reasons beyond the scope of this paper that do not invalidate our claims. Our library follows Yampa's specification.

With a *Reader* monad with time deltas in the environment, like in Arrowized FRP, *Stream*s can model FRP Signals. To allow for external sources, we nest IO with the monad, obtaining:

```
type Signal a = Stream (ReaderT DTime IO) a
```

A simple simulation of a ball moving around the mouse position could be written in Classic FRP style as follows:

```
ballInCirclesAroundMouse :: Signal (Int, Int)
ballInCirclesAroundMouse =
  addPair <$> mousePos <*> ballInCircles

ballInCircles :: Signal (Double, Double)
ballInCircles = (λx → (rad * cos x, rad * sin x)) <$> time
  where rad = 45   -- radius in pixels

mousePos :: Signal (Int, Int)
mousePos = liftS (λ() → lift getMousePos)

  -- Predefined
addPair      :: Num a ⇒ (a, a) → (a, a) → (a, a)
getMousePos :: IO (Double, Double)
time         :: Signal Time
```

With non-commutative monads like IO, additional measures must be taken to ensure referential transparency at the same conceptual time. If several signals depend on, for example, *mousePos*, the mouse position might be sampled twice, with different results at the same conceptual time. This can be addressed with a monad that caches results and enables garbage collection [19].

7.3 Limiting the impact of IO

MSFs over the IO monad limit out ability to apply equational reasoning and to debug programs. In contrast, the separation between IO and Signal Functions in pure Arrowized FRP makes it possible to record all inputs and replicate program behaviour precisely.

Monads are "contagious", and MSFs that do IO, directly or indirectly, must say so in their signature. We consider this a strength of our framework, as it allows, but discourages, impurity.

Additionally, we can limit side effects with a monad without support to lift arbitrary IO actions and a safer API. Monadic requirements can be abstracted into a type class, as follows:

```
class Monad m ⇒ GameMonad m where
  getMousePos :: m (Int, Int)
```

Signatures require additional type constraints, but implementations remain unchanged. We can now use a pure instance for debugging and the *IO* monad during normal execution. This makes our framework more flexible than pure Arrowized FRP, while maintaining its benefits in terms of testing and referential transparency.

8. Evaluation

We have implemented the ideas presented in this paper in two libraries: *Dunai* [2], a core implementation of Monadic Stream Functions, and *Bear River* [3], a Yampa replacement built on top of Dunai by using the ideas described in Section 7. Bear River defines Signal Functions parameterised over a monad, making this implementation more versatile than Yampa. In order to remain backwards compatible, we provide an *FRP.Yampa* module that reexports BearRiver and declares *SF a b = BearRiver.SF Identity a b*.

We have tested our library by compiling and executing existing games implemented in Yampa. We discuss two games: the commercial game Magic Cookies! and the open-source Haskanoid.

Magic Cookies! [3] is an FRP board game in which the user needs to turn off all the lights on a board, following a rule: touching any position toggles surrounding positions in a cross-like shape. The game uses SDL2 for multimedia, and is compiled via the unofficial GHC Android backend. This game is available from the official store, Google Play for Android.

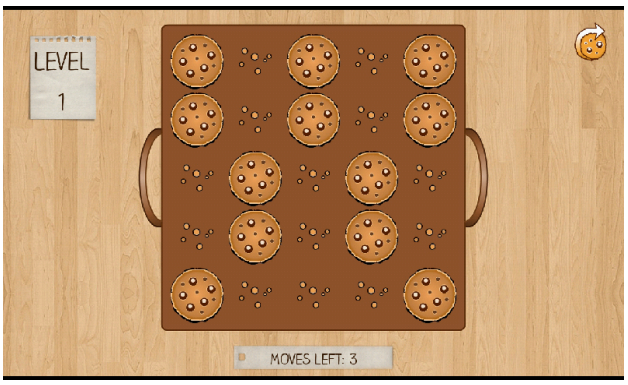


Figure 3. Screenshot of Magic Cookies! compiled BearRiver.

Haskanoid [2] is a clone of the game Arkanoid, written in Haskell using Yampa, SDL multimedia, and supporting input devices like Kinect and Wii remotes. This game implements a simple collision system with convex shapes like rectangles and circles.

Figure 4. Screenshot of the open-source Yampa game Haskanoid.

Our Yampa replacement *Bear River* can execute both of these programs in constant memory (Figure 5). In Haskanoid, memory consumption decreases as the blocks are removed during gameplay. Enabling compiler optimisations, Magic Cookies runs in 325KB of constant memory with BearRiver, while with Yampa it consumes 300KB. Haskanoid consumes a maximum of 2.2MB of memory with BearRiver, while with Yampa it requires approximately 2MB.

9. Related work

Our abstraction is a generalisation of Yampa's signal functions. One central difference is that Yampa has a fixed, notionally continuous time domain, with (broadly) all active signal functions sensing the same time flow. This is implemented by passing the time delta since the previous step to each signal function. In our framework, we can do the same using the Reader monad, but the time domain is no longer fixed, and it is easy to arrange for nested subsystems with different time domains and other variations as needed. Our

[2] `http://hackage.haskell.org/package/dunai`

[3] `http://hackage.haskell.org/package/bearriver`

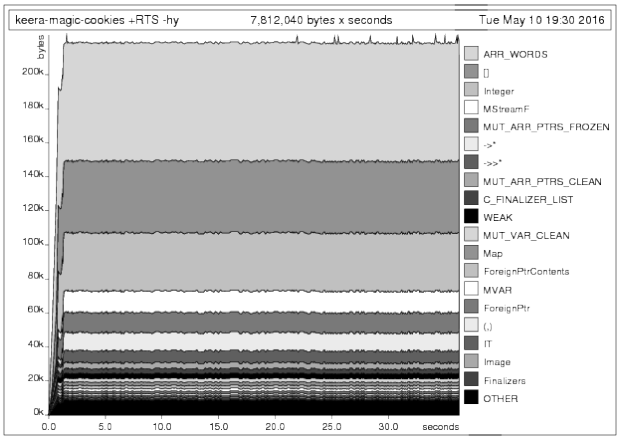


Figure 5. Heap profile of Magic Cookies! running with our library.

proposal also eliminates some of the bottlenecks of Yampa thanks to the flexibility offered by the monadic parametrisation.

We have tested several Yampa games with our library, through a mediating layer as outlined above, including games like Haskanoid [2] and the commercial game Magic Cookies! for Android [3]. Our tests show that our replacement library runs these games in constant memory, consuming about ten per cent more memory than Yampa. Yampa is optimised exploiting algebraic identities like the arrow laws [12, 17], suggesting that we could deliver comparative performance with similar optimisations.

Netwire, inspired by Yampa, is an FRP framework parametric over the time domain and a monad, in which signals can be inhibited (producing no output) and with a switch that maintains the global clock (unlike Yampa's switch, which resets time). We can implement equivalent behaviour using $EitherT$ to inhibit signals, and adding the total global time to the context in a $ReaderT$ layer.

Causal Commutative Arrows (CCA) [16] defines an abstraction of *causal stream transformer*, isomorphic to MSFs over the $Identity$ monad. Programs written in terms of causal stream transformers, or any CCA, can be optimised by orders of magnitude in speed. MSFs are valid CCAs for commutative monads [6], making it possible in principle to apply similar optimisations.

Multiple FRP libraries implement push or push/pull evaluation, such as Elerea, Sodium and reactive banana, all structured in terms of signals and/or event streams. These libraries are optimised to run efficiently with minimal data propagation, often using weak references and an IO monad behind the scenes. Additional measures must be taken to guarantee referential transparency and efficient garbage collection in the presence of non commutative monads like IO. These frameworks also include combinators to connect signals or event streams to external sources and sinks. This is a pragmatic choice, but the drawback is that the IO connection is not shown explicitly in the type. We consider the manifest reflection of all effects in the types a strength of our framework. For example, for pure MSFs (no general IO), we can perform reproducible tests and obtain more guarantees than in IO-based FRP implementations by connecting with automated testing tools like QuickCheck [7].

Monadic FRP is, despite its name, most similar to Yampa's Tasks (which we can model using MSFs using $EitherT$ transformer). Monadic FRP is not parametric over the monad, which makes our framework more general in that respect.

UniTi [22] is a hybrid-system simulation framework with local clocks in which signal functions can output debugging information. Our work can accommodate local clocks using a reader monad, as well as debugging facilities. Additionally, our framework enforces causality and can enforce temporal consistency, whereas UniTi can provide inconsistent results for past values of signals.

There are also stream-based programming libraries that share notions with our framework. Iteratees [14] are stream transformers parametrised over a monad and oriented towards memory-efficient processing of data streams gathered from network and file sources. The application domain is thus rather different from ours, and this is reflected in an asynchronous API with a somewhat imperative feel, centred around reading and writing individual stream elements. Pipes [4] describe interconnected data processors and their coordination in an asynchronous setting. At a higher-level, pipes are expressed in terms of constructs like $Producer$ and $Pipe$, conceptually similar to MSFs with an $EitherT$ monad transformer. three cases. Internally, pipes are based on a $Proxy$ type which is parametrised dually on the input and the output, making pipes bi-directional. Our framework can be used to describe bi-directional connections with an adapted reactimation function (Section 6).

Our approach can be used to implement discrete reactive programming like Reactive Values [21]. Reactive Values minimises forward propagation with push-based evaluation and change detection. This can be implemented in MSFs with an adapted push-based reactimate function. Expressing Reactive Values in terms of MSFs is also a way to define and study the semantics of the former.

Similarly, Wormholes [25] pairs whiteholes (monadic streams) with blackholes (sinks) to represent external resources. Reads and writes are sorted to guarantee referential transparency and commutativity. This could be achieved with custom monads in our setting. While they use a Monadic Stream Function representation to introduce IO, their types carry resource information, imposing additional constraints that require a customized Arrow type-class.

Hughes uses the same representation as our MSFs in a circuit simulator [13]. The approach is stream based, centred around data processors that actively wait for input events. Hughes does not explore the use of different pure monads or how they impact modularity. However, the work shows how to leverage active waits on the monad, something which is also applicable in our setting.

Finally, a representation similar to our MSFs was mentioned in [1], where the author briefly suggests using different monads to achieve different effects. To the best of our knowledge, that blog post did not spawn further work exploring such a possibility.

10. Conclusions and future work

We started this paper by observing how fractured the FRP landscape has become for a variety of reasons, including domain-specific aspects, the inflexibility of current FRP frameworks, and differences in opinions on how to best structure FRP systems. Indeed, today, it is not easy to pinpoint what FRP is. While diversity brings many benefits, there are also significant associated costs in terms of duplication of effort and end-user uncertainty about what specific FRP system or even approach to use, potentially hampering the use of FRP as such.

To address these concerns, this paper proposed to refactor FRP into a minimal core, capturing what arguably is the essence of FRP, but parametrised over a monad to make it open ended. Realisation of domain- and application specific features is then just a matter of picking an appropriate monad. We evaluated our approach both from a practical and theoretical perspective. For the practical evaluation, we reimplemented an existing FRP system, Yampa, using our minimal core, demonstrating good time and space performance over a selection of medium-sized open-source and commercial games, including Haskanoid and Magic Cookies!, on both standard mobile and desktop platforms. Additionally, we showed how a range of other features from other systems and proposed extensions from the FRP literature can be expressed. For the theoretical side,

we showed that our framework has an appropriate mathematical structure, such as satisfying the expected laws, preserving commutativity of the monad, and interoperating with monad morphisms.

So far, our implementation is unoptimised. We expect optimization techniques like those used in Yampa [8, 18] to carry over, bringing similar performance gains. We are further working on support for change propagation to avoid redundant computation when signals are unchanging. We anticipate that this will result in performance on par with push-based FRP implementations.

An advantage of making effects manifest, as when structuring code using arrows or monads, is that it becomes a lot easier to re-execute code. This in turn opens up for sophisticated approaches to automated testing. At present, we are exploiting this for Yampa to test commercial games and debug traces gathered from users' devices using QuickCheck [7] in combination with a custom language of temporal predicates. We are planning to provide similar capabilities to the setting of MSFs. Even when IO is needed, one could provide the needed IO capabilities through a wrapper that is responsible for carrying out the necessary logging and bookkeeping to also make such code re-executable.

Finally, we are currently exploring the addition of clock information to MSFs to express the speed at which *asynchronous* MSFs produce and consume data, and how to coordinate them precisely.

A. MSFs, Arrows and Arrow laws

The arrow laws and other properties hold for Monad Stream Functions [6]. To prove them, we model Haskell types as *complete partial orders* (CPOs), functions as continuous maps, and use *fixpoint induction* [11]. Each fundamental category and arrow combinator like id, $\ggg$, arr and *first* is a fixpoint of a corresponding *recursion function*. Below we include a shortened proof of one arrow law.

Definitions

For conciseness, we will leave out the value constructors, so the type of MSF is simply $MSF\ m\ a\ b = a \to m\ (b, MSF\ m\ a\ b)$.

$$
\begin{aligned}
&arr\ \ = fix\ arrRec\\
&arrRec\ rec\ f\ a = return\ (f\ a, rec\ f)\\
&(\ggg) = fix\ compRec\\
&compRec\ rec\ f\ g\ a = \mathbf{do}\ (b, f') \leftarrow f\ a\\
&\qquad\qquad\qquad\qquad\quad (c, g') \leftarrow g\ b\\
&\qquad\qquad\qquad\qquad\quad return\ (c, rec\ f'\ g')
\end{aligned}
$$

Proof: $arr\ (f \ggg g) \equiv arr\ f \ggg arr\ g$

The application of fixpoint induction means that if P is a predicate on values of some type a, and $f : a \to a$, and P holds for $\bot$, and for every value x we can infer $P\ x \Rightarrow P\ (f\ x)$, then P holds for $fix\ f\ =_{\text{def}}\ f\ (fix\ f)$. We define the predicate P as:

$$P\ (x, y)\ =_{\text{def}}\ \forall\ f\ g\ .\ (x\ f)\ `y`\ (x\ g) \equiv x\ (g \circ f)$$

We proceed as follows:

$$
\begin{aligned}
&P\ (\bot, \bot)\\
&\iff \{\ \textit{Definition of } P\ \}\\
&\qquad \forall\ f\ g\ .\ (\bot\ f)\ `\bot`\ (\bot\ g) \equiv \bot\ (g \circ f)\\
&\iff \{\ \textit{Application to } \bot\ \}\\
&\qquad \bot \equiv \bot\\
&\iff true\\
&P\ (arrRec\ x, compRec\ y)\\
&\iff \{\ \textit{Definition of } P\ \}\\
&\qquad \forall\ f\ g\ .\ (arrRec\ x\ f)\ `compRec\ y`\ (arrRec\ x\ g)\\
&\qquad\qquad \equiv arrRec\ x\ (g \circ f)\\
&\iff \{\ \textit{Definition of } compRec \textit{ and } arrRec \textit{ and } \beta\textit{-Reduction}\ \}\\
&\qquad \forall\ f\ g\ .\ lhs \equiv \lambda a \to return\ ((g \circ f)\ a, x\ (g \circ f))\ \mathbf{where}\\
&\qquad\quad lhs = \lambda a \to \mathbf{do}\\
&\qquad\qquad\quad (b, msf1') \leftarrow return\ (f\ a, x\ f)\\
&\qquad\qquad\quad (c, msf2') \leftarrow return\ (g\ b, x\ g)\\
&\qquad\qquad\quad return\ (c, msf1'\ `y`\ msf2')\\
&\iff \{\ \textit{Monad laws, function composition}\ \}\\
&\qquad \forall\ f\ g\ .\ \lambda a \to return\ (g\ (f\ a), x\ f\ `y`\ x\ g)\\
&\qquad\qquad\quad \equiv \lambda a \to return\ (g\ (f\ a), x\ (g \circ f))\\
&\Leftarrow\quad \forall\ f\ g\ .\ x\ f\ `y`\ x\ g \equiv x\ (g \circ f)\\
&\iff P\ (x, y)
\end{aligned}
$$

References

[1] Reification of time in FRP. `http://pchiusano.blogspot.co.uk/2010/07/reification-of-time-in-frp-is.html`.

[2] Haskanoid. `github.com/ivanperez-keera/haskanoid`.

[3] Magic Cookies! `https://play.google.com/store/apps/details?id=uk.co.keera.games.magiccookies`.

[4] Pipes. `https://hackage.haskell.org/package/pipes`.

[5] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter. A survey on reactive programming. *ACM Computing Surveys (CSUR)*, 45(4):52, 2013.

[6] M. Bärenz, I. Perez, and H. Nilsson. On the Mathematical Properties of Monadic Stream Functions. `http://cs.nott.ac.uk/~ixp/papers/msfmathprops.pdf`, 2016. (Unpublished).

[7] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM Sigplan Notices*, 46:53–64, 2011.

[8] A. Courtney and C. Elliott. Genuinely Functional User Interfaces. In *Proceedings of the 2001 Haskell Workshop*, pages 41–69, Sept. 2001.

[9] A. Courtney, H. Nilsson, and J. Peterson. The Yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 7–18, 2003.

[10] C. Elliott and P. Hudak. Functional Reactive Animation. In *ACM SIGPLAN Notices*, volume 32(8), pages 263–273, 1997.

[11] J. Gibbons and G. Hutton. Proof Methods for Corecursive Programs. *Fundamenta Informaticae Special Issue on Program Transformation*, 66(4):353–366, April-May 2005.

[12] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1):67 – 111, 2000.

[13] J. Hughes. Programming with Arrows. In V. Vene and T. Uustalu, editors, *Proceedings of the 5th International School on Advanced Functional Programming*, volume 3622, pages 73–129, 2005.

[14] O. Kiselyov. Iteratees. In *Functional and Logic Programming: 11th International Symposium, FLOPS 2012*, pages 166–181, 2012.

[15] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd Symposium on Principles of programming languages*, pages 333–343, 1995.

[16] H. Liu, E. Cheng, and P. Hudak. Causal commutative arrows and their optimization. In *ACM Sigplan Notices*, volume 44, pages 35–46, 2009.

[17] H. Nilsson. Dynamic Optimization for FRP using Generalized Algebraic Data Types. *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming*, pages 54–65, 2005.

[18] H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 51–64, 2002.

[19] G. Patai. Efficient and compositional higher-order streams. In *Functional and Constraint Logic Programming*, pages 137–154. 2010.

[20] R. Paterson. A new notation for arrows. In *International Conference on Functional Programming*, pages 229–240, Sept. 2001.

[21] I. Perez and H. Nilsson. Bridging the gui gap with reactive values and relations. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, pages 47–58, 2015.

[22] K. C. Rovers. *Functional model-based design of embedded systems with UniTi*. PhD thesis, 2011.

[23] N. Sculthorpe. *Towards Safe and Efficient Functional Reactive Programming*. PhD thesis, School of Computer Science, University of Nottingham, July 2011.

[24] Z. Wan and P. Hudak. Functional Reactive Programming from First Principles. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–252, 2000.

[25] D. Winograd-Cort and P. Hudak. Wormholes: Introducing Effects to FRP. In *Haskell Symposium*. ACM, September 2012.