August 2020

Asynchronous Monad for Reactive IoT Programming

Adam Bertrand Berger
University of Wisconsin-Milwaukee

ASYNCHRONOUS MONAD FOR REACTIVE IOT PROGRAMMING

by

Adam Bertrand Berger

A Thesis Submitted in

Partial Fulfillment of the

Requirements for the Degree of

Master of Science

in Computer Science

at

The University of Wisconsin-Milwaukee

August 2020

ABSTRACT

ASYNCHRONOUS MONAD FOR REACTIVE IOT PROGRAMMING

by

Adam Bertrand Berger

The University of Wisconsin-Milwaukee, 2020
Under the Supervision of Professor Tian Zhao

The widespread adoption of Internet of Things (IoT) has given rise to systems which must reactively process signal data, while maintaining latency and throughput constraints. While the reactive programming paradigm partially addresses this problem, it does not offer a complete solution. We propose a design for a monad, AsyncM, and apply it to implement push-pull functional reactive programming (FRP) where side-effect bound events are performed in low-latency push streams, and time-series computations are performed in demand-driven pull streams. The design uses implicitly threaded cancellation tokens and continuation passing style to allow for implementations in languages which have single-threaded event-loops as well as languages which have multi-threading.

We present our abstractions Stream, Signal, and Behavior and their functor, applicative, and (when applicable) monadic instances. We demonstrate how push streams (Stream) can be converted to pull streams (Signal) (and visa versa) to apply our model to practical use cases. We give a use case of a real-time IoT data analytics platform to demonstrate our design's industrial significance where signal sample rates can change dynamically based on unknown factors such as network speed and processor resources. Application of the design to dynamic languages is shown throughout the paper in a reference JavaScript implementation.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

Thank you to Professor Tian Zhao for supporting me and giving me the direction that I needed for this undertaking over the last three years. You kept me engaged in our work together, you taught me what it means to do research, and gave me a lifetime's worth of inspiration for learning and discovery. Also, to all the friends I have made along the way in school, I appreciate all the late nights studying and the way we motivated each other. It is this collective will to succeed that gives me ambition not only for the good of research, but for all that I do.

I am forever grateful for my parents. Thank you dad for buying me a website domain and teaching me how to use a website editor when I was in grade school. Thank you mom for keeping me organized and finding creative ways to keep me engaged in school early on. You have given me more than enough to succeed in life. There's simply no way to thank you enough.

To my wife, Allie, I am grateful for how patient you were with me while I was slogging away with research. Thank you for being your loving, and selfless self. I love you, and I always will.

To everyone in my life who has supported me: My brothers (who keep me laughing), the Braza family (who taught me what it means to support someone in the first place), all my friends, and even my dog, Chloe. Thank you.

1 Introduction

The Internet of Things (IoT) has grown to popularity in the last decade, connecting an ever-increasing number of devices to networks. IoT devices range from consumer IoT devices, to robust industrial electrical systems which demand for monitor and control capabilities. For those mission-critical electrical systems, it is important to perform real-time system diagnostics to have visibility into the system to avoid system failure. We refer to these diagnostics as key performance indicators (KPIs). KPIs are computations performed on signals which characterize a system's health. For example, a solar panel manufacturer may specify that a panel has 20% efficiency (converts 20% of the sun's energy into electricity). To ensure the panel is performing according to the rated efficiency, a KPI could be computed that takes the sun's energy and the panel's output power as input signals, and computes the efficiency as an output signal. The signal could be stored to a database and visualized on a dashboard. A user could monitor the efficiency KPI (along with many others) to know when components of their system need maintenance.

The efficiency KPI is a simple example, as it is a point-wise calculation (applied to one sample at a time) which can be run at a low resolution (as calculating efficiency even more than once per second could be redundant).

However, there exist KPIs which require high frequency signals at rates of 10KHz or above (depending on the input signal), such as total harmonic distortion (THD). Calculating THD of a signal requires performing a Fourier transform. This requirement means the sample rate must be high enough to reproduce the real continuous signal using discrete samples. The requirement for high sample rate puts a burden on the system, forcing it to either scale or fail. If high speed data is coming at a rate in which the KPI calculation can not keep up with, it accrues unprocessed data (called back-pressure) and must decide to buffer the data, or discard it. Neither are complete solutions as buffering could consume the entire system's memory, and discarding loses information. A robust solution is to have built-in measures to communicate (via events) in the system, so that sample rates could be reduced. This way, the system will not crash over lack of memory, and batches of data will not be lost. The problem will have been addressed at the source. A complete IoT monitoring system could capture, run KPI computations, store data, and visualize data all at once.

We propose a model for reactive programming that provides asynchronous computations for events with side-effects, synchronous computation for pure KPI calculations, composability via a monadic interface, and compatibility with dynamic languages such as JavaScript. Our design, inspired by Elliott's Push-Pull FRP [9], combines push-streams and pull-streams to provide low

latency for responding to events (such as receiving batches of data), and pull-streams for composability of KPI calculations. These push and pull streams can be composed using the `push2pull` and `reactimate` combinators to convert from push streams to pull streams and back. We improve on Elliott's design by providing the AsyncM monad which provides a more efficient and event-loop compatible design for Elliott's unambiguous choice (`unamb`) operator, which represents a future computation that may or may not happen depending on the progress of other asynchronous computations.

Roadmap For the rest of the thesis we will explain the background of Classic FRP, Push-Pull FRP, and related problems of functional reactive programming (FRP) in Chapter 2, where we also give a brief introduction to monads, the monad laws, and an example of a simple monad to provide intuition. Chapter 3 introduces `AsyncM` our abstraction for asynchronous computation. We describe our cancellation tokens, `Progress`, the monadic instance for `AsyncM`, present several `AsyncM` primitives, and provide a real-world use case for `AsyncM`. We present our design of push-streams (called `Streams`), along with its components (`Emitter` and `Channel`) in Chapter 4. We also present `Stream` primitives, the monadic instance of `Stream`, and show how `Stream` satisfies the monad laws. Chapter 5 introduces our design of pull-streams (called `Signals` and `Behaviors`), and their primitives. We

apply our design to a real-world use case in Chapter 6, show related work in Chapter 7, and finally conclude with Chapter 8.

2 Background

2.1 Classic FRP

Classic FRP (the original design of functional reactive programming, first defined in FRAN [10]) was a source of motivation for our design. It uses an entirely pull-based approach featuring two fundamental abstractions `Behaviors` and `Events` which represent time-varying values and time-value pairs respectively. A `Behavior` is a function from time to value (`Time -> a`) that is used for representing continuous signals where at any time, the signal value can vary.

`t -> sin(t)` is a simple `Behavior`, as every time `t` yields some value. The most primitive `Behavior` is `Time` defined as the identity function `t -> t`.

Events are discrete occurrences of some action that are either internal to the program (which can be created via the `predicate` operator) or external (which the library provides a default set of – such as `lbp` for a left mouse button press). The `at` operator is defined as a means of getting a `Behavior`'s value at an absolute time and the `occ` operator provides means of accessing the time and value of an event occurrence.

```
at :: Behavior -> Time -> a
occ :: Event -> (Time, a)
```

This approach is pure. `Behaviors` and `Events` have no side-effects, which allows them to be composed without interference. Functions can be lifted into `Behavior` combinators through lift functions (`lift0`, `lift1`, and so on for different arities). Listing 2.1 defines `efficiencyB`, the efficiency of a solar panel over time, provided a `Behavior` of the sun's energy `sunB` and of the solar panel's output power `powerB`. It uses `lift2` to create a division combinator for `Behaviors`.

```
divB = lift2 (/)
efficiencyB = divB sunB powerB
```

Listing 2.1: FRP-style definition of solar array efficiency

Part of the novelty of FRP is its reactivity. `Behaviors` can be combined to create more `Behaviors`. A graph can be constructed (known as a "signal graph") to represent the connections between `Behaviors`. Unlike previous designs, such as in synchronous data-flow programming, this graph is dynamic. Edges in the graph (control flow to different `Behaviors`) can change while the program is running. This is done through the `untilB` operator. This functionality is known as "switching".

```
untilB :: Behavior a -> Event (Behavior a) -> Behavior a
```

`untilB` produces a `Behavior` that uses an initial `Behavior` until some event occurs that contains another `Behavior`, in which case it "switches" to that `Behavior`. This concept is powerful, as it enables the signal graph to be dynamic instead of fixed. This concept enables our `Streams` to scale device sample rates to the system's throughput dynamically.

2.2 Push-Pull FRP

Push-Pull FRP [9] is a modernization and re-work of Classic FRP that gives lower latency responses to events where it is desirable, and keeps `Behaviors` for demand driven values. A core realization was there are two types of `Behaviors`: time functions, and *reactive value*s. Time functions are absolute functions which don't depend on events. They are well suited to be pull-based, unlike *reactive values* which rely on events (which make them change discretely). In Classic FRP, all `Behaviors` are pulled at some interval, meaning each `Behavior` has a latency which at worst could be equal to the polling interval. This is the cost paid for the composability of pull-streams. *Reactive values* maintain their composability, but don't pay the penalty of latency from the poll interval. Our reactive programming model adopts the hybrid push-pull model of Push-Pull FRP, while improving upon

the switching mechanism (known as the `unamb` operator) by supporting non-multi-threaded environments such as JavaScript.

2.3 Problems with FRP

FRP has some intrinsic issues to its design which can make some implementations inconsistent, and dangerous to run. Most issues have known solutions, for example by reducing the expressiveness (limiting the first-class nature of `Behaviors`).

Space-Time Leaks Classic FRP's design was too permissive to be safe. Recall the `at` operator which takes a `Time` and `Behavior` and returns a value. The `Time` parameter refers to an absolute time. This means any time in the past, present, or future. Knowing the value of a `Behavior` in the past that depends on events requires caching events, and knowing the future value is impossible (without delaying the computation until that time becomes the present time). The issue of accessing future times causes another fundamental problem in FRP called causality which we discuss in the section below.

The ability to reference arbitrary times in the past for `Behaviors` which use `Events` requires storing every `Behavior`'s complete history since the program began (if the `Behavior` is not a simple time function relies on

events). Moreover, `Behaviors` can be defined recursively where each recursion runs a computation on a previous time value until the beginning of time (a problem that affects both time functions and event dependent `Behaviors`). Additionally a computation could consume non-linear time or space, escalating the problem where performance space and time usage increase over time. Listing 2.2 defines `past`, a time transformer which looks one time-step into the past. Looking backwards by a constant amount can be manageable (requires some constant sized buffer). Then `past_` is defined, a time transformer which looks into the past an arbitrary amount requires an infinite buffer. Finally, `average` a recursively defined `Behavior` which samples a `Behavior` once per time-step to compute the sum of some `Behavior` then dividing by the number of elapsed time-steps. The helper function `sum` results in a linear space-time leak.

```
1 past b = \t -> b `at` (t - 1)
2 past_ b n = \t -> b `at` (t - n)
3 sum t0 step b = \t ->
4     if t0 <= t then
5         (sum (t0 + step) b) + (b `at` t)
6     else 0
7 average t0 b = \t -> (sum t0 1 b) / (lift0 t0-t)
```

Listing 2.2: Three progressively larger space-time leaks

Space-time leaks must be addressed to ensure program safety. Arrowized

FRP (AFRP) [21] uses signal functions (`SF a b`), abstracts functions from `Behavior a` to `Behavior b`, which are composed using Hughes's Arrow abstraction [11] which is a point-free style that doesn't expose `Behaviors` as first class values. Liu and Haduk demonstrated how AFRP prevents the class of space-time leak described above from occurring by using arrows as the base abstraction instead of first-class `Behaviors` [18]. Another approach is to allow first-class `Behaviors`, but redefine the semantics of FRP such that first-class `Behavior` can be used, but the time parameter is not persisted indefinitely over time [13]. Our model restricts access to continuous time (seen in Section 4.3.2 with the `fold` operator), and therefore prevents space-time leaks of this kind.

Causality The `at` operator causes another issue, which is the ability to depend on future or past values in the present. Unguarded access to an absolute time parameter allows for `Behaviors` which request values in the future. For example, Listing 2.3 defines a time transformer `futureB` which takes a `Behavior` and returns that `Behavior` which looks one time step into the future.

```
1 futureB b = \t -> b `at` (t + 1)
```

Listing 2.3: A behavior who's current value is always in the future

Supporting a look-ahead `Behavior` this way, would require delaying

evaluation of `Behaviors` until their times have actually arrived (in which case the future will have become the present). This causes a space-time leak and requires some initial values to be used until the `Behavior` actually has a value.

Glitches The order in which the signal graph is updated can cause an issue called a "glitch". For example, Listing 2.4 shows a computation (in a language with JavaScript syntax and reactive evaluation semantics) which could cause glitches in reactive programming implementations which don't explicitly handle them. `c` depends on the values `a` and `b`. If `a` is updated (via an event), the signal graph could update in two orders: `a`, then `c` or `c` then `a`. In the first case, `c` will correctly equal 2 (`3 - 1`). In the second case, `c` will incorrectly equal 0 (`1 - 1`).

```
1 var a = 1;
2 var b = a - 1;
3 var c = a + b;
4 b = 3;
```

Listing 2.4: Definition of a glitch-prone signal graph

Reactive programming models solve this problem by having a global scheduler that topologically sorts the signal graph before it updates the graph (ensuring the graph is updated in the correct order). Other designs have glitches, but provide some ad-hoc means of ignoring the updates of

glitches. Our model allows for glitches to occur in push-streams (due to their asynchronous nature), but prohibits them in pull-streams (`Signals` and `Behaviors`). This is sufficient in preventing unexpected errors in the computation.

2.4 Monadic Interface

Our reactive design features a monadic interface for AsyncM, the driving force for our push-streams (known in Push-Pull FRP as *reactive values*). Monads are a concept from category theory that was imported into functional programming by Phillip Walder [29] as a way of structuring computations in a consistent way which isolate a computation into two parts: the potentially side-effect bound code hidden behind the interface, and the code that utilizes the interface.

Monads abstract a common pattern of computation in which some type `m` (which represents a computational context) wraps another computation `a` which can be composed (`m a -> m b`) using functions of type `a -> m b` where wrapped computations are returned in functions (called Klesli arrows). An example of this pattern is the use of `flatMap` for lists (the relation is obvious in the type signature `list a -> (a -> list b) -> list b`), or in the context of reactive programming `switchMap` (seen in RxJS to

switch between push-streams [2]).

2.4.1 Monadic Laws

Utilizing a monadic interface ensures that if the monad's definition is sound (if it obeys a set of laws), computations are guaranteed to be correct when they are composed. This cannot be said about arbitrarily defined interfaces in imperative code. To get the composability guarantees, monads must obey the following three laws.

Left Identity The first law, left identity, requires that a monad must provide an operator, `return`, which produces an identity value such that

$$return\, x \gg\!= \lambda y.f \equiv f[x/y]$$

The notation used is analogous to Walder's original laws [29], but with Haskell syntax. $\gg\!=$ applies a Klesli arrow to a monad with the type `m a -> (a -> m b) -> m b`, where $[x/y]$ means "x is bound to y". Left identity implies that some binding occurs of the value inside the monad on the left side of the $\gg\!=$ operator to the Klesli arrow on the right. Left identity requires that lifting `x` to a monad and applying the $\gg\!=$ operator to a function $\lambda y.f$ is equivalent to just binding `x` to `y` in the function $\lambda y.f$.

Right Identity The second law, right identity, requires the same concept, but a `return` on the opposite side. That is, some monad `m` bound ($\gg=$) with some function that `return`s a value `x` is the same as just the original monad `m`.

$$m \gg= \lambda x.return\ x \equiv m$$

Associativity Associativity requires the ordering of parentheses with the $\gg=$ operator to be irrelevant. Note that on the left, `x` is bound in `g` and on the right `x` is free in `g`.

$$m \gg= (\lambda x.f \gg= \lambda y.g) \equiv (m \gg= \lambda x.f) \gg= \lambda y.g$$

In practice, monads can be defined which break these rules. Because there are no means of statically proving if a monad respects the rules, it is the responsibility of the implementer to ensure the rules are followed. Following the monad laws ensures that the monad's behavior is consistent, and that simple transformations to the code do not result in unexpected differences in the computation.

2.4.2 A Time Function Monad

To demonstrate the usefulness of a monadic interface, Listing 2.5 shows a Haskell implementation of a time function monad. Time functions are `Behaviors` that don't use `Events`. Without a monadic approach, time parameters would be scattered around the code which is both cumbersome to write, and allows for unprotected modification of the time parameter (leading to issues that arise from time transformations such as non-causal `Behaviors` and space-time leaks).

In this example, an animation is considered (inspired by FRAN [10]). A circular motion can be created by animating an object over time using `cos` and `sin` for the x and y coordinates. `circleB` demonstrates how a `Behavior` can be created only by composing pre-defined `Behavior` monads (`sinB` and `cosB`) to produce a `Behavior Vec2` (a two dimensional vector that can be indexed based on continuous time). `scaleB` defines a combinator which scales the x and y coordinates by some factor `amt` which can stretch the motion of any `Behavior Vec2`.

The `Behavior` monad implicitly threads a `Time` parameter throughout the entire computation. The `Time` can be retrieved by an identity function `Behavior` called `time`. This is similar to a `Reader` monad. A `Reader` monad is one of many standard monads (such as `Maybe`, `List`, `Either`, `Writer` and `State`) that define some well-understood behavior. In the

case of `Reader` it is for providing an implicitly threaded value which can be `asked` for, and also can be locally set. The `Behavior` monad in Listing 2.5 isn't a complete `Reader` monad because there is no way to locally set the `Time` (doing so would make it possible to implement time transformations which are a source of many problems, and a source of the ultimate flexibility that FRAN provided). The effects of the standard monads can be layered on top of each other without re-implementing their behavior by use of *monad transformers*. We will use this idea as a model for understanding the `AsyncM` implementation later.

```
type Time = Float;
data Behavior a = Behavior (Time -> a)

at :: Behavior a -> Time -> a
at (Behavior f) t = f t

instance Functor Behavior where
  fmap f b = Behavior $ \t -> f $ b `at` t
instance Applicative Behavior where
  pure x = Behavior $ \t -> x
  mf <*> mx = Behavior $ \t -> (mf `at` t) (mx `at` t)
instance Monad Behavior where
  return = pure
  b >>= f = Behavior $ \t -> (f $ (b `at` t)) `at` t
```

```
15
16 time = Behavior $ \t -> t
17 sinB = do
18   t <- time
19   return $ sin t
20 cosB = do
21   t <- time
22   return $ cos t
23
24 type Vec2 = (Float, Float)
25 circleB :: Behavior Vec2
26 circleB = do
27   x <- cosB
28   y <- sinB
29   return $ (x, y)
30
31 scaleB :: Float -> Behavior Vec2 -> Behavior Vec2
32 scaleB amtB b = do
33   (x, y) <- b
34   z <- amtB
35   return $ (x * z, y * z)
```

Listing 2.5: Time Function Monad Definition

The `scaleB` combinator can be used as `scaleB (return 2) circleB` to increase the circle's motion radius by two. A variable scaling can be

achieved by `scaleB (absB cosB) circleB` (assuming some absolute value combinator `absB`).

3 AsyncM

Our design builds on Push-Pull FRP [9] by providing a monad, `AsyncM`, which uses continuation-passing style (CPS), to create a concurrency model agnostic means of implementing *futures* (the concept behind *reactive values* in Push-Pull FRP). `AsyncM` uses nested `Progress` values to control the cancellation of composed `AsyncM` monads, which allows for expressing asynchronous operations that race against each other (`raceM`), wait for all to complete (`allM`), wait for any to complete (`anyM`), run in parallel without merging progress (`forkM`).

3.1 Progress

Progress of asynchronous computation is monitored in a nested fashion. These progress values (known as "cancellation tokens") are constructed as a cons list where each `Progress` has a head value indicating if the `Progress` has been cancelled, and optionally a tail (which is another `Progress` value). `Progress` values can either be a `ConsP` where the cell has a head value (which is true if the `Progress` is cancelled) and a tail value, or a `NilP` where there is only a head value.

Checking if a progress value is alive is a recursive algorithm (seen in

Listing 3.1) where the current progress value's `head` is checked, then its `tail` is checked recursively. If the `head` is true for any `Progress` value (including the nested ones), this indicates cancellation.

```
class ConsP extends Progress {
    isAliveP () {
        if (!this.head) { return this.tail.isAliveP(); }
        else return false;
    }
}
class NilP extends Progress {
    isAliveP () {
        return !this.head;
    }
}
```

Listing 3.1: Recursive liveliness check for `Progress` values

The significance of this design is that cancellation bubbles up from the inside out. For example, consider three `Progress` objects (A, B, and C) which are all alive (head is false) in Figure 3.1.

However in Figure 3.2, `Progress` B is cancelled. This cancellation automatically causes `Progress` C to be cancelled too. Even though the outer `Progress` is cancelled, `Progress` A will continue to be alive. This

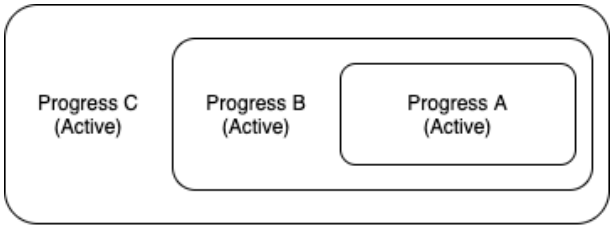

Figure 3.1: `Progress` value C with two child `Progress` values.

property is useful for `AsyncM` where we want to create a nested asynchronous computation who's cancellation is not affected by its outer scope (e.g. `scopeM`).

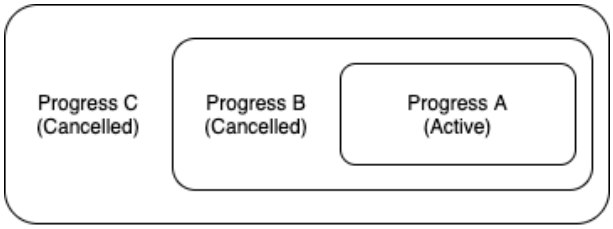

Figure 3.2: `Progress` C after cancelling `Progress` B.

3.2 AsyncM

3.2.1 Monadic Stack

`AsyncM` is a combination of two monad transformers (`ReaderT` and `ContT`) with a base `IO` monad that can be created with a type of `Progress -> (a -> IO()) -> IO ()`. That is, a function which takes an initial `Progress` value, a continuation `Progress` (the next computation),

and performs some side-effect. As mentioned in the monad section, monads can be combined using *monad transformers*. Transformers can be used directly in languages that support them (e.g. Haskell), otherwise writing a new monad with the effects of both is necessary. To understand `AsyncM` it is important to first consider the effects of the two monads `Reader` and `Continuation` and imagine stacking their effects on top of each other to create a single monad.

Reader Monad A `Reader` monad (as discussed in the Time Function Monad section) provides a $\gg=$ operator definition, `return` definition, and two helper functions `ask` and `local`. First consider the type signature of a `Reader` type: `e -> a` (where `e` is an environment and `a` is some output type). The $\gg=$ operator (`Reader e a -> (a -> Reader e b) -> Reader e b`) implicitly threads the environment `e` throughout functions. Note how the Klesli arrow takes a value of the output type `a` (`e` is hidden). `return` lifts some value into the monad by creating a function which takes `e` (which it ignores) and returns a constant value. The monad would be useless if it didn't have a means of accessing the environment `e`. `ask` serves this purpose by creating a `Reader` using an identity function (`e -> e`). `ask` has type `Reader e a -> Reader e e`. Although the monad is called "reader" there is a way of writing a new value (at least for any

child readers). `local` (type `(e -> e) -> Reader e a -> Reader e a`) can be used to map over the environment `e` so that a child reader is run with a modified environment.

Listing 3.2 shows a definition for $\gg=$ for a `Reader e a` type. This reveals the implicit threading of the environment value which is hidden to code that uses the monad.

```
1 >>= :: Reader e a -> (a -> Reader e b) -> Reader e b
2 m >>= f = Reader \e -> runReader (f (runReader m e)) e
```

Listing 3.2: Reader monad interface

Continuation Monad A continuation monad uses a monadic interface to implement continuations via continuation passing style (CPS). Continuations are a way to represent computations (along with all their state) that have yet to happen. Some languages implement continuations as first-class values (e.g. Scheme, Ruby). Languages with first-class functions can emulate continuations using CPS. This involves passing an additional function parameter to each participating function, and optionally calling that function with the result of the function when the computation completes. This style replaces `return` statements with function calls, giving the program more access to modify control-flow. However, CPS is cumbersome to write manually. Using a monad for this task is ideal, because the continuation parameter can be

passed implicitly (through the $\gg=$ operator).

A continuation monad instance has the type of `(a -> r) -> r` where `r` is some return type, and `a` is some input type. Often in Haskell, this will be presented as a monad transformer `ContT` which adds a monad parameter into the type signature `(a -> m r) -> m r`. We are concerned with when the monad is `IO` (such as `(a -> IO r) -> IO r`). The difference isn't necessary to distinguish in languages such as JavaScript where types are not checked statically. The first argument of the continuation type is called the "continuation" (that is `(a -> r)`). This is the continuation function which is implicitly propagated via $\gg=$ operator.

A continuation monad should provide three definitions: `return` which simply calls the continuation with a constant value, $\gg=$ :: `Cont r a -> (a -> Cont r b) -> Cont r b` which creates a new continuation that runs its first argument `Cont r a` and with the value it yields, calls the Klesli arrow with it (producing an inner continuation). Listing 3.3 shows the definition for $\gg=$ for a continuation type `Cont` whose type is `(a -> r) -> r`.

```
1 >>= :: Cont r a -> (a -> Cont r b) -> Cont r b
2 m >>= f = Cont (\k -> runCont m (\x -> runCont (f x) k))
```

Listing 3.3: Continuation monad interface

3.2.2 Monadic Definition

Having understood the `Reader` and `Continuation` monad, the `AsyncM` monad can be understood by layering their effects on top of each other. Listing 3.4 shows the monadic definition for `AsyncM`. The definition includes the functions `fmap` and `app` which correspond with `AsyncM`'s functor and applicative instances respectively. `fmap` can be used to create an `AsyncM` who's asynchronous result (that depends some hidden `Progress` value) has been applied to a function. `app` can be used to apply a function wrapped in some asynchronous computation to another asynchronous value. `pure` corresponds with `AsyncM`'s monadic `return` operator (also required for applicative instances).

```
1  class AsyncM {
2      // new :: (Progress -> (a -> IO ()) -> IO()) -> AsyncM a
3      constructor = run => { this.run = run; }
4      // fmap :: AsyncM a -> (a -> b) -> AsyncM b
5      fmap = f => new AsyncM (p => k => this.run(p)(x => k(f(x))));
6      // pure :: a -> AsyncM a
7      static pure = x => new AsyncM (p => k => k(x))
8      // app :: AsyncM (a -> b) -> AsyncM a -> AsyncM b
9      app = mx => this.bind(f => mx.bind(x => AsyncM.pure(f(x))))
10     // bind :: AsyncM a -> (a -> AsyncM b) -> AsyncM b
11     bind = f => new AsyncM (p => k =>
12         this.run(p)(x => f(x).run(p)(k)));
```

```
13 }
```

Listing 3.4: The definition of `AsyncM`

The `bind` implementation (≫=) merges the definitions of `Reader` and `Continuation` that we saw in the previous section. The `run` method serves the purpose as both `runReader` and `runCont` did in the previous section (it is a curried function where the first application has the same semantics as `runReader` and the second of `runCont`). In our Haskell implementation this monad can be expressed without any custom code by stacking the `ReaderT` and `ContT` *monad transformers*. The usefulness of this `bind` definition will be apparent after the definition of several utility functions.

3.2.3 Primitives

scopeM `scopeM` (similar to a `Reader` monad's `local` function) creates a new `AsyncM` with a modified `Progress` value. The `Progress` value is extended via adding to the `Progress` cons list. This way, cancelling the returned `AsyncM` will not cancel any child `AsyncM`. As explain in the Progress Section, the `AsyncM` will be cancelled whenever any child is cancelled.

```
1 const scopeM = m => new AsyncM(p => k => this.run(p.cons())(k));
```

unscopeM `unscopeM` performs the opposite of `scopeM`. That is, it creates a new `AsyncM` which has a modified `Progress` value that uses the tail of the `Progress` cons list.

```
const unscopeM = m =>
    new AsyncM(p => k => this.run(p.tail ? p.tail : p)(k));
```

forkM `forkM` runs the `AsyncM` and the continuation `k` with a new `Progress` value which is cancelled if the parent is ever cancelled. The `AsyncM`'s side-effects are run in parallel along with the continuation's side-effects. This is a race; however, the `Progress` values are not linked which makes `forkM` distinct from `raceM`. An identity function is used for when running the `AsyncM` as the final continuation, as we are running it for its side-effects (has type `IO ()`). Note the the `Progress` value of the forked `AsyncM` is exposed as a means of cancelling the forked computation.

```
const forkM = scopeM(m => new AsyncM (p => k => {
    m.run(p)(x => x);
    k(p);
}));
```

ifAliveM `ifAliveM` creates a checkpoint where `Progress` of the given `AsyncM` is checked. If the `AsyncM` has been cancelled (using the recursive condition specified in the Progress Section), the `AsyncM`'s continuation is

not called. This causes the computation to terminate immediately after the checkpoint.

```
const ifAliveM = new AsyncM(p => k => p.ifAliveP(k));
```

cancelM `cancelM` cancels the current `Progress` and continues the computation (calls the continuation `k`). This primitive is used for indicating cancellation to parent or racing `AsyncM`, but continuing the current computation.

```
const cancelM = new AsyncM(p => k => { if (p.cancelP()) k(); });
```

commitM `commitM` combines `ifAliveM` and `cancelM` to only continue the computation if the `Progress` is alive, and then cancelling the `Progress`. This is primitive is useful in cases, such as just after an `raceM`, to ensure the continuation is only called once.

```
const commitM = ifAliveM.bind(_ => cancelM);
```

timeout `timeout` run calls the `AsyncM`'s continuation in `n` milliseconds. This function forms the base for `reactimateing` pull-streams (`Signals`) into push-streams `Streams`. The JavaScript implementation uses `setTimeout`, but the concept works with any form of concurrency (event-loop based or multi-threading for example).

```
const timeout = n => new AsyncM(p => k => setTimeout(k, n));
```

raceM `raceM` runs two `AsyncM`s in parallel with a newly scoped `Progress` value. Scoping the `Progress` allows child `AsyncM`s to use either `commitM` or `cancelM` directly to cancel itself to allow for only one to continue after the race. If no cancellation is made, the continuation `k` will be called twice.

```
const raceM = m1 => m2 => scopeM(new AsyncM(p => k => {
    m1.run(p)(k);
    m2.run(p)(k);
}));
```

anyM `anyM` runs two `AsyncM`s that race with each other, where if the first `m1` succeeds, the computation continues with a `Either` value of `Left x1` otherwise it continues with a value of `Right x2`. `commitM` is used so that the continuation of `raceM` is only called once (by the `AsyncM` that won the race). Using `Either` allows the succeeding computation to know which `AsyncM` completed.

```
const anyM = m1 => m2 => raceM
    (m1.bind(x1 => commitM.bind(_ => AsyncM.pure(Left(x1)))))
    (m2.bind(x2 => commitM.bind(_ => AsyncM.pure(Right(x2)))))
```

allM `allM` runs two `AsyncM` in parallel, joins their return values together, and continues when both `AsyncM` have completed.

```
const allM = m1 => m2 => new AsyncM (p => k => {
```

```
        let count = 0, r = [];
        const k1 = x => {
                r[0] = x;
                if (++count >= 2) k(r);
        };
        const k2 = x => {
                r[1] = x;
                if (++count >= 2) k(r);
        };
        m1.run(p)(k1);
        m2.run(p)(k2);
});
```

3.2.4 Monad Laws

Section 2.4.1 presented the *monad laws* which were three laws which monads are to follow to ensure consistent behavior. `AsyncM` should obey these laws if we claim it to be a monad. Our monadic definition of `AsyncM` utilized the concept of *monad transformers*, where conceptually two monad transformers `ReaderT` and `ContT` are "stacked" on top of an IO monad to create `AsyncM`. It is well-established that stacking monad transformers on top of a monad such as IO monad results in another monad [17], which means that `AsyncM` obeys the monad laws.

3.2.5 Use Case: Cancelable Web Requests

`AsyncM` primitives can be combined for general purpose asynchronous computations. The code that interfaces with `AsyncM` can be written in a declarative style (in Haskell using do-notation, in JavaScript through `bind`), but represents complex relationships between cancelable asynchronous processes.

Consider an example where some data is being loaded from an API, and the user has a cancel button which stops the loading from happening. The HTTP request could be sent with some delay (`timeout`), and if the cancel button was not pressed before the delay, the resources would be loaded. Listing 3.5 shows an example of this where a list of URLs is loaded in five seconds, unless the user clicks a button. Not only is this effect achieved, but the resulting `AsyncM`, `cancelableFetchM`, is composable. Code that uses the example can `bind cancelableFetchM`. If the value is a `Left`, the button cancelled the data from being loaded, otherwise a `Right` value that contains the result of the web requests is received. `getM` defines an `AsyncM` that fetches a JSON via a HTTP request (using the JavaScript `fetch` API), and continues the computation with the JSON value. `clickM` is an `AsyncM` which continues when a button is pressed (given some DOM node).

```
1 const getM = url => new AsyncM(p => k =>
2         fetch(url).then(r => r.json().then(k)));
3 const clickM = el => new AsyncM(p => k =>
```

```
        el.addEventListener('click', k, { once: true }));
const fetchDataM = allM(getM('/user/1'))(getM('/user/2'));
const cancelableFetchM = anyM(
        clickM(document.querySelector('#cancel')))
        (timeout(5000).bind(_ =>
            ifAliveM().bind(_ => fetchDataM)));
```

Listing 3.5: An `AsyncM` which fetches data unless a cancel button is pressed.

4 Streams

`AsyncM` can be used as the foundation for creating a push-pull model of FRP. This model of FRP will inherit the benefits of using `AsyncM`. `AsyncM`'s usage of CPS will enable the FRP system to support implementations in languages regardless of their concurrency strategy (e.g. multi-threading, single-threaded event loops), and `AsyncM`'s cancellation semantics will enable the FRP system to create exit checkpoints where asynchronous computations can exit safely and reliably.

The new FRP design must not only be equipped with `AsyncM`, but also have several new primitive objects. For implementation of an IoT analytics system (as we demonstrate in Chapter 6), it is important to have switching (for adjusting the computation to dynamic factors), a hybrid push-pull approach (where data which should be low latency is pushed, and composable time-series computations are pulled), and a means of converting between push and pull streams. To accomplish this, we layer several objects on top of `AsyncM`: `Emitter`, `Channel`, `Stream`, `Signal`, and `Behavior`.

4.1 Emitter

An `Emitter` is a stateful class which implements the core of push-streams. `Emitters` maintain a list of listeners (which are `AsyncM` continuations to call upon emission), and the last emitted value. `Emitters` have three basic primitive methods: `emit`, `listen`, and `wait`. `listen` creates a new `AsyncM` who registers itself to the emitter (the continuation is called during emission), `emit` calls each continuation (stored in `listeners`) with a given value, `wait` creates an `AsyncM` which either continues with the latest value of the emitter (if it has one), otherwise, registers the continuation with the emitter.

```
1  class Emitter {
2      constructor() {
3          this.now = null;
4          this.listeners = [];
5      }
6      emit = x => {
7          this.now = x;
8          const listeners = this.listeners;
9          this.listeners = [];
10         for (const listener of listeners) listener(x);
11     }
12     listen = () => new AsyncM(p => k => this.listeners.push(k))
13     wait = () => new AsyncM(p => k => {
```

```
14              if (this.now !== null) k(this.now);
15              else this.listeners = [k];
16      });
17  }
```

Listing 4.1: Emitter class definition

`Emitter` objects are used to propagate values throughout the signal graph via pushing. The latency of `Emitter` objects is low, as the listeners are called immediately after an event is emitted. Pull-based models could have delay that is at worst case as long as the sample rate; while this push-based approach doesn't introduce any additional latency.

An `AsyncM` utility function `spawnM` is defined that uses `Emitter` (found in Listing 4.2). `spawnM` creates a new `AsyncM` which yields an inner `AsyncM`, who continues whenever the parameter `AsyncM`, `m`, continues. The synchronization between `AsyncM` is done via a shared `Emitter` object. This becomes useful for our definition of switching for push-streams, when we need to refer the next continuation of an `AsyncM` as an `AsyncM` value.

```
1  const spawnM = m => new AsyncM(p => k => {
2    const e = new Emitter();
3    m.run(p)(x => e.emit(x));
4    k(e.wait());
5  });
```

Listing 4.2: Definition of `spawnM`

4.2 Channel

`Channel` is a stateful class which implements the core of pull-streams. `Channels` have a list of data (called the "buffer"), and a list of continuations (called "listeners"). The purpose of a `Channel` is to provide a queue (FIFO) of buffered values which are written asynchronously (an `AsyncM` would continue immediately) and read synchronously (an `AsyncM` will not continue until there is a value). `Channels` have two primitive operators: `read` and `write`. `read` takes a continuation, and if the `Channel` has data, dequeues an item from the buffer, and calls the continuation with that item. If there is no data, the continuation is added as a listener, so that it is called immediately when a `write` occurs. `write` adds an item to the buffer (if there are no listeners), otherwise immediately dequeues the next listener and calls it with the item.

```
class Channel {
    constructor() {
        this.data = [];
        this.listeners = [];
    }
    // read :: (a -> IO()) -> IO()
    read = k => this.data.length > 0 ?
        k(this.data.shift()) :
        this.listeners.push(k);
```

```
10      // write :: a -> IO()
11      write = x => this.listeners.length > 0 ?
12          this.listeners.shift()(x) :
13          this.data.push(x);
14  }
```

As consecutive calls to `write` occur, items accumulate indefinitely in the buffer. This accumulation is called *back-pressure*. Back-pressure is essential in systems which must process data at rates that are dynamic or could exceed the processing limits of some computation. While back-pressure enables such computations to occur, it can also be the source of a memory-leak in programs who can not process buffered data at a rate greater than the rate of incoming data. In our IoT Analytics use case we address this problem by measuring the data rate, and altering the sampling rate to match the computation's throughput. Maintaining a safe amount of back-pressure is a matter of ensuring that `writes` from channels occur at less than or equal to the rate that `reads` occur.

4.3 Stream

`Stream` is our implementation of push-streams (in the spirit of Push-Pull FRP) which is comprised of a cons list of `Next` values linked together by `AsyncMs`, and terminated with an `End` value. The head value of each item

in the list is a `Maybe` type where there can either be `Just` some value, or `Nothing`. The use of a `Maybe` type is due to streams of IoT data having no reasonable initialization value (the data is only known after the data is received). To illustrate why `Maybe` is necessary, imagine a `Stream` of voltage signals, and a KPI which divides by voltage. Now the most reasonable of initialization values, zero, causes division by zero. To avoid this, there must be some way of skipping a computation when there is no value (to not force an initial value). This way a `Stream` can be initialized with a `nothing` value, until data comes in. The tail of a `Next` type is an `AsyncM` of `Stream` (a `Stream` which occurs in the future). This cons list structure where the tail is an asynchronous computation can continue indefinitely (by always yielding `Next` values), or can terminate via a `End` value (which has no tail).

Similar to `AsyncM`, `Stream` has a monadic interface, but unlike `AsyncM`, the monad is not comprised of monadic transformers. Listing 4.3 defines the monadic interface for `Stream` (along with `fmap`, and `app` for functor and applicative instances). `fmap` applies a function to each head of the cons list recursively. `Next`'s `fmap` implementation can continue indefinitely, as `this.next` is an `AsyncM` value. This is desired behavior to support `Streams` which never terminate (e.g. background processing, real-time visualizations). `maybe_a` represents the `Maybe` type, which is a separate monad consisting of the current value of the `Stream`. This means `Maybe`'s

`fmap` function will not apply the function parameter to `Stream` values who are a `Nothing` value (it only applies the function to `Just` values). `Maybe`'s `maybe` method takes two arguments (curried), the first being the value if the `maybe` is a `Nothing` type, the second being the value if the `maybe` is a `Just` type.

`bind` is implemented in a similar fashion to a `List` monad where the monad is applied to `fmap`, then flattened (with `join`). To "flatten" is to take some type `T` of `T` and yield a type of just `T`. For example, taking a `Stream` of `Streams` (a `Stream` which produces `Streams`), and create a single `Stream`. While the concept is simple, the way that the flattening occurs determines the semantics of the monad. In this case, the flattening determines the semantics of push-stream switching. The `switchS` function is discussed in Section 4.3.1 which is omitted in this listing, and is responsible for the majority of the `bind` implementation. Finally, the `app` function uses `bind` to apply a `Stream` of functions to a `Stream` of any value.

```
1 class Stream {
2   constructor(a) { this.maybe_a = a; }
3   static next = (a, m) => new Next(a, m)
4   static end = a => new End(a)
5   app = sx => this.bind(f => sx.bind(x => Stream.pure(f(x))))
6   static pure = x => Stream.end(Maybe.just(x))
7 }
```

```
class Next extends Stream {
  constructor (a, m) { super(a); this.next = m; }
  fmap = f => Stream.next(
    this.maybe_a.fmap(f),
    this.next.fmap(s => s.fmap(f)))
  bind = k => this.fmap(k).join()
  join = () => this.maybe_a.maybe
    (Stream.next(Maybe.nothing, this.next.fmap(ss => ss.join())))
    (s => s.switchS(this.next))
  run = k => ifAliveM.bind(_ =>
    AsyncM.liftIO(k1 => k1(this.maybe_a.maybe (unit) (k)))
      .bind(_ => this.next.bind(s => s.run(k))))
}
class End extends Stream {
  constructor (a) { super(a); }
  fmap = f => Stream.end(this.maybe_a.fmap(f))
  bind = k => this.maybe_a.maybe (this) (k)
  join = () => this.maybe_a.maybe (this) (s => s)
  run = k => ifAliveM.bind(_ =>
    AsyncM.liftIO(k1 => k1(this.maybe_a.maybe (unit) (k))))
}
```

Listing 4.3: Monadic instance for `Stream`

The `run` method can be used to recursively call the `Stream` for its values, calling each value with the continuation `k`. This way, values in the `Stream`

can be exposed to functions which perform side-effects.

4.3.1 Switching

In Listing 4.3, we presented the monadic instance for `Stream`. However, it omitted the implementation of `switchS`. Listing 4.4 shows extensions to the `Stream` classes to provide switching (separated from Listing 4.3 for clarity). Note `either` is a method of an `Either` object where the first function parameter is called if the `Either` value is of type `Left`, and calls the second function parameter if the value is of type `Right`. `switchS` is the final interface function (seen in its usage in `bind`), while the other functions are utility functions.

```
1  class Stream {
2    static _switchA = ms => mss => {
3      const f = choice =>
4        choice.either(ss => ss.join()) (s => s._switch(mss));
5      return anyM(mss)(ms.unscopeM()).fmap(f);
6    }
7  }
8  class Next extends Stream {
9    switchS = mss1 => Stream
10     .next(this.maybe_a,
11       mss1.spawnM()
12         .bind(mss2 => Stream._switchA(this.next)(mss2)))
```

```
13   _switch = mss => Stream
14     .next(this.maybe_a, Stream._switchA(this.next)(mss))
15 }
16 class End extends Stream {
17   switchS = mss => this._switch(mss)
18   _switch = mss =>
19     Stream.next(this.maybe_a, mss.fmap(ss => ss.join()))
20 }
```

Listing 4.4: Extensions to `Stream` classes for switching

Switching decides which `Stream` in the `Stream` of `Streams` to use by racing the first `Stream` with the `Stream` of `Streams`. If the initial `Stream` wins, that `Stream` produces values for the parent `Stream`. Otherwise, if the `Stream` of `Streams` wins, the next value of that `Stream` is used as the next initial value, and the racing continues in this manner until an `End` value is reached.

`spawnM` is used in `Next`'s `switchS` method to create a reference to the next `AsyncM` that will occur after the given one (not as a continuation function as usual, but as another `AsyncM` object). `_switchA` must call `unscopeM` to use the inner `Progress` object of `ms`, as `anyM` would otherwise cause it to be cancelled via `commitM` (as seen in the `anyM` definition).

4.3.2 Primitives

receive `receive` takes an `Emitter`, and creates a `Stream` of values which are continuously pushed by the `Emitter`. The effect can be conceptualized as a converter from `Emitter` to a `Stream`.

```
const receive = e => {
  const h = _ => e
    .listen()
    .bind(a => ifAliveM.bind(_ =>
      AsyncM.pure(Stream.next(Maybe.just(a), h()))));
  return Stream.next(Maybe.nothing, h());
}
```

repeatS `repeatS` takes a `AsyncM` and creates a `Stream` which continuously runs that `AsyncM`. Between each run of the `AsyncM`, the `Progress` value is checked. If the `AsyncM` has been cancelled, the computation does not continue. The utility function `repeatA` recursively constructs the `Stream`, and `repeatS` initializes the `Stream` with a `Nothing` value.

```
const repeatS = m => Stream.next(Maybe.nothing, repeatA(m))
const repeatA = m => m.bind(a =>
  ifAliveM.bind(_ =>
    AsyncM.pure(Stream.next(Maybe.just(a), repeatA(m)))))
```

broadcast `broadcast` runs a `Stream` `s` and `emits` all values from the `Stream` into a new `Emitter` object. Emission does not block, as it is run in parallel with `forkM`. The resulting `Emitter` object (along with its `Progress`) is exposed in the resulting `AsyncM`, to be `listened` to by another `AsyncM` (or converted to a `Stream` by `receive`). Note that the `Stream` `s` will run indefinitely. To allow cancellation, `forkM` exposes the `Progress` value and passes on in the result (`AsyncM [a, Progress]`).

```
const broadcast = s => AsyncM.liftIO(k => k(new Emitter()))
    .bind(e => s.run(x => e.emit(x))
    .forkM()
    .bind(p => AsyncM.pure([e,p])))
```

multicast `multicast` enables multiple `Streams` to be constructed which have the values of a single `Emitter`. It uses `receive` to create a `Stream` from an `Emitter` and yields a `Stream [Stream a, Progress]`. The `Progress` value of the input `Stream` is exposed, similar to `broadcast`, as the `Stream` is run indefinitely until the `Progress` is cancelled, or the `Stream` reaches an `End`.

```
const multicast = s => Stream.next(Maybe.nothing,
            broadcast(s).bind(([e,p]) =>
              AsyncM.pure(Stream.pure([e.receive(), p]))))
```

fetch `fetch` `Streams` to be buffered, which allows asynchronous actions which have variable running times (such as network requests) to be buffered in a `Channel` so that `AsyncM` can be executed in parallel and buffered, instead of waiting for each `AsyncM` to complete. In practice this means network requests which take two seconds to capture one second of IoT data (due to a one second latency in the network) could be made in parallel so that every second requests are made instead of every two seconds.

`fetch` uses a `Channel` to `write` a `Stream` of asynchronous results (`AsyncM`), and yields a new `Stream` which reads the results continuously. `fetch` provides clean integration between `AsyncM` based computations and `Streams`. For example, `fetch` can take batched HTTP requests represented as a `Stream (AsyncM Request)` and provide a more natural type of `Stream Request`.

```
// fetch :: Stream (AsyncM a) -> Stream a
const fetch = s => {
  const m = AsyncM.liftIO (k => k(new Channel())).bind(c => {
    const w = s.bind(m => m.spawnM()
                .liftS()).run(m1 => c.write(m1)).forkM();
    return w.bind(p =>
              repeatA(AsyncM.liftIO(k => c.read(k)).join()));
  })
  return Stream.next(Maybe.nothing, m)
}
```

fold `fold`, similar to the `reduce` or `fold` functions commonly seen in functional languages, accumulates some value over a list. In our case, the list is a `Stream` of functions (`a -> a`). This way, a `Stream` can persist some state over the course of its lifetime. When the `Stream` ends, a final accumulated value is provided.

Recall two of the issues common in FRP: non-causality and space-time leaks. Our implementation of `fold` avoids one important class of space-time leak by only persisting the current value of the accumulator, instead of persisting all events emitted (this is similar to how AFRP avoids this space-time leak). Also, our function is causal, as you cannot refer to times in the future directly. You can directly refer to future values by using `timeout` to delay computation; however, that will resolve when the future time has become the present.

```
1
2  class Stream {
3    // fold :: Int -> a -> Stream (a -> a) -> AsyncM a
4    fold = n => c =>
5      this.accumulate(c).last(timeout(n)).fmap(r => r.fromJust())
6    last = m => m.spawnM().bind(m1 => this._last(m1))
7  }
8  class Next extends Stream {
9    accumulate = a => {
10     const a1 = this.maybe_a.maybe(a)(f => f(a));
```

```
11      return Stream
12        .next(Maybe.just(a1),
13              this.next.fmap(s => s.accumulate(a1)));
14    }
15    _last = m => anyM(this.next)(m)
16      .bind(r => r.either
17        (s => s._last(m))
18        (_ => AsyncM.pure(this.maybe_a)))
19  }
20  class End extends Stream {
21    accumulate = a =>
22      Stream.end(Maybe.just(this.maybe_a.maybe(a)(f => f(a))))
23    _last = m => m.fmap(_ => this.maybe_a)
24  }
```

Listing 4.5: Extensions to `Stream` class to provide `fold`

`fold` uses a utility function, `accumulate`, which is specialized for `Next` and `End` types, where `Next` recursively accumulates and `End` accumulates once and terminates. `last` returns either the last event of a `Stream`, or if some given `AsyncM m` completes first, returns the last event that was emitted before the race was won.

`fold`'s integer parameter corresponds with how many milliseconds to accumulate a value before providing the final value. For example, Listing 4.6 defines `count` which yields an `AsyncM` of how many events occurred in `n`

milliseconds.

```
const count = s => n => s.fmap(_ => c => c+1).fold(n)(0);
```

Listing 4.6: Definition of `count` using `fold`

4.3.3 Monad Laws

Wadler defines that there are seven laws that when satisfied, are equivalent to the three monad laws we defined earlier [29]. These laws are better suited for monads which define a `fmap` and `join` function, as our `Stream` monad does, so long as the $\gg=$ operator is defined as:

```
m >>= f = join (fmap f m)
```

The seven laws are as follows:

1. $fmap\, id = id$
2. $fmap\, (f \circ g) = fmap\, f \circ fmap\, g$
3. $fmap\, f \circ return = return \circ f$
4. $fmap\, f \circ join = join \circ fmap\, (fmap\, f)$
5. $join \circ return = id$
6. $join \circ fmap\, return = id$
7. $join \circ fmap\, join = join \circ join$

Law one, law two, and law three The first three laws deal primarily with the behavior of `fmap`. Proving the first will provide the intuition that the next two laws follow. Recall `fmap` applies some function `f` recursively to a `Stream`. We can substitute `f` for the identity function `id`, as seen in Listing 4.7.

```
1 fmap id (Next a ms) = Next (fmap id a) (fmap id ms)
2 fmap id (End a) = End (fmap id a)
```

Listing 4.7: Substituation of identity function in `fmap`

When `fmap` is applied with the identity function (`a -> a`), we can derive that `fmap id` on a `Stream` applies the inner `Maybe` value with an identity function, which is equivalent to just `id` of the `Stream` itself. Notice that `fmap` is only ever applied to either a `Maybe` type or recursively (which results in an `fmap` to a `Maybe` type). Because `Maybe` upholds the monad laws and all recursive calls use `Maybe fmap`, it follows that our monad upholds the monad laws for `fmap`. The third law needs one more clarification, which is that `return x` simply creates an `End` stream of `Just x` (which reduces it again to `fmaps` of `Maybe`).

Law four Law four requires applying `fmap` to a nested monad (in our case `Stream (Stream a)`), then applying `fmap` once more is the same as `joining` the nested monad, then applying `fmap`. This law is implied

by the type of `join` (`Stream (Stream a) -> Stream a`), which is parametric to the type variable of the monad [28].

Law five Law five says that the `joined return` of some `Stream` is equivalent to the identity function. To prove this, we need a concrete representation of the `join` operator. The `join` operator is represented in a more algebraic form than our previous JavaScript formulation in Listing 4.8.

```
1 join :: Stream (Stream a) -> Stream a
2 join (Next (Nothing) mss) = (Next (Nothing) (fmap join mss))
3 join (Next (Just s) mss) = switch s mss
4 join (End (Just s)) = s
```

Listing 4.8: Defintion of `join` for `Stream`

Because `return x` creates a `End (Just x) Stream`, it is equivalent to `join (End (Just x)) = id x`. As seen in Listing 4.8, `join` of an `End Stream` equals the value of the `Just` type. This reduces `join` to be an identity function when applied with an `End Stream`, and therefore, satisfies the law.

Law Six Law six holds for a similar reason as law five. `fmap return` is applied to a `Stream`, which produces a `Stream` of `End Streams`. As seen in Listing 4.9, `join` of an `End Stream` constructs a `Next Stream` where the value is equivalent to the `End Stream`.

Law Seven The final law says that `joining` the resulting `Stream` of `fmap join` (with a `Stream` of `Stream` of `Stream`) is equivalent to `joining` twice. In words, this says that the ordering in which `join` occurs has no importance (`fmap` first `joins` the inner `Streams` first). This is *associativity* for the `join` operator.

To prove this, we must first understand the complete definition of `join`. `join` uses `switch` (which was ommitted from Listing 4.8) to implement flattening of the nested `Stream`. `switch` is represented in Listing 4.9 in a more algebraic form.

```
switch :: Stream a -> AsyncM (Stream (Stream a)) -> Stream a
switch (Next a ms) mss = Next a (h ms =<< spawnM mss)
    where h ms mss =
      let
        f (Left ss) = join ss
        f (Right (Next a ms')) = Next a (h ms' mss)
      in
        fmap f (anyM mss (unscopeM ms))
switch (End a) mss = Next a (fmap join mss)
```

Listing 4.9: Definition of `switch` for `Stream`

The semantics of `switch` are to race `Stream s` (using `anyM`) until an event occurs on the nested `Stream` of `Streams mss`. `switch` reduces to calls of `anyM` where a `Stream` is raced against a `Stream` of `Streams`. If

`anyM` holds this property (where the order of races is arbitrary), then `join` too has this property.

$$anyM\ (anyM\ m1\ m2)\ m3 \equiv anyM\ m1\ (anyM\ m2\ m3)$$

Our `anyM` primitive is associative. Racing a nested race with an `AsyncM` is the same as racing an `AsyncM` with some nested race (it has no preference for order). This is because all three `AsyncM` (`m1`, `m2`, or `m3`) are raced simultaneously, instead of racing the nested race, then racing that against the next `AsyncM` in some order. Because `anyM` is associative, and `join` is defined in terms of `anyM`, `join` is associative and satisfies the seventh law.

5 Signals and Behaviors

Push streams are ideal for applications which must be low latency and asynchronous such as requesting batches of IoT data, visualizing data, and pushing data to a database. However, they are not a well suited abstraction for synchronous time-series computations such as KPIs. KPI computations fall into two categories: point-wise computations and windowed computations. Point-wise computations operate on single samples at a time, for example efficiency (input power divided by output power) or power calculations (voltage times current). Windowed computations operate on many samples at a time, and include windowed averages and THD calculations.

Data can be requested from web-servers using the `fetch` primitive of a `Stream` in batches. `Streams` cannot reliably be used for KPI computations, as multiple `Streams` may be combined for a single KPI (for example power uses both voltage and current). Each request of data batches will take a different amount of time to resolve leading to another problem. Moreover, time-series KPI computations are concerned with the data's sample time, instead of the system time of `Stream` events. The solution is to buffer batches into a `Channel`, using some demand driven interface (which can be composed), and coordinate related samples based off sample time. We

call this solution `Signals`. `Signals` are optimal for point-wise computations; however, we will find that `Signals` are not flexible enough to express signals with variable sampling rates and another abstraction is necessary. The signals with variable sampling rates can be represented as `Behaviors` which are related to `Signal`, but have a time parameter which allows for re-sampling.

5.1 Signal

`Signal` is an applicative functor which implements demand-driven computations. `Signals` can be composed with others using an `app` operator (`Signal (a -> b) -> Signal a -> Signal b`). They can be created from `Streams` via a `push2pull` offering interoperability between push-streams and pull-streams. `Signals` represent the real signals from IoT devices, or the resulting computations based on signals or other computations. Listing 5.1 defines `Signal` and the applicative instance (via `pure` and `app`).

```
1 class Signal {
2   // run :: (a -> IO()) -> IO()
3   constructor(run) {  this.run = run; }
4   fmap = f => new Signal(k => this.run(x => k(f(x))));
5   static pure = a => new Signal(k => k(a))
```

```
  app = gx => new Signal(k => this.run(f => gx.run(x => k(f(x)))))
}
```

Listing 5.1: Definition of `Signal`

`app` is used to apply a `Signal` of `a -> b` to another `Signal` of `a`. This operator is more powerful than `fmap` because it allows a mapping to occur that within the `Signal` type. `Signals` themselves can be composed with `app` which gives `Signals` enough power to express point-wise KPI computations. Listing 5.2 shows an example power KPI calculation with two input `Signals` `voltageS` and `currentS`. A curried multiplication function is lifted into the applicative, and applied twice to the `Signals`.

```
const power = Signal.pure(x => y => zip(x, y).map([x, y] => x * y))
  .app(voltageS)
  .app(currentS);
```

Listing 5.2: Example power KPI using applicative interface

`Streams` can be converted to `Signals` via the `push2pull` operator. The conversion uses a `Channel` to buffer events from the `Stream`. This causes a source of back-pressure. If the `Signal` does not pull data at a rate on average equal to the `Stream`'s push rate, data will consume memory until the `Signal`'s rate increases or the `Stream`'s rate decreases. This issue is addressed in the next section on `Behaviors`.

```
const push2pull = s =>
```

```
  AsyncM.liftIO(k => k(new Channel())).bind(c =>
    s.run(x => c.write(x)).forkM().bind(p =>
      AsyncM.pure(new Signal(k => c.read(k)))));
```

`Streams` can be converted to `Signals` via the `reactimate` operator. `reactimate` takes an amount of delay to wait between pulling from the input `Signal`, and produces a `Stream` who samples the `Signal` at that rate.

5.1.1 Event Signals

`Signals` are concerned with the timing of the data instead of the timing of events themselves. `Events` can be defined as `Signals` of time-value pairs (`Signal (Time, a)`) where time is the sampling period. `fetchE` takes a sample period `dt`, and a request, creates a `Stream` of values paired with the sampling period, and finally converts that `Stream` to a `Signal`.

```
// fetchE  :: Time -> (Time -> Stream (AsyncM a)) -> AsyncM (Event a)
const fetchE  = dt => reqFun => fetchES(dt)(reqFun).push2pull()
// fetchES :: Time -> (Time -> Stream (AsyncM a)) -> Stream (Time, a)
const fetchES = dt => reqFun => reqFun(dt).fetch().fmap(x => [dt, x])
```

Listing 5.3: Definition of `fetchE`

Using `fetchE` allows computations to be conscious of how much time

a sample represents. However, the data returned by `fetchE` could be improved by `unbatch`ing it. If `fetchE`'s request function returns a batch of samples, it should be converted into an `Event Signal` of samples instead of batches (`Signal (Time, a)` instead of `Signal (Time, [a])`). This allows the data to use the applicative interface completely, and to make resampling data easier.

```
class Signal {
  // unbatch :: Signal (Time, [a]) -> Signal (Time, a)
  unbatch = () => new Signal(
    k => {
      this._run(([dt, lst]) => {
            const a = lst.shift();
            if (lst.length > 0) this.cache = [dt, lst];
            k([dt, a])
      })
    }
  )
  _run = k => {
    const cache = this.cache;
    if (cache) {
      this.cache = null;
      k(cache)
    }
    else this.run(k);
```

```
19    }
20 }
```

Listing 5.4: Extensions to `Signal` class to implement `unbatch`

`unbatch` takes a batched `Event Signal`, and creates an unbatched `Event Signal` by iterating over the batch in order. The `_run` utility function keeps a stateful cache of batch values which have not yet been processed. In our pure functional version, `Signal` is made pure by returning `(a, Signal a)`. That is, running a `Signal` returns the value along with a new `Signal` which continues the computation. In JavaScript, creating so many additional object would be wasteful (for memory and garbage collection) which is why we chose a stateful approach. Notice how the sample period of the input `Event Signal` is propagated to the output `Event Signal` (each value is wrapped with the same sample period as the batch).

5.2 Behavior

`Behaviors`, similar to `Signals`, are applicative functors which represent KPI computations. They differ from `Signals` in the way that they can adjust the sampling period `dt` for windowing and re-sampling. Listing 5.5 shows the definition of the applicative instance for `Behavior`.

```
1 class Behavior {
```

```
  // run :: Time -> (a -> IO()) -> IO()
  constructor(run) { this.run = run; }
  fmap = f => new Behavior(dt => k => this.run(dt)(x => k(f(x))))
  static pure = a => new Behavior(_ => k => k(a))
  app = bx => new Behavior(dt => k =>
    this.run(dt)(f => bx.run(dt)(x => k(f(x)))))
}
```

Listing 5.5: Definition of `Behavior`

Computations can be expressed in terms of the `Behavior`'s `app` and `pure` functions. Listing 5.6 shows the same power calculation seen in Listing 6.4, except with `Behaviors`. Notice how `Behaviors` are not concerned with batches of data, so no `zip` is necessary on data before processing it, which makes the code clearer.

```
const power = pure(x => y => x * y)
  .app(voltageB)
  .app(currentB);
```

Listing 5.6: A power calculation using `Behavior`

`Behavior` can be created by using a `stepper` function to summarize an unbatched `Event Stream(Signal (Time, a))` via a summary function `[(Time, a)] -> a`. Listing 5.7 shows an implementation of `stepper`. Note that `Signal` uses a shared state between `unbatch` and `stepper` with the `signal.cache` list. This list propagates the remaining items to process

in the `Signal`, which makes `Signals` not reusable (must be instantiated for each use). This is a trade-off between performance and usability which had to be made for JavaScript, but can be avoided in languages which have cheaper costs for recursively constructing objects (such as Haskell). Also note that the recursion of `h` requires an implementation of JavaScript which optimizes recursive tail calls. Otherwise, an implementation could be written which uses trampolining.

```
// stepper :: Signal (Time, a) -> ([(Time, a)] -> a) -> Behavior a
const stepper = signal => summary => {
  const f = lst => (lst.length == 1) ? lst[0][1] : summary(lst);
  const h = lst => dt1 => k => signal._run(([dt2, a]) => {
    if (dt1 == dt2) k(f(lst.concat([[dt1,a]])))
    else if (dt1 < dt2) {
      k(f(lst.concat([[dt1,a]])));
      signal.cache = [dt2-dt1, a];
    } else h(lst.concat([[dt2,a]]))(dt1-dt2)(k);
  });
  return new Behavior (h([]));
}
```

Listing 5.7: Definition of `stepper`

`Behaviors` can be converted to `Streams` via the `reactimate` operator. `reactimate` uses a `Behavior`, a delay (in which the resulting `Stream` should pull data from the `Behavior`), and a sampling period `dt`.

The result is a `Stream` of the sampling period with the value (`Stream (Time, a)`). In an IoT system, `reactimate` is used for taking a KPI computation (as a `Behavior`) and sending it to a visualization, or to a database. The sampling period can be chosen which gives `reactimate` the ability to choose the resolution of data to receive. `Signals` also have a `reactimate` operator which provides a conversion into a `Stream`; however, it does not allow for a sampling period parameter, as it is fixed.

```
// reactimate :: Behavior a -> Int -> Time -> Stream (Time, a)
const reactimate = b => delay => dt => {
  const h = timeout(delay).bind(_ =>
    ifAliveM.bind(_ =>
      AsyncM.liftIO(k => b.run(dt)(k)).bind(a =>
        AsyncM.pure(Stream.next(Maybe.just([dt, a]), h)))));
  return Stream.next(Maybe.nothing, h)
}
```

`Behaviors` can be `upsampled`, `downsampled`, and `windowed`. `upsample` allows for a `Behavior`'s sample rate to be increased by some factor. Upsampling at a higher resolution than the `Stream` provides can only provide artificial values (e.g. repeating previous values) as seen in the behavior of list (`Behavior [a]`) output type where the length is equal to the number of repeated samples. To achieve higher sample rates of authentic data, a `Stream` could be used which sends web-requests to devices to provide data

at a higher rate (Section 6 handles this case). `downsample` decreases the `Behavior`'s sample period by a factor. It uses a summary function (similar to `stepper`) to implement the logic of down sampling, as some use cases may require an averaging of values while others could take the first or last sample. `window` uses a `Behavior` to create an `Event Signal` of batches with the given size (first parameter), and stride (second parameter). The stride indicates how many samples to skip between each window. `window` is useful for computing a THD KPI or moving averages where an exact time window is necessary. Listing 5.8 provides the type signatures (implementations are omitted for brevity) for the `upsample`, `downsample`, and `window` operators.

```
1 upsample :: Behavior a -> Int -> Behavior [a]
2 downsample  :: Behavior a -> Int -> ([(Int, a)] -> a) -> Behavior a
3 window :: Behavior a -> Int -> Int -> Time -> Event [a]
```

Listing 5.8: Type signatures for `upsample`, `downsample`, and `window`

`Behaviors` allow for the sampling period of computations to be separate from the sampling period of the captured data itself. This provides a layer of abstraction where `Streams` are concerned with getting the appropriate data at the devices full resolution, then `Behaviors` can re-sample the data at a rate in which the KPI computations can handle without producing excessive back-pressure in `Channels`. This re-sampling can be dynamic. For example,

it could be based off of the throughput of data by `counting` the number of incoming events of a `Stream`. We will see in the next section a design for an IoT analytics system which implements this method of scaling.

6 Use Case: IoT Analytics

Equipped with `AsyncM`, `Stream`, `Signal`, and `Behavior` we can address the use case of an IoT analytics system, where remote IoT devices provide access to their signals which can be accessed via web-requests. Those signals, which are to be captured in parallel, should support multiple KPI computations (e.g. to check each device's health). The resulting KPIs are then stored in a database for archival, and visualized on a dashboard for real-time display. Figure 6.1 shows the environment in which our IoT analytics system will be placed in. There is a web-server which provides a `/capture` endpoint which gives access to captured signals on several devices.

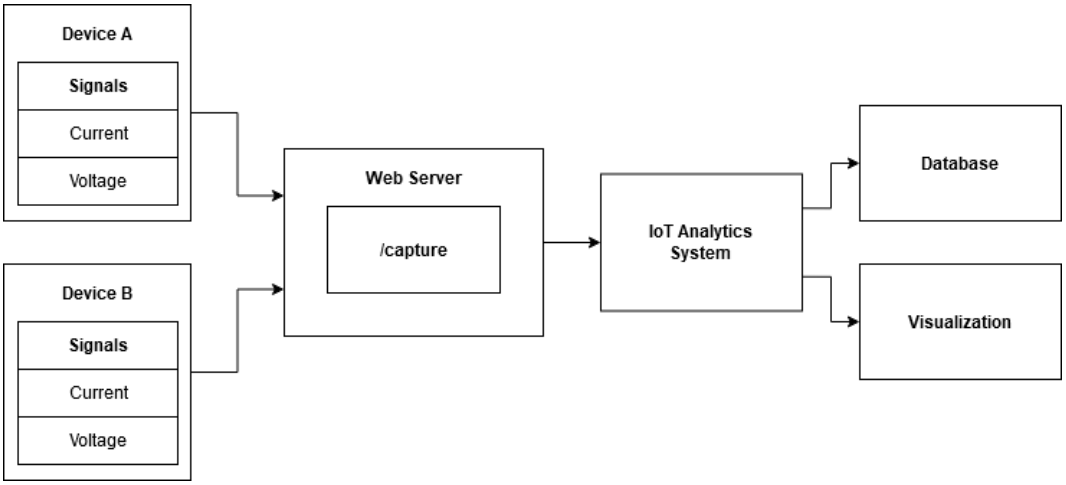

Figure 6.1: Block diagram of the IoT analytics system environment

We will use all of the components defined in this paper to pull data from

the web-server via a buffered `Stream` (a `Stream` which uses a `Channel`). Figure 6.2 shows a block diagram of the components involved of making a request to the web-server, converting the request to a `Stream`, then to a `Signal`, and using the applicative interface to compute the power of the signal. We use a new function `request` to make requests to a web-server, `fetch` to buffer requests in a `Channel`, `push2pull` to convert a `Stream` to a `Signal`, and `app` to compute KPIs.

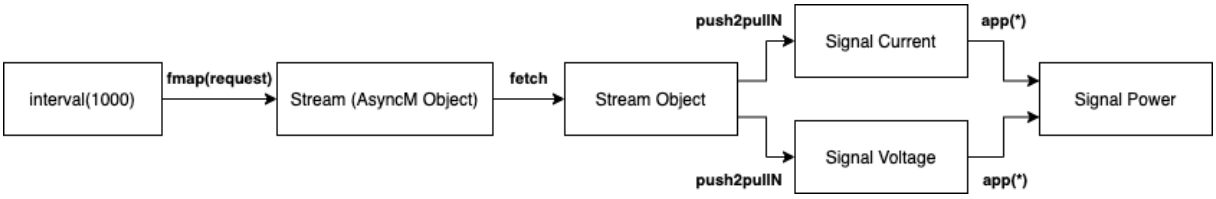

Figure 6.2: Block diagram of a power KPI

We demonstrate the composability of `Signals` by computing the THD of a current signal and joining it with the power KPI `Signal` to create a stream consisting of all our KPIs `Stream (Power, THD)`. The current `Signal` is converted to a `Behavior` by using a stepper function. A windowed computation is run on the current `Behavior` to produce a `Behavior` of THD. The `Behavior THD` is then converted to a `Signal THD` by `batch`, where it can be combined into a single `Signal` with the power calculation (`Signal (Power, THD)`). Finally, the `Signal` of KPIs can be `reactimated` into a `Stream`.

A `Stream` of KPIs is the ideal abstraction to use when sending KPIs to

Figure 6.3: Block diagram of a THD KPI

a database and visualization. A function `send` could be constructed which produces a `Stream` that performs the side-effect of sending the data.

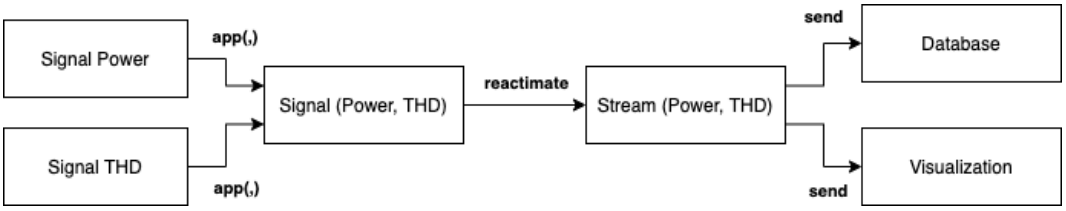

Figure 6.4: Sending KPIs to a database and visualization

As expressed in the introduction, there is an issue of back-pressure where if the incoming sampling rate of signals is too high for a system to handle, the KPI computations (specifically THD) may not be able to keep up with the devices. Recall that the average of the output samples must be equal to or less than the number of input samples. Otherwise memory usage grows indefinitely. We address this problem with another primitive `control` which drives the input `Stream` to ensure our system can scale back device sample rates.

6.1 Web Server

Our web server has one end-point `/capture` which accepts JSON requests with a `device_name`, `signal_name`, `sample_period`, and `window_size`. The `device_name` specifies which devices the server should capture a signal from, `signal_name` specifies which signal to capture (e.g. voltage or current), `sample_period` is the inverse of the sample rate, and `window_size` is how many milliseconds of data to capture. Listing 6.1 shows an example request (the details of the `request` function will be examined later) which asks for the voltage of device "a" where the signal should be captured once every ten milliseconds, over the course of a second. Once the capture is complete, the server replies with one hundred samples worth of data.

```
// Request
request({
  device_name: 'a',
  signal_names: ['voltage', 'current'],
  sample_period: 10,
  window_size: 1000,
}, k);

// Response
{ "voltage": <100 item list>, "current": <100 item list> }
```

Listing 6.1: An example request and response

Our example server is presented in Python using the minimal web framework, `Flask`, in Listing 6.2. The server simply accepts JSON requests, and directs their parameters to a `capture` function. In a simulation system, `capture` could run an electrical system simulation to generate signals. In a testing system, `capture` could read previously captured electrical signals from a file (or database). In a real-time system, `capture` could either have direct access to signals via hardware or have an additional software layer which buffers real signal data. In any case, the interface to `capture` could remain the same. Depending on the robustness requirements of the capturing processes, a request identifier (or timestamp) could be present with each request which allows for the server to cache samples for the client in the event that the client missed samples due to errors (e.g. client crashed, network link disconnected).

```
1  from flask import Flask, request, jsonify
2  from iot import capture
3
4  app = Flask(__name__)
5
6  @app.route('/capture', methods=['POST'])
7  def signals():
8      jo = request.json
9      device_name = jo['device_name']
10     signal_names = jo['signal_names']
```

```
    sample_period = jo['sample_period']
    window_size = jo['window_size']
    signals = capture(device_name, signal_names, \
                      sample_period, window_size)
    return jsonify(dict(zip(signal_names, signals)))
```

Listing 6.2: Web server which captures IoT signals

6.2 External Sources

Application of our system requires that we define several new utility functions which will act as a bridge between our library and external sources (the web server, database, and visualization). `post`, defined in Listing 6.3, uses the JavaScript `fetch` API (an HTTP client) to make JSON requests a web-server. Specializations are made for `request` which sends data to our web server (as defined above), and two more for sending to the database and visualization. It is expected that `body` has the appropriate fields that the web server accepts. `k` is a continuation to call once a JSON response is received which contains the signal data.

```
const post = url => body => k => {
  fetch(url, {
    method: 'POST',
    headers: {
```

```
5         'Content-Type': 'application/json',
6       },
7       body: JSON.stringify(body),
8     }).then(r => r.json().then(x => k(x)));
9 }
10 const request = post(CAPTURE_ENDPOINT_URL);
```

Listing 6.3: Definition of web request utility functions

6.3 Point-wise KPIs

Point-wise KPI calculations can be computed by creating a `Stream` of requests (which return batches of data), converting them to either `Signals` or `Behaviors`, then applying the computation with the `app` operator. In our case, each request can contain more than one signal. For example, requesting two signal names for voltage and current will yield two signals in a signal request (of type `Stream ([Voltage], [Current])`). When converted to a pull-stream, this format does not allow the applicative instance of `Signal` or `Behavior` to express the computation, so a utility function `push2pullN` is defined in Listing 6.4. `push2pullN` takes a list of signal names, a request (which returns some object who's fields match the given signal names), and returns an `AsyncM` of list of `Signal`. Now with two `Signals`, the applicative instance regains its composability (as `Signal`

`Voltage` and `Signal Current` can be combined naturally, while `Signal ([Voltage], [Current])` cannot)).

```
const powerKPI = req_stream =>
  push2pullN(['voltage', 'current'])(req_stream)
    .bind(([voltageS, currentS]) =>
      AsyncM.pure(Signal.pure(x => y =>
        zip(x)(y).map(([x, y]) => x * y))
                          .app(voltageS)
                          .app(currentS)));
}
// push2pullN :: [String] -> Stream Object -> AsyncM ([Signal a])
const push2pullN = names => s =>
  AsyncM.liftIO(k =>
    k(range(names.length)
      .map(_ => new Channel()))
      .bind(channels =>
    s.run(x => zip(names)(channels)
      .map(([n, c]) => c.write(x[n]))
    ).forkM().bind(p =>
      AsyncM.pure(channels.map(c => new Signal(k => c.read(k))))))
const zip = a1 => a2 => a1.map((k, i) => [k, a2[i]]);
const range = n => {
    let xs = [];
    for (let i=0; i < n; ++i) xs.push(i);
    return xs;
```

```
};
```

Listing 6.4: Definition of `powerKPI`

`powerKPI` takes a request stream to the `/capture` endpoint, converts the response to two `Signals` (a `Signal Voltage` and a `Signal Current`), applies the `Signals` together using `app` (with a zipped multiplication function on the entire batch at a time), and finally returns a `Signal Power`.

Figure 6.5 defines driver code which uses `powerKPI` to compute the power of device "a" by fetching requests with a `AsyncM` `requestM` (a wrapper around `request`). A clock sends requests to the web server for batches of one second long data (with a sampling period of ten milliseconds). `fetch` is used on the request `Stream` for its buffering capabilities, so if the request takes longer than one second to complete due to network latency, requests are run simultaneously to save time. `powerA` is a `Signal Power`, which can be `reactimated` into a `Stream`, and sent to an external source (e.g. database) to complete the end-to-end process of this KPI.

```
const requestM =
  device_name => signal_names =>
  sample_period => window_size =>
  new AsyncM(p => k =>
    request({ device_name, signal_names,
```

```
6                sample_period, window_size })(k));
7 const clock = timeout(1000).repeatS();
8 const requestS = req => clock.fmap(x => req);
9 const deviceS = req => requestS(req).fetch();
10 const requestA = requestM('a')(['voltage', 'current'])(10)(1000);
11 const powerA = powerKPI(deviceS(requestA));
```

Listing 6.5: Computing a power KPI using `powerKPI`

6.4 Windowed KPIs

Windowed computations be expressed with either `Signals` or `Behaviors` using their applicative interfaces. `Behaviors` have an advantage that when driven by a clock, it can produce batches with variable sample period (seen in Listing 6.7), contrasted with `Signals` which operate directly on batches of data. `Signals` can be more optimal for computations which strictly require fixed batch sizes (which don't scale with the system's performance) or do not rely on operating on time windows.

Listing 6.6 defines `thdE`, an `AsyncM (Signal (Time, THD))` which performs a THD calculation over a window of 1000 samples, which skips 1000 samples per window (due to the stride value), with a sampling period of 100 microseconds. A utility function `requestCurrent` is defined which converts a request to retrieve the current signal from the `/capture` endpoint

to a simple `Signal Current`. This example uses a fixed driver which has a sampling period of 100 microseconds. We will see in Section 6.5 an approach to make the sampling-period a variable.

```
const requestCurrent = dt =>
    requestS(requestM('b')(['current'])(dt)(1000))
      .fmap(m => m.fmap(o => o['current']));
const driver = Stream.fetchE(0.1)(requestCurrent);
const currentB = driver.fmap(s =>
    s.unbatch().stepper(x => x[0]));
const thdE = currentB.fmap(b =>
    b.window(1000)(1000)(0.1).fmap(thd))
```

Listing 6.6: Definition of `thdE`

6.5 Scaling Sampling Rate

Using `Behaviors` allows for expressing computations over variable sampling periods. This enables for windowed computations to remain consistent even when the input `Event Signal` changes its sampling rate. `speedControl` is a `Stream` primitive which allows a `Stream` to scale its output rate based on the `Stream`'s performance. It uses another primitive, `speed`, to `fold` over an `Event Stream`'s time, to compute the amount of time an `Event Stream` has processed. A time parameter `n` is accepted which represents

how much time the `Stream` should process. A data rate is computed by dividing the amount of time processed by `n`.

With the data rate provided by `speed`, `speedControl` can be defined which takes an additional function `Double -> Int` to determine if the rate is too slow, too fast, or acceptable. The function parameter takes the data rate, and returns `-1` if the rate is too slow, `1` if too fast, or `0` if it is acceptable. `speedControl` uses this function to create a `AsyncM Bool` which is `true` when the `Stream` is operating too fast, and `false` when it is too slow. As long as the `Stream` is operating at an acceptable rate, the `AsyncM` won't continue as no action must be taken. The `AsyncM Bool` can be used in the system to modify the amount of input data or the resolution of the data (depending on which effect is desired).

A more automatic, but less powerful primitive, `control`, can be used for monitoring the speed of web requests and responses, and adjust sampling periods to reach a stable throughput. The request and response `Streams` are `folded` to count the number of events emitted from the input and output over some time window. Based on this mechanism, `Streams` can dynamically alter their sampling periods when performance is not sustainable because `control` has access to a request function which parameterizes the sampling period.

`control` takes a request function (`Time -> (Stream (AsyncM a))`),

a duration, an initial sampling period, an adjustment function (`Bool -> Time -> Time`), and returns a `Stream` which automatically adjusts its own sampling period based off the input throughput and output throughput. Adjustments are made based off of the given `adjust` function, which receives a boolean indicating if the output rate is less than the input rate (which means an unsustainable request rate is taking place), the current sampling period, and returns a new sampling period. A simple `adjust` function is to multiply or divide the sample period by some constant factor (in this case 1.1), so that incrementally the sampling period moves towards a sustainable value. The duration tells `control` how long to measure throughput until `adjust`ing. The new `driver` function can be used to replace the previous THD KPI driver from Listing 6.6 to provide a scalable computation.

```
1  const driver = dt0 => Stream.fetchES(dt0)(requestCurrent)
2                              .multicast().bind(([s1, p1]) => {
3    return s1.push2pull().liftS().bind(g => {
4      const b = g.unbatch().stepper(avg);
5      const f = dt => {
6        const thdS = b.windowing(1000)(1000)(dt).fmap(thd).reactimate(dt);
7        return thdS.multicast().bind(([s2, p2]) => {
8          return Stream.next(Maybe.just(s2),
9                  s2.speedControl(dt)(x => x < 1.0 ? -1 : 0).bind(_ => {
10           p2.cancelP();
11           return AsyncM.pure(f(dt * 1.5));
```

```
        }))
      });
    };
    return f(dt0).join();
  });
});
```

Listing 6.7: Alternate driver for `thdS` that scales to `Stream` performance

6.6 External Sinks

The final step of the IoT analytics platform is to send data to its external sinks (e.g. databases, or visualizations). Listing 6.8 defines `kpiS`, a function which combines the KPIs we defined in the previous sections (power and THD) into a single JavaScript object to create a `Stream Object`. Notice how the `Signals` are first combined using the `app` function, and then converted into a `Stream` using `reactimate`. `sendKPIs` is the final interface function which takes a `send` function of type `Object -> IO()`, which receives each processed batch of KPIs and sends it to the relevant external sinks.

```
const kpiS = powerA.bind(power =>
  thdE.bind(thd =>
    AsyncM.pure(
```

```
4       Signal.pure(x => y => { 'power': x, 'thd': y })
5         .app(power)
6         .app(thd)
7         .reactimate(1000))));
8
9  function sendKPIs(send) {
10     kpiS.run(Progress.nil())(send)
11 }
```

Listing 6.8: Definition of `kpiS` and `sendKPIs`

7 Related Work

FRP [10] is a general framework for continuously changing behaviors. The power of FRP roots in its precise treatment of time, but unmitigated access to time leads to practical problems such as space/time leaks, violation of causality, and the difficulties in interacting with the outside world. A value in the present cannot depend on events of the future or have arbitrary memory of the past. The precise timing of events also limits concurrency, where asynchronous events occur in uncertain orders. Time is also relative depending the frame of reference. For IoT applications, the data time when the sensor signals are captured is different from the system time when the signals are analyzed. The proposed model avoids some of the problems in the treatment of time by separating data time from system time. Our push-stream handles asynchronous events in system time but it cannot access time like a behavior, has no arbitrary memory of the past, or depends on events of the future. The primary function of push-stream is to push IoT data to a buffer from which a pull-signal can pull data. The pull-signal has access to data time like a behavior, but it is first order and does not interact with asynchronous events. Pull-signals using the same data time are driven by the same clock to pull data synchronously from their respective buffers. Space

leaks may occur in the buffers, though it can be controlled through dynamic switching of the push-streams to ensure that the IoT data is processed in real-time.

Push vs Pull Classic FRP is pull-based, which detects events through polling in discrete time steps with the implication that the event latency depends on the step size and the behaviors are checked for possible switching every time step. Push-based FRP such as FrTime [6], Flapjax [20], Scala React [19], ReactiveX [1], and Elm [8] wait on event occurrences and only run when an event occurs. Though this provides timely responses to events and avoids re-computation when events do not occur, there may be glitches where the events propagated from the same source are not evaluated the same time. Solutions to this problem usually involve some kind of central planner that oversees the event dispatching or propagation. For example, in Elm, the first-order signals form a graph where a global dispatcher takes events from their sources and push updates through the graph. In Flapjax, where a dataflow graph is used, the graph nodes are updated based on topological order. In Monadic FRP [26], a program runs in a loop where at each iteration, it collects a set of future events, uses blocking IO to wait for one of the events to occur, and then starts the next iteration with the event. These types of solutions add complexity to their implementations. Our push-stream does

not prevent glitch in itself, but the data events pushed to the pull-signals will not have glitches when they are driven by the same clock.

Push-Pull Our work was influenced by push-pull FRP [9], which was a modernization of Fran [10]. Push-pull FRP models a behavior as a reactive time function, where a push-event can cause a pull-based behavior to switch to another one. `Reactive` is recursively defined as a value followed by a future `Reactive` where the racing of future values is implemented with threads. Our push-stream shares the same structure as `Reactive` and its monadic interface. The difference is that we implement the future value using `AsyncM`. Racing future values with `AsyncM` is lightweight, does not involve threads so that it is suitable for dynamic languages with event loops. Moreover, a stream is also run as an `AsyncM`, which can be shared through `multicast`.

Variations of classic FRP First-class behaviors can lead to space-time leaks and wasteful re-computation. Jeltsch [12] used phantom types to tie discrete push-signals to specific start time to avoid restarting a signal after switching and used memoization to avoid duplicated computation of signals. The paper's motivation is related to stateful signals such as the one that counts network traffic. Such a signal is recomputed if used in multiple places and gets restarted after switching. Our push-stream does not prevent this

type of issue through types since our model is aimed for dynamic languages. Instead, we can `multicast` a stream so that multiple uses will not cause re-computation, and switching will not cause restart. Krishnaswami [14] used a static approach to ensure that past values cannot be accessed and Patai [22] achieved similar goals by distinguishing streams and streams of streams at the type level. FRP Now [25] provided a variation to Fran that does not cause space leaks and also supports asynchronous IO. This approach erases past values with an optimization based on mutable memory. It handles asynchronous IO in a behavior by running the IO on a new thread, which passes the results as an event to the next round of the clock that runs the behavior.

Arrowized FRP Another type of solution to the space-time leak problem is to use the Arrows abstraction [18]. Yampa is an arrowized FRP variant which composes signal functions using arrow combinators where signals are not first-class values. A drawback of the arrowized approach is that it requires inputs and outputs be threaded throughout the entire program, and imposes a point-free style of programming [7]. Scalable FRP [4] improved on Yampa by providing an imperative implementation which has most of the expressiveness of Yampa with better performance. Arrowized FRP has been generalized into a monad stream function in Ivan Perez's Dunai [23], which

can model FRP signals and stateful reactive programming by stacking different monads. A later version called Rhine [3] introduced type-level clocks for processing data at different rates, where synchronous processes are run with an atomic clock on signal functions while asynchronous processes are run with schedules on resampling buffers. Rhine statically checks for correct composition involving clocks, and concurrent data is processed by threads that pass results through channels.

Dataflow Languages Before FRP, dataflow languages (e.g. Lucid [27]) and synchronous dataflow languages (e.g. Lustre [24] and SIGNAL [15]) provided an efficient and correct solution to real-time processing of signals. However, they are limited in power, as their dataflow graphs are static, and they do not support a form of first-class signals. In these models, signals use implicit time based on ordering of events, rather than an explicit continuous time or discrete time interval. Without switching operator, adjusting sample rates to external factors is not possible with these languages.

Concurrency monad The use of a continuation monad for a form of concurrency was demonstrated by Claessen [5] in the context of Haskell. This idea was later adopted by Li and Zidancwic [16] in their design for scalable network services that provides type-safe abstractions for both events and threads. They use a continuation monad to build traces that are scheduled

by event loops. This work uses a continuation monad to provide concurrency control for asynchronous IO that is run by event loops or threads.

8 Conclusion

We have created a model of reactive programming structured around IoT use cases by layering multiple composable abstractions on top of each other. First `AsyncM` was presented, offering a foundation for representing asynchronous computations. We demonstrated how `AsyncM`'s monadic interface is useful for combining chains of `bind` applications and other `AsyncM` together to create new behavior. The cancellation token structure which is implicit to `AsyncM` was presented, which allows `AsyncM` to have liveliness checkpoints by use of `ifAliveM`, and to race each other with `anyM` (among many other primitives).

We presented our design of push-streams with the `Stream` class. `Streams` provide low latency push capabilities where `Emitters` are used to broadcast events throughout the program. We defined a monadic instance for `Stream` which used a primitive called "switching" to add a dynamic structure to the program's dataflow graph. We showed how this behavior is well-behaved through its satisfaction of the monad laws.

Two pull-based abstractions (`Signal` and `Behavior`) were presented to serve two discrete use cases. `Signals` are for computations on batches of data which do not use the sample's time explicitly (such as point-wise

computations), and `Behavior` are for better abstracting time-bound computations (such as windowed computations). We demonstrated how both are applicative functors which can be composed via an `app` operator.

Finally, we applied our model to a real-world use case of an IoT analytics system. Primitives were added to our model (`control` and `speedControl`) for scaling the KPI computations with the performance of the system (to avoid excessive back-pressure). More primitives were added for interfacing with external sources and sinks, acting as a bridge between our model and the outside world (e.g. web servers, databases, and visualizations). We outlined the process from data capture to data display and storage, where data is fetched from external sources as a `Streams` of input, is converted to `Signals` and `Behaviors` (where KPIs are computed), then converted to a `Stream` of KPIs, and finally, sent to external sinks.

Bibliography

[1] Reactive extensions. `http://reactivex.io/`. Accessed: 2020-07-02.

[2] Rxjs switchmap. `https://rxjs.dev/api/operators/switchMap`. Accessed: 2020-07-02.

[3] Manuel Bärenz and Ivan Perez. Rhine: Frp with type-level clocks. *SIGPLAN Not.*, 53(7):145–157, September 2018.

[4] Guerric Chupin and Henrik Nilsson. Functional reactive programming, restated. In *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages 2019*, pages 1–14, 2019.

[5] Koen Claessen. A poor man's concurrency monad. *J. Funct. Program.*, 9(3):313–323, May 1999.

[6] Gregory H Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *European Symposium on Programming*, pages 294–308. Springer, 2006.

[7] Antony Courtney, Henrik Nilsson, and John Peterson. The yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 7–18, 2003.

[8] Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 411–422, New York, NY, USA, June 2013. ACM Press.

[9] Conal Elliott. Push-pull functional reactive programming. In *Haskell Symposium*, 2009.

[10] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 263–273, 1997.

[11] John Hughes. Generalising monads to arrows. *Science of computer programming*, 37(1-3):67–111, 2000.

[12] Wolfgang Jeltsch. Signals, not generators! In *Trends in Functional Programming*, 2009.

[13] Neelakantan R Krishnaswami. Higher-order functional reactive programming without spacetime leaks. *ACM SIGPLAN Notices*, 48(9):221–232, 2013.

[14] Neelakantan R. Krishnaswami. Higher-order functional reactive programming without spacetime leaks. *SIGPLAN Not.*, 48(9):221–232, September 2013.

[15] Paul LeGuernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire. Programming real-time applications with signal. *Proceedings of the IEEE*, 79(9):1321–1336, 1991.

[16] Peng Li and Steve Zdancewic. Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. *SIGPLAN Not.*, 42(6):189–199, June 2007.

[17] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, page 333–343, New York, NY, USA, 1995. Association for Computing Machinery.

[18] Hai Liu and Paul Hudak. Plugging a space leak with an arrow. *Electronic Notes in Theoretical Computer Science*, 193:29–45, 2007.

[19] Ingo Maier and Martin Odersky. Deprecating the observer pattern with scala. react. Technical report, 2012.

[20] Leo A Meyerovich, Arjun Guha, Jacob Baskin, Gregory H Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: a programming language for ajax applications. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 1–20, 2009.

[21] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop*, 01 2002.

[22] Gergely Patai. Efficient and compositional higher-order streams. In Julio Mariño, editor, *Functional and Constraint Logic Programming*, pages 137–154, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[23] Ivan Perez, Manuel Bärenz, and Henrik Nilsson. Functional reactive programming, refactored. *ACM SIGPLAN Notices*, 51(12):33–44, 2016.

[24] Daniel Pilaud, N Halbwachs, and JA Plaice. Lustre: A declarative language for programming synchronous systems. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (14th POPL 1987). ACM, New York, NY*, volume 178, page 188, 1987.

[25] Atze van der Ploeg and Koen Claessen. Practical principled frp: Forget the past, change the future, frpnow! *SIGPLAN Not.*, 50(9):302–314, August 2015.

[26] Atze van der Ploeg. Monadic functional reactive programming. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*, Haskell '13, page 117–128, New York, NY, USA, 2013. Association for Computing Machinery.

[27] William W Wadge and Edward A Ashcroft. *LUCID, the dataflow programming language*, volume 198. Academic Press London, 1985.

[28] Philip Wadler. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 347–359, 1989.

[29] Philip Wadler. Monads for functional programming. In *International School on Advanced Functional Programming*, pages 24–52. Springer, 1995.