Asynchronous Monad for Reactive IoT Programming

Tian Zhao
tzhao@uwm.edu
University of Wisconsin-Milwaukee
Milwaukee, Wisconsin, USA

Adam Berger
bergerab@uwm.edu
University of Wisconsin-Milwaukee
Milwaukee, Wisconsin, USA

Yonglun Li
yli@uwm.edu
University of Wisconsin-Milwaukee
Milwaukee, Wisconsin, USA

Abstract

Many industrial IoT applications process sensor data over distributed networks to monitor devices in real-time. Since the sensor telemetries are transmitted over networks as events, imperative and event-driven programs are often used to handle IoT data. However, the inverted control flow and shared global states of these imperative programs make them difficult to interface with synchronized computation on IoT data. This problem is further complicated for high-frequency data such as electric signals, which may require dynamic adjustment to data sampling rate to operate under the constraints of network and system.

In this paper, we propose a push-pull reactive programming model for IoT application to address this challenge. This model uses push-streams for asynchronous computation such as data capturing and user controls and uses pull streams for synchronized computation such as data analysis. This model is simpler than push-based models by avoiding the complexity of glitch prevention through re-sampling in pull-streams. It is also more flexible than pull-based models by allowing dynamic adjustment of the sampling rate to maintain real-time speed of the IoT computation. The push-stream has a monadic interface, which converts to a pull stream through buffering. A pull stream converts to a push-stream when driven by a clock. The dynamic switching of our streams is based on a monadic abstraction called AsyncM that uses continuation passing style and a form of cancellation token for asynchronous control. Our model is simple and can use threads or event callbacks for concurrency.

CCS Concepts: • **Computing methodologies → Concurrent computing methodologies**; • **Software and its engineering → Concurrent programming structures**.

Keywords: FRP, asynchronous programming, IoT

REBLS '20, November 16, 2020, Virtual, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8188-8/20/11…$15.00
https://doi.org/10.1145/3427763.3428314

ACM Reference Format:
Tian Zhao, Adam Berger, and Yonglun Li. 2020. Asynchronous Monad for Reactive IoT Programming. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS '20), November 16, 2020, Virtual, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3427763.3428314

1 Introduction

With the proliferation of the web, increasingly more electronics are being attached to networks. Industrial electrical systems which once required offline controls and displays are now able to communicate and be controlled through the web. The increased demand for these IoT devices causes an increased demand in their reliability too. Occasionally the sole purpose of being connected to the web is to run these reliability checks. We call these "checks" *key performance indicators* (KPI). KPIs are calculations performed on signals (such as voltages or current readings from sensors) which can tell an end-user the health of the system. As a simple example, a system could perform an efficiency calculation for an internet-enabled solar panel to determine whether the solar panel is performing according to the manufacturer's efficiency rating. If the panel is not, the user would know to further investigate the device.

However, KPIs are often more complex than just a low sampling-rate point-wise calculation. Some KPIs can push networks to their limits. For example, a total harmonic distortion (THD) calculation of a current signal requires discrete sampling at a rate that can reproduce the original continuous signal. Depending on the input signal's frequency, it could force the sampling rate to be 10Khz or more. On top of that, the calculation involves a moving window, whose window size can vary per signal. A complete system could have many devices each with many signals each with their own KPI calculations. To monitor the health of such a system, software must be able to notify users of changes to the KPIs and react to changes with low latency without causing excessive back-pressure.

A monitoring system that runs computations on high frequency data from multiple devices must be concurrent, reactive, and composable. Concurrency is necessary so that one device's computations do not block another device's, as well as to keep the latency low by processing multiple KPIs at once. Reactivity allows the system to scale dynamically in unstable environments. For example, if devices can set sampling rates on signals, during a decrease in network speed it would be advantageous to alter the sampling rate of the signals to avoid back-pressure. Composability is important to ensure program correctness is preserved when combining KPI processes on different signals.

1.1 Challenges of IoT Data Processing

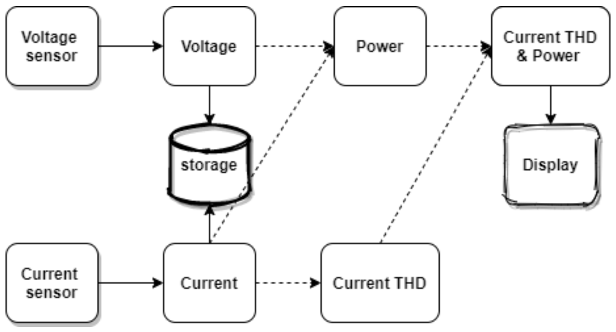

Figure 1. A program that monitors the current THD and the power of an inverter. The solid lines represent asynchronous tasks and the dashed lines represent synchronous tasks.

An industrial IoT application runs in real-time with large amount of data, performs frequent IO operations, and reacts to asynchronous events. Figure 1 shows a workflow that monitors the performance of a power inverter by capturing the voltage and current signals of the inverter, computing KPIs such as the power of the inverter and the THD of the current, storing the raw signals in a database, and displaying the KPIs on a dashboard. These tasks have different constraints on their speed and run with different sampling rate. For example, the signals may be captured and stored at a sampling rate of 20KHz but down-sampled to 10KHz for power and THD calculation, and the KPIs are displayed at 60Hz. To reduce latency, some tasks such as data capturing, storage, and display should be implemented asynchronously. However, there cannot be glitches for tasks such as inverter power, which must multiple the voltage and current sampled at the same time. Moreover, if the network and system load increases, some of the tasks may slow down, which can cause latency, memory leak or data loss. To deal with this, each task can update its sampling rate to maintain its speed, which means that some tasks need to re-sample their input.

1.2 Functional Reactive Programming

A reactive programming model is a natural choice to process signals and allow for dynamic reactions to data. Reactive programming is a broad area of study ranging from pull-based FRP (e.g. Fran [9], Yampa [6], monadic stream functions [1, 19], FRP Now [21]), to push-based designs (e.g. Reactive Extensions [16], FrTime [5], Flapjax [15], Scala React [14], Elm [7], Monadic FRP [23]), to hybrid model (e.g. push-pull FRP [8]). However, none of the existing models has the ideal characteristics that are specialized for IoT data that can scale sampling rates to its environment.

The core concepts of FRP [9] are behaviors and events, where a behavior is a continuous time function that can switch on events. In the existing FRP designs, the events and the behaviors have the same time domain. However, in IoT applications, the time parameter of IoT data analysis is the *data time* (i.e. when the data is sampled at the remote sensors, which may be in the arbitrary past if it is historical data), not the *system time* when the network events containing the IoT samples are received by the IoT programs.

The push-based models [7, 14, 15] use separate mechanisms such as signal graph to prevent harmful glitches by ensuring global ordering of events. While this works well for applications such as graphic interface, it is not suitable for IoT data. For example, the current and voltage data in Figure 1 should match their sampling time but the network events containing the data do not need to be globally ordered. Some pull-based models [1, 21] use scheduler or type-level clocks to combine asynchronous events and synchronous data at different sampling rate. However, they do not distinguish event time from the behavior time, which is necessary for adjusting sampling rate of the data in response to the changes in network and system load.

1.3 Proposed solution

Our solution is based on the observation that separate stages of IoT computation can be implemented by either push or pull models, which can be connected by simple mechanisms like buffering and timeout loops (or clocks). We define push-based streams for handling system events. The push-streams can be converted to pull-based streams through buffering. We then use those pull-streams for pure computations such as KPI calculations. The pull-streams are driven by clocks to form push-streams for asynchronous computation related to storage, display, and user interface.

With this design, the push-streams do not need to maintain global ordering of the asynchronous events since any synchronous computation that they are part of is driven by the same clock that pulls data from buffers, which ensures their ordering. The pull-streams do not interact with asynchronous IO directly since they only pull data from buffers.

Figure 2 illustrates a push-pull implementation of the workflow in Figure 1. The voltage (current) data is captured from the voltage (current) sensor using a push stream, which sends a stream of requests to capture the data and then emits the responses in the order in which the requests are sent. The stream sends data requests asynchronously to reduce latency and can adjust the sampling rate of the requested data to sustain its speed. The push streams are converted to pull streams (i.e. behaviors) through a buffering and a stepper function and the pull streams are used to compute power KPI. The conversion to pull-streams is to ensure that the voltage and current data can be synchronized and to allow re-sampling

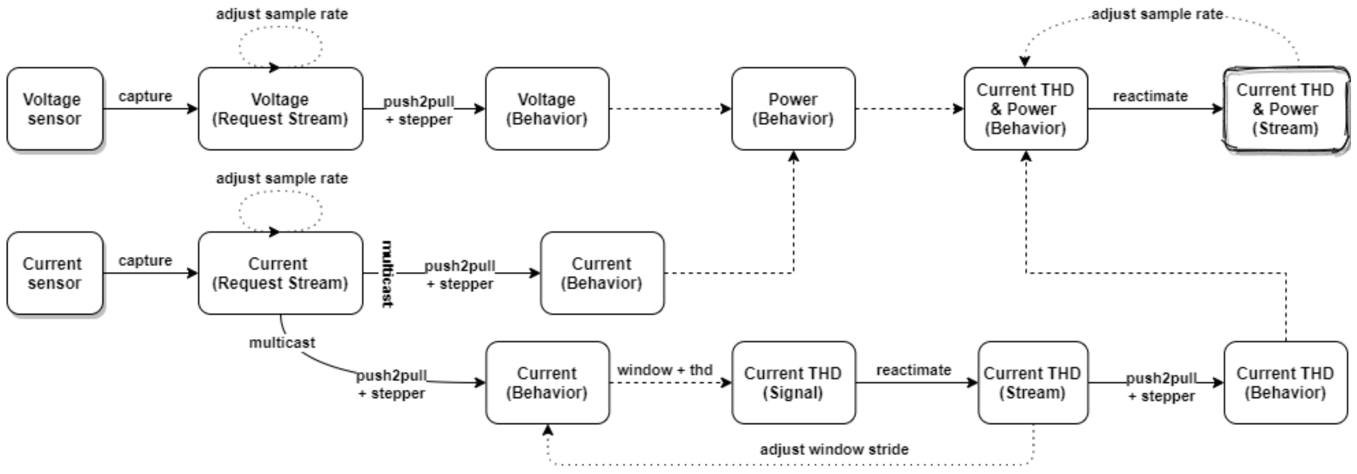

Figure 2. A use case of the push-pull model, where the solid lines represent push computation or push/pull conversion, the dashed lines represent pull computation, and the dotted lines represent dynamic adjustment to the push-streams.

since the sampling rate of the push streams may change over time. The current data is also used to compute the THD of the current. The THD calculation includes Fourier transform on a sliding window of the current, which is computationally intensive and may need to increase the stride of the sliding window to maintain real-time speed. Since the current data for THD and power is pulled independently, the push stream for current is multicast into two push-streams before converting to pull streams. The power and THD of current are paired together and sent to display but since the display is asynchronous, the power and THD behavior is converted to a push stream first with a reactimate function that samples the data around 60Hz. Similar method can be used to save the data to a storage (the details are omitted).

1.4 Contributions

Our design for push-stream is adapted from the *Reactive Value* abstraction proposed in push-pull FRP [8]. A reactive value is a value followed by another reactive value that occurs in the future. Reactive values are composable through a monadic interface. Joining higher-order reactive values requires racing two future reactive values. The original proposal was to use Haskell's threads. However, many IoT applications are implemented in dynamic languages that use event loops for concurrency. Also, our push streams can be shared through multicast or be converted to pull streams through buffering. This creates multiple streams with independent controls. A stream may continue to run even if its results are no longer used, which can cause memory leak.

Thus, our **first contribution** is a design of a reactive stream that represents future values using `AsyncM`, which is a form of continuation monad that implicitly carries a list of cancellation tokens. This design is lightweight and works with event loops or threads. Using `AsyncM`, we can race or cancel future values. If we cancel an `AsyncM` value, then any nested `AsyncM` values are cancelled as well, which is crucial for the dynamic switching of push-pull streams.

The **second contribution** is a hybrid push-pull design that supports high-throughput real-time data processing. The push-streams can make independent adjustment to their computation parameters to maintain real-time data speed. The pure functions on data can be implemented as synchronous computation on pull-streams.

For the rest of the paper, we first introduce the design of a cancellable continuation monad in Section 2, which is the foundation for our hybrid push-pull model. In Section 3, we describe the monadic interface of our push-based stream and how it can be used to process IoT data in real-time. In Section 4, we show how pull-based signals/behaviors can be derived from push-streams, how to run signals/behaviors as push streams, how to run computations with independently adjusted sampling rates. We discuss implementation and performance in Section 5 and related work in Section 6. The implementation is at https://github.com/tianzhao/asyncm.

2 Cancellable Continuation Monad

To implement an asynchronous and reactive model, we need an abstraction to represent asynchronous computation. The abstraction should not depend on threads so that it can be implemented in a single-threaded language like JavaScript. The abstraction should also support cancellation semantics so that a reactive stream can run independently, be shareable, and be stopped when it is no longer needed.

Our design is a continuation monad called `AsyncM`, which supports collaborative cancellation. Each AsyncM `m` runs with a progress value `p` so that if `p` is cancelled, `m` will stop at a checkpoint where it checks the cancellation status of

p. The cancellation semantics is modular as AsyncM is composed with monadic bind and the progress value is threaded through the bind operator.

AsyncM is defined in Listing 1, which is a function that takes a Progress value and a continuation callback k, and passes its asynchronous result to k. A Progress value is a list of cancellation tokens, each of which has the "MVar ()" type, which either is empty (meaning alive) or has the unit value () (meaning cancelled).

```
type AsyncM a = Progress -> (a -> IO ()) -> IO ()

type Progress = [MVar ()]
```

Listing 1. The definition of AsyncM

AsyncM is an instance of Functor, Monad, and MonadIO (where an IO action can be lifted to an AsyncM using liftIO) since it can be defined with monad transformers as follows.

```
newtype AsyncM a = AsyncM {
    runAsyncM :: ReaderT Progress (ContT () IO) a
} deriving (Functor, Applicative, Monad, MonadIO)
```

We can run an AsyncM for its side effect by calling it with an empty continuation.

```
-- run an AsyncM for its side effect
runM :: AsyncM a -> IO ()
runM m = m [] (\_ -> return ())
```

For example, the download function defined below downloads an HTTP document and saves it to a file, but if the download is not completed in 2 seconds, the downloaded document is not saved.

```
download :: String -> Path -> AsyncM ()
download url path = do r <- anyM (http url) (timeout 2000)
                       case r of Left str -> save path str
                                 Right _ -> return ()

timeout :: Int -> AsyncM ()
timeout n = \p k -> do async (threadDelay (n*10^3) >>= k)
                       return ()

http :: String -> AsyncM ByteString
http url = \p k -> do async (simpleHttp url >>= k)
                      return ()

save :: Path -> ByteString -> AsyncM ()
save path str = \p k -> writeFile path str >> k ()
```

The functions for racing and canceling AsyncMs are defined in Listing 2. Racing m1 and m2 is just to run them in parallel with a new progress value, where m1 and m2 should check whether the progress is still alive (with aliveM) and cancel it (with cancelM) before completion; otherwise the continuation k may be called twice.

```
anyM :: AsyncM a -> AsyncM b -> AsyncM (Either a b)
anyM m1 m2 = raceM (do x1 <- m1
                       commitM
                       return (Left x1))
                   (do x2 <- m2
                       commitM
                       return (Right x2))

commitM = ifAliveM >> cancelM
```

```
raceM :: AsyncM a -> AsyncM a -> AsyncM a
raceM m1 m2 = \p k -> do p' <- consP p
                         m1 p' k
                         m2 p' k

ifAliveM :: AsyncM ()
ifAliveM = \p k -> do b <- isAliveP p
                      if b then k () else return ()

cancelM :: AsyncM ()
cancelM  = \p k -> do b <- cancelP p
                      if b then k () else return ()
```

Listing 2. Concurrency control with AsyncM.

The functions for Progress values are in Listing 3. To cancel a Progress value, we put () in the head token, and to test whether a progress is alive, we check all of that value's cancellation tokens so an AsyncM can be cancelled in any context.

```
-- extend the progress p with a new cancellation token
consP :: Progress -> IO Progress
consP p = (:p) <$> newEmptyMVar

-- test whether a progress is cancelled
isAliveP [] = return True
isAliveP (v:p) =
    do b <- isEmptyMVar v
       if b then isAliveP p else return False

-- try cancel a progress and return true if succeeds
cancelP :: Progress -> IO Bool
cancelP (v:_) = tryPutMVar v ()
```

Listing 3. Make, test, or cancel a progress value.

As another example, download' attempts to download a document from a list of alternative URLs and if any download completes in 2 seconds, it is saved, and the program exits. Otherwise, the next URL in the list is tried.

```
download' :: [String] -> Path -> AsyncM ()
download' [] _ = return ()
download' (url:rest) path =
    do r <- anyM (http url) (timeout 2000)
       case r of Left str -> save path str
                 Right _ -> download' rest path
```

Since download' contains the checkpoint isAliveM, we can stop the entire download process (e.g. after 5 seconds) as follows.

```
download'' :: [String] -> Path -> AsyncM ()
download'' lst path =
    anyM (download' lst path) (timeout 5000)
```

Note that for a finitely nested anyM, the cost of ifAliveM is not significant compared to the IO within the anyM. However, if a function is called recursively within an anyM, the size of the progress grows with each recursive call and the cost of ifAliveM increases correspondingly. To avoid performance issues, long-running recursive calls should occur outside of anyM (e.g. the push-stream's monadic join in Section 3.2).

3 Push-based Reactive Stream

In many classic FRP implementations [8, 9], a behavior is a function from time to value, and an event source is a list of time/value pairs. A behavior can switch to new behaviors by reacting to the occurrence of events using a switch operator. Classic FRP samples the behavior values and detects event occurrences synchronously. However, IoT sensors are often distributed and synchronous sampling of sensor telemetries can cause unacceptable delay due to network latency. In addition, the overhead of synchronous sampling is unsuitable for high-frequency data such as the electrical signals that may be sampled at 10KHz or more. Thus, an asynchronous implementation is necessary for reactive IoT computations.

We adopt a push-based design for representing the stream of discrete events (similar to the `Reactive` value of push-pull FRP [8]). A value of the type "`Stream a`" contains a "`Maybe a`" value and a future stream "`AsyncM (Stream a)`".

```
-- the reactive stream
data Stream  a = Next (Maybe a) (AStream a)

-- the future reactive stream
type AStream a = AsyncM (Stream a)
```

The `Maybe` type is used for stream values because not all streams in IoT applications have sensible initial values when started. For example, if we set the initial value of a voltage stream to 0, there could be a KPI calculation which divides some signal by the voltage – producing a division by zero. In this case, `Nothing` should be the initial value, which will be skipped in KPI calculation.

```
-- a stream that starts with Nothing
repeatS :: AsyncM a -> Stream a
repeatS m = Nothing `Next` repeatA m

 -- an asynchronous stream by repeating m
repeatA :: AsyncM a -> AStream a
repeatA m = do a <- m
               ifAliveM -- cancellation checkpoint
               return (Just a `Next` repeatA m)
```

Listing 4. Make a stream by running an `AsyncM` repeatedly.

We can make a stream with `repeatS m` (Listing 4) that waits for the value of m, checks the progress status, and then repeats itself. For instance, `repeatS (timeout 1000)` is a stream of 1 second intervals.

For the rest of the paper, we assume there exists a function `getData` that captures sensor data by a sampling period in seconds and a duration in milliseconds.

```
getData :: String -- name of the sensor
        -> Int    -- capturing time in milliseconds
        -> Double -- sampling period in seconds
        -> AsyncM [Double] -- resulting data batch
```

For example, `repeatS (getData "Va" 1000 0.0002)` is a stream of data batches, where each batch contains 1000 milliseconds of voltage data with a sampling period of 0.0002 seconds (or 5000 Hz). Streams like this can capture data from multiple sensors without accumulative delay.

A stream can be run with `runS` that sends the stream events to a function `k` which has side effects (e.g. saving data), where the `Nothing` events are skipped.

```
runS :: Stream a       -- stream to run
     -> (a -> IO ())   -- side-effecting function
     -> AsyncM ()      -- resulting computation

runS (a `Next` ms) k =
     do ifAliveM       -- cancellation checkpoint
        liftIO (f a)   -- run 'k' with event 'a'
        s <- ms
        runS s k
  where -- no effect for the Nothing event
        f Nothing = return ()
        f (Just x) = k x
```

Listing 5. Run a stream with a callback function.

3.1 Functor

`Stream` is a functor, where its `fmap` method (`<$>`) recursively applies `f` to the stream events. Since `a` is a `Maybe` value, "`f <$> a`" is `Nothing` if `a` is `Nothing` and is "`Just (f x)`" if `a` is "`Just x`".

```
instance Functor Stream where
  fmap f (a `Next` ms) = (f <$> a) `Next` (fmap f <$> ms)
```

For example, the code below calculates the KPI of a sensor signal, saves it to a database, and at the same time, displays it on a user interface. The function `kpi` calculates a KPI value for every second of sensor samples, and `forkM` starts an `AsyncM` without waiting for it to complete.

```
let s = kpi <$> repeatS (getData "Va" 1000 0.0002)
in do forkM (runS s writeDB)
      forkM (runS s display)

-- run m with a new progress p' and return p' immediately
forkM :: Async a -> Async Progress
forkM m = \p k -> do -- p' extends p with a new token
                     p' <- consP
                      -- discard the result of m
                     m p' (\_ -> return ())
                     k p' -- return p' right away
```

Listing 6. Run an `AsyncM` and return its progress value.

However, this example has a flaw since it captures sensor data and computes its KPI twice. A better version below avoids recomputing the stream `s` by broadcasting its values to an Emitter `e`, and then events from `e` are received by two separate streams for saving to a database and displaying.

```
let s = kpi <$> repeatS (getData "Va" 1000 0.0002)
in do (e, _) <- broadcast s
      let s' = receive e
      forkM (runS s' writeDB)
      forkM (runS s' display)
```

The call "`broadcast s`" creates a new emitter `e`, emits each value of `s` to `e`, and returns `e`, whose values are received by the stream "`receive e`". Since we use `runS` to broadcast the stream `s`, the `Nothing` events in `s` are skipped.

```
-- broadcast the events of a stream to an emitter
broadcast :: Stream a -> AsyncM (Emitter a, Progress)
broadcast s = do e <- liftIO newEmitter
```

```
                 -- keep emitting events of s to e
                 p <- forkM (runS s (emit e))
                 -- return progress for cancellation
                 return (e, p)

receive :: Emitter a -> Stream a
receive e = Nothing `Next` h
  where h = do a <- listen e -- listen for event on e
               ifAliveM -- cancel checkpoint
               return (Just a `Next` h)
```

Listing 7. Broadcast a stream to an emitter to enable sharing.

Both broadcast and receive can be cancelled since both contain the checkpoint isAliveM (where the checkpoint of broadcast is in runS). In addition, broadcast can be cancelled explicitly through the Progress value it returns.

The function listen takes an emitter e and returns an AsyncM that yields the future value of e by registering a callback on e. The call "emit e a" writes a to e, fires any callbacks registered on e, and then clears the callback list.

```
data Emitter a = Emitter (MVar a) -- the previous event
                         (MVar [a -> IO ()]) -- callbacks

newEmitter = pure Emitter <*> newEmptyMVar <*> newMVar []

-- listen for an event on an emitter
listen :: Emitter a -> AsyncM a
listen (Emitter _ kv) =
    -- add 'k' to the callback list of the emitter
    \_ k -> modifyMVar_ kv (\lst -> return (k : lst))

-- emit an event 'a' to the emitter
emit :: Emitter a -> a -> IO ()
emit (Emitter av kv) a = do
    tryTakeMVar av  -- clear previous event
    putMVar av a    -- set current event
    lst <- swapMVar kv [] -- clear callback list
    forM_ lst (\k -> k a) -- fire registered callbacks
```

Listing 8. Create, listen, or emit an event to an emitter.

3.2 Monad

Stream has a monad interface defined in Listing 9, where 'return x' is a stream with just x followed by neverM, which is an AsyncM that never completes. The bind operator >>= is defined with a join function that flattens a stream of stream.

```
instance Monad Stream where
   -- neverM is an AsyncM that never completes
  return x = Just x `Next` neverM

  -- join the Stream of Stream 'fmap k s'
  s >>= k  = join (fmap k s)

join :: Stream (Stream a) -> Stream a
join (Nothing `Next` mss) = Nothing `Next` (join <$> mss)
join (Just s  `Next` mss) = switch s mss
```

Listing 9. Monad interface of reactive stream

The join function has two cases. In the first case, the inner stream is Nothing and we skip it, and continue to join the future outer stream. In the second case, we use the switch function in Listing 10 to run the inner stream s until the future outer stream mss emits.

```
1  switch :: Stream a -> AStream (Stream a) -> Stream a
2  switch (a `Next` ms) mss = a `Next` (h ms =<< spawnM mss)
3    where h ms mss =
4      let
5          f (Left ss) = join ss
6          f (Right (a `Next` ms')) = a `Next` h ms' mss
7      in
8          f <$> anyM mss (unscopeM ms)
```

Listing 10. Switch the inner stream when the future outer stream emits.

The switch function races the future inner stream ms with the future outer stream mss (line 8) using anyM. If mss wins the race with a new outer stream ss, then we abandon ms and continue with "join ss" (line 5). If ms wins the race with an event a, then a is emitted and we continue the race of the next future inner stream ms' with mss (line 6).

The switch function calls the local function h (defined at line 3) to perform the race. If ms wins the race, the call reduces to "h ms' mss" (line 6), where ms' is the next future inner stream. Notice that mss is reused in the recursion, which is problematic for two reasons. First, mss may contain an asynchronous request (e.g. a timeout) that returns after the response is received. In this case, each run of mss will start a new request with a new completion time. Second, mss may be a composite AsyncM (e.g. another race), so restarting it wastes runtime. Therefore, when switch starts, it runs "h ms =<< spawnM mss" at line 2 to cache the result of mss using "spawnM mss", which starts mss and returns an AsyncM that waits for the result of mss.

We race ms and mss in "anyM mss (unscopeM ms)" (line 8), where unscopeM runs ms with the progress outside anyM. This is needed since anyM cancels its progress (and any AsyncM running with the progress) once the race completes but there may be a pending AsyncM started inside ms (e.g. using spawnM) that has not completed when ms wins the race.

```
1  neverM :: AsyncM a  -- never complete by not calling 'k'
2  neverM = \p k -> return ()
3
4  -- avoid cancelling 'm' inside a race
5  unscopeM :: AsyncM a -> AsyncM a
6  unscopeM m = \p k -> m (tail p) k
7
8  -- start m and return an AsyncM waiting for m's result
9  spawnM :: AsyncM a -> AsyncM (AsyncM a)
10 spawnM m = \p k -> do e <- newEmitter
11                       m p (emit e)
12                       k (wait e)
13
14 -- try reading a future event from the emitter and
15 -- register a callback if the event is not available
16 wait :: Emitter a -> AsyncM a
17 wait (Emitter av kv) = \p k ->
18              do a <- tryReadMVar av
19                 case a of Just x -> k x
20                           Nothing -> swapMVar kv [k]
```

Listing 11. Auxiliary functions to run AsyncM.

The function spawnM in Listing 11 starts an AsyncM with a callback that writes its result to an emitter e at line 11, and then returns "wait e" at line 12 to wait for the result. To avoid memory leaks, "`wait e`" only keeps one callback in `e` (using `swapMVar` at line 19) since `wait e` (returned from `spawnM`) can be started many times, but only the last instance is needed.

Use of Monad Interface. Our monadic interface is convenient for switching. For example, we can define a stream below that displays the KPI of a sensor chosen by users, where `sensorSource` is an `AsyncM` that waits on user input to choose the sensor for KPI calculation.

```
let s = do src <- repeatS sensorSource
           repeatS (getData src 1000 0.0002)
in runS (kpi <$> s) display
```

Using `switch`, we can define functions such as `stopS` that stops a stream after `n` milliseconds.

```
stopS n s = s `switch` do timeout n
                          return (Nothing `Next` neverM)
```

3.3 Buffered Stream

There is usually some latency between capturing IoT sensor data and the KPI calculation. For example, it may take 2 seconds to run "`getData "Va" 1000 0.0002`" to retrieve 1 second of samples (with 1 second of network delay). If we only send a request after receiving the response from the previous request, then the samples between two successive requests will be lost. We may be able to hide this latency using a queue and two streams: the first stream sends the requests at a regular interval and pushes the future response for each request to the front of the queue. The second stream extracts the future response from the end of the queue and waits for it to resolve. If the latency is not due to the lack of network bandwidth, it may be able to be hidden this way.

The example below creates a `request` stream at line 3 that sends a data request every second. Another stream `response` at line 5 is created using the `fetchS` function (Listing 12).

```
1 let clock = repeatS (timeout 1000)
2     -- request  :: Stream (AsyncM [Double])
3     request = getData "Va" 1000 0.0002 <$ clock
4     -- response :: Stream [Double]
5     response = fetchS request
6 in  runS (fmap kpi response) display
```

The `fetchS` function in Listing 12 spawns an `AsyncM` for each request in the `request` stream, and writes it to a channel (line 3). It returns a new stream (line 6) that reads each `AsyncM` from the channel and waits for it to yield a result. The workflow of `fetchS` is shown in Figure 3.

```
1  -- take a request stream and return a response stream
2  fetchS :: Stream (AsyncM a) -> Stream a
3  fetchS sm = Nothing `Next` do
4      c <- liftIO newChan
5      forkM $ runS (sm >>= liftS . spawnM) (writeChan c)
6      repeatA $ join $ liftIO (readChan c)
7
8  -- lift an AsyncM to a Stream
9  liftS :: AsyncM a -> Stream a
10 liftS m = Nothing `Next` (m >>= return . return)
```

Listing 12. Fetch data by sending requests in a stream.

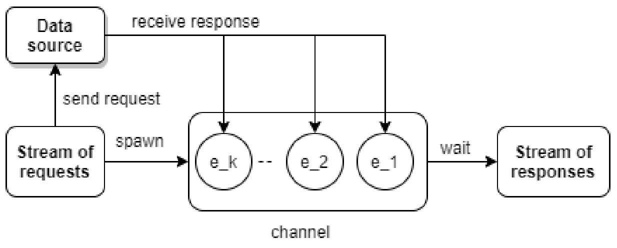

Figure 3. Illustration of the workflow of `fetchS`.

In Listing 12, the expression "`sm >>= liftS . spawnM`" at line 5 converts the stream of requests `sm` to a stream of future responses using `spawnM`. At line 6, "`liftIO (readChan c)`" has the type `AsyncM (AsyncM [Double])`, which is flattened by `join` to `AsyncM [Double]` that reads a response from the channel and waits for it to complete.

3.4 Real-time Push Stream

The `fetchS` function can help reduce the latency and increase the throughput of IoT data retrieval. However, the data retrieval throughput is also subject to limitations such as network bandwidth. For the previous example, if the amount of data being transmitted exceeds the network bandwidth, then the latency between IoT data capturing and KPI calculation will gradually increase, and the KPI calculation will no longer be in real-time. One way to prevent this problem is to monitor the speed of the response stream and if it is lower than the speed of the request stream, then we reduce the amount of data for each request (e.g. by lowering the sensor sampling rate).

To measure the speed of the response stream, we can use the `countS` function in Listing 13 to count the events of a stream during a time interval. The `countS` function is based on `foldS`, which folds a stream of functions for a duration, and returns the last event.

```
-- count the number of events of s in n milliseconds
countS :: Int -> Stream a -> AsyncM Int
countS n s = foldS n 0 ((+1) <$ s)

-- fold s for n milliseconds with the initial value c
foldS :: Int -> a -> Stream (a -> a) -> AsyncM a
foldS n c s = do r <- lastS (accumulate c s) (timeout n)
                 return $ fromJust r
```

Listing 13. Count the events of a stream in a period of time.

The function `foldS` calls `accumulate` (Listing 14) to recursively apply a stream of functions to an initial value, then uses `lastS` to stop the accumulation and returns the last value. Note that `countS` does not include `Nothing` since `accumulate` emits the same value if `f` is `Nothing`.

```
-- accumulate s with the initial value a
accumulate :: a -> Stream (a -> a) -> Stream a
accumulate a (f `Next` ms) =
    let a' = maybe a ($ a) f
    in Just a' `Next` (accumulate a' <$> ms)
```

```
-- return the last event of s when m returns
lastS :: Stream a -> AsyncM () -> AsyncM (Maybe a)
lastS s m = spawnM m >>= h s
  where h (a `Next` ms) m = do
          r <- anyM ms m
          case r of Left s -> h s m
                    Right () -> return a
```

Listing 14. Accumulate a stream and take a snapshot.

Using countS, we can implement a stream that makes runtime adjustments to sustain its speed. For example, the getBatch function in Listing 15 returns a stream of sample batches from a data source, where the sampling rate is adjusted based on the request/response speed. The function req_fun (line 4) takes the sampling-period dt and returns a request stream for 1 second of data sampled at 1/dt Hz. The call to controlS at line 6 compares the number of requests sent and responses received within 10 seconds and adjusts dt by a ratio of 1.1 if the numbers are not equal.

```
1 getBatch :: String -> Stream (Double, [Double])
2 getBatch src =
3    let clock = repeatS (timeout 1000)
4        req_fun dt = getData src 1000 dt <$ clock
5        adjust b dt = if b then dt*1.1 else dt/1.1
6    in controlS req_fun (10^4) 0.0002 adjust
```

Listing 15. A data stream with adjustable sampling period.

The function controlS (Listing 16) takes a request function req_fun, a duration, a request parameter dt, an adjustment function, and outputs a response stream that can self-adjust based on the relative request/response speed. The controlS function builds a stream of stream at line 23 with a response stream that runs until a new stream is emitted from mss (when the speed of request and response differs). The nested stream is joined at line 6, and the request parameter dt is added to the response stream at line 23. The basic workflow of controlS is shown in Figure 4.

```
1  controlS :: (t -> Stream (AsyncM a)) -- request function
2           -> Int                      -- duration (in ms)
3           -> t                        -- request parameter
4           -> (Bool -> t -> t)         -- adjust function
5           -> Stream (t, a)            -- result stream
6
7  controlS req_fun duration dt adjust = join $ h dt
8    where h dt = do
9      -- multicast creates shareable streams
10     (request,  p1) <- multicast $ req_fun dt
11     (response, p2) <- multicast $ fetchS request
12
13     let mss = do -- account for initial latency
14                 timeout duration
15                 -- measure request/response speed
16                 (x, y) <- allM (countS duration response)
17                                (countS duration request)
18                 if x == y then mss
19                 else do -- cancel multicast streams
20                         liftIO (cancelP p1 >> cancelP p2)
21                         -- restart adjusted stream
22                         return $ h $ adjust (x < y) dt
23     -- make a stream of response stream
24     Just ((,) dt <$> response) `Next` mss
25
26 -- run two AsyncMs and wait for both to return
27 allM :: AsyncM a -> AsyncM b -> AsyncM (a, b)
```

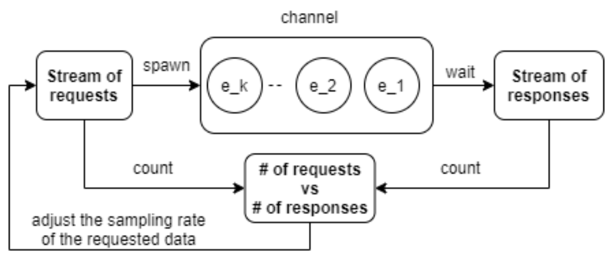
channel
Stream of requests
spawn
e_k
--
e_2
e_1
wait
Stream of responses
count
# of requests vs # of responses
count
adjust the sampling rate of the requested data

Figure 4. Illustration of the workflow of controlS.

Listing 16. Make a request stream that can adjust to the speed of the response.

The controlS function uses the multicast function in Listing 17 to make shareable streams such as request and response. However, the multicast streams will keep running even if they are not used. To avoid redundant computation, we cancel them by cancelling their Progress values at line 19 (Listing 16) when the stream switching occurs.

```
multicast :: Stream a -> Stream (Stream a, Progress)
multicast s =
    Nothing `Next` do (e, p) <- broadcast s
                      return $ return (receive e, p)
```

Listing 17. Make a shareable and cancellable stream.

4 Pull-based Data Stream

While IoT data is often captured, stored, and displayed asynchronously, it is more convenient to analyze IoT data (e.g. calculating KPI) synchronously. For example, suppose we capture the voltage and current of an inverter in two push streams, whose data arrive in the following order.

```
Time     0    1     2     3     4     5
voltage  V1         V2          V3
current       I1          I2          I3
```

If we use the monad interface of Stream to compute the inverter power, we will produce incorrect output below.

```
Time     0    1     2     3     4     5
power       V1*I1 V2*I1 V2*I2 V3*I2 V3*I3
```

Since the voltage and current of the same data time may arrive at different system times, we have to match the voltage and current events by their sampling timestamps, which is costly. But, if the two streams are buffered, they can be processed synchronously by pulling their events from the respective buffers to yield the correct output.

```
Time     0    1     2     3     4     5
power       V1*I1       V2*I2       V3*I3
```

For this purpose, we define the Signal type in Listing 18, which is a recursive data structure that provides discrete data events on demand. Signal is an Applicative Functor, where the app operation gf <*> ga is defined as pairwise applications of the function signal gf to argument signal ga.

```
-- Pull-based stream
newtype Signal m a = Signal {runSignal :: m (a, Signal a)}

instance (Monad m) => Functor (Signal m) where
  fmap f g = Signal $ do (a, g') <- runSignal g
                         return (f a, fmap f g')

instance (Monad m) => Applicative (Signal m) where
  pure a = Signal $ return (a, pure a)

  gf <*> gx = do (f, gf') <- runSignal gf
                 (x, gx') <- runSignal gx
                 return (f x, gf' <*> gx')
```

Listing 18. The definition of pull-based `Signal` stream.

4.1 Push Pull Conversion

The function `push2pull` (Listing 19) converts a push-stream `s` to a pull-signal by sending the events of `s` to a channel buffer and returning a signal that reads from the channel. We run a signal using the `reactimate` function that returns a `Stream` that emits the signal values with a fixed delay.

```
push2pull :: Stream a -> AsyncM (Signal IO a)
push2pull s =  do
     -- make an event buffer
     c <- liftIO newChan
     -- write events of 's' to the buffer
     forkM $ runS s $ writeChan c
     -- read events from the buffer
     let g = Signal $ do a <- readChan c
                         return (a, g)
     return g

-- run signal with event delay
reactimate :: Int -> Signal IO a -> Stream a
reactimate delay g = Nothing `Next` h g
  where h g = do timeout delay
                 ifAliveM
                 (a, g') <- liftIO $ runSignal g
                 return $ Just a `Next` h g'
```

Listing 19. Conversion between Stream and Signal.

It is more convenient to calculate KPI with `Signals`. For example, the function `power` below computes the power of an inverter and returns the results in a stream of batches. Each event in "`power 1000 0.0002`" is a batch of 5000 samples since the sampling period is 0.0002 seconds, and each batch contains 1000 milliseconds of data.

```
power :: Int -> Double -> Stream [Double]
power duration dt = do
   clock <- multicast $ repeatS (timeout duration)
   -- a stream of requests for a given source
   let req src = getData src duration dt <$ clock

   -- convert a request stream to a signal
   let req2signal s = liftS (push2pull (fetchS s))

   voltage <- req2signal (req "Va") -- Signal IO [Double]
   current <- req2signal (req "Ia") -- Signal IO [Double]

   let power = pure (zipWith (*)) <*> voltage <*> current

   reactimate 1 power -- run Signal as a push stream
```

Signal of data batches is not suitable for computation that needs to change sampling rate or operate on a moving window of data samples. We can create the voltage (and current) streams by calling `fetchS` on a stream of data requests that capture 1 sample per request. However, this is inefficient due to the overhead of sampling and transmission. A more realistic solution is to fetch data as a signal of batches and then convert the batch signal into sample signal.

In Section 3.3, we gave an example of a stream of voltage batches with varying sampling rate to match the speed of request and response. This creates a problem when we have a stream of voltages and a stream of currents with possibly different sampling rates. We need to re-sample the data before inverter power can be calculated, which requires the sampling period be included in the signal events. Moreover, KPIs such as THD (which can measure the quality of an inverter signal) take the sampling period as input. For these reasons, we need to have signals with time.

4.2 Event

`Event` is a signal of sampling-period and value pairs. Since the sensor signal converted from a request stream may have varying sampling rate, the sampling period should be included in the signal, which forms a sequence of delta-time and value pairs.

```
type DTime = Double -- sampling period

-- Event is a signal of sampling-period and value pairs
type Event a = Signal IO (DTime, a)
```

IoT data can be retrieved as an `Event` with a given sampling period using the function `fetchE` in Listing 20, which uses `fetchS` to retrieve a stream of the responses to the requests sent with a sampling period `dt`, and then pairs the results with `dt`.

```
-- fetch Event Signal
fetchE :: (DTime -> Stream (AsyncM a)) -- request stream
       -> DTime                        -- sampling period
       -> AsyncM (Event a)             -- event signal
fetchE req_fun dt =
    push2pull $ (,) dt <$> fetchS (req_fun dt)
```

Listing 20. Convert a request Stream to an Event.

An `Event` of batches can be easily converted to an `Event` of samples using the `unbatch` function in Listing 21, where the sampling period of a batch is repeated in the unbatched samples. When the sampling period of a batch changes, the corresponding sampling period of the unbatched samples changes as well.

```
-- flatten the Event of batches to an Event of samples
unbatch :: Event [a] -> Event a
unbatch eb = Signal $ do
     ((dt, b), eb') <- runSignal eb
     h dt b eb'
  where h _ [] eb' = runSignal $ unbatch eb'
        h dt (a:b) eb' =
          return ((dt, a), Signal $ h dt b eb')
```

Listing 21. Convert batch Event to sample Event.

An event of samples is not only easier for KPI calculations, but also allows re-sampling so that IoT data of varying sampling rate can be used in a computation with a constant sampling rate through re-sampling.

4.3 Behavior

To support re-sampling, below we define the Behavior type as a signal that takes a sampling-period as input. Each value of the behavior is the summary of some sample values over a given sampling period.

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

stepper :: ([(DTime, a)] -> a) -- summary function
        -> Event a             -- input Event
        -> Behavior a          -- output Behavior

stepper sum ev = Signal $ ReaderT $ \t -> h [] t ev
   where h lst t ev = do
         ((t', a), ev') <- runSignal ev
         if (t == t')
         then return (sum ((t,a):lst), stepper sum ev')
         else if (t < t')
              then return (sum ((t,a):lst), stepper sum
                  $ Signal $ return ((t'-t, a), ev'))
              else h ((t',a):lst) (t-t') ev'
```

A stepper function can be defined to convert an Event to a Behavior with the help of a summary function that summarizes a sequence of time/value pairs to a value. The idea is to repeat the sample value of the Event for up-sampling (when the sampling period of the Behavior is shorter than that of the Event), and use the summary function for down-sampling (when the sampling period of the Behavior is longer). The summary function depends on the IoT data, which may be numeric values or discrete states.

Using the Behavior abstraction, we can calculate inverter power using two streams with variable sampling rates. For example, below is a code snippet that first obtains two self-adjusting streams by calling getBatch (Listing 15) and then converts the streams to behaviors using stream2behavior (Listing 22). The workflow of computing inverter power is shown in Figure 5, where getBatch produces voltage/current batches Stream (DTime, [Double]), which are streams of sampling period and data batch pairs.

```
do voltage <- stream2behavior (getBatch "Va")
   current <- stream2behavior (getBatch "Ia")
   let power = pure (*) <*> voltage <*> current
   runS (reactimateB 1 0.001 power) display
```

The stream2behavior function goes through the steps of converting a push-stream to an Event of batches, to an Event of samples, and to a Behavior.

```
avg :: [(DTime, Double)] -> Double -- weighted average

stream2behavior :: Stream (DTime, [Double])
                -> AsyncM (Behavior Double)
stream2behavior s =
                  (stepper avg . unbatch) <$> push2pull s
```

Listing 22. Convert a batch stream to a behavior

We run a behavior using the function reactimateB in Listing 23, which turns a behavior to a push-stream given a delay and sampling period.

```
reactimateB :: Int                  -- delay between pulls
            -> DTime                -- sampling period
            -> Behavior a           -- input behavior
            -> Stream (DTime, a)    -- output stream

reactimateB delay dt b = Next Nothing (h b)
  where h b = do
      timeout delay
      ifAliveM
      (a, b') <- liftIO $ (runReaderT $ runSignal g) dt
      return $ Just (dt, a) `Next` h b'
```

Listing 23. Run a behavior as a stream.

KPI calculations can be expensive. For example, calculating THD for dozens of electrical signals can overwhelm some systems. To avoid this problem, we can measure the data speed by adding up the sampling periods of the stream reactimated from a behavior and divide it by the system time used. If the ratio is less than 1 (or a number less than 1 considering runtime overhead), then it is operating at less than real-time speed and computation load should be reduced. The speedS function in Listing 24 is for this purpose.

```
-- measure the total amount of sample time
-- within an given interval of system time
speedS :: Int                -- duration in milliseconds
       -> Stream (DTime, a)  -- stream of time/value pairs
       -> AsyncM Double      -- relative data speed

speedS n s =
    f <$> (foldS n 0.0 $ (\(dt,_) t -> t + dt) <$> s)
  where f t = t * 1000.0 / fromIntegral n
```

Listing 24. Measure the data speed of a Behavior.

Note that the sampling-rate of a stream reactimated from a behavior can be independent from the sampling-rate of the streams that the behavior depends on. For example, suppose that a stream of voltage samples is captured at 10KHz and is saved to a database. If the behavior for KPI calculations cannot run in real-time for this amount of data, then the behavior can be reactimated at a lower sampling rate (e.g. 5Khz). The data can still be captured at 10KHz and be saved to the database, but when the 10KHz stream is converted to the behavior, it is down-sampled by the stepper function.

Additional operations such as up/down sampling can be defined to support computation that operates at different frequency. A window function (Listing 25) is especially useful for IoT data since KPIs such as THD takes batches of samples in order to calculate the harmonics of the electric signals. The window function can generate data batches based on a specific window size and stride (the number of samples to skip between two consecutive batches).

```
-- convert a Behavior into an Event of data batches
window :: Int                       -- window size
       -> Int                       -- stride
       -> DTime                     -- sampling period
       -> Behavior a -> Event [a]
```

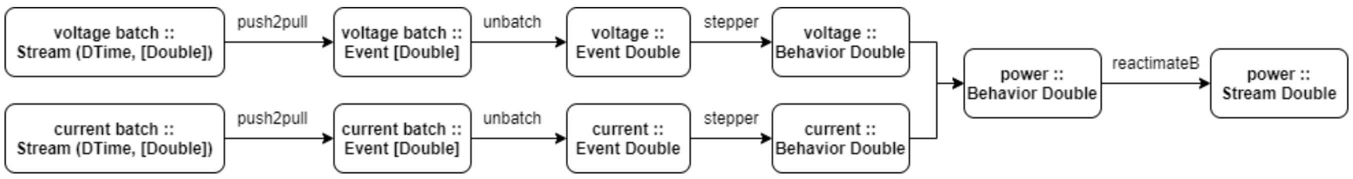
voltage batch :: Stream (DTime, [Double])
push2pull
voltage batch :: Event [Double]
unbatch
voltage :: Event Double
stepper
voltage :: Behavior Double
current batch :: Stream (DTime, [Double])
push2pull
current batch :: Event [Double]
unbatch
current :: Event Double
stepper
current :: Behavior Double
power :: Behavior Double
reactimateB
power :: Stream Double

Figure 5. Illustration of the workflow of computing inverter power.

```haskell
-- up-sample a Behavior by a factor
upsample :: Int                   -- up-sample factor
         -> Behavior a -> Behavior [a]
-- down-sample a Behavior by a factor
downsample :: Int                  -- down-sample factor
           -> ([(DTime, a)] -> a) -- summary function
           -> Behavior a -> Behavior a
```

Listing 25. Utility functions for `Behavior`

Assume we have a `thd` function that takes a sampling period and a list of values and returns the THD value. The `thd_stream` function below produces a stream of THD values from an inverter current behavior by calling the `window` function to make batch input to the `thd` function.

```haskell
thd :: DTime -> [Double] -> Double -- return THD value

thd_stream :: Int -> Int -> DTime
           -> Behavior Double
           -> Stream (DTime, Double)

thd_stream size stride dt behavior =
    reactimate 1 $ f <$> window size stride dt behavior
  where f (dt', w) = (dt', thd dt w)
```

We can use `speedS` function to measure the data speed of a stream and make runtime adjustment. The code below computes the THD of inverter current by first creating a stream of sliding windows of 5000 samples with the sampling period of 0.0002 seconds (line 5). The stride of the sliding window is 100, which means that one THD value is computed every $100 \times 0.0002 = 0.2$ seconds. Since larger stride means less computation, we can adjust the stride (line 10) if the data speed is outside the range of 0.9 to 1.1.

```haskell
1  adjust :: Bool -> Int -> Int -- adjust the stride
2
3  do current <- stream2behavior (getBatch "Ia")
4     let f stride = do
5          stream <- thd_stream 5000 stride 0.0002 current
6          (s, p) <- multicast stream
7          let mss = do x <- speedS 1000 s -- measure speed
8                       if 0.9 < x && x < 1.1 then mss
9                       else do liftIO cancelP p
10                              f $ adjust (x < 0.9) stride
11         Just s `Next` mss
12    join $ f 100
```

5 Discussion

Our definition of `Stream` is similar to the monadic stream `MStream` in [19]. In particular, `AsyncM (Stream a)` can be defined as `MStream AsyncM (Just a)`. However, the monadic interface of `Stream` depends on `AsyncM`. Also, for better efficiency, the `Stream` type in our implementation includes an

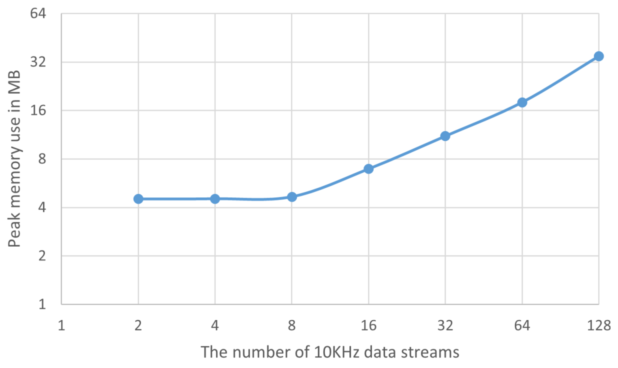

Figure 6. The peak memory use (MB) of the test programs containing 2 to 128 push streams of 10KHz samples.

`End` case. This change avoids the need to race any `AsyncM` with `neverM` even though the latter can never win a race.

```haskell
data Stream a = Next (Maybe a) (AStream a)
              | End (Maybe a)

instance Monad Stream where
-- return x = Just x `Next` neverM
   return x = End (Just x)
```

The definition of `Signal` is structurally identical to `MStream`. We choose this name to mean sensor signal instead of the FRP signal [6], which is a time to value function.

This design intends to work with high-frequency IoT data (e.g. 5–20KHz) but our push streams are driven by clocks of much lower frequency. IoT sensor data is often transmitted in batches due to the limitation of the sensors and the transmission links. Our design is to handle the batches of high-frequency samples asynchronously (e.g. data capture, storage, display) and to process the individual samples synchronously (e.g. KPI calculation). The clocks that drive the push streams may operate in the range of 1–100Hz.

Performance. To evaluate the overhead of our design, we ran tests that use several push streams to emit 10KHz of numeric data. The push streams are converted to behaviors, which are combined into one behavior using `Behavior`'s applicative interface and arithmetic operators. The final behavior is `reactimated` to a push-stream that prints the results. The tests primarily measure the memory overhead of push/pull conversion, unbatching, and resampling. The peak memory use of the tests is shown in Figure 6, where the memory use is close to linear to the number of data streams.

The tests were run with one physical thread on a Dell laptop with Intel i7 processor (4 cores). We do not have precise measurements of the runtime overhead but for 128 data streams, the CPU usage is about 4% and for 64 data streams, it is about 2%.

6 Related Work

Push-Pull. Our work was influenced by push-pull FRP [8], which was a modernization of Fran [9]. 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 [10] 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. Instead, we can `multicast` a stream so that multiple uses will not cause re-computation and switching will not cause restart. Krishnaswami [11] used a static approach to ensure that past values cannot be accessed and Patai [18] achieved similar goals by distinguishing streams and streams of streams at the type level. FRP Now [21] 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 [13]. 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 [6]. 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 [19], which can model FRP signals and stateful reactive programming by stacking different monads. A later version called Rhine [1] 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 [24]) and synchronous dataflow languages (e.g. Lustre [20] and SIGNAL [12]) 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 the 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. Hiphop.js [3] is a dataflow programming language that builds on the programming model of Esterel [2] and allows mixing of synchronous and asynchronous programming. Hophop.js focuses on Web orchestration with a declarative interface while our design is on real-time data processing.

Distributed Reactive Programming. Distributed FRP focuses on solving issues such as glitch-freedom, scalability, and fault-tolerance that arise differently compared to non-distributed systems. QPROP [17] is a propagation algorithm designed for distributed systems. It works by exploring the graph to find the dependency nodes before start propagation and its variant supports dynamic graph changes. XFRP [22] is a distributed FRP language based on actor model that compiles to Erlang. Though our model can interact with remote data sources and sinks, it is not a distributed design.

7 Conclusion

In this paper, we presented a push-pull reactive programming model for IoT data processing which uses asynchronous streams for processing events and synchronous signals for processing data. Separating asynchronous streams from synchronous signals allows our model to isolate side-effecting computation which can run asynchronously such as fetching batches of data via HTTP requests from the pure synchronous computation of processing data.

Furthermore, we demonstrated how the model can be used for real-time processing of high sampling-rate signals while reacting to changes in processing speed by adjusting the sampling rate. This dynamic switching is afforded by the `AsyncM` monad, a continuation monad with implicitly threaded cancellation tokens, which is the basis for our asynchronous computations that allows for multi-threaded as well as single-threaded event loop implementations.

References

[1] Manuel Bärenz and Ivan Perez. 2018. Rhine: FRP with Type-Level Clocks. *SIGPLAN Not.* 53, 7 (Sept. 2018), 145–157. https://doi.org/10.1145/3299711.3242757

[2] Gérard Berry. 1999. The Constructive Semantics of Pure Esterel. (1999).

[3] Gérard Berry and Manuel Serrano. 2020. HipHop. js:(A) Synchronous reactive web programming.. In *PLDI*. 533–545.

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

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

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

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

[8] Conal Elliott. 2009. Push-pull functional reactive programming. In *Haskell Symposium*. http://conal.net/papers/push-pull-frp

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

[10] Wolfgang Jeltsch. 2009. Signals, Not Generators!. In *Trends in Functional Programming*.

[11] Neelakantan R. Krishnaswami. 2013. Higher-Order Functional Reactive Programming without Spacetime Leaks. *SIGPLAN Not.* 48, 9 (Sept. 2013), 221–232. https://doi.org/10.1145/2544174.2500588

[12] Paul LeGuernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire. 1991. Programming real-time applications with SIGNAL. *Proc. IEEE* 79, 9 (1991), 1321–1336.

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

[14] Ingo Maier and Martin Odersky. 2012. *Deprecating the observer pattern with Scala. react.* Technical Report.

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

[16] Microsoft. 2020. Reactive Extensions. http://reactivex.io/. Accessed: 2020-07-02.

[17] Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. 2019. Distributed reactive programming for reactive distributed systems. *arXiv* (2019), arXiv–1902.

[18] Gergely Patai. 2011. Efficient and Compositional Higher-Order Streams. In *Functional and Constraint Logic Programming*, Julio Mariño (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 137–154.

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

[20] Daniel Pilaud, N Halbwachs, and JA Plaice. 1987. 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*, Vol. 178. 188.

[21] Atze van der Ploeg and Koen Claessen. 2015. Practical Principled FRP: Forget the Past, Change the Future, FRPNow! *SIGPLAN Not.* 50, 9 (Aug. 2015), 302–314. https://doi.org/10.1145/2858949.2784752

[22] Kazuhiro Shibanai and Takuo Watanabe. 2018. Distributed functional reactive programming on actor-based runtime. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. 13–22.

[23] Atze van der Ploeg. 2013. Monadic Functional Reactive Programming. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell (Haskell '13)*. Association for Computing Machinery, New York, NY, USA, 117–128. https://doi.org/10.1145/2503778.2503783

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