Rhine - FRP with type-level clocks

Let signals and streams flow together at the correct speed

Manuel Bärenz
Postdoctoral Researcher
Mathematische Fakultät
Universität Wien
maths@manuelbaerenz.de

Abstract

Rhine is a library for synchronous and asynchronous Functional Reactive Programming (FRP). It separates the aspects of clocking, scheduling and resampling from each other, and ensures clock-safety on the type level.

A general-purpose framework is presented that can be used for game development, media applications, GUIs and embedded systems. It offers a flexible API with many reusable components. Side effects can be reasoned about at the type level, allowing for deterministic scheduling. Through the generality of clocks in Rhine, events and behaviours can be unified, and easily resampled. Concurrent communication is encapsulated safely.

Diverse reactive subsystems can be combined in a coherent, declarative data flow framework, while the interoperability of the different data rates is guaranteed by type level clocks.

CCS Concepts •**Software and its engineering → Functional languages;** *Data flow languages; Concurrent programming structures; Domain specific languages;*

Keywords functional reactive programming, haskell, reactive programming, asynchronous programming

ACM Reference format:
Manuel Bärenz. 2017. Rhine - FRP with type-level clocks. In *Proceedings of Haskell Symposium, Oxford, UK, September 07–08, 2017 (Haskell'17),* 11 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 Introduction

Complex reactive programs often process data at different rates. For example, games, GUIs and media applications typically output audio and video signals, and receive user input at unpredictable times. Additionally, internal processing may take time. Coordinating these different rates is in general a hard problem. Consider this nonexhaustive list of typical bugs that can occur in such applications, if not enough care is taken:

- Buffer underruns and overflows
- Space and time leaks
- Accidental synchronisation of independent subsystems, and subsequent lags
- Concurrency issues such as deadlocks

Assume the following example, which will lead us through the article: A media application plays back a sound file, while a visualisation of the sound is produced. The user may mute the sound with a mouse click. The audio samples should be produced at a rate of 48000 samples per second, whereas the video frames have to be emitted at 60 frames per second. Assume for the sake of the example that the sampling ratio between audio and visualisation should be stable, i.e. for every 800 sound samples, one visualisation frame should be created. The mouse clicks can occur at any time and should be thought of as events.

An ad-hoc implementation may immediately suffer from bugs: If audio and video are accidentally synchronised, i.e. a video frame is costly calculated in the time step when an audio sample should be processed, the audio may lag. If the sampling ratio between audio and video is not kept stable (as would be the case when handling audio and video concurrently), but the video application constantly draws 800 samples for every frame, a buffer may be underrun or overflowed. The straightforward solution to the latter problem would be manually upsampling the video signal by means of `Maybe`, and only emitting a `Just` value for every 800th frame. This forces the programmer to mix implementation details of the scheduling with the actual data processing. Transporting the mouse input to the sound subsystem would usually require concurrent communication, with all its pitfalls.

All the mentioned problems arise by forcing implementation details of scheduling, resampling and communication onto the programmer, who is busy enough with the implementation of the data flow.

The solution is a framework that separates the aspects of clocking, scheduling and resampling from each other, and from data processing itself. It should be *modular*, such that small resampling and scheduling components can be reused and composed without introducing new bugs. It should be *strongly typed*, and incorrectly clocked programs should be rejected at compile time.

Rhine is such a framework. It gains its strength by annotating the signal processing components of the program

Haskell'17, Oxford, UK
2017. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

with the information when data will be input, processed and output. This information is called a *clock*.

Different components of the signal network will become active at different times, or work at different rates, and they will be said to "run under different clocks". If components running under different clocks need to communicate, it has to be decided when each component becomes active, and how data is transferred between the different rates. The former aspect is called *scheduling*, and the latter *resampling*.

Rhine separates all these aspects from each other, and from the individual signal processing of each subsystem. It offers a flexible API to all of them and implements several reusable standard solutions. In the places where these aspects need to intertwine, typing constraints on clocks come into effect, enforcing *clock safety*.

In the media application example, three clocks are present:

Certainly the video rate at 60 frames per second constitutes a clock, but so does the audio sample rate. Synchronising these clocks violates clock safety, since the rates are different, and a program attempting this should be rejected. Still, the two clocks must be scheduled *deterministically* in a constant ratio. Determinism of schedules can be ensured by Rhine, since it offers a way to track side effects as type parameters.

The mouse events can be thought of as a clock as well, albeit an unconventional one: It does not usually tick at regular intervals and thus does not approximate temporal continuity, but it does tick every time a particular subsystem needs to become active, namely the one that processes the mouse click. According to our definition, event sources thus constitute clocks as well. This insight leads to the unification of events and behaviours (in classic FRP terminology), drastically simplifying the framework.

In the following, it will be shown how to…

- … build each of the three subsystems individually in a modular way, without concerns about resampling or scheduling aspects.
- … synchronise audio and visualisation deterministically and without side effects.
- … schedule the mouse event clock safely in a separate thread.
- … keep the overview over the whole program and see the direction of the data flow.
- … enforce clock safety on all subsystems and schedules.

The clock type system can be neatly embedded into Haskell's type system, which enables us to immediately reuse existing libraries and backends to write real world applications in Rhine. Often, little or no wrapper code has to be written to access other backends.

Implementations of clocks, schedules and resampling strategies for standard use cases are available in Rhine, as well as example applications.

The separation of clocking aspects from data aspects, and type-level clocks, distinguish Rhine from all other general-purpose FRP libraries written in Haskell. Over classical FRP frameworks, Rhine offers in particular the advantage of unifying events and behaviours into a single concept – clocked signals.

Outline

In Section 2, synchronous FRP will be revised, with an emphasis on arrowized FRP as embodied by Yampa. Side effects are introduced into the picture in Section 3 by discussing Dunai, a library that implements *monadic stream functions*.

Section 4 is devoted to the synchronous sector of Rhine, which builds on the concept of *type level clocks*. Scheduling and asynchronous data flow is covered in Section 5. Section 6 gives an overview over the various clocks, schedules and resampling buffers that Rhine has to offer. It also discusses connections to other frameworks.

In Section 7, the performance of Rhine is compared to Dunai. Qualitative comparisons to other frameworks and discussion of related work are found in 8.

2 Synchronous, Arrowized FRP

The semantic idea behind a *behaviour*, or *signal*, is that of a value varying with time[1]:

```
type Signal a = Time -> a
```

Straightforward implementations of this idea typically suffer from time and space leaks. While *classical* FRP frameworks try to alleviate the issue with a variety of techniques, *arrowized* FRP frameworks such as Yampa (Nilsson et al. 2002) avoid the issue conceptually. Taking the "Functional" in FRP very seriously, the arrowized approach emphasises *Signal Functions*, which semantically model functions from signals to signals, but are implemented in continuation passing style:

```
data YSF a b = YSF (DTime -> a -> (b, YSF a b))
```

This exemplary implementation of Yampa-style signal functions is of course a simplification that leaves out certain optimisation techniques, but it serves as a demonstration of the inner workings of a hybrid system: Data is processed in discrete time steps, or *ticks*, although the system does not prescribe a particular step size. Rather, the response depends on it continuously, in form of the first parameter of type `DTime`. Then a value of type `a` from the input signal is sampled, and an output sample of type `b` is created, together with a continuation that processes the next tick.

Note that such a framework is necessarily *synchronous*, i.e. the output signal is sampled at the same rate as the input signal. Since this implies that *all* signal functions in the whole reactive program need to be sampled at the same

[1] In some frameworks, `Time` is merely a type synonym for `Double`. In Rhine, it is implemented as a type class for maximum flexibility.

times, this is obviously a severe restriction, which will be resolved here.

The closest approximation to a (pure) signal is a signal function with trivial input:

```
type YSignal a = YSF () a
```

Another prominent concept in classical FRP is that of an *Event stream*, presented semantically as a stream of time-stamped values:

```
type Event a = [ (Time, a) ]
```

They are typically modelled in Yampa as a signal that may be absent:

```
type YEvent a = YSignal (Maybe a)
```

This is at least aesthetically displeasing: Care needs to be taken in order to sample the whole signal network at every time when an event occurs, and each event stream needs to be filled with an amount of `Nothing`s that diverges as the sampling rate increases. This problem will also be resolved here.

Yampa's signal functions are instances of the `Arrow` type class, which in turn extends `Category`. Thus there is an identity arrow, and pure functions can be lifted to signal functions, which in turn can be composed parallelly and sequentially:

```
id    :: YSF a a
arr   :: (a -> b) -> YSF a b
(>>>) :: YSF a b -> YSF b c -> YSF a c
(***) :: YSF a b -> YSF c d -> YSF (a, c) (b, d)
```

Additionally, Yampa provides initialised feedback loops (amongst many other signal function primitives):

```
loopPre :: c -> YSF (a, c) (b, c) -> YSF a b
```

Thus, data can be delayed and memory built up. Fundamental signal processing components such as integrators can then be implemented.

Control flow in Yampa is achieved by *mode switching*, or event handling. It will not be shown here since it is surpassed by exception handling in Dunai, which will introduced in the next section.

Running the main event loop in Yampa requires an unwieldy separation of the program into data-producing sensors and data-consuming actuators (both of which are in `IO`), and a pure signal function. Again, a simpler, improved version will be recapitulated in the next section.

3 Dunai: Synchronous FRP is Stream Processing with Side Effects

Dunai (Perez et al. 2016) generalises Yampa in one pivotal aspect: Instead of reading the sample time step each tick, we allow *any* side effect to be executed:

```
data MSF m a b = MSF (a -> m (b, MSF m a b))
```

The abbreviation `MSF` stands for *monadic stream function*, since they are most powerful when `m` is a monad. Then, `MSF m` is again an instance of `Arrow`, and initialised feedback loops can again be implemented. Strikingly, not only pure functions can be lifted to `MSF`s, but also Kleisli arrows:

```
arrM :: (a -> m b) -> MSF m a b
```

Dunai does not introduce hidden side effects, as many other frameworks do: The particular side effect used is visible in the type signature, and one may constrain the signal network to be completely pure.

The special case of Yampa signal functions arises for the `Reader` monad:

```
type YSF' a b = MSF (Reader DTime) a b
```

Control flow and termination, which may seem ad-hoc in Yampa, now become entirely natural with the `Either` monad. If a monadic stream function wishes to terminate with a result value `e`, it simply produces `Left e` in the tick. The exception may be handled by calculating a continuing stream function:

```
catch :: MSF (Either e1) a b
      -> (e -> MSF (Either e2) a b)
      ->       MSF (Either e2) a b
```

With this technique, `MSF`s become monads in the exception type, which leads to a very convenient interface for control flow.

Many other convenient possibilities offer themselves: Several components of the signal network may share state and environment variables without passing them around explicitly. The list monad allows for branching computations. Finally, with the `IO` monad, sensors and actuators are just special cases of `MSF`s:

```
type Sensor a = MSF IO () a
type Actuator b = MSF IO b ()
```

A sensor produces data by means of side effects, and an actuator consumes them while creating side effects. An `MSF` can be pre- and post-composed with sensors and actuators, yielding a *closed* stream function with no open inputs or outputs, i.e. one of type `MSF m () ()`. Running the main loop is now a much simpler task as in Yampa:

```
reactimate :: MSF m () () -> m ()
```

A closed monadic stream function is run indefinitely (or e.g. until an exception is thrown in the `Either` monad), and its behaviour is given by the side effects it produces. The framework doesn't insist on controlling the main loop, though, and can be called from within other frameworks as well.

Different effects can be combined through *monad transformers*. As one can escape transformer layers in ordinary code, one can do so in Dunai:

```
runReaderS :: MSF (ReaderT r m)    a  b
           -> MSF            m (r, a) b
```

Instead of implicitly reading the value from the environment, it is then passed as an input. Similar effect handlers exist

for all other usual transformers. For example, escaping an `EitherT e` effect means handling exceptions of type `e` and thus control flow, as has been demonstrated. `ListT` can be escaped by continuous concatenation:

```
runListS :: MSF (ListT m) a b -> MSF m a [b]
```

Combining the Yampa aspect with arbitrary further effects gives *effectful synchronous signal functions*:

```
type YMSF m a b = MSF (ReaderT DTime m) a b
```

As the outermost effect, the time step can be retrieved from the environment, whereas the programmer is free to use any further monads inside the transformer.

It is possible to write nontrivial FRP programs, for example arcade games, with such signal functions (Perez 2017; Perez et al. 2016). Of course, Yampa programs embed into this approach by choosing the identity monad for `m`, but more complex applications are possible.

The only clock-related code that has to be added by hand is the supply of `DTime` values to escape the outermost `Reader` layer: The final program `ymsf :: YMSF m () ()` is transformed to `runReaderS ymsf :: MSF m (DTime, ()) ()`, which needs to be precomposed with a stream of `DTime`s before it can be run as the main loop. In the next section, this will be done in a methodological fashion.

4 Clock-Safety in Rhine

For the purposes of the library, a *running clock* is an effectful stream of both time stamps and possibly further information about the clock state or the nature of the tick, called the *tag*:

```
type RunningClock m tag = MSF m () (Time, tag)
```

A running clock may produce side effects such blocking, or polling the system clock or another device. A running clock is said to *tick* at the time stamps it outputs. In the case of real-time clocks, it is the obligation of the clock implementation to correctly observe the system time, and to wait, if necessary, until the correct time has been reached.

The additional data supplied by the tags prove useful in real world applications, as they may specify *why* the clock ticked (e.g. the occurrence of a particular event), or *how* it ticked (e.g. whether an attempt at soft real-time was successful).

In this section, we will see how Rhine's *Clocks* supply the time steps to the synchronous subsystems in a *clock-safe* way, i.e. such that every subsystem will be supplied with the correct time steps, and subsystems with different clocks cannot be synchronised by accident.

The key idea is to let the type checker verify the clock-safety. Thus, a type class `Clock` is supplied, and its instances will be called *clock types*. Two concepts have to be distinguished:

The clock type specifies all relevant properties of the clock, in particular:
- when, or in particular how fast, it ticks,
- in which monad its side effects take place,
- what kind of `tag`s it can produce.

The clock value, i.e. a value of a clock type, holds all relevant information necessary to *run* the clock, such as event sockets or device addresses. If no such information is required, the clock is a singleton.

Combining both aspects, clocks are implemented as a type class in Rhine:

```
class Clock cl m where
  type Tag cl
  runClock :: cl -> RunningClock m (Tag cl)
```

The actual implementation is only slightly more involved to allow for more complex initialisation actions of the clock.

`Time` and `DTime` are in fact implemented as a type class:

```
class (Num (Diff Time), Ord time)
  => TimeDomain time where
    type Diff time
    diffTime :: time -> time -> Diff time
```

This is the minimal interface necessary for Rhine. For conciseness though, we will suppress the type class in this article.

From a running clock, we can derive the time steps as differences of the absolute times, using `diffTime`. For convenience, the absolute time and the time since the first tick[2] will also be supplied, together with the tag:

```
data TimeInfo cl = TimeInfo
  { sinceLast  :: DTime
  , sinceStart :: DTime
  , absolute   :: Time
  , tag        :: Tag cl
  }
```

The fundamental components of Rhine's signal networks are synchronous, effectful signal functions annotated with a clock type:

```
type SyncSF m cl a b
   = MSF (ReaderT (TimeInfo cl) m) a b
```

Two synchronous signal functions can only be composed if their clock types agree.

A closed synchronous signal function can be run together with a clock value of the correct type:

```
reactimateSync :: (Monad m, Clock m cl)
               => cl -> SyncSF m cl () () -> m ()
```

The clock is run, and the resulting stream of `TimeInfo`s escapes the `ReaderT` context of the signal function.

4.1 Unifying Events and Behaviours

Since signals in Rhine can be sampled at different speeds, events and behaviours can be unified compellingly by simply distinguishing them by their clocks:

[2] Of course, the time since the first tick could also be accumulated from the time steps, but the rounding errors can grow indefinitely as the program runs.

Event clocks are those clocks that tick precisely when the originating event occurs. The original event data is stored in the tag of the tick.

Frequent clocks are those clocks that approximate continuity, usually by ticking at regular intervals.

Synchronous signals under an event clocks are called *clocked events*:

```
type ClEvent m eventcl a = SyncSF m eventcl () a
```

There is no enforced distinction between event clocks and frequent clocks, and many examples indeed blur the line between the two concepts. For example, we may run a cleanup task at regular intervals, and additionally whenever a certain event occurs.

As an advantage of this approach, we can now immediately see that two events will occur simultaneously if they have the same clock, which is visible in the type signature. Combinations of non-simultaneous events will be treated in Section 5.

A special role is played by those signals that can be sampled under any clock. These reflect the spirit of the hybrid paradigm represented by Yampa: Continuity is approached by defining a discretised signal that can be sampled at arbitrary times. Such signals may thus truly be called *Behaviours*:

```
type Behaviour m a = forall cl. SyncSF m cl () a
```

4.2 Examples

Already now, we can implement the synchronous subsystems of the envisaged media application.

Handling the mouse input is a typical application of event clocks. For the SDL library, which supports mouse input, a clock for the internal event queue has been implemented. Rhine provides subclocks parametrised by a main clock and a subevent type. They can be created from a main clock value and a subevent selector function:

```
data SubClock cl a = SubClock
  { mainClock :: cl
  , select    :: Tag cl -> Maybe a
  }
```

We select only mouse clicks:

```
type MouseClock = SubClock MainEvClock (Int, Int)
mouseClock :: MouseClock
mouseClock = SubClock mainEventClock select
  where
    select (MouseClick (x, y)) = Just (x, y)
    select _                   = Nothing
```

Determining whether the current state of the audio signal should be muted or not is a simple matter of counting the events and checking whether the count is odd:

```
muted :: Monad m => SyncSF m MouseClock () Bool
muted  = count >>> arr odd
```

No side effects have been introduced, and this is visible in type signature as polymorphism in `m`. The signal function `muted` will be called precisely when the event occurs, so it is not necessary to artificially resample them as `Maybe` values as we needed in Yampa.

For conciseness, and since they are not specific to Rhine, the implementations of the audio and visualisation signals are not presented here. It will suffice to note that the audio samples are drawn by means of `IO`, whereas the visualisation is calculated purely from a buffer of audio samples:

```
samples :: SyncSF IO SPS48000 ()       Double
vis     :: SyncSF m  FPS60    [Double] Image
```

The clock `SPS48000` ticks with a rate of 48000 samples per second, likewise `FPS60` at 60 frames per second. It is not possible to accidentally synchronise the two subsystems. One could try to compile the following expression:

```
samples >>> arr return >>> vis
```

But it will lead to an error:

```
Couldn't match type 'FPS60' with 'SPS48000'
[...]
```

The type error is very clear and helpful. Furthermore, the clock types serve as useful documentation of the processing rates of the different subsystems.

5 Schedules and Resampling for Asynchronicity

5.1 Schedules

The *scheduling problem* is the question when to execute the ticks of several differently clocked synchronous subsystems. There is no universal solution it. Different situations call for different schedules.

In our example, the audio system has to be *deterministically* scheduled with the visualisation, in the sense that for every 800 audio samples exactly one video frame has to emitted. On the other hand, it is of course impossible to predict how many audio samples may pass between two mouse events. To schedule the mouse clock together with the audio clock, a side effect needs to be introduced: concurrency. (In other words, we rely on the operating system to solve the scheduling problem.)

Consequently, Rhine gives the programmer the freedom to implement their own schedules, and at the same time supplies several common implementations to cover typical use cases.

In general, a *binary schedule* for two clocks `cl1` and `cl2` should have the semantics of *universal clock* such that `cl1` and `cl2` are its subclocks. In other words, if we consider the partially ordered set of clocks with the subclock relation, schedules should correspond to *joins*.

Depending on the clocks, a binary schedule might not exist (if e.g. the side effects required by the clocks are incompatible) or not be unique (if e.g. certain ticks of the clocks

have to occur simultaneously, and an execution order has to be fixed). Schedules can be polymorphic in the clocks (such as the concurrency schedule mentioned before), or polymorphic in the side effects (such as deterministic schedules).

A binary schedule for two clocks `cl1` and `cl2` with side effects in `m`, together with values for the individual clocks, gives a value of type `CombinedClock m cl1 cl2`. Combined clocks are used to type signal systems *sequentially* composed of two subsystems, where the subsystem under `cl1` produces data that the subsystem under `cl2` consumes.

It will also be possible to compose signal networks *parallelly*, and again a schedule is needed to clock such compositions. Such a schedule, and values for the individual clocks, yield a value of type `ParClock m cl1 cl2`.

The subclocks may themselves be combined of further clocks, resulting finally in an ordered tree with atomic clocks on the leaves and clock-safe schedules on the nodes. Rhine supplies type families `Leftmost` and `Rightmost`, which calculate the leftmost and rightmost atomic clocks of the tree. They correspond to the rate at which data enters, or leaves the system, respectively.

The reader may be concerned that Rhine's schedules combine exactly two clocks, and not more. The reason for this comes from the directedness of data, which is particular to arrowized FRP frameworks. Scheduling typically occurs at the boundary between two subsystems, where one component produces data and the other consumes it.

In our example, all schedules can be taken from the library.

The visualisation clock is a subclock of the audio clock, by selecting every 800th tick:

```
type FPS60 = SubClock SPS48000 ()
sched800 :: SyncSF m SPS48000 () (Maybe ())
sched800 = count
  >>> arr (\n -> guard $ (n+1) `mod` 800 == 0)

fps60 :: Monad m => FPS60 m
fps60 = SubClockS SPS48000 sched800
```

There is a builtin deterministic schedule that combines subclocks with their main clocks, here assumed to be singletons:

```
joinSubClock :: Schedule m cl (SubClock cl a)
```

It ticks when a tick of the main clock occurs, and emits an additional tick for the subclock whenever the selector function outputs a `Just` value.

There is no deterministic schedule for the mouse clock and the audio clock. The simplest solution is to defer the problem to the GHC scheduler by forking separate threads for the two clocks and to collect the ticks from a shared variable in the foreground thread. This is provided by a clock-polymorphic schedule:

```
concurrently :: Schedule IO cl1 cl2
```

It encapsulates all concurrent communication and thus eliminates typical pitfalls, since no threads or shared variables need to be created by the programmer.

5.2 Resampling

Scheduling describes the *clock aspect* of running several systems at different rates, whereas resampling describes the *data aspect*. One fundamental advantage that Rhine has over any other FRP framework in Haskell, to the knowledge of the author, is the separation of these two concerns. The resampling problem can be treated entirely separately from the scheduling problem. Of course, the two problems are often intertwined, and some data can only be resampled under the assumption of specific clocks. This is reflected by clock type constraints in Rhine.

As for scheduling, there is no single solution to the resampling problem, so the choice is again with the programmer, and popular resampling techniques are already given in the library.

The fundamental building block of resampling provided by Rhine is a *resampling buffer*:

```
data ResBuf m cl1 cl2 a b = ResBuf
  { put :: TimeInfo cl1 -> a
        -> m (   ResBuf m cl1 cl2 a b)
  , get :: TimeInfo cl2
        -> m (b, ResBuf m cl1 cl2 a b)
  }
```

The idea is akin to monadic stream functions, in that continuation passing style is used, but resampling buffers are fundamentally asynchronous: Input and output never happen simultaneously. This allows them to be the connecting link between two subsystems under a schedule: The schedule decides which subsystem ticks. If the left subsystem produces data, it is stored in the resampling buffer, together with a timestamp from the schedule, via `put`. If the right subsystem requires data, it is retrieved (again timestamped) from the resampling buffer via `get`.

Buffers that can accept via `put` and `get` calls at any time are clock-polymorphic, and buffers that cannot (e.g. in order to meet space requirements) can constrain the clocks in the type signature. Resampling buffers can range from interpolation (such as linear, cubic or sinc) to e.g. FIFO queues.

The astute reader may have noticed that `ResBuf`s are very similar to (monadic stream) functions that accept lists (or sequences) of data as inputs. All `put` calls can be queued, and processed in one step when `get` is called:

```
pureResBuf :: ([a] -> b) -> ResBuf m cl1 cl2 a b
```

The most general implementation allows internal state and side effects, and adds a single time stamp for `get` and time stamps for each `put`:

```
msfResBuf ::
     MSF m (TimeInfo cl2, [(TimeInfo cl1, a)]) b
  -> ResBuf m cl1 cl2 a b
```

While all `put` calls are queued, memory bounds may be exceeded, resulting in a space leak, and when all of the queued data is processed at once during a `get` call, a time leak can occur. Therefore, functions or **MSF**s with list inputs are not an efficient simplification for resampling buffers, justifying the implementation of **ResBuf** in terms of continuation passing style.

In our example, data has to be resampled in two places: At the boundary between the mouse and audio subsystems, and between the audio and visualisation subsystems.

From mouse to audio, we will simply remember the last value for the mute state that was emitted from the mouse event:

```
keepLast :: a -> ResBuf m cl1 cl2 a a
```

Scheduling mishaps in which mouse clicks would be lost cannot occur since each click is processed under the mouse clock. Only the final result is forwarded to the audio system.

From audio to visualisation, we collect all values in a list:

```
collect :: ResBuf m cl1 cl2 a [a]
```

To ensure that precisely 800 samples are collected for each frame, we can constrain the type signature to specific clocks:

```
collect800 :: ResBuf m SPS48000 FPS60 a [a]
collect800 = collect
```

5.3 Asynchronous signal functions

For a given clock tree, an asynchronous signal function is a tree of the same shape, with synchronous signal functions under the corresponding atomic clocks on the leaves, and clock-safe resampling buffers on those nodes corresponding to (sequentially) combined clocks. They are of type **SF** `m cl a b`, where `m` is the type of side effects, `cl` is the clock type, and `a` and `b` are input and output, respectively.

SFs can be composed sequentially by providing a schedule and a resampling buffer that match the clocks of the two **SF**s. They can also be composed parallelly in two cases: If the clock types match, the **SF**s can be composed without additional data. If they do not match, a schedule must be provided, and the input types must match instead, since otherwise there would be no guarantee that the currently active subsystem receives input of the correct type.

Unfortunately, it does not seem possible to implement an **Arrow** instance, due to the extra clock type parameters. To alleviate the situation, certain combinators (extending the **Arrow** combinators) are provided by Rhine, that allow for composition of **SF**s and clocks in an intuitive and readable fashion.

A closed asynchronous signal function, together with a value of the correct clock type is called, as a polyseme, a *Rhine* (since many streams flow in it):

```
data Rhine m cl a b = Rhine
  { sf    :: SF m cl a b
  , clock :: cl
  }
```

A closed Rhine can be run as a main loop:

```
flow :: ( Monad m, Clock m cl )
     => Rhine m cl () () -> m ()
```

The main loop simply waits for the next tick of the top schedule, and then navigates through the tree to step the corresponding synchronous signal function on the leaf, putting and getting data to and from the neighbouring resampling buffers.

Syntactic sugar allows us to create the synchronous subsystems together with their clocks:

```
mutedC :: Rhine m MouseClock () Bool
mutedC = muted @@ mouseClock
```

The operator `@@` combines a synchronous signal function with a clock of the correct value.

```
audio :: Rhine IO SPS48000 Bool Double
audio = (@@ SPS48000) $ proc mute -> do
  sample <- samples -< ()
  returnA -< if mute then 0 else sample
```

Using **Arrow** syntax, we have specified the complete audio subsystem. First, it is fixed to run under the audio clock. Then, the parameter `mute` is received. A `sample` is retrieved, and depending on the `mute` variable, it is output or replaced by 0.

Assume that the graphics backend has been already implemented, or taken from another library. Then the complete graphics subsystem is defined as follows:

```
backend  :: Image -> IO ()
graphics :: Rhine  IO  FPS60   [Double] ()
graphics = (vis >>> arrM backend) @@ fps60
```

With further syntactic sugar, we can join the subsystems:

```
res800 = collect800 -@- joinSubClock
audioVis = audio >-- res800 --> graphics

keepMuteState = keepLast False -@- concurrently
system = mutedC >-- keepMuteState --> audioVis
```

The operator `-@-` combines a resampling buffer and a schedule to a *resampling point*. The operators `>--` and `-->` compose two **Rhine**s along a resampling point, building up the clock tree and the signal function tree simultaneously.

Finally, the whole reactive program can be run:

```
main :: IO ()
main = flow system
```

5.4 Wormholes and Concurrency

While Rhine supplies concurrent scheduling, the data processing by default happens in a single thread. This is a potential pitfall when scheduling expensive computations that consume more than their allotted step time. Processing the

data concurrently can in principle alleviate the issue, but brings up the need for communication across threads.

Wormholes (Winograd-Cort and Hudak 2012) have been presented as a means of communication between threads in the context of arrowized FRP.

They have been ported to Dunai:

```
data Wormhole m a = Wormhole
  { sink   :: MSF m a  ()
  , source :: MSF m () a
  }
```

A wormhole encapsulates a communication primitive, and allows data to be sent from the sink to the source by means of side effects. Implementations based on **IORef**s, **Chan**s and **TChan**s exist.

Without clocks, wormholes already serve their purpose well and make concurrency in Dunai convenient and safe. It is of course possible to simply use them in Rhine, since it is based on Dunai, but it is possible to do better.

It is a common reactive programming pattern to defer an expensive computation or blocking side effect to a separate thread, and be notified when the data is ready. Rhine enables this through a slight variation on wormholes:

```
data ClockWormhole m td a = ClockWormhole
  { clockSink   :: MSF m a  ()
  , clockSource :: MSF m () (td, a)
  }
```

By the slight extra expense of time-stamping every output value, **ClockWormhole**s become instances of the **Clock** type class. While data can be sent from one thread and received on another, the receiving end defines a clock which ticks exactly when the data arrives. It may then be processed and resampled to different rates with the standard Rhine tools that have already been demonstrated. To send data back to the first thread, simply create a second **ClockWormhole**.

6 A Brief Tour through the Library

6.1 Clocks

Several soft real time clocks exist that aim to tick at a given fixed rate, as well as a **Busy** clock which tries to tick as fast as possible. These clocks use the system time. The **DataKinds** extension can be used to specify the clock rate as a type level natural number:

```
exampleSyncSF :: SyncSF m (Millisecond 42) a b
```

Real time audio synthesis is a technically hard problem, but implementing audio clocks is conceptually bearable in Rhine. An audio clock is implemented that drives the computation in buffers, and aligns with system time at the end of each calculation.

There is a clock that can be rate-controlled through effects in the **Reader** monad. One may supply the rate by a synchronous signal function, the *rate-controlling subsystem* and combine both two in a new clock:

```
data ControlledReaderClock m cl r =
  ControlledReaderClock
    { clock :: cl
    , env   :: SyncSF (ReaderT r m) cl () r
    }
```

(The code has been slightly simplified as compared to the library.) If one starts the rate-controlled clock, the synchronous signal function `env` is started simultaneously, and supplies the clock with data on which it can base the decision when to tick next. An application is the *continuous collision detection* monad (Perez et al. 2016, Section 7.1.3). Any part of the rate-controlling subsystem may require a calculation step at a certain critical time (typically if a collision in a physical simulation occurs), and the clock can respond by emitting an additional tick at that time.

Deterministic simulation clocks exist that tick at given rates, which can again be specified by type level natural numbers.

Given a schedule, two clocks can be combined to a parallel clock, which ticks whenever either of the clocks ticks.

For the Simple DirectMedia Layer (SDL) library, an event clock is implemented.

As has been demonstrated in the example, a subclock can be created from any clock by analysing its tag.

6.2 Schedules

Any two clocks may be scheduled concurrently, at the expense of introducing **IO**.

Fixed rate clocks using type level natural numbers can be scheduled in a deterministic fashion, providing an infinite family of side-effect-free schedules. Likewise, there exist deterministic schedules between a subclock and its main clock, and between two subclocks of the same main clock.

To provide a declarative interface to effectful clocks that can be automatically scheduled, a new monad transformer is defined:

```
data ScheduleT m a
  = Pure a
  | Wait DTime (m (ScheduleT m a))
```

It is constructed as a free monad over a formal "waiting" operation. Consequently, the **ScheduleT** layer adds the functionality of formally waiting for a certain time before resuming with the effectful computation. If a concrete blocking effect in the encapsulated monad is provided, the **ScheduleT** transformer can be escaped:

```
runScheduleT :: Monad m
             => ScheduleT m a
             -> (DTime -> m ()) -> m a
```

If two *arbitrary* clocks are implemented in the **ScheduleT** monad, they can always be scheduled, using a single, polymorphic schedule:

```
schedule :: Schedule (ScheduleT m) cl1 cl2
```

Depending on the particular side effect type `m`, this schedule may be deterministic.

6.3 Resampling buffers

Several general purpose buffers are implemented, such as `collect` and the `keepLast`, which have already been presented in the example. Furthermore, there exist bounded and unbounded FIFO and LIFO queues:

```
fifo        ::             ResBuf m a (Maybe a)
fifoBounded :: Integer -> ResBuf m a (Maybe a)
```

For numerical data, the standard linear, cubic and sinc interpolations are implemented.

Helper functions build `ResBuf`s from side effectful getter and setter functions, simplifying the implementation of custom buffers:

```
mkResBuf :: Monad m
         => (s -> a -> m s)
         -> (s -> m (b, s))
         -> s
         -> ResBuf m cl1 cl2 a b
```

Furthermore, the convenience functions `pureResBuf` and `msfResBuf` build resampling buffers from monadic stream function or pure function accepting lists of values as inputs, respectively. As already discussed in Section 5.2, they can introduce space and time leaks, if no care is taken to control the size of the lists, and typically, leak-free alternatives exist.

6.4 Utilities

Several combinators for `SF`s and `Rhine`s exist. The combinators for sequential composition have already been shown in the examples. It is also possible to parallelly compose `SF`s on the same clock:

```
(***) :: SF m cl a b -> SF m cl c d
      -> SF m cl (a, c) (b, d)
```

If two `SF`s are on different clocks, but have the same input type, they can be parallelly composed as well:

```
(*-*) :: SF m cl1 a b -> SF m cl2 a c
      -> SF m (ParClock m cl1 cl2) a (Either b c)
```

6.5 Other frameworks

There are wrappers for common SDL functions, and SDL event clocks, which facilitate event handling.

To connect to Gloss, Rhine provides a side-effect-free Gloss clock. Signal functions under this clock can be simulated in Gloss, which makes it possible to painlessly develop simple OpenGL applications in Rhine.

For the Linux audio library Pulse, a clock and a simple backend have been created.

Wrappers for GUI libraries are under development.

Benchmark	Dunai	Dunai/Reader	Rhine
Sine	0.270 s	0.467 s	0.847 s
Empty	0.040 s	0.084 s	0.215 s

Table 1. Benchmarks for the sine generator, and for the empty signal generator. One second's worth of audio samples was produced. Best of 100 runs on a Gentoo Linux-4.10.11 machine with an Intel®Core™i7-4700MQ CPU.

7 Performance

A naïve sine generator has been implemented to compare the performance of synchronous Rhine programs to Dunai's monadic stream functions:

```
sine = loopPre 1 $ proc (_, posOld) -> do
  let acc = - posOld
  vel <-                  integral -< acc
  pos <- arr (+ 1)   <<< integral -< vel
  exitWhen (> 48000) <<< count    -< ()
  returnA                         -< (pos, pos)
```

A feedback loop is built in which a harmonic oscillator is simulated. The acceleration is the negative of the initial position, and velocity and new position are Euler-integrated from the acceleration. The starting position is set to 1. Additionally, the number of ticks is counted and causes the simulation to exit when 48000 samples have been produced.

This sine generator is run in three different settings, which are summarised in Table 1:

Dunai It is run in Dunai where the integral is implemented manually, i.e. the second line of the definition of `sine` is altered to:

```
vel <- sumS -< acc * dTime
```

The stream function `sumS` sums up its input over time, and `dTime` is a fixed time step.

Dunai/Reader It is run again in Dunai, but the integral is implemented using the `ReaderT` abstraction described in Section 3 to handle the time steps.

Rhine The sine generator is run in the Rhine framework with a fixed rate side-effect-free clock.

The results show that the `ReaderT` abstraction introduces a performance penalty of roughly a factor of two, and so do Rhine's additional wrappers. This is significant, though just about bearable for the particular situation. Since 48000 samples are produced, they would need to be consumed in a whole second, and each benchmark avoids this threshold. Of course, for a more elaborate audio processing pipeline or a less powerful machine, the threshold may be exceeded.

Additionally, the same benchmark was run with the sine generator removed, and just the counting of the samples left in place, so an empty signal was produced. It shows that the overhead was roughly 4.5 μs per tick in the synchronous Rhine simulation. Therefore, in other situations such as video processing, the relative overhead introduced by the

framework has to be expected to be much milder, since the frame rate is much lower. For GUI applications, the overhead should not be detectable by the user anymore.

The benchmark is meant to measure the overhead introduced by the support for asynchronicity, as compared to synchronous monadic stream functions. Meaningful comparisons to other asynchronous frameworks are hard to produce, since example benchmark programs tend to be incomparable, and will be deferred to future work.

8 Related work

Rhine has several unique advantages over other FRP frameworks in Haskell.

The vast majority of frameworks do not supply type-level clocks, making it hard to reason about the speeds of the different subsystems. To the knowledge of the author, the single exception is CλaSH (Baaij et al. 2010), which only provides fixed-rate logical subclocks of a main clock. Furthermore, it specifically targets hardware circuits and does not support `IO`. Rhine provides clocks of arbitrary rates and interoperates with standard Haskell code.

Likewise, most frameworks do not explicitly distinguish clocks and data, in particular, they do not separate the scheduling aspect from the resampling aspect. This separation allows for much of Rhines modularity and the reusability of its components.

Most classical frameworks hide `IO` primitives under their abstractions, typically `IORefs` such as in Sodium (Blackheath 2012), Elerea (Patai 2011) and Reactive Banana (Apfelmus 2011), or even `unsafePerformIO` such as FRPNow (Ploeg and Claessen 2015). The Rhine framework is completely pure since its abstractions are universally quantified over all monads. This is very useful since it is then possible to simulate signal networks, e.g. for testing purposes, without user interaction. Additionally, one can clearly reason about the behaviour of the reactive program since there are no hidden side effects. For example, the Gloss library (Lippmeier 2010) provides a pure interface, and Rhine can in fact connect to it. The benefits of `IO` can be brought back by simply instantiating the framework with `IO` components.

Classical FRP frameworks inherit the separation into events and behaviours from Fran (Elliott and Hudak 1997). Rhine unifies and generalises the two concepts, which simplifies programming and allows for transfer between the two.

Dunai, Netwire (Söylemez 2016), varying (Scivally 2015) and Auto (Le 2015) have a monad type parameter to allow reasoning about the possible side effects, but they are all synchronous in nature.

Pipes (Gonzalez 2012) and Conduit (Snoyman 2011) are asynchronous and also quantify over monads. They are powerful in their application domain, but lack any treatment of time and can therefore not be regarded as FRP frameworks. In particular, they lack the separation of scheduling and resampling.

Outside the Haskell ecosystem, Lucid Synchrone (Caspi and Pouzet 2000) takes the concept of type level clocks seriously, and supplies useful clock combinators. In fact, it was one of the main inspirations for Rhine. Like CλaSH, it is aimed at hardware circuits, and therefore its clocks are logical, i.e. subclocks of a particular main clock. It also lacks quantification over the side-effect type. Finally, it is not interoperable with Haskell in any convenient way.

9 Summary

We have seen that the separation of aspects into synchronous data flow, clocks, schedules and resampling allowed us to develop a complex media application step by step. Synchronous subsystems could be combined to an asynchronous network through reusable general purpose components such as schedules and resampling buffers. The interoperability of the different components is ensured by the clock type system. It is not necessary anymore to resample signals and events by hand, as it was in Yampa and Dunai, and the programmer is spared the pitfalls of concurrency. Instead, they could be treated in a unified framework. For more complex applications, concurrent data processing through wormholes and universal schedules through the `ScheduleT` transformer are available. Still, the code stays intuitive and modular, in a data-flow fashion.

Rhine introduces a performance overhead, but it is still bearable. It offers unique advantages over existing FRP frameworks.

As future work, more wrappers for GUI and media backends must be written, to lower the bar for real-life FRP development even more. Bigger example applications have to be constructed in order to find out how well the abstractions scale.

Acknowledgements

First and foremost, the author wishes to thank Ivan Perez for years of discussions and collaboration on these topics. For helpful discussions, thank you, Henrik Nilsson, Michael Mendler, Marc Pouzet, Adrien Guatto, Timothy Bourke, the participants of SYNCHRON 2016, the FP Lab Nottingham, and the Monday Afternoon Club in Bamberg.

References

Heinrich Apfelmus. 2011. Reactive-banana. https://github.com/HeinrichApfelmus/reactive-banana. (2011).

Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. 2010. CλaSH: Structural Descriptions of Synchronous Hardware using Haskell. In *Proceedings of the 13th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools*. IEEE Computer Society, 714–721. http://doc.utwente.nl/73124/

Stephen Blackheath. 2012. Sodium. https://github.com/SodiumFRP/sodium. (2012).

Paul Caspi and Marc Pouzet. 2000. *Lucid Synchrone, a functional extension of Lustre.* Technical Report. Université Pierre et Marie Curie, Laboratoire LIP6.

Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*. ACM, New York, NY, USA, 263–273. DOI: http://dx.doi.org/10.1145/258948.258973

Gabriel Gonzalez. 2012. Pipes. https://github.com/Gabriel439/Haskell-Pipes-Library. (2012).

Justin Le. 2015. Auto. https://github.com/mstksg/auto. (2015).

Ben Lippmeier. 2010. Gloss. https://github.com/benl23x5/gloss. (2010).

Henrik Nilsson, Antony Courtney, and John Peterson. 2002. Functional Reactive Programming, Continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell '02)*. ACM, New York, NY, USA, 51–64. DOI: http://dx.doi.org/10.1145/581690.581695

Gergely Patai. 2011. Efficient and Compositional Higher-order Streams. In *Proceedings of the 19th International Conference on Functional and Constraint Logic Programming (WFLP'10)*. Springer-Verlag, Berlin, Heidelberg, 137–154. http://dl.acm.org/citation.cfm?id=2008270.2008280

Ivan Perez. 2017. Back to the Future: FRP with Reversible Time. *unpublished* (2017).

Ivan Perez, Manuel Bärenz, and Henrik Nilsson. 2016. Functional Reactive Programming, Refactored. In *Proceedings of the 9th International Symposium on Haskell (Haskell 2016)*. ACM, New York, NY, USA, 33–44. DOI: http://dx.doi.org/10.1145/2976002.2976010

Atze van der Ploeg and Koen Claessen. 2015. Practical Principled FRP: Forget the Past, Change the Future, FRPNow!. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, New York, NY, USA, 302–314. DOI: http://dx.doi.org/10.1145/2784731.2784752

Schell Scivally. 2015. varying. https://github.com/schell/varying. (2015).

Michael Snoyman. 2011. Conduit. https://github.com/snoyberg/conduit. (2011).

Ertugrul Söylemez. 2016. Netwire. https://github.com/esoeylemez/netwire. (2016).

Daniel Winograd-Cort and Paul Hudak. 2012. Wormholes: Introducing Effects to FRP. In *Proceedings of the 2012 Haskell Symposium (Haskell '12)*. ACM, New York, NY, USA, 91–104. DOI: http://dx.doi.org/10.1145/2364506.2364519