

Asynchronous Reactive Programming with Modal Types in Haskell

Patrick Bahr, Emil Houlborg, and Gregers Thomas Skat Rørdam

IT University of Copenhagen, Copenhagen, Denmark
{paba,ehou,grtr}@itu.dk

Abstract. The implementation of asynchronous systems, in particular graphical user interfaces, is traditionally based on an imperative model that uses shared mutable state and callbacks. While efficient, the combination of shared mutable state and callbacks is notoriously difficult to reason about and prone to errors. Functional reactive programming (FRP) provides an elegant alternative and recent theoretical advances in modal FRP suggest that it can be efficient as well.

In this paper, we present *Async Rattus*, an FRP language embedded in Haskell. The distinguishing feature of *Async Rattus* is a modal type constructor that enables the composition of asynchronous subsystems by keeping track of each subsystem’s clock at compile time which in turn enables dynamically changing clocks at runtime. The central component of our implementation is a Haskell compiler plugin that, among other aspects, checks the stricter typing rules of *Async Rattus* and infers compile-time clocks. This is the first implementation of an asynchronous modal FRP language. By embedding the language in Haskell we can exploit the existing language and library ecosystem as well as rapidly experiment with new language features and library design. We hope that such experimentation with *Async Rattus* sparks further research in modal FRP and its applications.

Keywords: Functional reactive programming · Modal types · Space leaks

1 Introduction

Functional reactive programming (FRP) [14] provides an elegant, high-level programming paradigm for reactive systems. This is achieved by making time-varying values (also called *signals* or *behaviours*) first-class objects that are easily composable. For example, assuming a type *Sig a* that classifies signals of type *a*, an FRP library may provide a function $map :: (a \rightarrow b) \rightarrow Sig\ a \rightarrow Sig\ b$ that allows us to manipulate a given signal by applying a function to it.

Haskell has a rich ecosystem of expressive and flexible FRP libraries [1, 7, 8, 15, 19, 22, 30–32, 35]. Devising such FRP libraries is challenging as their APIs must be carefully designed to ensure that reactive programs are *causal* and are not prone to *space leaks*. A reactive program is causal if the value of any output signal at any time *t* only depends on the value of input signals at time *t* or earlier.

Due to the high-level nature of FRP programs, they can suffer from *space leaks*, i.e. they keep data in memory for too long. Haskell FRP libraries tackle these issues by providing a set of *abstract* types (i.e. their definitions are not exposed) to represent signals, signal functions, events etc. and only expose a carefully selected set of combinators to manipulate elements of these types.

Over the last decade an alternative to this library-based approach has been developed [3, 4, 21, 24, 26, 28, 29] that uses a modal type operator \bigcirc (pronounced “later”) to express the passage of time at the type level. This type modality allows us to distinguish a value of type a , which is available now, from a value of type $\bigcirc a$, which represents data of type a arriving in the next time step. A language with such a modal type operator \bigcirc has been recently implemented as an embedded language in Haskell called *Rattus* [2].

In *Rattus*, signals can be implemented as follows:¹

```
data Sig a = a ::: ( $\bigcirc$ (Sig a))
```

That is, a signal of type *Sig a* consists of a value of type a now and a signal of type *Sig a* later, thus separating consecutive values of the signal by one time step. Instead of hiding the definition of *Sig* from the user, *Rattus* ensures the operational guarantees of causality and absence of space leaks via its type system.

However, the use of the \bigcirc modality limits *Rattus* to *synchronous* reactive programs where all components of the program progress according to a *global clock*. This is witnessed by the fact that we can implement the following function that takes two delayed integers and produces their delayed sum:

```
add ::  $\bigcirc$ Int →  $\bigcirc$ Int →  $\bigcirc$ Int
add x y = delay (adv x + adv y)
```

This only works because the two delayed integers x and y are guaranteed to arrive at the same time, namely at the next tick of the global clock.

Computing according to a *global clock* is a reasonable assumption for many contexts such as simulations and games as well as typical application domains of synchronous (dataflow) languages [6, 9, 33] such as real-time and embedded systems. However, for many applications, e.g. GUIs and concurrent systems, the notion of a global clock may not be natural and may lead to inefficiencies. A global clock would have to run at least as fast as the updates to the fastest input signal. For example, a GUI may have a component (say a text field) that should react only to keyboard input, while another component (say a button) should only react to mouse clicks. However, in a synchronous implementation of this, both components have to react and perform a computation whenever the user performs a mouse click or uses the keyboard, even though the reaction of one of the components is merely to report that no updates need to be performed on it.

In this paper, we present *Async Rattus*, an embedded modal FRP language that replaces the *single global clock* of *Rattus* with *dynamic local clocks* that enable asynchronous computations. *Async Rattus* is based on the *Async RaTT*

¹ The data constructor `:::` for *Sig* is written as an infix operator, similarly to `:` for lists.

calculus for asynchronous FRP that has recently been proposed by Bahr and Møgelberg [5] and has been shown to ensure causality and absence of space leaks. Moreover, `Async Rattus` is implemented as a shallowly embedded language in Haskell, which means that `Async Rattus` programs can seamlessly interact with regular Haskell code and thus also have access to Haskell’s rich library ecosystem.

Similarly to `Rattus`, the implementation of `Async Rattus` consists of a library that implements the primitives of the language along with a plugin for the Glasgow Haskell Compiler (GHC). The plugin is needed to check the language’s more restrictive variable scope rules and to ensure the eager evaluation strategy that is necessary to obtain the operational properties. However, `Async Rattus` requires an additional novelty: The underlying core calculus of `Async Rattus` requires explicit *clocks annotations* in the program. These annotations are necessary to keep track of the dynamic data dependencies in FRP programs at runtime. Instead of relying on the programmer to provide these clock annotations, the `Async Rattus` plugin infers them and then inserts them into the intermediate code generated by GHC from the `Async Rattus` code.

The remainder of the paper is structured as follows: Section 2 describes the syntax and semantics of `Async Rattus` with a particular focus on its non-standard typing rules. Section 3 illustrates the expressiveness of `Async Rattus` and its interaction with Haskell with the help of a selection of example programs. Section 4 describes how `Async Rattus` is implemented as an embedded language in Haskell. Finally, Section 5 and section 6 discuss related and future work, respectively.

`Async Rattus` is available as a package on Hackage [18]. Apart from the language implementation itself, the package also contains an FRP library implemented in `Async Rattus` along with example programs using this library. In particular, it contains the full source code of all examples presented in this paper.

2 Introduction to `Async Rattus`

`Async Rattus` differs from Haskell in two major ways. Firstly, `Async Rattus` is eagerly evaluated. This difference in the operational semantics is crucial for the language’s ability to avoid space leaks. Secondly, `Async Rattus` extends Haskell’s type system with two type modalities, \bigcirc and \square . A value of type $\bigcirc a$ is a delayed computation that waits for an event upon which it will produce a value of type a , whereas a value of type $\square a$ is a thunk that can be forced at any time, now or in the future, to produce a value of type a .

Each value $x :: \bigcirc a$ waits for an event to occur before it can be evaluated to a value of type a . Intuitively, an element of type $\bigcirc a$ is a pair (θ, f) consisting of a (local) *clock* θ and a thunk f , so that f can be forced to compute a value of type a as soon as the clock θ ticks. This intuition is witnessed by the two functions $cl :: \bigcirc a \rightarrow Clock$ and $adv :: \bigcirc a \rightarrow a$ that project out these two components. Conversely, we can construct a value of type $\bigcirc a$ by providing these two components using the function $delay :: Clock \rightarrow a \rightarrow \bigcirc a$. Using these functions, we can implement a function that takes a delayed integer and increments it:

$$\begin{array}{c}
\frac{\Gamma, \checkmark_{\text{cl}(t)} \vdash t :: A}{\Gamma \vdash \text{delay}_{\text{cl}(t)} t :: \bigcirc A} \quad \frac{\checkmark \notin \Gamma' \text{ or } A \text{ stable}}{\Gamma, x :: A, \Gamma' \vdash x :: A} \quad \frac{\Gamma \vdash t :: \bigcirc A}{\Gamma \vdash \text{unbox } t :: A} \quad \frac{\Gamma^\square \vdash t :: A}{\Gamma \vdash \text{box } t :: \bigcirc A} \\
\\
\frac{\Gamma \vdash s :: \bigcirc A \quad \Gamma \vdash t :: \bigcirc B \quad \checkmark \notin \Gamma'}{\Gamma, \checkmark_{\text{cl}(s)} \sqcup \text{cl}(t), \Gamma' \vdash \text{select } s t :: \text{Select } A B} \quad \frac{\Gamma \vdash t :: \bigcirc A \quad \checkmark \notin \Gamma'}{\Gamma, \checkmark_{\text{cl}(t)}, \Gamma' \vdash \text{adv } t :: A} \quad \frac{}{\Gamma \vdash \text{never} :: \bigcirc A} \\
\\
\text{where} \quad \begin{array}{l} \cdot^\square = \cdot \\ (\Gamma, \checkmark_\theta)^\square = \Gamma^\square \end{array} \quad (\Gamma, x :: A)^\square = \begin{cases} \Gamma^\square, x :: A & \text{if } A \text{ stable} \\ \Gamma^\square & \text{otherwise} \end{cases}
\end{array}$$

Fig. 1. Select typing rules for Async Rattus.

$$\begin{array}{l}
\text{incr} :: \bigcirc \text{Int} \rightarrow \bigcirc \text{Int} \\
\text{incr } x = \text{delay}_{\text{cl}(x)} (\text{adv } x + 1)
\end{array}$$

This makes explicit the fact that the integers produced by the delayed computations $x :: \bigcirc \text{Int}$ and $\text{incr } x :: \bigcirc \text{Int}$ become available at the same time. We write the first argument of `delay` as a subscript. As we will see shortly, these clock arguments are annotations that can always be inferred from the context.

Intuitively speaking, a clock θ represents a form of temporal data dependency. In the above example, `incr x` produces an integer as soon as x does. A reactive program processes data received from some input channels (e.g. keyboard, mouse, network connection). A clock θ is simply a set of such input channels, and a tick on θ means that data has been received on some input channel $c \in \theta$. Hence, `delayθ t` means that we delay evaluation of t until we have received data on some input channel $c \in \theta$.

2.1 Typing rules for delayed computations

The type signatures that we have given for `delay` and `adv` above are a good starting point to understand what `delay` and `adv` do, but they are too permissive. If `delay` were simply a function of type `delay :: Clock → a → ⋄a`, we could delay arbitrary computations – and the data they depend on – into the future, which will cause space leaks. Similarly, `adv` cannot be simply a function of type `⋄a → a` as this would allow us to execute future computations now, which would break causality.

Instead, the type system of Async Rattus constrains the use of `delay` and `adv` to ensure that programs are causal and do not cause space leaks. Figure 1 shows the most important typing rules of Async Rattus, where we write Γ for typing contexts; s, t for terms; x, y, z for variables; A, B, C for types, and θ for clocks.

Async Rattus uses a *Fitch-style* type system [11], which means that a typing context Γ may contain *tokens* of the form \checkmark_θ (pronounced “tick of clock θ ” or just “tick”) in addition to the usual typing assumptions of the form $x :: A$. We can think of \checkmark_θ as denoting the passage of one time step on the clock θ , i.e. all variables to the left of \checkmark_θ are one time step older w.r.t. the clock θ compared to

those to the right of \checkmark_θ . The rule for `delay` introduces a token $\checkmark_{\text{cl}(t)}$ in the typing context Γ . This means that t sees the variables in Γ as one time step older w.r.t. clock $\text{cl}(t)$, thus matching the intuitive semantics of `delay` which delays evaluation of t by one time step on the clock $\text{cl}(t)$.

The variable introduction rule explains how ticks influence which variables are in scope: A variable occurring to the left of a tick is no longer in scope unless it is of a type that is time-independent. We call these time-independent types *stable* types, and in particular all base types such as *Int* and *Bool* are stable as are any types of the form $\Box a$. For instance, function types are not stable, and thus functions cannot be moved into the future, which means that the type checker must reject the following definition:

$$\begin{aligned} \text{mapLater} &:: (a \rightarrow b) \rightarrow \bigcirc a \rightarrow \bigcirc b \\ \text{mapLater } f \ x &= \text{delay}_{\text{cl}(x)} (f (\text{adv } x)) \quad \text{-- } f \text{ is out of scope} \end{aligned}$$

The problem is that functions may store time-dependent data in their closure and thus moving functions into the future could lead to space leaks. We shall return to stable types later when we discuss the \Box type modality in section 2.2.

The typing rule for `adv` only allows us to advance a delayed computation $t :: \bigcirc A$, if we know that the clock of t has already ticked, which is witnessed by the token $\checkmark_{\text{cl}(t)}$ in the context. That is, `delay` looks ahead one time step on a clock θ and `adv` then allows us to go back to the present. Variable bindings made in the future, i.e. the context Γ' in the typing rule for `adv`, are therefore not accessible once we returned to the present.

We can now see why the `add` function from section 1 does not type check:

$$\begin{aligned} \text{add} &:: \bigcirc \text{Int} \rightarrow \bigcirc \text{Int} \rightarrow \bigcirc \text{Int} \\ \text{add } x \ y &= \text{delay}_\theta (\text{adv } x + \text{adv } y) \quad \text{-- no suitable clock } \theta \end{aligned}$$

The problem is that there is no clock θ so that both subexpressions `adv x` and `adv y` type check. The former only type checks if $\theta = \text{cl}(x)$ and the latter only type checks if $\theta = \text{cl}(y)$. It might very well be that the clocks of x and y are the same at runtime, e.g. if $y = \text{incr } x$, but that is not guaranteed at compile time.

In order to deal with more than one delayed computation, *Async Rattus* allows us to form the union $\theta \sqcup \theta'$ of two clocks θ and θ' . The new clock $\theta \sqcup \theta'$ ticks whenever θ or θ' ticks. We interact with such clocks via the `select` primitive, which takes two delayed computations $s :: \bigcirc A$ and $t :: \bigcirc B$ as arguments, given that a tick on the clock $\text{cl}(s) \sqcup \text{cl}(t)$ is in the context. In return, `select` produces a value of type *Select A B*, which is defined as follows:²

$$\mathbf{data} \ \text{Select } a \ b = \text{Fst } a (\bigcirc b) \mid \text{Snd } (\bigcirc a) \ b \mid \text{Both } a \ b$$

Given that $\text{cl}(s) \sqcup \text{cl}(t)$ has ticked, `select s t` produces a value with constructor *Fst*, *Snd*, or *Both* if $\text{cl}(s)$ ticked before, after, or at the same time as $\text{cl}(t)$, respectively. For example, the following function waits for two integers and returns the integer that arrives first:

² *Async Rattus* is a strict language and all type definitions are strict by default.

$$\begin{aligned}
& \mathit{first} :: \bigcirc \mathit{Int} \rightarrow \bigcirc \mathit{Int} \rightarrow \bigcirc \mathit{Int} \\
& \mathit{first} \ x \ y = \mathit{delay}_{\text{cl}(x) \sqcup \text{cl}(y)} \ (\mathbf{case\ select} \ x \ y \ \mathbf{of} \\
& \qquad \mathit{Fst} \ x' \ _ \rightarrow x' \\
& \qquad \mathit{Snd} \ _ \ y' \rightarrow y' \\
& \qquad \mathit{Both} \ x' \ _ \rightarrow x')
\end{aligned}$$

With the help of `select`, we can also implement the `add` function from the introduction, but we have to revise the return type:

$$\begin{aligned}
& \mathit{add} :: \bigcirc \mathit{Int} \rightarrow \bigcirc \mathit{Int} \rightarrow \bigcirc (\mathit{Int} \oplus \bigcirc \mathit{Int}) \\
& \mathit{add} \ x \ y = \mathit{delay}_{\text{cl}(x) \sqcup \text{cl}(y)} \ (\mathbf{case\ select} \ x \ y \ \mathbf{of} \\
& \qquad \mathit{Fst} \ x' \ y' \rightarrow \mathit{Inr} \ (\mathit{delay}_{\text{cl}(y')} \ (x' + \mathit{adv} \ y')) \\
& \qquad \mathit{Snd} \ x' \ y' \rightarrow \mathit{Inr} \ (\mathit{delay}_{\text{cl}(x')} \ (\mathit{adv} \ x' + y')) \\
& \qquad \mathit{Both} \ x' \ y' \rightarrow \mathit{Inl} \ (x' + y'))
\end{aligned}$$

where \oplus is the (strict) sum type with the two constructors $\mathit{Inl} :: a \rightarrow a \oplus b$ and $\mathit{Inr} :: b \rightarrow a \oplus b$. The type now reflects the fact that we might have to wait two ticks (of two different clocks) to obtain the result. From now on we will elide the clock annotations for `delay` as it will always be obvious from the context what the annotation needs to be. Indeed, `Async Rattus` will infer the correct clock annotation and insert it automatically during compilation.

2.2 Typing rules for stable computations

As we have seen above, only variables of *stable types* can be moved across ticks and thus into the future. A type A is stable if all occurrences of \bigcirc and function types in A are guarded by \square . For example $\mathit{Int} \oplus \mathit{Float}$, $\square(\mathit{Int} \rightarrow \mathit{Float})$, and $\square(\bigcirc \mathit{Int}) \oplus \mathit{Int}$ are stable types, but $\square \mathit{Int} \rightarrow \mathit{Float}$, $\bigcirc \mathit{Int}$, and $\bigcirc(\square \mathit{Int})$ are not. That is, the \square modality can be used to turn any type into a stable type, thus making it possible to move functions into the future safely without risking space leaks. Using \square , we can implement the `map` function for \bigcirc :

$$\begin{aligned}
& \mathit{mapLater} :: \square(a \rightarrow b) \rightarrow \bigcirc a \rightarrow \bigcirc b \\
& \mathit{mapLater} \ f \ x = \mathit{delay} \ (\mathit{unbox} \ f \ (\mathit{adv} \ x))
\end{aligned}$$

where `unbox` is simply a function of type $\square a \rightarrow a$.

A value of type \square can only be constructed using the introduction form `box`. The typing rule for `box`, shown in Figure 1, restricts the typing context to Γ^\square , which is obtained from Γ by removing all variables of non-stable types and all \surd_{θ} tokens. Hence, the type system ensures that boxed values may only refer to variables of a stable type.

2.3 Recursive definitions

Similarly to `Rattus` and other synchronous FRP languages [3, 26], signals can be defined in `Async Rattus` by the following definition:

$$\mathbf{data} \ \mathit{Sig} \ a = a :: (\bigcirc(\mathit{Sig} \ a))$$

That is, a signal of type $Sig\ a$ consists of a current value of type a and a future update to the signal of type $\bigcirc(Sig\ a)$. We can define a map function for signals, but similarly to the $mapLater$ function on the \bigcirc modality, the function argument has to be boxed:

$$\begin{aligned} map &:: \Box(a \rightarrow b) \rightarrow Sig\ a \rightarrow Sig\ b \\ map\ f\ (x \::\ xs) &= \text{unbox } f\ x \::\ \text{delay } (map\ f\ (\text{adv } xs)) \end{aligned}$$

In order to ensure productivity of recursive function definitions, `Async Rattus` requires that recursive function calls, such as $map\ f\ (\text{adv } xs)$ above, have to be guarded by a `delay`. More precisely, such a recursive occurrence may only occur in a context Γ that contains a \surd_θ .

It may seem that we can get around the need to box the function argument to map by instead using a nested recursive function run :

$$\begin{aligned} map &:: (a \rightarrow b) \rightarrow Sig\ a \rightarrow Sig\ b \\ map\ f &= run\ \mathbf{where} \\ run &:: Sig\ a \rightarrow Sig\ b \\ run\ (x \::\ xs) &= f\ x \::\ \text{delay } (run\ (\text{adv } xs)) \quad \text{-- } f \text{ is out of scope} \end{aligned}$$

However, in this definition, f would no longer be in scope. To type check a recursive definition in a typing context Γ , the body of the recursive definition has to type check in the typing context Γ^\square . For example, for $run \::\ Sig\ a \rightarrow Sig\ b$ to type check in the context $\Gamma = f \::\ a \rightarrow b$, the body of run has to type check in the context $\Gamma^\square = \cdot$, i.e. f is no longer in scope in the body of run .

While the definition of the signal type looks superficially the same as in synchronous FRP languages, its semantics is quite different because \bigcirc has a different semantics: Updates to a signal do not come at the rate given by a global clock, but rather by some local clock, which may in turn change dynamically. For example, we can implement the constant signal function as follows:

$$\begin{aligned} const &:: a \rightarrow Sig\ a \\ const\ x &= x \::\ \text{never} \end{aligned}$$

where $\text{never} \::\ \bigcirc b$ is simply a delayed computation with a clock that will never tick. The $const$ signal function might seem pointless, but we can combine it with a combinator that switches from one signal to another signal:

$$\begin{aligned} switch &:: Sig\ a \rightarrow \bigcirc(Sig\ a) \rightarrow Sig\ a \\ switch\ (x \::\ xs)\ d = x \::\ \text{delay } (\mathbf{case\ select } xs\ d\ \mathbf{of} \\ &\quad Fst\ xs'\ d' \rightarrow switch\ xs'\ d' \\ &\quad Snd\ _ \ d' \rightarrow d' \\ &\quad Both\ xs'\ d' \rightarrow d') \end{aligned}$$

A signal $switch\ s\ e$ first behaves like s , but as soon as the clock of e ticks the signal behaves like the signal produced by e . For example, given a value $x \::\ a$ and a delayed value $y \::\ \bigcirc a$, we can produce a signal that first has the value x and then, as soon as y arrives, has the value that y produces:

$$\begin{aligned} \text{step} &:: a \rightarrow \bigcirc a \rightarrow \text{Sig } a \\ \text{step } x \ y &= \text{switch } (\text{const } x) (\text{delay } (\text{const } (\text{adv } y))) \end{aligned}$$

2.4 Operational semantics

One of the main features of **Async Rattus** is that it avoids space leaks. To this end, its type system prevents us from moving arbitrary computations into the future. In addition, the operational semantics is carefully designed so that computations are executed as soon as the data they depend on is available. In short, this means that **Async Rattus** uses an eager evaluation semantics except for **delay** and **box**. That is, arguments are evaluated to values before they are passed on to functions, but special rules apply to **delay** and **box**. Furthermore, **Async Rattus** requires strict data types and any use of lazy data types will produce a warning.

Following the temporal interpretation of the \bigcirc modality, its introduction form delay_θ does not eagerly evaluate its argument since we have to wait until input data arrives, namely when the clock θ ticks. For example, in the expression $\text{delay } (\text{adv } x + 1)$, we cannot evaluate $\text{adv } x + 1$ until the delayed integer value of $x :: \bigcirc \text{Int}$ arrives, which is one time step from now on the clock $\text{cl}(x)$. However, evaluation is only delayed until the clock $\text{cl}(x)$ ticks, and this delay is reversed by **adv**. For example, $\text{adv } (\text{delay } (1 + 1))$ evaluates immediately to 2.

Several modal FRP calculi [3–5, 26] have a similar operational semantics to achieve the same memory property that **Async Rattus** has. However, similarly to **Rattus** [2], **Async Rattus** uses a slightly more eager evaluation strategy for **delay**: Recall that $\text{delay}_\theta t$ delays the computation t by one time step and that **adv** reverses such a delay. The operational semantics reflects this intuition by first evaluating every term t that occurs as $\text{delay}_{\text{cl}(t)} (\dots \text{adv } t \dots)$ before evaluating **delay**. In other words, $\text{delay}_{\text{cl}(t)} (\dots \text{adv } t \dots)$ is equivalent to

$$\text{let } x = t \text{ in } \text{delay}_{\text{cl}(x)} (\dots \text{adv } x \dots)$$

Similarly, $\text{delay}_{\text{cl}(s) \sqcup \text{cl}(t)} (\dots \text{select } s \ t \dots)$ is equivalent to

$$\text{let } x = s; y = t \text{ in } \text{delay}_{\text{cl}(x) \sqcup \text{cl}(y)} (\dots \text{select } x \ y \dots)$$

This generalisation of the operational semantics of **delay** allows us to lift some of the restrictions present in the **Async RaTT** calculus [5] on which **Async Rattus** is based: **Async Rattus** allows more than one \checkmark_θ in the typing context, i.e. **delay** can be nested; it does not prohibit lambda abstractions in the presence of a \checkmark_θ ; and both **adv** and **select** can be used with arbitrary terms, not just variables. In section 4, we introduce the technical device that makes it safe to lift these restrictions of **Async RaTT**, namely the *single tick* program transformation.

3 Reactive Programming in Async Rattus

In this section, we demonstrate the expressiveness of **Async Rattus** with a number of examples. The full source code of abridged examples along with further example programs can be found in the **Async Rattus** package [18].

3.1 A simple FRP application

To support FRP using the *Sig* type, we implement a library of standard FRP combinators [5] in `Async Rattus`. Figure 2 lists a small subset of this library.

In addition to the core language constructs of `Async Rattus` presented in section 2, the language also features primitives for interacting with its environment via *input channels* and *output channels*. The simplest input channel is a timer that ticks at a fixed interval (given in milliseconds):

$$timer :: Int \rightarrow \square(\bigcirc())$$

Input channels have the type $\square(\bigcirc a)$, but they can always be turned into signals:

$$\begin{aligned} mkSig &:: \square(\bigcirc a) \rightarrow \bigcirc(Sig\ a) \\ mkSig\ b &= \mathit{delay}\ (\mathit{adv}\ (\mathit{unbox}\ b) :: mkSig\ b) \\ timerSig &:: Int \rightarrow Sig\ () \\ timerSig\ n &= () :: mkSig\ (timer\ n) \end{aligned}$$

That is, the signal *timerSig n* produces a new value every *n* microseconds. As an example, this timer input channel can be used for implementing the *derivative* and *integral* combinators in Figure 2 (as in Bahr and Møgelberg [5]).

More general input channels can be constructed using

$$getInput :: IO\ (\square(\bigcirc a), (a \rightarrow IO\ ()))$$

which produces an input channel of type $\square(\bigcirc a)$ that we can feed from Haskell by using the callback function of type $a \rightarrow IO\ ()$. Whenever the callback function is called with an argument *v*, the input channel will produce a new value *v*. Library authors can use *getInput* to provide an `Async Rattus` interface to external resources. For example, we can implement an input channel for the console with the following Haskell code:

```
consoleInput :: IO (\square(\bigcirc Text))
consoleInput = do (inp, cb) <- getInput
                 let loop = do line <- getLine; cb line; loop
                 forkIO loop
                 return inp
```

For output channels, `Async Rattus` provides the function

$$setOutput :: Sig\ a \rightarrow (a \rightarrow IO\ ()) \rightarrow IO\ ()$$

which, if given a signal *s* and a callback function *f*, calls *f v* whenever *s* produces a new value *v*. To support a variety of programming styles beyond the *Sig* type, the type of *setOutput* is in fact more general:

$$setOutput :: Producer\ p\ a \Rightarrow p \rightarrow (a \rightarrow IO\ ()) \rightarrow IO\ ()$$

```

current  :: Sig a → a
future   :: Sig a →  $\bigcirc$ (Sig a)
map      ::  $\square(a \rightarrow b) \rightarrow \text{Sig } a \rightarrow \text{Sig } b$ 
mapD     ::  $\square(a \rightarrow b) \rightarrow \bigcirc(\text{Sig } a) \rightarrow \bigcirc(\text{Sig } b)$ 
const    :: a → Sig a
scan     :: (Stable b) ⇒  $\square(b \rightarrow a \rightarrow b) \rightarrow b \rightarrow \text{Sig } a \rightarrow \text{Sig } b$ 
zipWith  :: (Stable a, Stable b) ⇒  $\square(a \rightarrow b \rightarrow c) \rightarrow \text{Sig } a \rightarrow \text{Sig } b \rightarrow \text{Sig } c$ 
interleave ::  $\square(a \rightarrow a \rightarrow a) \rightarrow \bigcirc(\text{Sig } a) \rightarrow \bigcirc(\text{Sig } a) \rightarrow \bigcirc(\text{Sig } a)$ 
switch   :: Sig a →  $\bigcirc(\text{Sig } a) \rightarrow \text{Sig } a$ 
switchS  :: Sig a →  $\bigcirc(a \rightarrow \text{Sig } a) \rightarrow \text{Sig } a$ 
derivative :: Sig Float → Sig Float
integral  :: Float → Sig Float → Sig Float

```

Fig. 2. Simple FRP library.

Instances of *Producer p a* are types *p* that produce values of type *a* over time. In particular, we have instances *Producer* ($\bigcirc(\text{Sig } a)$) *a* and *Producer* (*Sig a*) *a*.

Similarly to *getInput*, library authors can use *setOutput* to provide an *Async Rattus* interface for external resources. For example, we may wish to provide a way to output a signal of integers by printing each new value to the console:

```

intOutput :: Producer p Int ⇒ p → IO ()
intOutput sig = setOutput sig print

```

As a simple example, we implement a console application that waits for the user to enter a line, and then outputs the length of the user’s input:³

```

main = do inp ← mkSignal ($) consoleInput      -- inp ::  $\bigcirc(\text{Sig Text})$ 
           let outSig = mapD (box length) inp    -- outSig ::  $\bigcirc(\text{Sig Int})$ 
           intOutput outSig
           startEventLoop

```

In the last line we call *startEventLoop* :: *IO ()* which starts the event loop that executes output actions registered by *setOutput*. We will look at a more comprehensive example in section 3.3.

3.2 Filtering functions

As Bahr and Møgelberg [5] have observed, the *Sig* type does not support a filter function *filter* :: $\square(a \rightarrow \text{Bool}) \rightarrow \text{Sig } a \rightarrow \text{Sig } a$. The problem is that a signal of type *Sig a* must produce a value of type *a* at every tick of its current clock. In order to check the predicate *p* :: $\square(a \rightarrow \text{Bool})$ we must wait until the input signal ticks and produces a new value *v* :: *a*. Hence, we must produce a value for the output signal for that tick as well, regardless of whether *p v* is true or not. Instead, we can implement a variant of *filter* with the following type:⁴

³ (\$) is the infix notation for the function *fmap* :: *Functor f* ⇒ (*a* → *b*) → *f a* → *f b*

⁴ *Maybe'* is a strict variant of the standard *Maybe* type.

```

filterMap  ::  $\Box(a \rightarrow \text{Maybe}' b) \rightarrow \text{Sig } a \rightarrow \text{IO } (\Box(\bigcirc(\text{Sig } b)))$ 
filterMapD ::  $\Box(a \rightarrow \text{Maybe}' b) \rightarrow \bigcirc(\text{Sig } a) \rightarrow \text{IO } (\Box(\bigcirc(\text{Sig } b)))$ 
filter     ::  $\Box(a \rightarrow \text{Bool}) \rightarrow \text{Sig } a \rightarrow \text{IO } (\Box(\bigcirc(\text{Sig } a)))$ 
filterD    ::  $\Box(a \rightarrow \text{Bool}) \rightarrow \bigcirc(\text{Sig } a) \rightarrow \text{IO } (\Box(\bigcirc(\text{Sig } a)))$ 
trigger    ::  $(\text{Stable } a, \text{Stable } b) \Rightarrow \Box(a \rightarrow b \rightarrow c) \rightarrow \text{Sig } a \rightarrow \text{Sig } b \rightarrow \text{IO } (\Box(\text{Sig } c))$ 
triggerD   ::  $\text{Stable } b \Rightarrow \Box(a \rightarrow b \rightarrow c) \rightarrow \bigcirc(\text{Sig } a) \rightarrow \text{Sig } b \rightarrow \text{IO } (\Box(\bigcirc(\text{Sig } c)))$ 

```

Fig. 3. Filter functions in Async Rattus.

```

filter' ::  $\Box(a \rightarrow \text{Bool}) \rightarrow \text{Sig } a \rightarrow \text{Sig } (\text{Maybe}' a)$ 
filter' p = map (box ( $\lambda x \rightarrow \text{if } \text{unbox } p \ x \ \text{then } \text{Just}' \ x \ \text{else } \text{Nothing}'$ ))

```

This is somewhat unsatisfactory but workable. We can provide an implementation of standard FRP combinators (like those in Figure 2) that work with signals of type $\text{Sig } (\text{Maybe}' a)$ instead of $\text{Sig } a$. However, this introduces inefficiencies since programs that work with signals of type $\text{Sig } (\text{Maybe}' a)$ have to explicitly check for the *Nothing'* case for each tick.

A possible solution is to replace the modal operator \bigcirc with the derived operator F that may take several ticks to produce a result:

```

data F a = Now a | Wait ( $\bigcirc(F a)$ )
data SigF a = a ::F ( $\bigcirc(F (\text{Sig}_F a))$ )

```

That is, a value of type $F a$ is the promise of a value of type a in 0 or more (possibly infinitely many) ticks. Then the definition of Sig_F replaces \bigcirc with the composition of \bigcirc and F . That is, a signal has a current value and the promise that it will update in one or more ticks. With this type, we can implement a function $\text{filter} :: \Box(a \rightarrow \text{Bool}) \rightarrow \text{Sig}_F a \rightarrow F (\text{Sig}_F a)$ as well as corresponding versions of the functions in Figure 2.

Sadly, $\text{Sig}_F a$ still suffers from the same inefficiency as the $\text{Sig } (\text{Maybe}' a)$ type. To implement a more efficient filter function, we instead make use of the *getInput* and *setOutput* functions. By composing the two, we can turn a signal of type $\text{Sig } (\text{Maybe}' a)$ into a signal of type $\text{Sig } a$:

```

mkInputSig ::  $\text{Producer } p \ a \Rightarrow p \rightarrow \text{IO } (\Box(\bigcirc(\text{Sig } a)))$ 
mkInputSig p = do (out, cb) ← getInput
                  setOutput p cb
                  return box (mkSig out)

```

Since we have $\text{Producer } (\text{Sig } (\text{Maybe}' a)) \ a$, we can implement *filter* as follows:

```

filter ::  $\Box(a \rightarrow \text{Bool}) \rightarrow \text{Sig } a \rightarrow \text{IO } (\Box(\bigcirc(\text{Sig } a)))$ 
filter p xs = mkInputSig (filter' p xs)

```

Figure 3 lists further filter functions that can be implemented in this fashion: *filterMap* essentially composes the *filter* function with the *map* function. *trigger f xs ys* is a signal that produces a new value *unbox f x y* whenever *xs*

produces a new value, where x and y are the current values of xs and ys respectively. One can think of *trigger* as a left-biased version of *zipWith*. Finally, we also have versions of these functions that work with delayed signals. We will put this library to use in the next section.

3.3 Extended example

To demonstrate the use of the FRP library that we have developed above, we implement a simple interactive application. To this end, we extend our simple IO library, which so far consists of *consoleInput* and *intOutput*, with

```
setQuit :: Producer p a => p -> IO ()
setQuit sig = setOutput sig (\_ -> exitSuccess)
```

which quits the program as soon as the argument *sig* produces its first value.

Figure 4, shows an interactive console application that uses our simple IO API. The application maintains an integer counter *nats* that increments each second. At any time, we can show the current value of the counter by typing “show” in the console: The *showSig* signal triggers output on *showNat*. Moreover, we can manipulate the counter by either writing “negate” or a number “n” to the console, which multiplies the counter with -1 or adds n to it, respectively. Finally, we can quit the application by writing “quit”.

This example demonstrates the use of the different filter functions to construct new signals (see *quitSig*, *showSig*, *negSig*, *numSig*), the use of *interleave* and *trigger* to combine several signals (see *sig* and *showNat*, respectively), and the use of *switchS* to dynamically change the behaviour of a signal (see *nats'*).

4 Embedding Async Rattus in Haskell

The embedding of Async Rattus in Haskell consists of two main components: (1) the definition of the language’s syntax in the form of standard Haskell type and function definitions, and (2) a plugin for GHC which implements the typing rules for the modal type operators and performs the necessary program transformations to obtain the desired operational semantics for Async Rattus. Some parts of the implementation of Async Rattus are similar to that of Rattus [2], including the \square modality, the *Stable* type constraint, and the check for guarded recursion. We, therefore, focus our attention on the \bigcirc modality as it requires a novel and significantly different approach due to the presence of clocks.

Syntax. Figure 5 shows the implementation of the syntax of \bigcirc . A value *Delay* θ f of type $\bigcirc a$ consists of a clock θ and a delayed computation f . In turn, a clock θ is a finite set of input channel identifiers, and we say that θ ticks whenever any of the input channels in θ produces a new value. As soon as θ ticks, by virtue of an input channel $c \in \theta$ producing a new value v of some type b , we can run the delayed computation f by passing both c and v to f as an argument

```

everySecond :: Sig ()
everySecond = () :: mkSig (timer 1000000)
readInt :: Text → Maybe' Int
readInt text = case decimal text of Right (x, rest) | null rest → Just' x
              _ → Nothing'

nats :: Int → Sig Int
nats init = scan (box (λn _ → n + 1)) init everySecond
main = do
  console :: ○(Sig Text) ← mkSig ($) consoleInput
  quitSig :: ○(Sig Text) ← unbox ($) filterD (box (≡ "quit")) console
  showSig :: ○(Sig Text) ← unbox ($) filterD (box (≡ "show")) console
  negSig :: □(○(Sig Text)) ← filterD (box (≡ "negate")) console
  numSig :: □(○(Sig Int)) ← filterMapD (box readInt) console
  let sig :: □(○(Sig (Int → Int)))
      sig = box (interleave (box (○))
        (mapD (box (λ_ n → -n)) (unbox negSig))
        (mapD (box (λm n → m + n)) (unbox numSig)))
      let nats' :: Int → Sig Int
          nats' init = switchS (nats init)
              (delay (λn → nats' (current (adv (unbox sig)) n)))
  showNat :: □(○(Sig Int)) ← triggerD (box (λ_ n → n)) showSig (nats' 0)
  setQuit quitSig
  intOutput showNat
  startEventLoop

```

Fig. 4. Example reactive program.

of type *InputValue*. Note that the return type a of f and the type b of v are independent from each other. This is achieved by the fact that the type variable b in the definition of *InputValue* is existentially quantified. Finally, the introduction and elimination forms `delay`, `adv`, and `select` are simply implemented as \perp . These dummy implementations are replaced by their correct implementations in a program transformation performed by the compiler plugin. The correct implementations are inserted later since they depend on compile time clocks, which are inferred by the plugin and are thus not part of the surface syntax.

Scope & clock check. GHC provides a rich API that allows us to insert custom logic in several phases of its pipeline, which is sketched in Figure 6. During the type checking phase, GHC will use a constraint solver for the *Stable* type constraint provided by the plugin. Afterwards, the plugin checks the stricter scoping rules of *Async Rattus* and infers clock annotations. Checking the scoping rules for *Async Rattus* is similar to *Rattus* [2]: Variables may no longer be in scope because `delay` introduced a tick, because `adv` or `select` dropped the context to the right of a tick, or because of the restricted context Γ^\square , e.g. in the rule for `box`. The clock inference algorithm introduces an existentially quantified clock variable θ

```

data InputValue where                                -- Not exported
  InputValue :: ChanId → b → InputValue
type Clock = Set ChanId                               -- Not exported
data  $\bigcirc a = \text{Delay Clock (InputValue → a)}$       -- Constructor not exported
delay :: a →  $\bigcirc a$     adv ::  $\bigcirc a \rightarrow a$     select ::  $\bigcirc a \rightarrow \bigcirc b \rightarrow \text{Select } a b$ 
delay _ =  $\perp$           adv _ =  $\perp$                 select _ _ =  $\perp$ 

```

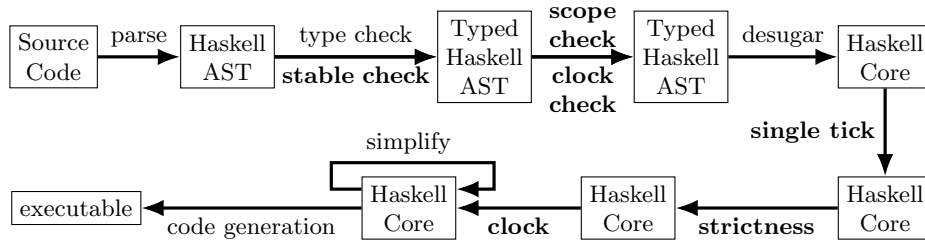
Fig. 5. Implementation of the syntax of \bigcirc .

Fig. 6. Simplified pipeline of GHC extended with Async Rattus plugin (in bold).

for each occurrence of `delay`, which is then instantiated to $\text{cl}(t)$ or $\text{cl}(s) \sqcup \text{cl}(t)$ as soon as it encounters an occurrence of `adv` t or `select` s t , respectively.

Single tick, strictness & clock transformation. GHC desugars the typed Haskell AST into the Core intermediate language, on which it then performs various simplification and optimisation steps. The *Async Rattus* plugin adds three additional transformations. Figure 7 lists the rewrite rules that are applied during these three transformations. In these rewrite rules, we use K to denote a term with a single hole that does not occur in the scope of `delay`, `adv`, `select`, `box`, or lambda abstraction, and we write $K[t]$ to denote the term obtained from K by replacing its hole with the term t .

The *single tick transformation* rules preserve the typing of the program and once the rules have been exhaustively applied, the resulting program is typable with the more restrictive typing rules of the *Async RaTT* calculus [5], which only allows at most one tick in the context, requires that `adv` and `select` be applied to variables, and disallows lambda abstractions in the scope of a tick. As a consequence, *Async Rattus* benefits from a less restrictive type system compared to *Async RaTT*, while still retaining the operational properties that have been proved for the latter. One can prove that the single tick transformation produces a well-typed *Async RaTT* program using an argument similar to Bahr [2].

The *strictness transformation* replaces lambda abstractions and let bindings so that *Async Rattus* programs have a call-by-value semantics.

The clock inference performed during an earlier compilation phase established that we can find suitable clock annotation for each occurrence of `delay`. The *clock transformation* inserts these clock annotation and thereby also replaces

Single tick transformation:

$$\begin{aligned}
\text{delay}(K[\text{adv } t]) &\longrightarrow \text{let } x = t \text{ in } \text{delay}(K[\text{adv } x]) && \text{if } t \text{ is not a variable} \\
\text{delay}(K[\text{select } s t]) &\longrightarrow \text{let } x = s \text{ in } \text{delay}(K[\text{select } x t]) && \text{if } s \text{ is not a variable} \\
\text{delay}(K[\text{select } s t]) &\longrightarrow \text{let } x = t \text{ in } \text{delay}(K[\text{select } s x]) && \text{if } t \text{ is not a variable} \\
\lambda x. K[\text{adv } t] &\longrightarrow \text{let } y = \text{adv } t \text{ in } \lambda x. (K[y]) \\
\lambda x. K[\text{select } s t] &\longrightarrow \text{let } y = \text{select } s t \text{ in } \lambda x. (K[y])
\end{aligned}$$

Strictness transformation:

$$\begin{aligned}
\lambda x. t &\longrightarrow \lambda x. \text{case } x \text{ of } _ \rightarrow t \\
\text{let } x = s \text{ in } t &\longrightarrow \text{case } s \text{ of } x \rightarrow t
\end{aligned}$$

Clock transformation:

$$\begin{aligned}
\text{delay}(K[\text{adv } x]) &\longrightarrow \text{Delay } (\text{cl}(x)) (\lambda v \rightarrow K[\text{adv}' x v]) \\
\text{delay}(K[\text{select } x y]) &\longrightarrow \text{Delay } (\text{cl}(x) \sqcup \text{cl}(y)) (\lambda v \rightarrow K[\text{select}' x y v])
\end{aligned}$$

Fig. 7. Transformation rules.

$$\begin{aligned}
\text{adv}' &:: \bigcirc a \rightarrow \text{InputValue} \rightarrow a \\
\text{adv}' (\text{Delay } _ f) & \text{ inp} = f \text{ inp} \\
\text{select}' &:: \bigcirc a \rightarrow \bigcirc b \rightarrow \text{InputValue} \rightarrow \text{Select } a b \\
\text{select}' a @ (\text{Delay } \theta_1 f) b @ (\text{Delay } \theta_2 g) v @ (\text{InputValue } ch _) & \\
= \text{if } ch \in \theta_1 \text{ then if } ch \in \theta_2 \text{ then Both } (f v) (g v) & \\
\text{else } \text{Fst } (f v) b \quad \text{else } \text{Snd } a (b v) &
\end{aligned}$$

Fig. 8. Implementation of `adv` and `select`.

the dummy functions `delay`, `adv`, and `select` from Figure 5 with their actual implementations shown in Figure 8. In addition to inserting the correct clocks, this transformation also propagates the identity of the input channel that caused the tick of the clock as well as the input value that it produced. The identity of the input channel is required for the implementation of `select` as it needs to check which of the two delayed computations to advance (see `select'` in Figure 8), whereas the input value is used to implement the `getInput` function.

5 Related Work

The use of modal types for FRP has seen much attention in recent years [2–4, 10, 16, 20, 23, 24, 26, 28, 29]. The first implementation of a modal FRP language we are aware of is `AdjS` [25], which compiles FRP programs into JavaScript. The language is based on the synchronous modal FRP calculus of Krishnaswami [26] and uses linear types to interact with GUI widgets [27]. To address the discrepancy between the synchronous programming model of `AdjS` and the inherently asynchronous nature of GUIs, the λ_{Widget} calculus of Graulund et al. [16] combines linear types with an asynchronous modal type constructor

\diamond . Similarly to `Async Rattus`, two values $x : \diamond A$ and $y : \diamond B$ arrive at some time in the future, but not necessarily at the same time and thus λ_{Widget} provides a `select` primitive to observe the relative arrival time. However, we are not aware of an implementation of a language based on λ_{Widget} .

`Async Rattus` is based on the `Async RaTT` calculus of Bahr and Møgelberg [5], which proposes the modal operator \boxplus to model asynchronous signals (in `Async Rattus` we use the simpler notation \circ instead of \boxplus). Like the synchronous calculus of Krishnaswami [26] (on which `AdjS` [25] is based), but unlike the asynchronous λ_{Widget} calculus of Graulund et al. [16], `Async RaTT` comes with a proof of operational guarantees: All `Async RaTT` programs are causal, productive, and do not have space leaks. `Async Rattus` generalises the typing rules of `Async RaTT` in three ways: It allows (1) more than one tick to occur in contexts (thus allowing nested occurrences of `delay`), (2) function definitions to occur in the scope of ticks in the context, and (3) `adv` and `select` to be applied to arbitrary terms instead of just variables. The soundness of this generalisation is based on the single tick program transformation (cf. section 4), introduced by Bahr [2], that is performed by the compiler plugin so that the resulting program will type check using the stricter typing rules of `Async RaTT`. The implementation of `Async Rattus` borrows much from the implementation of `Rattus` [2], which is based on a synchronous modal FRP calculus. However, the asynchronous setting required three key additions: Inference of clocks during type checking, an additional program transformation that inserts inferred clocks into the Haskell code, and finally a new runtime system that allows `Async Rattus` and Haskell to interact. The latter is enabled by the clock transformation (cf. section 4) and is provided to the user in the form of the `getInput` and `setOutput` functions.

6 Discussion and Future Work

The implementation of `Async Rattus` as an embedded language further demonstrates the power of GHC’s plugin API [12, 13, 17, 34]. In addition to customising the type checker and using program transformations to tweak the operational semantics, we can also use it to implement program elaboration mechanisms like `Async Rattus`’ clock inference.

Our goal is to use `Async Rattus` to further experiment with asynchronous modal FRP. Interesting topics for further work include: evaluation of asynchronous modal FRP for implementing concurrent programs and GUI applications; library design for asynchronous modal FRP in general as well as specific problem domains; and extending the language with explicit clocks and clock quantification.

Bibliography

- [1] Apfeldmus, H.: Reactive Banana (2011), URL <https://hackage.haskell.org/package/reactive-banana>
- [2] Bahr, P.: Modal FRP for all: Functional reactive programming without space leaks in Haskell. *Journal of Functional Programming* **32**, e15 (2022), ISSN 0956-7968, 1469-7653, publisher: Cambridge University Press
- [3] Bahr, P., Graulund, C.U., Møgelberg, R.E.: Simply RaTT: A Fitch-style modal calculus for reactive programming without space leaks. *Proceedings of the ACM on Programming Languages* **3**(ICFP), 1–27 (2019)
- [4] Bahr, P., Graulund, C.U., Møgelberg, R.E.: Diamonds are not forever: liveness in reactive programming with guarded recursion. *Proceedings of the ACM on Programming Languages* **5**(POPL), 2:1–2:28 (Jan 2021), 00002
- [5] Bahr, P., Møgelberg, R.E.: Asynchronous Modal FRP. *Proceedings of the ACM on Programming Languages* **7**(ICFP), 205:476–205:510 (2023)
- [6] Berry, G., Cosserat, L.: The estrel synchronous programming language and its mathematical semantics. In: Brookes, S.D., Roscoe, A.W., Winskel, G. (eds.) *Seminar on Concurrency*, pp. 389–448, Springer Berlin Heidelberg, Berlin, Heidelberg, DE (1985), ISBN 978-3-540-39593-5
- [7] Blackheath, S.: Sodium (2012), URL <https://hackage.haskell.org/package/sodium>
- [8] Bärenz, M., Perez, I.: Rhine: FRP with type-level clocks. In: *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*, pp. 145–157, Haskell 2018, Association for Computing Machinery, New York, NY, USA (Sep 2018), ISBN 978-1-4503-5835-4
- [9] Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.A.: Lustre: A declarative language for real-time programming. In: *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 178–188, POPL '87, ACM, New York, NY, USA (1987), ISBN 0-89791-215-2
- [10] Cave, A., Ferreira, F., Panangaden, P., Pientka, B.: Fair Reactive Programming. In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 361–372, POPL '14, ACM, San Diego, California, USA (2014), ISBN 978-1-4503-2544-8
- [11] Clouston, R.: Fitch-style modal lambda calculi. In: Baier, C., Dal Lago, U. (eds.) *Foundations of Software Science and Computation Structures*, vol. 10803, pp. 258–275, Springer, Springer International Publishing, Cham (2018), ISBN 978-3-319-89366-2
- [12] Diatchki, I.S.: Improving Haskell types with SMT. In: *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, pp. 1–10, Haskell '15, Association for Computing Machinery (Aug 2015)
- [13] Elliott, C.: Compiling to categories. *Proceedings of the ACM on Programming Languages* **1**(ICFP), 27:1–27:27 (Aug 2017)

- [14] Elliott, C., Hudak, P.: Functional reactive animation. In: Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming, pp. 263–273, ICFP '97, ACM, New York, NY, USA (1997), ISBN 0-89791-918-1
- [15] Elliott, C.M.: Push-pull Functional Reactive Programming. In: Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell, pp. 25–36, Haskell '09, ACM, New York, NY, USA (2009), ISBN 978-1-60558-508-6, 00145 event-place: Edinburgh, Scotland
- [16] Graulund, C.U., Szamozvancev, D., Krishnaswami, N.: Adjoint reactive gui programming. In: FoSSaCS, pp. 289–309 (2021)
- [17] Gundry, A.: A typechecker plugin for units of measure: domain-specific constraint solving in GHC Haskell. In: Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell, pp. 11–22, Haskell '15, Association for Computing Machinery, New York, NY, USA (Aug 2015)
- [18] Houlborg, E., Rørdam, G., Bahr, P.: Async Rattus (2023), URL <https://hackage.haskell.org/package/AsyncRattus>
- [19] Hudak, P., Courtney, A., Nilsson, H., Peterson, J.: Arrows, Robots, and Functional Reactive Programming. In: Advanced Functional Programming, Lecture Notes in Computer Science, vol. 2638, Springer Berlin / Heidelberg (2003), ISBN 978-3-540-40132-2
- [20] Jeffrey, A.: LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs. In: Claessen, K., Swamy, N. (eds.) Proceedings of the sixth workshop on Programming Languages meets Program Verification, PLPV 2012, Philadelphia, PA, USA, January 24, 2012, pp. 49–60, ACM, Philadelphia, PA, USA (2012), ISBN 978-1-4503-1125-0
- [21] Jeffrey, A.: Functional reactive types. In: Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), pp. 54:1–54:9, CSL-LICS '14, ACM, New York, NY, USA (2014), ISBN 978-1-4503-2886-9
- [22] Jeltsch, W.: Grapefruit (2007), URL <https://hackage.haskell.org/package/grapefruit>
- [23] Jeltsch, W.: Towards a common categorical semantics for linear-time temporal logic and functional reactive programming. *Electronic Notes in Theoretical Computer Science* **286**, 229–242 (2012)
- [24] Jeltsch, W.: Temporal logic with "until", functional reactive programming with processes, and concrete process categories. In: Proceedings of the 7th Workshop on Programming Languages Meets Program Verification, pp. 69–78, PLPV '13, ACM, New York, NY, USA (2013), ISBN 978-1-4503-1860-0
- [25] Krishnaswami, N.R.: AdjS compiler (2013), URL <https://github.com/neel-krishnaswami/adjS>
- [26] Krishnaswami, N.R.: Higher-order Functional Reactive Programming Without Spacetime Leaks. In: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming, pp. 221–232, ICFP '13, ACM, Boston, Massachusetts, USA (2013), ISBN 978-1-4503-2326-0

- [27] Krishnaswami, N.R., Benton, N.: A semantic model for graphical user interfaces. In: Proceedings of the 16th ACM SIGPLAN international conference on Functional programming, pp. 45–57, ICFP '11, Association for Computing Machinery, New York, NY, USA (Sep 2011), ISBN 978-1-4503-0865-6
- [28] Krishnaswami, N.R., Benton, N.: Ultrametric semantics of reactive programs. In: 2011 IEEE 26th Annual Symposium on Logic in Computer Science, pp. 257–266, IEEE Computer Society, Washington, DC, USA (June 2011), ISSN 1043-6871
- [29] Krishnaswami, N.R., Benton, N., Hoffmann, J.: Higher-order functional reactive programming in bounded space. In: Field, J., Hicks, M. (eds.) Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22–28, 2012, pp. 45–58, ACM, Philadelphia, PA, USA (2012), ISBN 978-1-4503-1083-3
- [30] Patai, G.: Efficient and Compositional Higher-Order Streams. In: Mariño, J. (ed.) Functional and Constraint Logic Programming, pp. 137–154, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2011), ISBN 978-3-642-20775-4
- [31] Perez, I., Bärenz, M., Nilsson, H.: Functional reactive programming, refactored. In: Proceedings of the 9th International Symposium on Haskell, pp. 33–44, Haskell 2016, Association for Computing Machinery, New York, NY, USA (Sep 2016), ISBN 978-1-4503-4434-0
- [32] Ploeg, A.v.d., Claessen, K.: Practical principled FRP: forget the past, change the future, FRPNow! In: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, pp. 302–314, ICFP 2015, Association for Computing Machinery, Vancouver, BC, Canada (Aug 2015), ISBN 978-1-4503-3669-7, 00019
- [33] Pouzet, M.: Lucid synchrone, version 3. Tutorial and reference manual. Université Paris-Sud, LRI 1, 25 (2006)
- [34] Prott, K.O., Teegen, F., Christiansen, J.: Embedding Functional Logic Programming in Haskell via a Compiler Plugin. In: Hanus, M., Incezan, D. (eds.) Practical Aspects of Declarative Languages, pp. 37–55, Lecture Notes in Computer Science, Springer Nature Switzerland (2023)
- [35] Trinkle, R.: Reflex (2016), URL <https://reflex-frp.org>