Asynchronous Reactive Programming with Modal Types in Haskell

Patrick Bahr, Emil Houlborg, and Gregers Thomas Skat Rørdam

IT University of Copenhagen

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.

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 describes signals of type a, an FRP library may provide a function $map :: (a \to b) \to Sig\ a \to Sig\ b$ 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 its API 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 times 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:

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

That is, a signal of type $\mathit{Sig}\ a$ is an element of type a now and a signal of type $\mathit{Sig}\ a$ later, thus separating consecutive elements 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:

$$\begin{aligned} &\mathit{add} :: \bigcirc \mathit{Int} \to \bigcirc \mathit{Int} \to \bigcirc \mathit{Int} \\ &\mathit{add}\ x\ y = \mathsf{delay}\ (\mathsf{adv}\ x + \mathsf{adv}\ y) \end{aligned}$$

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

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 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) 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. Our implementation of Async Rattus infers these clock annotations and transforms the GHC Core code generated from the Async Rattus code accordingly.

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 $\Box$. 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 $\Box a$ 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 $cl :: \bigcirc a \to \mathit{Clock}$ and $\mathsf{adv} :: \bigcirc a \to a$ that project out these two components. Conversely, we can construct a value of type $\bigcirc a$ by providing these two components using the function $\mathsf{delay} :: \mathit{Clock} \to a \to \bigcirc a$. Using these functions, we can implement a function that takes a delayed integer and increments it:

$$
\begin{aligned}
&\mathit{incr} :: \bigcirc \mathit{Int} \to \bigcirc \mathit{Int} \\
&\mathit{incr}\ x = \mathsf{delay}_{\mathsf{cl}(x)}\ (\mathsf{adv}\ x + 1)
\end{aligned}
$$

This makes explicit the fact that the integers produced by the delayed computations $x :: \bigcirc \mathit{Int}$ and $\mathit{incr}\ x :: \bigcirc \mathit{Int}$ 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.

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 and we have to reign them in to ensure that Async Rattus programs are causal and

$$\frac{\Gamma, \checkmark_{\mathsf{cl}(t)} \vdash t :: A}{\Gamma \vdash \mathsf{delay}_{\mathsf{cl}(t)}\ t :: \bigcirc A} \qquad \frac{\checkmark \notin \Gamma' \quad \text{or} \quad A \text{ stable}}{\Gamma, x :: A, \Gamma' \vdash x :: A} \qquad \frac{\Gamma \vdash t :: \Box A}{\Gamma \vdash \mathsf{unbox}\ t :: A} \qquad \frac{\Gamma^{\Box} \vdash t :: A}{\Gamma \vdash \mathsf{box}\ t :: \Box A}$$

$$\frac{\Gamma \vdash s :: \bigcirc A \quad \Gamma \vdash t :: \bigcirc B \quad \checkmark \notin \Gamma'}{\Gamma, \checkmark_{\mathsf{cl}(s) \sqcup \mathsf{cl}(t)}, \Gamma' \vdash \mathsf{select}\ s\ t :: \mathit{Select}\ A\ B} \qquad \frac{\Gamma \vdash t :: \bigcirc A \quad \checkmark \notin \Gamma'}{\Gamma, \checkmark_{\mathsf{cl}(t)}, \Gamma' \vdash \mathsf{adv}\ t :: A} \qquad \frac{}{\Gamma \vdash \mathsf{never} :: \bigcirc A}$$

where $\cdot^{\Box} = \cdot$ $\qquad (\Gamma, \checkmark_{\theta})^{\Box} = \Gamma^{\Box}$ $\qquad (\Gamma, x :: A)^{\Box} = \begin{cases} \Gamma^{\Box}, x :: A & \text{if } A \text{ stable} \\ \Gamma^{\Box} & \text{otherwise} \end{cases}$

Fig. 1. Select typing rules for Async Rattus.

do not cause space leaks. If delay was simply a function of type $\mathsf{delay} :: \mathit{Clock} \to a \to \bigcirc a$, we could delay arbitrary computations – and the data they depend on – into the future, which will cause space leaks. Figure 1, shows the most important typing rules of Async Rattus.

Async Rattus uses a *Fitch-style* type system [11], which manifests itself by the presence of *tokens* of the form $\checkmark_{\theta}$ (pronounced "tick of clock θ" or just "tick") in typing contexts. We can think of $\checkmark_{\theta}$ as denoting the passage of one time step on the clock θ, i.e. all variables to the left of $\checkmark_{\theta}$ are one time step older w.r.t. the clock θ compared to those to the right of $\checkmark_{\theta}$. The rule for delay introduces a token $\checkmark_{\mathsf{cl}(t)}$ in the typing context Γ. This means that t sees the variables in Γ as one time step older w.r.t. clock $\mathsf{cl}(t)$, thus matching the intuitive semantics of delay which delays evaluation of t by one time step on the clock $\mathsf{cl}(t)$.

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:

```
mapLater :: (a → b) → ○a → ○b
mapLater f x = delay_cl(x) (f (adv x))      -- f is out of scope
```

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.

Also adv cannot be simply a function of type $\bigcirc a \to a$ as this would allow us to simply execute future computations now, which would break causality. Instead, 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 $\checkmark_{\mathsf{cl}(t)}$ 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. those in 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:

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

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

In order to deal with more than one delayed computation, Async Rattus provides the select primitives, which takes two delayed computation $s :: \bigcirc A$ and $t :: \bigcirc B$ as arguments, given a tick on the clock $\mathsf{cl}(s) \sqcup \mathsf{cl}(t)$. It produces a value of type $\mathit{Select}\ A\ B$, which is defined as follows:[1]

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

A clock of the form $\theta \sqcup \theta'$ ticks whenever θ or θ' ticks. That is, $\mathsf{select}\ s\ t$ waits for a tick on either of the two clocks $\mathsf{cl}(s)$ and $\mathsf{cl}(t)$ and depending on whether $\mathsf{cl}(s)$ ticks before, after, or at the same time as $\mathsf{cl}(t)$, it returns *Fst*, *Snd*, or *Both*, respectively. For example, the following function waits for two integers and returns the integer that arrives first:

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

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

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

where $\oplus$ is the (strict) sum type. 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

[1] Async Rattus is a strict language and all type definitions are strict by default.

types in A are guarded by $\Box$. For example $\mathit{Int} \oplus \mathit{Float}$, $\Box(\mathit{Int} \to \mathit{Float})$, and $\Box(\bigcirc \mathit{Int}) \oplus \mathit{Int}$ are stable types, but $\Box \mathit{Int} \to \mathit{Float}$, $\bigcirc \mathit{Int}$, and $\bigcirc(\Box \mathit{Int})$ are not. That is, the $\Box$ 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 $\Box$, we can implement the *map* function for $\bigcirc$:

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

where unbox is simply a function of type $\Box a \to a$.

A value of type $\Box$ can only be constructed using the introduction form box, whose typing rule ensures that boxed values may only refer to variables of a stable type. The notation $\Gamma^{\Box}$ denotes the typing context that is obtained from Γ by removing all variables of non-stable types and all $\checkmark_\theta$ tokens. Thus, for a well-typed term box t, we know that t only accesses variables of 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:

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

That is, a signal of type $\mathit{Sig}\ a$ consists of a current value of type a and a future update to the signal of type $\bigcirc(\mathit{Sig}\ a)$. 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:

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

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

While the definition of the signal type looks superficially the same as in synchronous languages, its semantics is quite different: Updates to a signal do not come at the rate given by the 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:

$$\begin{aligned}&\mathit{const} :: a \to \mathit{Sig}\ a\\&\mathit{const}\ x = x ::: \mathsf{never}\end{aligned}$$

where $\mathsf{never} :: \bigcirc b$ 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:

```
switch :: Sig a → ◯(Sig a) → Sig a
switch (x ::: xs) d = x ::: delay (case select xs d of
                                     Fst  xs' d' → switch xs' d'
                                     Snd  _   d' → d'
                                     Both xs' d' → d')
```

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:

```
step :: a → ◯a → Sig a
step x y = switch (const x) (delay (const (adv y)))
```

2.4 Operational semantics

One of the goals of Async Rattus is to avoid space leaks. To this end, its typing system prevents us from moving arbitrary computations into the future. In addition, also 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. In addition, Async Rattus requires strict data types and any use of lazy data types will produce a warning. The resulting eager evaluation strategy ensures that we do not have to keep intermediate values in memory for longer than one time step.

Following the temporal interpretation of the $\bigcirc$ modality, its introduction form delay_θ does not eagerly evaluate its argument since we may have to wait until input data arrives, namely when the clock θ ticks. For example, in the following function, we cannot evaluate $\mathsf{adv}\ x{+}1$ until the integer value of $x{::}\bigcirc Int$ arrives, which is one time step from now:

```
delayInc :: ◯Int → ◯Int
delayInc x = delay (adv x + 1)
```

However, evaluation is only delayed until the clock $\mathsf{cl}(x)$ ticks, and this delay is reversed by adv. For example, adv (delay $(1 + 1)$) evaluates immediately to 2.

The modal FRP calculi of Krishnaswami [26] and Bahr et al. [3, 4], Bahr and Møgelberg [5] have a similar operational semantics to achieve same memory property that Async Rattus has. However, similarly to Rattus, Async Rattus uses a slightly more eager evaluation strategy for delay: Recall that $\mathsf{delay}_\theta\ t$ 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 $\mathsf{delay}_{\mathsf{cl}(t)}$ (... adv t ...) before evaluating delay. In other words, $\mathsf{delay}_{\mathsf{cl}(t)}$ (... adv t ...) is equivalent to

$$\textbf{let}\ x = t\ \textbf{in}\ \mathsf{delay}_{\mathsf{cl}(x)}\ (\ldots\ \mathsf{adv}\ x\ \ldots)$$

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

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

This generalisation of the operational semantics of delay allows us to lift the restrictions present in the Async RaTT calculus [5] on which Async Rattus is based: Async Rattus allows more than one $\checkmark_\theta$ in the typing context, i.e. delay can be nested; it does not prohibit lambda abstractions in the presence of a $\checkmark_\theta$; and both adv and select can be used with arbitrary terms, not just variables.

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. Async Rattus interacts with its environment via input channels (sources that produce signals) and output channels (sinks that consume signals). The simplest input channel is a timer that ticks at a fixed interval (given in milliseconds):

$$\mathit{timer} :: \mathit{Int} \to \Box(\bigcirc())$$

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

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

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

$$\mathit{getInput} :: \mathit{IO}\ (\Box(\bigcirc a), (a \to \mathit{IO}\ ()))$$

which produces an input channel of type $\Box(\bigcirc a)$ that we can feed from Haskell by using the callback function of type $a \to \mathit{IO}\ ()$. 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:

$$\begin{aligned}&\mathit{consoleInput} :: \mathit{IO}\ (\Box(\bigcirc \mathit{Text}))\\ &\mathit{consoleInput} = \textbf{do}\ (\mathit{inp}, \mathit{cb}) \leftarrow \mathit{getInput}\end{aligned}$$

```
current     :: Sig a → a
future      :: Sig a → ◯(Sig a)
map         :: □(a → b) → Sig a → Sig b
mapD        :: □(a → b) → ◯(Sig a) → ◯(Sig b)
const       :: a → Sig a
scan        :: (Stable b) ⇒ □(b → a → b) → b → Sig a → Sig b
zipWith     :: (Stable a, Stable b) ⇒ □(a → b → c) → Sig a → Sig b → Sig c
interleave  :: □(a → a → a) → ◯(Sig a) → ◯(Sig a) → ◯(Sig a)
switch      :: Sig a → ◯(Sig a) → Sig a
derivative  :: Sig Float → Sig Float
integral    :: Float → Sig Float → Sig Float
```

Fig. 2. Simple FRP library.

```
    let loop = do line ← getLine; cb line; loop
    forkIO loop
    return inp
```

Any time the callback function *cb* returned by *getInput* is called with an argument v, the input channel *inp* will produce a new value v.

For output channels, Async Rattus provides the function

```
setOutput :: Sig a → (a → IO ()) → 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 ⇒ p → (a → IO ()) → IO ()
```

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

For example, we may wish to process an output signal of integers by simply 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:[2]

```
main = do inp ← mkSignal ⟨$⟩ consoleInput
          let outSig :: ◯(Sig Int)
              outSig = mapD (box length) inp
          intOutput outSig
          startEventLoop
```

[2] $\langle\$\rangle$ is the infix notation for the function *fmap* :: *Functor* $f \Rightarrow (a \to b) \to f\ a \to f\ b$

In the last line we call $\mathit{startEventLoop} :: \mathit{IO}\ ()$ 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 $\mathit{filter} :: \Box(a \to \mathit{Bool}) \to \mathit{Sig}\ a \to \mathit{Sig}\ a$. The problem is that a signal of type $\mathit{Sig}\ a$ must produce a value of type a at every tick of its current clock. In order to check the predicate $p :: \Box(a \to \mathit{Bool})$ 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:[3]

$$\begin{aligned}&\mathit{filter}' :: \Box(a \to \mathit{Bool}) \to \mathit{Sig}\ a \to \mathit{Sig}\ (\mathit{Maybe}'\ a)\\&\mathit{filter}'\ p = \mathit{map}\ (\mathsf{box}\ (\lambda x \to \mathbf{if}\ \mathsf{unbox}\ p\ x\ \mathbf{then}\ \mathit{Just}'\ x\\&\qquad\qquad\qquad\qquad\qquad\mathbf{else}\ \mathit{Nothing}'))\end{aligned}$$

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 $\mathit{Sig}\ (\mathit{Maybe}'\ a)$ instead of $\mathit{Sig}\ a$. However, this introduces inefficiencies since programs that work with signals of type $\mathit{Sig}\ (\mathit{Maybe}'\ a)$ have to explicitly check for the $\mathit{Nothing}'$ 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:

$$\begin{aligned}&\mathbf{data}\ F\ a = \mathit{Now}\ a \mid \mathit{Wait}\ (\bigcirc(F\ a))\\&\mathbf{data}\ \mathit{Sig}_F\ a = a :::_F (\bigcirc(F\ (\mathit{Sig}_F\ a)))\end{aligned}$$

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 $\mathit{filter} :: \Box(a \to \mathit{Bool}) \to \mathit{Sig}_F\ a \to F\ (\mathit{Sig}_F\ a)$ as well as corresponding versions of the functions in Figure 2.

Sadly, $\mathit{Sig}_F\ a$ still suffers from the same inefficiency as the $\mathit{Sig}\ (\mathit{Maybe}'\ a)$ 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 $\mathit{Sig}\ (\mathit{Maybe}'\ a)$ into a signal of type $\mathit{Sig}\ a$:

$$\begin{aligned}&\mathit{mkInputSig} :: \mathit{Producer}\ p\ a \Rightarrow p \to \mathit{IO}\ (\Box(\bigcirc(\mathit{Str}\ a)))\\&\mathit{mkInputSig}\ p = \mathbf{do}\ (\mathit{out}, \mathit{cb}) \leftarrow \mathit{getInput}\\&\qquad\qquad\qquad\qquad\mathit{setOutput}\ p\ \mathit{cb}\\&\qquad\qquad\qquad\qquad\mathit{return}\ \mathsf{box}\ (\mathit{mkSig}\ \mathit{out})\end{aligned}$$

[3] Maybe' is a strict variant of the standard *Maybe* type.

$$
\begin{array}{ll}
\mathit{filterMap} & :: \Box(a \to \mathit{Maybe}'\ b) \to \mathit{Sig}\ a \to \mathit{IO}\ (\Box(\bigcirc(\mathit{Sig}\ b))) \\
\mathit{filterMapD} & :: \Box(a \to \mathit{Maybe}'\ b) \to \bigcirc(\mathit{Sig}\ a) \to \mathit{IO}\ (\Box(\bigcirc(\mathit{Sig}\ b))) \\
\mathit{filter} & :: \Box(a \to \mathit{Bool}) \to \mathit{Sig}\ a \to \mathit{IO}\ (\Box(\bigcirc(\mathit{Sig}\ a))) \\
\mathit{filterD} & :: \Box(a \to \mathit{Bool}) \to \bigcirc(\mathit{Sig}\ a) \to \mathit{IO}\ (\Box(\bigcirc(\mathit{Sig}\ a))) \\
\mathit{trigger} & :: (\mathit{Stable}\ a, \mathit{Stable}\ b) \Rightarrow \Box(a \to b \to c) \to \mathit{Sig}\ a \to \mathit{Sig}\ b \to \mathit{IO}\ (\Box(\mathit{Sig}\ c)) \\
\mathit{triggerD} & :: \mathit{Stable}\ b \Rightarrow \Box(a \to b \to c) \to \bigcirc(\mathit{Sig}\ a) \to \mathit{Sig}\ b \to \mathit{IO}\ (\Box(\bigcirc(\mathit{Sig}\ c)))
\end{array}
$$

Fig. 3. Filter functions in Async Rattus.

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

$$
\begin{array}{l}
\mathit{filter} :: \Box(a \to \mathit{Bool}) \to \mathit{Sig}\ a \to \mathit{IO}\ (\Box(\bigcirc(\mathit{Sig}\ a))) \\
\mathit{filter}\ p\ xs = \mathit{mkInputSig}\ (\mathit{filter}'\ p\ xs)
\end{array}
$$

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

$$
\begin{array}{l}
\mathit{setQuit} :: (\mathit{Producer}\ p\ a) \Rightarrow p \to \mathit{IO}\ () \\
\mathit{setQuit}\ \mathit{sig} = \mathit{setOutput}\ \mathit{sig}\ (\lambda\_ \to \mathit{exitSuccess})
\end{array}
$$

which quits execution as soon as it receives the first value from the producer.

Figure 4, shows an interactive console application that uses our simple IO API. The application maintains an integer counter that increments each second (*nats*). 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

```
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.

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. Since the implementation of the $\Box$ modality and the *Stable* type constraint is similar to Rattus [2], we focus our attention on the $\bigcirc$ modality as it requires a significantly different approach due to the presence of clocks.

Syntax. Figure 5 shows the implementation of the syntax of $\bigcirc$. A value 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, we can run the delayed computation f by passing both c and v to f as an argument of type *InputValue*. The introduction and elimination forms delay, adv, and select are simply implemented as $\bot$. 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.

```
data InputValue where                                   -- Not exported
  InputValue :: ChanId → a → InputValue
type Clock = Set ChanId                                 -- Not exported
data ◯a = Delay Clock (InputValue → a)                  -- Constructor not exported
delay :: a → ◯a          adv :: ◯a → a          select :: ◯a → ◯b → Select a b
delay _ = ⊥              adv _ = ⊥              select _ _ = ⊥
```

Fig. 5. Implementation of the syntax of ◯.

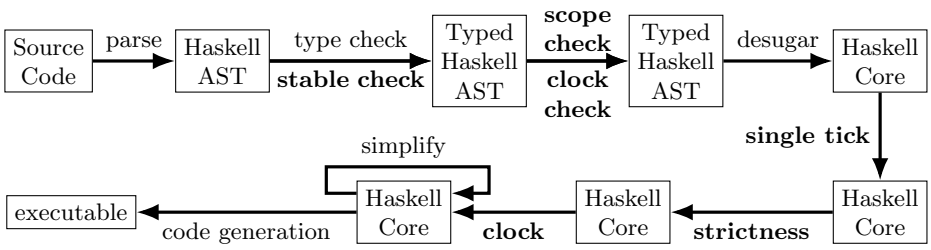

Fig. 6. Simplified pipeline of GHC extended with Async Rattus plugin (in bold).

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 Γ', or because they were dropped when transforming a context Γ to $\Gamma^{\Box}$, e.g. in the rule for box. The clock inference algorithm introduces an existential clock variable θ for each occurrence of delay, which is then instantiated to $\mathsf{cl}(t)$ or $\mathsf{cl}(s) \sqcup \mathsf{cl}(t)$ as soon as it encounters occurrences of adv t or select s t, respectively.

Single tick, strictness & clock transformation. GHC desugars the typed Haskell AST into the Haskell 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 step 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.

Single tick transformation:

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

Strictness transformation:

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

Clock transformation:

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

Fig. 7. Transformation rules.

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 the dummy functions delay, adv, and select from Figure 5 with their actual implementations from 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 former is important for the implementation of select as it needs to check which of the two delayed computations to advance, whereas the latter 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 $\lambda_{\mathsf{Widget}}$ 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 $\lambda_{\mathsf{Widget}}$ provides a select primitive to observe the relative arrival time. However, we are not aware of an implementation of a language based on $\lambda_{\mathsf{Widget}}$.

```
adv' :: ◯a → InputValue → a
adv' (Delay _ f) inp = f inp
select' :: ◯a → ◯b → InputValue → Select a b
select' a@(Delay θ1 f) b@(Delay θ2 g) v@(InputValue ch _)
  = if ch ∈ θ1 then if ch ∈ θ2 then Both (f v) (g v)
    else Fst (f v) b              else Snd a (b v)
```

Fig. 8. Implementation of adv and select.

Async Rattus is based on the Async RaTT calculus of Bahr and Møgelberg [5], which proposes the modal operator ⋵ to model asynchronous signals (we use the simpler notation ◯ in Async Rattus). Like the synchronous calculus of Krishnaswami [26] (on which AdjS [25] is based), but unlike the asynchronous $\lambda_{\mathsf{Widget}}$ calculus of Graulund et al. [16], Async RaTT comes with a proof of operational guarantees: All Async RaTT programs are causal, productive, and don't 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 typecheck 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]. Not only does it allow us to customise the type checker and use program transformations to tweak the operational semantics. We can also 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 or simplifying select so that more than two delayed computation can be easily tracked.

Bibliography

[1] Apfelmus, 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 esterel 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., Inclezan, 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`