DRAFT
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 recently introduced 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 embedded the language in Haskell, we can exploit the existing language and library ecosystem as well as rapidly experiment with new features. We hope that such experimentation with Async Rattus sparks further research in modal FRP and its applications.

1 Introduction

Functional reactive programming (FRP) [7] provides an elegant, high-level programming paradigm for implementing reactive programs. 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. These libraries are 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 depend 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 the programmer with a set of *abstract* types 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 approach to safe FRP has been developed [2, 3, 10, 12, 14, 16, 17] 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. Such a language with a modal type operator $\bigcirc$ has been recently implemented as an embedded language in Haskell called Rattus [1].

In Rattus, signals can be implemented as follows:

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

That is, a signal of type *Sig a* is an element of type a now and a signal of type *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:

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

Computing according to a *global clock* is a reasonable assumption for many contexts such as simulations or games, and indeed much of the application domain of synchronous data flow languages CITE. However, for many applications, e.g. GUIs, the notion of a global clock may not be natural and may lead to inefficiencies.

In this paper, we present Async Rattus an embedded 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 [4] 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 GHC Haskell compiler 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 the FRP program. Our implementation of Async Rattus infers these clock annotations and transforms the GHC Core code generated from the Async Rattus code accordingly.

2 Introduction to Async Rattus

$$\frac{\Gamma, \checkmark_{\mathsf{cl}(t)} \vdash t :: A}{\Gamma \vdash \mathsf{delay}_{\mathsf{cl}(t)}\ t :: \bigcirc A} \qquad \frac{\Gamma' \text{ tick-free} \ \text{ or } \ A \text{ stable}}{\Gamma, x :: A, \Gamma' \vdash x :: A} \qquad \frac{\Gamma \vdash t :: \bigcirc A \qquad \Gamma' \text{ tick-free}}{\Gamma, \checkmark_{\mathsf{cl}(t)}, \Gamma' \vdash \mathsf{adv}\ t :: A}$$

$$\frac{\Gamma \vdash s :: \bigcirc A \qquad \Gamma \vdash t :: \bigcirc B \qquad \Gamma' \text{ tick-free}}{\Gamma, \checkmark_{\mathsf{cl}(s) \sqcup \mathsf{cl}(t)}, \Gamma' \vdash \mathsf{select}\ s\ t :: \mathit{Select}\ A\ B} \qquad \frac{\Gamma^{\Box} \vdash t :: A}{\Gamma \vdash \mathsf{box}\ t :: \Box A} \qquad \frac{\Gamma \vdash t :: \Box A}{\Gamma \vdash \mathsf{unbox}\ t :: A}$$

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

Fig. 1. Select typing rules for Async Rattus.

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

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

This makes explicit the fact that both $x :: \bigcirc \mathit{Int}$ and $\mathit{incr}\ x :: \bigcirc \mathit{Int}$ become available at the same time. We write the first argument $\mathit{cl}\ x$ of delay as a subscript. As we will see shortly, we can think of these clock arguments as 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 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.

The rule for delay is a characteristic example of a Fitch-style typing rule [6]: It introduces the *token* $\checkmark_{\mathsf{cl}(t)}$ (pronounced "tick of clock $\mathsf{cl}(t)$") in the typing context Γ. A typing context consists of type assignments of the form $x :: A$, but it can also contain several occurrences of ticks $\checkmark_\theta$. 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 than those to the right w.r.t. the clock θ. In the above typing rule, the term t does not have access to these "old" variables in Γ. There is, however, an exception: If a variable in the typing context is of a type that is time-independent, we still allow t to access them – even if the variable is one time step old. 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 $\Box a$. We shall return to stable types later when we discuss the $\Box$ type modality in section 2.2.

From the variable introduction rule, we can see that if x is not of a stable type and appears to the left of a $\checkmark_\theta$, then it is no longer in scope. 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:

$$
\begin{array}{l}
\mathit{mapLater} :: (a \to b) \to \bigcirc a \to \bigcirc b \\
\mathit{mapLater}\ f\ x = \mathsf{delay}_{\mathsf{cl}(x)}\ (f\ (\mathsf{adv}\ x)) \quad \text{-- f is out of scope}
\end{array}
$$

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.

Also adv cannot be simply a function of type $\bigcirc a \to a$ as this would allow us to simply execute delayed computations now, effectively looking into the future. 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 above typing rule, are therefore not accessible once we returned to the present.

Let's return to the *add* function from the introduction and see why it does not type check in Async Rattus:

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

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 at runtime the clocks of x and y are the same, e.g. if $y = \mathit{incr}\ x$, but that is not guaranteed.

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 clock $\mathsf{cl}(s) \sqcup \mathsf{cl}(t)$ that ticks whenever either $\mathsf{cl}(s)$ or $\mathsf{cl}(t)$ ticks. It produces a value of type *Select A B*, which is defined as follows:[1]

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

That is, select s t waits for a tick on either of the two clocks $\mathsf{cl}(s)$ and $\mathsf{cl}(t)$ (in other words a tick on the clock $\mathsf{cl}(s) \sqcup \mathsf{cl}(t)$) 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:

$$\begin{array}{l}
\mathit{first} :: \bigcirc \mathit{Int} \to \bigcirc \mathit{Int} \to \bigcirc \mathit{Int} \\
\mathit{first}\ x\ y = \mathsf{delay}_{\mathsf{cl}(x) \sqcup \mathsf{cl}(y)}\ (\textbf{case}\ \mathsf{select}\ x\ y\ \textbf{of}\ \begin{array}[t]{l} \mathit{Fst}\ \ x'\ _ \to x' \\ \mathit{Snd}\ \ _\ y' \to y' \\ \mathit{Both}\ x'\ _ \to x') \end{array}
\end{array}$$

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

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

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 moves 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 $\mathit{Int} \oplus \mathit{Float}$, $\square(\mathit{Int} \to \mathit{Float})$, and $\square(\bigcirc \mathit{Int}) \oplus \mathit{Int}$ are stable types, but $\square \mathit{Int} \to \mathit{Float}$, $\bigcirc \mathit{Int}$, and $\bigcirc(\square \mathit{Int})$ are not. That is, the $\square$ modality can be used to turn any type into a stable type and thus make it possible to move functions into the future safely without risking space leaks. Using $\square$, we can implement the *map* function for $\bigcirc$:

$$\begin{array}{l}
\mathit{mapLater} :: \square(a \to b) \to \bigcirc a \to \bigcirc b \\
\mathit{mapLater}\ f\ x = \mathsf{delay}\ (\mathsf{unbox}\ f\ (\mathsf{adv}\ x))
\end{array}$$

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

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

The typing rule for the introduction of $\Box$ ensures that boxed values may only refer to variables of a stable type. Here, $\Gamma^{\Box}$ 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 $\mathsf{box}\ t$, we know that t only accesses variables of stable type.

2.3 Recursive definitions

Similarly to Rattus and similar synchronous FRP languages (e.g. Krishnaswami [14]), 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 $\mathit{Sig}\ a$ 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:

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

In order to ensure productivity of recursive function definitions, Async Rattus requires that recursive function calls, like $\mathit{map}\ f\ (\mathsf{adv}\ \mathit{xs})$ 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 is similar to synchronous languages, their 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:

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

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

$$\begin{aligned}
&\mathit{switch} :: \mathit{Sig}\ a \to \bigcirc(\mathit{Sig}\ a) \to \mathit{Sig}\ a\\
&\mathit{switch}\ (x ::: \mathit{xs})\ d = x ::: \mathsf{delay}\ (\mathbf{case}\ \mathsf{select}\ \mathit{xs}\ d\ \mathbf{of}\ \begin{array}[t]{l}\mathit{Fst}\ \ \mathit{xs}'\ d' \to \mathit{switch}\ \mathit{xs}'\ d'\\ \mathit{Snd}\ \ _\ \ d' \to d'\\ \mathit{Both}\ \mathit{xs}'\ d' \to d')\end{array}
\end{aligned}$$

A signal $\mathit{switch}\ s\ e$ 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 is first a

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

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 is 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 *theta* 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:

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

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

The modal FRP calculi of Krishnaswami [14] and Bahr et al. [2, 3], Bahr and Møgelberg [4] 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 $\mathsf{delay}_{\mathsf{cl}(t)}\ (\ldots\ \mathsf{adv}\ t\ \ldots)$ before evaluating delay. In other words, $\mathsf{delay}_{\mathsf{cl}(t)}\ (\ldots\ \mathsf{adv}\ t\ \ldots)$ is equivalent to

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

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

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

This adjustment of the operational semantics of delay is important, as it allows us to lift the restrictions present in the Async RaTT calculus [4] on which Async Rattus is based: Unlike Async RaTT, 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.

$$\begin{array}{ll}
\mathit{current} & :: \mathit{Sig}\ a \to a \\
\mathit{future} & :: \mathit{Sig}\ a \to \bigcirc(\mathit{Sig}\ a) \\
\mathit{map} & :: \Box(a \to b) \to \mathit{Sig}\ a \to \mathit{Sig}\ b \\
\mathit{mapD} & :: \Box(a \to b) \to \bigcirc(\mathit{Sig}\ a) \to \bigcirc(\mathit{Sig}\ b) \\
\mathit{const} & :: a \to \mathit{Sig}\ a \\
\mathit{scan} & :: (\mathit{Stable}\ b) \Rightarrow \Box(b \to a \to b) \to b \to \mathit{Sig}\ a \to \mathit{Sig}\ b \\
\mathit{zipWith} & :: (\mathit{Stable}\ a, \mathit{Stable}\ b) \Rightarrow \Box(a \to b \to c) \to \mathit{Sig}\ a \to \mathit{Sig}\ b \to \mathit{Sig}\ c \\
\mathit{interleave} & :: \Box(a \to a \to a) \to \bigcirc(\mathit{Sig}\ a) \to \bigcirc(\mathit{Sig}\ a) \to \bigcirc(\mathit{Sig}\ a) \\
\mathit{switch} & :: \mathit{Sig}\ a \to \bigcirc(\mathit{Sig}\ a) \to \mathit{Sig}\ a \\
\mathit{derivative} & :: \mathit{Sig}\ \mathit{Float} \to \mathit{Sig}\ \mathit{Float} \\
\mathit{integral} & :: \mathit{Float} \to \mathit{Sig}\ \mathit{Float} \to \mathit{Sig}\ \mathit{Float}
\end{array}$$

Fig. 2. Simple FRP library.

3 Reactive Programming in Async Rattus

3.1 A simple FRP application

To support FRP using the *Sig* type, we can implement[2] a library of standard FRP combinators [4] in Async Rattus. We have listed a small subset of the library in Figure 2. This functions can be implemented in Async Rattus, e.g.

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

Async Rattus also features built-in clocks that tick at a fixed interval:

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

The delayed computation produced by *timer n* will tick every n microseconds. Boxed delayed computations, such as the ones produced by *timer*, can be turned into signals:

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

In turn, these built-in timer clocks can be used for implementing the *derivative* and *integral* combinators in Figure 2 (as in Bahr and Møgelberg [4]).

By virtue of being a shallowly embedded language, Async Rattus can seamlessly call Haskell code. Conversely, Haskell code can also interface with Async Rattus code. This is necessary to run Async Rattus code so that it can actually interact with some environment, e.g. a GUI. To this end, the Async Rattus library provides two functions:

[2] See `https://github.com/pa-ba/AsyncRattus/blob/master/src/AsyncRattus/Signal.hs`

$$\begin{array}{l} \mathit{getInput} :: \mathit{IO}\ (\Box(\bigcirc a), (a \to \mathit{IO}\ ())) \\ \mathit{setOutput} :: \mathit{Producer}\ p\ a \Rightarrow p \to (a \to \mathit{IO}\ ()) \to \mathit{IO}\ () \end{array}$$

The function *getInput* provides an input channel of type $\Box(\bigcirc a)$ that we can feed from Haskell by using the callback function of type $a \to \mathit{IO}\ ()$. This function can be used by library authors to provide an Async Rattus interface. For example, we may want to provide an input channel for the console:

$$\begin{array}{l} \mathit{consoleInput} :: \mathit{IO}\ (\Box(\bigcirc \mathit{Text})) \\ \mathit{consoleInput} = \mathbf{do}\ (\mathit{inp}, \mathit{cb}) \leftarrow \mathit{getInput} \\ \qquad\qquad\qquad\qquad \mathbf{let}\ \mathit{loop} = \mathbf{do}\ \mathit{line} \leftarrow \mathit{getLine}; \mathit{cb}\ \mathit{line}; \mathit{loop} \\ \quad \mathit{forkIO}\ \mathit{loop} \\ \quad \mathit{return}\ \mathit{inp} \end{array}$$

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

Conversely, we can use *setOutput* so that Haskell code can process the output produced by an Async Rattus program. Instances of $\mathit{Producer}\ p\ a$ consist of signal types p that produce values of type a. Concretely, this means that such a producer can be turned into as signal of Maybe' values, which is a strict variant of the standard *Maybe* type:

$$\begin{array}{l} \mathbf{class}\ \mathit{Producer}\ p\ a\ \mathbf{where} \\ \quad \mathit{prod} :: p \to \mathit{Sig}\ (\mathit{Maybe}'\ a) \end{array}$$

In particular, we have instances $\mathit{Producer}\ (\bigcirc(\mathit{Sig}\ a))\ a$ and $\mathit{Producer}\ (\mathit{Sig}\ a)\ a$.

For example, we may wish to process an output signal of integers by simply printing each new value to the console:

$$\begin{array}{l} \mathit{intOutput} :: \bigcirc(\mathit{Sig}\ \mathit{Int}) \to \mathit{IO}\ () \\ \mathit{intOutput}\ \mathit{sig} = \mathit{setOutput}\ \mathit{sig}\ \mathit{print} \end{array}$$

Then we can implement a simple console application that waits for the user to input a line into the console prompt and outputs the length of the user input:

$$\begin{array}{l} \mathit{main} = \mathbf{do}\ \mathit{inp} \leftarrow \mathit{consoleInput} \\ \qquad\qquad\ \mathbf{let}\ \mathit{consoleSig} :: \bigcirc(\mathit{Sig}\ \mathit{Text}) \\ \qquad\qquad\qquad \mathit{consoleSig} = \mathit{mkSignal}\ \mathit{inp} \\ \qquad\qquad\qquad \mathit{newSig} :: \bigcirc(\mathit{Sig}\ \mathit{Int}) \\ \qquad\qquad\qquad \mathit{newSig} = \mathit{mapD}\ (\mathsf{box}\ \mathit{length})\ \mathit{intSig} \\ \qquad\qquad\ \mathit{intOutput}\ \mathit{newSig} \\ \qquad\qquad\ \mathit{startEventLoop} \end{array}$$

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.

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

Fig. 3. Filter functions in Async Rattus.

3.2 Filtering functions

As Bahr and Møgelberg [4] have observed in their Async RaTT calculus, the *Sig* type does not support a filter function of type

$$\mathit{filter} :: \Box(a \to \mathit{Bool}) \to \mathit{Sig}\ a \to \mathit{Sig}\ a$$

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. However, once the clock of the input signal ticks, we obtain a value $v :: a$ and if the predicate $p :: \Box(a \to \mathit{Bool})$ is not satisfied for v, the output signal cannot produce a value and we have to wait until the next tick. Instead, we can implement *filter* with the following type:

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

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, we present two alternatives that are preferable to this solution based on $\mathit{Sig}\ (\mathit{Maybe}'\ a)$: The first solution replaces the modal operator $\bigcirc$ with the derived operator F that may take several ticks to produce a result:

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

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.

However, we are going to focus on the second solution, which is more efficient and works with the existing *Sig* type. Recall the *getInput* and *setOutput* functions to get access to external input and to produce external output from

Async Rattus. We can combine the two to produce a new signal of type $Sig\ a$ from a signal of type $Sig\ (Maybe'\ a)$:

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

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

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

Figure 3 shows Async Rattus implementation of further filter functions: *filterMap* essentially composes the *filter* function with a *map* function. *trigger f xs ys* is a signal that produces a new value whenever *xs* produces a new value and when it does the new value it produces is the produced by applying f to the current value of *xs* and *ys*. One can think of *trigger* as a left-biased version of *zipWith*. Finally, we provide a "*D*" version of these functions that work with delayed signals.

3.3 Extended example

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

$$
\begin{aligned}
&setQuit :: (Producer\ p\ a) \Rightarrow p \rightarrow IO\ ()\\
&setQuit\ sig = setOutput\ sig\ (\lambda\_ \rightarrow exitSuccess)
\end{aligned}
$$

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

Figure 4, shows an interactive console application that uses the 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 value 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".

The purpose of this example is to demonstrate 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 *switchS* to dynamically change the behaviour of a signal (see *nats'*).

```
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.

4 Related Work

The use of modal types for FRP has seen much attention in recent years [1–3, 5, 8, 9, 11, 12, 14, 16, 17]. The first implementation of a modal FRP language we are aware of is AdjS [13], which compiles FRP programs into JavaScript. The language is based on the synchronous modal FRP calculus of Krishnaswami [14] and uses linear types to interact with GUI widgets [15]. To address the discrepancy of the synchronous programming model of AdjS and the inherently asynchronous nature of GUIs, the $\lambda_{\mathsf{Widget}}$ calculus of Graulund et al. [8] 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 $\lambda_{\mathsf{Widget}}$.

Async Rattus is based on the Async RaTT calculus of Bahr and Møgelberg [4], which proposes the modal operator $\exists\!\!\!\bigcirc$ to model asynchronous signals (we use the simpler notation $\bigcirc$ in Async Rattus). Like the synchronous calculus of Krishnaswami [14], but unlike the asynchronous calculus $\lambda_{\mathsf{Widget}}$ of Graulund et al. [8], 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 two places: It allows more than one tick to occur in contexts (thus allowing nested occurrences of delay), function definitions occurring in the scope of ticks in the context, and adv and select to be applied to arbitrary terms instead of just variables. The soundness of this is based on the program transformation, introduced by Bahr [1], 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 [1], 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 provided to the programmer by the *getInput* and *setOutput* functions.

Bibliography

[1] 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, `https://doi.org/10.1017/S0956796822000132`, URL `https://www.cambridge.org/core/journals/journal-of-functional-programming/article/modal-frp-for-all-functional-reactive-programming-without-space-leaks-in-haskell/9BE20E8D61E9B74811CF3CF97B5D10C7`, publisher: Cambridge University Press

[2] 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)

[3] Bahr, P., Graulund, C.U., Møgelberg, R.E.: Diamonds are not forever: Liveness in reactive programming with guarded recursion (Jan 2021), pOPL 2021, to appear

[4] Bahr, P., Møgelberg, R.E.: Asynchronous Modal FRP. Proceedings of the ACM on Programming Languages **7**(ICFP), 205:476–205:510 (2023), `https://doi.org/10.1145/3607847`, URL `https://dl.acm.org/doi/10.1145/3607847`

[5] 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, `https://doi.org/10.1145/2535838.2535881`

[6] 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

[7] 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, `https://doi.org/10.1145/258948.258973`, URL `http://doi.acm.org/10.1145/258948.258973`

[8] Graulund, C.U., Szamozvancev, D., Krishnaswami, N.: Adjoint reactive gui programming. In: FoSSaCS, pp. 289–309 (2021)

[9] 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, `https://doi.org/10.1145/2103776.2103783`, URL `https://doi.org/10.1145/2103776.2103783`

[10] 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, https://doi.org/10.1145/2603088.2603106, URL http://doi.acm.org/10.1145/2603088.2603106
[11] 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), https://doi.org/10.1016/j.entcs.2012.08.015
[12] 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, https://doi.org/10.1145/2428116.2428128, URL http://doi.acm.org/10.1145/2428116.2428128
[13] Krishnaswami, N.R.: AdjS compiler (2013), URL https://github.com/neel-krishnaswami/adjs
[14] 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, https://doi.org/10.1145/2500365.2500588
[15] 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, https://doi.org/10.1145/2034773.2034782, URL https://doi.org/10.1145/2034773.2034782
[16] 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, https://doi.org/10.1109/LICS.2011.38
[17] 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, https://doi.org/10.1145/2103656.2103665, URL https://doi.org/10.1145/2103656.2103665