Push-Pull Modal Functional Reactive Programming

Lasse Faurby Klausen, Philip Kristian Møller Flyvholm, and Patrick Bahr

IT University of Copenhagen, Copenhagen, Denmark
lakl@itu.dk, philipflyvholm@gmail.com, paba@itu.dk

Abstract. Functional reactive programming (FRP) offers an expressive programming paradigm for building reactive systems in a functional style. In recent years, several FRP languages have introduced modal types to ensure that programs are free from space leaks despite their high level of abstraction. So far these modal FRP languages only offer a data-driven (or *push*) execution model, where computation is driven by discrete events triggered by the environment. However, many applications benefit from a demand-driven (or *pull*) execution model, where computation is initiated by the consumer of data.
In this paper, we use Elliott's push-pull evaluation model to build an FRP library in a programming language with modal types. Additionally, we propose and implement several refinements to Elliott's push-pull model to extend its expressiveness. We evaluate our push-pull approach by implementing a GUI framework and a case study of GUI applications.

Keywords: Functional reactive programming · Modal types · Space leaks

1 Introduction

Functional reactive programming (FRP) is based on a simple but powerful principle: It models dynamic behaviour using a first-class type of signals, which we can manipulate using the expressive power of functional programming. However, finding an *efficient* implementation of such a first-class signal type is far from simple. An attractive approach that has seen considerable development in recent years is the use of modal types to capture the temporal aspects of FRP [12,13,15,3,4,5]. It uses the later type modality $\bigcirc$ to express that a value of type $\bigcirc A$ is the promise of a value of type A in the next time step. The type systems of such modal FRP languages are able to keep track of *when* data is available to ensure that all programs are operationally well-behaved and can thus be implemented efficiently without introducing space leaks [15,3].

Modal FRP languages typically define signals as streams of data where each discrete step of a stream is separated by one time step using the $\bigcirc$ modality:

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

Here, as in the rest of the paper, we use Haskell syntax, and we use ::: as an infix constructor. That is, a signal of type $Sig\ a$ consists of a value of type a now and the promise of a new signal in the next time step.

By its very nature, this model of signals is event-driven: A signal is updated whenever a suitable event happens, which triggers the 'next time step' indicated by the $\bigcirc$ modality. This event-driven (or *push*) model is suitable for many types of systems. For example, most aspects of GUIs are indeed event-driven, where discrete user interactions – such as button presses or keyboard inputs – cause updates in the GUI. However, in other application domains, we would prefer a demand-driven (or *pull*) model, where the consumer samples a signal. For example, we need a demand-driven model for smooth animations where a signal must be sampled according to the refresh rate of the display.

In this paper, we show that modal FRP languages can indeed support both models. To this end, we implement a hybrid FRP library based on Elliott's *push-pull* FRP approach [9] in the modal FRP language Async Rattus [11,5,7]. By implementing push-pull FRP in Async Rattus, we can make use of the fact that it is implemented as an embedded language in Haskell and thus has access to Haskell's library ecosystem. This allows us to test push-pull modal FRP by using it to implement a small GUI library along with a small case study in the form of several minimal GUI applications. The push-pull FRP library, the GUI library built on top of it, and the case study is available in an online repository[1].

The remainder of this paper is structured as follows: We give an introduction to modal FRP in Async Rattus in Sect. 2, review Elliott's Haskell-based push-pull approach in Sect. 3.1, and present a simple Async Rattus implementation of this push-pull approach in Sect. 3.2. We then revise this simple implementation of modal push-pull FRP in Sect. 4 in order to extend its expressiveness. This revised push-pull FRP library is then used in Sect. 5 to implement a simple GUI library along with a simple example GUI. We conclude with an overview of related work and a final discussion in Sects. 6 and 7, respectively.

2 Modal FRP

This section gives a brief introduction to the modal FRP language Async Rattus [11]. Readers familiar with the language can skip this section.

There are two major differences between standard Haskell and Async Rattus. Firstly, Async Rattus is eagerly evaluated, in contrast to the lazy evaluation semantics of Haskell. Eager evaluation is a central component in the prevention of space leaks. Secondly, Async Rattus features a non-standard type system with two type modalities, $\bigcirc$ and $\Box$, also called the 'later' and 'box' modalities, respectively. The later modality expresses the passage of time at the type level. This makes it possible to differentiate between the type A, which classifies values that are available now, and the type $\bigcirc A$, which classifies values of type A that are available in the future. The box modality ensures that values can be safely and efficiently moved across time. A value of type $\Box A$ is a time-independent computation that produces a value of type A, i.e., this computation can be moved arbitrarily far into the future without causing space leaks. The most

[1] Available from `https://github.com/pa-ba/AsyncRattus/tree/push-pull`

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

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

$$\frac{}{\Gamma \vdash \mathsf{chan} :: C\ (\mathit{Chan}\ A)} \qquad \frac{\Gamma \vdash t :: \mathit{Chan}\ A}{\Gamma \vdash \mathsf{wait}\ t :: \bigcirc A} \qquad \frac{\Gamma \vdash t :: \bigcirc (C\ A)}{\Gamma \vdash \mathsf{run}\ t :: \bigcirc A} \qquad \frac{}{\Gamma \vdash \mathsf{time} :: C\ \mathit{Time}}$$

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

Fig. 1. Select typing rules for Async Rattus.

important typing rules for the Async Rattus language are shown in Fig. 1. We describe the type modalities and the typing rules that govern them in more detail below.

While Async Rattus is implemented as an embedded language in Haskell, this embedding has to account for the fundamental differences in semantics and type system. To this end, Async Rattus is implemented by a combination of a Haskell library, which implements the basic primitives and types of the language, and a compiler plugin. The compiler plugin transforms the code so that it matches the eager evaluation semantics of Async Rattus, and it performs an additional type-checking pass to enforce the stricter typing rules of the language. Async Rattus also provides strict variants of Haskell's standard data types such as list ($\mathit{List}\ a$), product ($a \times b$), and sum types ($a \oplus b$).

A value of type $\bigcirc A$ represents a delayed computation that will produce a value of type A in the future. Conceptually, a value of type $\bigcirc A$ is a pair (θ, f) consisting of a so-called *clock* θ that tells us when the value of type A is available and a delayed computation f that will produce the value of type A at the time promised by θ. An Async Rattus program may receive data from several input channels such as the keyboard or a button in a GUI. A clock θ is a set of such input channels, e.g., $\theta = \{\kappa_{keyboard}, \kappa_{ok\_button}\}$, and a *tick* on θ means that data has been received on some input channel $\kappa \in \theta$. This mechanism ensures that delayed computations respect temporal causality, i.e., a delayed computation is performed only when appropriate according to the ticking of its associated clock, which in turn indicates that new data relevant to the delayed computation has been received.

The two components θ and f of a delayed computation of type $\bigcirc A$ are accessible via two functions $\mathsf{cl} :: \bigcirc a \to \mathit{Clock}$ and $\mathsf{adv} :: \bigcirc a \to a$, respectively. However, cl is not directly accessible to the programmer, and adv is subject to the typing rule in Fig. 1, which we turn to shortly. Conversely, to construct a delayed computation, we can use delay, which we can think of as having type $\mathsf{delay} :: \mathit{Clock} \to a \to \bigcirc a$ for now. That is, it takes a clock θ and a computation

f producing a and returns a delayed computation (θ, f) that will yield the value of type a once θ ticks. While Async Rattus is eagerly evaluated by default, delay does not evaluate its argument of type a, because it represents a delayed computation that may only be performed once the associated clock θ ticks.

Using these functions that interact with the later modality, we can implement a function that takes a delayed integer and increments it:

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

The function takes $x :: \bigcirc \mathit{Int}$ as an argument and then calls delay using the clock of x to produce a delayed computation. According to the typing rule for delay, this changes the typing context from $x :: \bigcirc \mathit{Int}$ to $x :: \bigcirc \mathit{Int}, \checkmark_{\mathsf{cl}(x)}$. That is, the typing context contains the token $\checkmark_{\mathsf{cl}(x)}$, which indicates that time has passed on the clock $\mathsf{cl}(x)$ and that x is now one time step older. The presence of this token $\checkmark_{\mathsf{cl}(x)}$ in the typing context allows us to use adv, whose typing rule states that it can only be used on an argument t if the typing context contains a token $\checkmark_{\mathsf{cl}(t)}$. Moreover, t itself may only use variables that occur to the left of that $\checkmark_{\mathsf{cl}(t)}$.

This interaction between delay and adv demonstrates that the former moves ahead in time, indicated by the ✓ token, whereas the latter moves back in time, indicated by the removal of the ✓ token. In addition, when we use adv, all variables that were available in the future – i.e., to the right of ✓ – are no longer available once we move back in time again. This typing discipline is important to avoid FRP programs that are non-causal such as the following function, which makes a future value already available now:

$$\begin{aligned} &\mathit{now} :: \bigcirc \mathit{Int} \to \mathit{Int} \qquad \text{-- Does not type-check since there is} \\ &\mathit{now}\ x = \mathsf{adv}\ x \qquad\quad \text{-- no token } \checkmark_{\mathsf{cl}(x)} \text{ in the context.} \end{aligned}$$

In the definition of *incr* we have included the clock annotation $\mathsf{cl}(x)$ on delay only for the purpose of explaining the typing rule. The Async Rattus type checker will infer the correct clock annotation if there exists one, so that the definition of *incr* is in fact written as follows:

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

In addition to adv, Async Rattus also features select, which tries to advance *two* delayed computations. As two delayed computations may have different clocks, their delayed values may not be available at the same time. To account for that, select returns a value of type *Select* that covers the three possible cases of when two delayed values will arrive. Either the first one arrives first, the second one arrives first, or both arrive at the same time:

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

Signals are represented by the recursive type we have seen in the introduction:

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

This type allows us to implement common combinators on signals such as *map* to apply a function to a signal and *switch* to dynamically switch between two signals. However, the naive implementation of *map* does not work:

```
mapLeaky :: (a → b) → Sig a → Sig b      -- f is no longer in scope below
mapLeaky f (x ::: xs) = f x ::: delay (mapLeaky f (adv xs))
```

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. Therefore, function types like $a \to b$ are not considered *stable* types. Only variables that have a stable type can be moved into the future. This can be seen in the typing rule for variables. If a variable $x :: A$ occurs to the left of a $\checkmark$ token in the typing context, we can only use x if A is a stable type. All base types such as *Int* and *Bool* are stable, as are product, sum, and recursive types that combine stable types. By contrast, types of the form $\bigcirc a$ and $a \to b$ are not stable, as moving values of this type into the future may cause space leaks. But we can turn any type into a stable type using the $\Box$ modality. In particular, $\Box(a \to b)$ is stable.

```
map :: □(a → b) → Sig a → Sig b
map f (x ::: xs) = unbox f x ::: delay (map f (adv xs))
```

In this revised implementation, f is still in scope under the delay since it is of a stable type. But we need to use unbox to turn $f :: \Box(a \to b)$ into a function of type $a \to b$ before applying it to $x :: a$. The corresponding introduction form box for $\Box$ makes sure that its argument t only references variables of stable type by requiring t to be typed in the modified context $\Gamma^{\Box}$. This context $\Gamma^{\Box}$ is obtained from Γ by removing all variables $x :: A$ where A is not stable.

In addition to the $\Box$ modality to construct stable types, Async Rattus also has the *Stable* type constraint that allows us to restrict type variables to stable types. For example, we can implement a *buffer* combinator that takes a signal and moves it one time step into the future by keeping the current signal value one time step longer:

```
buffer :: Stable a ⇒ Sig a → ○(Sig a)
buffer (x ::: xs) = delay (x ::: buffer (adv xs))
```

The *buffer* function type-checks since type a is stable, and therefore $x :: a$ is still in scope under the delay.

Finally, Async Rattus features the monad C to accommodate limited side effects, which is necessary due to Haskell's purity. We can observe the current time with time, and we can allocate channels with chan. In turn, a channel of type $c :: \mathit{Chan}\ A$ can be used to receive data of type A. To wait for the arrival of such data we write wait c, which gives us a delayed computation of type $\bigcirc A$, which will tick as soon as data is received on channel c. All channel types $\mathit{Chan}\ A$ are stable, and thus channels can be freely moved into the future. For example, any channel gives rise to a delayed signal of the same type:

$chanSig :: Chan\ a \to \bigcirc(Sig\ a)$
$chanSig\ c = \mathsf{delay}\ (\mathsf{adv}\ (\mathsf{wait}\ c) ::: chanSig\ c)$

Due to the limited nature of the side effect encapsulated by C, Async Rattus allows such computations to be run in the presence of a tick via the primitive $\mathsf{run} :: \bigcirc(C\ a) \to \bigcirc a$. For example, we can implement a combinator that allows us to look up the time in any delayed computation:

$withTime :: \bigcirc(Time \to a) \to \bigcirc a$
$withTime\ df = \mathsf{run}\ (\mathsf{delay}\ (\mathsf{adv}\ df\ \langle\$\rangle\ \mathsf{time}))$

In the above definition, we use the fact that any monad m is also a functor with an application operator $\langle\$\rangle :: (a \to b) \to m\ a \to m\ b$.

3 Simple Push-Pull FRP

Elliott's push-pull FRP [9] combines discrete, event-driven FRP as exemplified by the *Sig* type found in modal FRP systems with a continuous, demand-driven version of FRP as exemplified by the *Behavior* type in the original work on FRP [8]. In this hybrid approach, *events* represent discrete values and use push-based evaluation, while *behaviours* represent continuous time-varying values using both push- and pull-based evaluation. We first review the essence of Elliott's approach in its original implementation language, namely plain Haskell, and then show how it can be implemented in Async Rattus.

3.1 Push-Pull FRP in Haskell

Elliott's push-pull approach features a type *Reactive* that corresponds to the *Sig* type found in modal FRP but without the modal typing discipline. This type forms the basis of the 'push' half of 'push-pull':

data $Reactive\ a \quad = a\ `Stepper`\ Future\ (Reactive\ a)$
newtype $Future\ a = Fut\ (FTime, a)$

In this definition, *Future* plays the role that $\bigcirc$ plays in Async Rattus. It is a pair (t, v) consisting of a (lazy) value v and a time stamp t indicating when the value v is available.

Because *Reactive* is event-driven, it can be used to model discrete events:

type $Event\ a = Future\ (Reactive\ a)$

To model demand-driven behaviours, *Reactive* values are combined with function types:

type $Behaviour\ a = Reactive\ (Time \to a)$

That is, similarly to classic FRP [8], a behaviour is a function from time to values. But unlike classic FRP, these function can be updated by discrete events that

produce new functions from time to values. That is, *Behaviour* a is isomorphic to $(\mathit{Time} \to a) \times \mathit{Event}\ (\mathit{Time} \to a)$.

Finally, this type of behaviours is refined to allow limited forms of symbolic representations of time functions that enable optimizations:

$$\textbf{type}\ \mathit{Behaviour}\ a = \mathit{Reactive}\ (\mathit{Fun}\ \mathit{Time}\ a)$$
$$\textbf{data}\ \mathit{Fun}\ t\ a = K\ a \mid \mathit{Fun}\ (t \to a)$$

Using these simple building blocks, Elliott is able to implement a rich and expressive FRP library. However, Haskell lacks the type system to ensure important operational properties such as causality and absence of space leaks. For example, we can implement the counterexamples *now* and *mapLeaky* from Sect. 2, which Async Rattus rules out:

$$\mathit{now} :: \mathit{Future}\ a \to a$$
$$\mathit{now}\ (t, x) = x$$
$$\mathit{mapLeaky} :: (a \to b) \to \mathit{Reactive}\ a \to \mathit{Reactive}\ b$$
$$\mathit{mapLeaky}\ f\ (x\ \text{`}\mathit{Stepper}\text{`}\ xs) = f\ x\ \text{`}\mathit{Stepper}\text{`}\ (\mathit{mapLeaky}\ f\ (\mathit{now}\ xs))$$

3.2 Push-Pull FRP in Async Rattus

The type definitions of Elliott's push-pull FRP can be translated directly into Async Rattus by simply replacing *Future* with $\bigcirc$. Moreover, in anticipation of the refinements we introduce in Sect. 4, we rename the *Fun* type to *Pull*:

$$\textbf{type}\ \mathit{Ev}\ a\ \ = \bigcirc(\mathit{Sig}\ a)$$
$$\textbf{type}\ \mathit{Beh}\ a = \mathit{Sig}\ (\mathit{Pull}\ a)$$
$$\textbf{data}\ \mathit{Pull}\ a = K\ a \mid \mathit{Fun}\ (\Box(\mathit{Time} \to a))$$

In addition, we also revise the definition of *Fun* so that it uses a boxed function type. We shall see an example of why this is necessary shortly.

Using this simple definition for behaviours and events, we can implement a basic FRP library (cf. Fig. 2) that provides many of the signal combinators found in modal FRP languages [11,2,3,15]. However, compared to standard modal FRP libraries, this library has a clear distinction between discrete and continuous signals, which are now represented by *Ev* and *Beh*, respectively. Moreover, the library can express arbitrary continuous behaviour using *cont*.

Events. Since events are just delayed signals, the three combinators that only manipulate events – *mapE*, *interleave*, *scan*, and *chanEv* – can be implemented in the same way as corresponding combinators on *Sig*. For example:

$$\mathit{mapE} :: \Box(a \to b) \to \mathit{Ev}\ a \to \mathit{Ev}\ b$$
$$\mathit{mapE}\ f\ xs = \mathsf{delay}\ (\textbf{let}\ (x :::\ xs') = \mathsf{adv}\ xs\ \textbf{in}\ \mathsf{unbox}\ f\ x :::\ \mathit{mapE}\ f\ xs')$$

$$
\begin{array}{ll}
\mathit{mapE} & :: \Box(a \to b) \to \mathit{Ev}\ a \to \mathit{Ev}\ b \\
\mathit{interleave} & :: \Box(a \to a \to a) \to \mathit{Ev}\ a \to \mathit{Ev}\ a \to \mathit{Ev}\ a \\
\mathit{scan} & :: \mathit{Stable}\ b \Rightarrow \Box(b \to a \to b) \to b \to \mathit{Ev}\ a \to \mathit{Ev}\ b \\
\mathit{chanEv} & :: \mathit{Chan}\ a \to \mathit{Ev}\ a \\
\mathit{cont} & :: \Box(\mathit{Time} \to a) \to \mathit{Beh}\ a \\
\mathit{const} & :: a \to \mathit{Beh}\ a \\
\mathit{discr} & :: a \to \mathit{Ev}\ a \to \mathit{Beh}\ a \\
\mathit{mapB} & :: \Box(a \to b) \to \mathit{Beh}\ a \to \mathit{Beh}\ b \\
\mathit{zipWith} & :: (\mathit{Stable}\ a, \mathit{Stable}\ b) \Rightarrow \Box(a \to b \to c) \to \mathit{Beh}\ a \to \mathit{Beh}\ b \to \mathit{Beh}\ c \\
\mathit{switch} & :: \mathit{Beh}\ a \to \bigcirc(\mathit{Beh}\ a) \to \mathit{Beh}\ a \\
\mathit{switchS} & :: \mathit{Stable}\ a \Rightarrow \mathit{Beh}\ a \to \bigcirc(a \to \mathit{Beh}\ a) \to \mathit{Beh}\ a \\
\mathit{switchR} & :: \mathit{Stable}\ a \Rightarrow \mathit{Beh}\ a \to \mathit{Ev}\ (a \to \mathit{Beh}\ a) \to \mathit{Beh}\ a
\end{array}
$$

Fig. 2. Simple Async Rattus push-pull FRP library.

The *interleave* function interleaves the occurrences of two events using a tie-breaker function for the case that both events occur simultaneously:

$$
\begin{array}{l}
\mathit{interleave} :: \Box(a \to a \to a) \to \mathit{Ev}\ a \to \mathit{Ev}\ a \to \mathit{Ev}\ a \\
\mathit{interleave}\ f\ \mathit{xs}\ \mathit{ys} = \mathsf{delay}\ (\mathbf{case}\ \mathsf{select}\ \mathit{xs}\ \mathit{ys}\ \mathbf{of} \\
\quad \mathit{Fst}\ \ (x ::: \mathit{xs}')\ \mathit{ys}' \qquad \to x \qquad\qquad ::: \mathit{interleave}\ f\ \mathit{xs}'\ \mathit{ys}' \\
\quad \mathit{Snd}\ \ \mathit{xs}' \qquad (y ::: \mathit{ys}') \to y \qquad\qquad ::: \mathit{interleave}\ f\ \mathit{xs}'\ \mathit{ys}' \\
\quad \mathit{Both}\ (x ::: \mathit{xs}')\ (y ::: \mathit{ys}') \to \mathsf{unbox}\ f\ x\ y ::: \mathit{interleave}\ f\ \mathit{xs}'\ \mathit{ys}')
\end{array}
$$

The select primitive of Async Rattus takes two delayed computations, *xs* and *ys*, and advances whichever of the two arrives first or both if they arrive at the same time. In the latter case, we use the tie-breaker function *f* and apply it to the values of the two values that we observe. For example, if we call $\mathit{interleave}\ (\mathsf{box}\ (\lambda x\ y \to x))\ \mathit{ev}_1\ \mathit{ev}_2$, we bias the interleaving to always pick the event occurrences from ev_1 whenever both ev_1 and ev_2 fire.

The *scan* function is similar to Haskell's *scanl* function on lists. For example, we can us it to implement a function that sums up the integers received from an event:

$$
\begin{array}{l}
\mathit{sum} :: \mathit{Ev}\ \mathit{Int} \to \mathit{Ev}\ \mathit{Int} \\
\mathit{sum} = \mathit{scan}\ (\mathsf{box}\ (+))\ 0
\end{array}
$$

Behaviours. The simplest behaviours are produced by *cont* and *const*, which use $\mathsf{never} :: \bigcirc a$ to construct behaviours that never receive push updates:

$$
\begin{array}{ll}
\mathit{cont} :: \Box(\mathit{Time} \to a) \to \mathit{Beh}\ a & \mathit{const} :: a \to \mathit{Beh}\ a \\
\mathit{cont}\ f = \mathit{Fun}\ f ::: \mathsf{never} & \mathit{const}\ x = K\ x ::: \mathsf{never}
\end{array}
$$

The *discr* function turns any event *ev* into a behaviour that changes its value every time the event *ev* fires and remains constant otherwise:

$$
\begin{aligned}
&\mathit{discr} :: a \to \mathit{Ev}\ a \to \mathit{Beh}\ a \\
&\mathit{discr}\ \mathit{initial}\ \mathit{ev} = (K\ \mathit{initial} ::: \mathit{mapE}\ (\mathsf{box}\ K)\ \mathit{ev})
\end{aligned}
$$

Similar to *mapE* on events, we can pointwise apply boxed functions to behaviours via a corresponding *mapB* combinator, which in turn is implemented by a corresponding *mapP* combinator on *Pull*:

$$
\begin{aligned}
&\mathit{mapB} :: \Box(a \to b) \to \mathit{Beh}\ a \to \mathit{Beh}\ b \\
&\mathit{mapB}\ f\ (x ::: xs) = \mathit{mapP}\ f\ x ::: \mathsf{delay}\ (\mathit{mapB}\ f\ (\mathsf{adv}\ xs)) \\
&\mathit{mapP} :: \Box(a \to b) \to \mathit{Pull}\ a \to \mathit{Pull}\ b \\
&\mathit{mapP}\ f\ (K\ a)\ \ = K\ (\mathsf{unbox}\ f\ a) \\
&\mathit{mapP}\ f\ (\mathit{Fun}\ t) = \mathit{Fun}\ (\mathsf{box}\ (\mathsf{unbox}\ f \circ \mathsf{unbox}\ t))
\end{aligned}
$$

Since behaviours are continuous and thus have a value at any point in time, we can pointwise combine two behaviours similarly to the familiar list functions *zip* and *zipWith*. That is, given two behaviours, $as :: \mathit{Beh}\ a$ and $bs :: \mathit{Beh}\ b$ along with a function $f : \Box(a \to b \to c)$, *zipWith* produces a new behaviour of type $\mathit{Beh}\ c$ that at each time t has the value $\mathsf{unbox}\ f\ a\ b$, where a and b are the values of as and bs at time t. To implement this combinator, we first, generalize *mapP* to two *Pull* arguments:

$$
\begin{aligned}
&\mathit{mapP2} :: (\mathit{Stable}\ a, \mathit{Stable}\ b) \Rightarrow \Box(a \to b \to c) \to \mathit{Pull}\ a \to \mathit{Pull}\ b \to \mathit{Pull}\ c \\
&\mathit{mapP2}\ f\ (K\ x)\ \ (K\ y)\ \ = K\ \ (\mathsf{unbox}\ f\ x\ y) \\
&\mathit{mapP2}\ f\ (\mathit{Fun}\ x)\ (\mathit{Fun}\ y) = \mathit{Fun}\ (\mathsf{box}\ (\lambda t \to \mathsf{unbox}\ f\ (\mathsf{unbox}\ x\ t)\ (\mathsf{unbox}\ y\ t))) \\
&\mathit{mapP2}\ f\ (\mathit{Fun}\ x)\ (K\ y)\ \ = \mathit{Fun}\ (\mathsf{box}\ (\lambda t \to \mathsf{unbox}\ f\ (\mathsf{unbox}\ x\ t)\ y)) \\
&\mathit{mapP2}\ f\ (K\ x)\ \ (\mathit{Fun}\ y) = \mathit{Fun}\ (\mathsf{box}\ (\mathsf{unbox}\ f\ x \circ \mathsf{unbox}\ y))
\end{aligned}
$$

The last two equations of the definition above are the reason for requiring the two types a and b to be stable: In the third equation, $y :: b$ is moved into a box, and in the fourth equation $x :: a$ is moved into a box.

We can use *mapP2* to implement our *zipWith* function as follows:

$$
\begin{aligned}
&\mathit{zipWith} :: (\mathit{Stable}\ a, \mathit{Stable}\ b) \Rightarrow \Box(a \to b \to c) \to \mathit{Beh}\ a \to \mathit{Beh}\ b \to \mathit{Beh}\ c \\
&\mathit{zipWith}\ f\ (x ::: xs)\ (y ::: ys) = \\
&\quad \mathit{mapP2}\ f\ x\ y ::: \mathsf{delay}\ (\mathbf{case}\ \mathsf{select}\ xs\ ys\ \mathbf{of} \\
&\qquad\qquad\qquad\qquad \mathit{Fst}\ \ xs'\ lys \to \mathit{zipWith}\ f\ xs' \qquad\ (y ::: lys) \\
&\qquad\qquad\qquad\qquad \mathit{Snd}\ \ lxs\ ys' \to \mathit{zipWith}\ f\ (x ::: lxs)\ ys' \\
&\qquad\qquad\qquad\qquad \mathit{Both}\ xs'\ ys' \to \mathit{zipWith}\ f\ xs' \qquad\ ys')
\end{aligned}
$$

In addition to requiring the *Stable* constraints on a and b for the use of *mapP2*, we also need them in order to move $x :: \mathit{Pull}\ a$ and $y :: \mathit{Pull}\ b$ into the delay so that they can be used in the recursive call to *zipWith*. Here we make use of the fact that *Stable* a implies *Stable* (*Pull* a). However, this is only true because we used the boxed function type $\Box(\mathit{Time} \to a)$ in the definition of *Fun* rather than just the plain function type $\mathit{Time} \to a$, which is not stable.

Finally, behaviours can be switched dynamically using the *switch* combinator:

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

This implementation, is in fact the same as the *switch* combinator found in asynchronous modal FRP libraries [5,11,7] since the switching itself is essentially event-driven. The *switch* combinator also has two variants: *switchS* allows the new behaviour to depend on the last value of the original behaviour, and *switchR* generalizes this further by allowing the second argument to produce new behaviours several times instead of only once.

To implement *switchS*, we need two additional ingredients. First, we implement a helper function that allows us to sample a *Pull* value at a given time:

```
at :: Pull a → Time → a
at (K x) _   = x
at (Fun f) t = unbox f t
```

Second, we need to obtain the current time so that we can sample the current value of a behaviour. To this end, we use $withTime :: \bigcirc(Time \to a) \to \bigcirc a$ from the end of Sect. 2, which allows us to read the time in the next time step:

```
switchS :: Stable a ⇒ Beh a → ◯(a → Beh a) → Beh a
switchS (x ::: xs) d = x ::: withTime (delay (λt →
  case select xs d of Fst  xs' d' → switchS xs' d'
                      Snd  _   f  → f (x `at` t)
                      Both _   f  → f (x `at` t)))
```

As soon as the delayed computation ticks and gives us a function $f :: a \to Beh\ a$, we apply f to the current value of the old behaviour $(x ::: xs)$. We obtain that value by sampling $x :: Pull\ a$ at the current time. Unlike *switch*, the *switchS* combinator requires a to be stable since we have to move x into the future when f has arrived.

The generalization to *switchR* works in a similar manner. But instead of just switching to the new behaviour for good, it switches anew whenever a new function $f :: a \to Beh\ a$ is produced by the event argument:

```
switchR :: Stable a ⇒ Beh a → Ev (a → Beh a) → Beh a
switchR (x ::: xs) ev = x ::: withTime (delay (λt →
  case select xs ev of
    Fst  xs' ev'        → switchR xs' ev'
    Snd  _   (f ::: ev') → switchR (f (x `at` t)) ev'
    Both _   (f ::: ev') → switchR (f (x `at` t)) ev'))
```

$$
\begin{array}{ll}
\mathit{filter} & :: \Box(a \to \mathit{Bool}) \to \mathit{Ev}\ a \to \mathit{Ev}\ a \\
\mathit{filterMap} & :: \Box(a \to \mathit{Maybe}'\ b) \to \mathit{Ev}\ a \to \mathit{Ev}\ b \\
\mathit{sample} & :: \mathit{Stable}\ a \Rightarrow \Box(a \to b \to c) \to \mathit{Ev}\ a \to \mathit{Beh}\ b \to \mathit{Ev}\ b \\
\mathit{stop} & :: \Box(a \to \mathit{Bool}) \to \mathit{Beh}\ a \to \mathit{Beh}\ a \\
\mathit{stopWith} & :: \Box(a \to \mathit{Maybe}'\ a) \to \mathit{Beh}\ a \to \mathit{Beh}\ a \\
\mathit{derivative} & :: \mathit{Beh}\ \mathit{Float} \to C\ (\mathit{Beh}\ \mathit{Float}) \\
\mathit{integral} & :: \mathit{Float} \to \mathit{Beh}\ \mathit{Float} \to C\ (\mathit{Beh}\ \mathit{Float})
\end{array}
$$

Fig. 3. Extended FRP library supported by the refined definition of *Beh* and *Ev*.

4 Extended Push-Pull FRP

The simple push-pull modal FRP library from Sect. 3.2 (summarized in Fig. 2) provides the basic combinators we expect to find. In order to support more advanced combinators, we have to refine the definition of behaviours and events. As a result, we can extend the FRP library from Fig. 2 with the combinators listed in Fig. 3.

4.1 Events

We first consider the *filter* combinator: An event *filter* p e only contains those event occurrences of e that satisfy the predicate p. It is not possible to implement this combinator with the simple definition of events. The issue is that we can only consume a delayed computation – and thus check whether the value it produces satisfies a predicate – if we also promise to construct a new delayed computation. In other words, we cannot skip a $\bigcirc$ modality from an input. In particular, there is no general operation of type $\bigcirc(\bigcirc a) \to \bigcirc a$. As a consequence, the *filter* function must have a return type reflecting that some event occurrences may be skipped:

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

This definition uses a strict variant of the standard *Maybe* type with constructors $\mathit{Nothing}' :: \mathit{Maybe}'\ a$ and $\mathit{Just}' :: a \to \mathit{Maybe}'\ a$.

An issue similar to the one for *filter* above also occurs when we try to implement a *sample* combinator that samples a behaviour with an event. The best we can do is give *sample* a return type $\mathit{Ev}\ (\mathit{Maybe}'\ b)$ instead of $\mathit{Ev}\ b$:

$$
\begin{array}{l}
\mathit{sample} :: \mathit{Stable}\ b \Rightarrow \Box(a \to b \to c) \to \mathit{Ev}\ a \to \mathit{Beh}\ b \to \mathit{Ev}\ (\mathit{Maybe}'\ c) \\
\mathit{sample}\ f\ \mathit{ev}\ (x ::: \mathit{xs}) = \mathsf{run}\ x\ \mathit{ev}\ \mathit{xs}\ \mathbf{where} \\
\quad \mathsf{run}\ x\ \mathit{ev}\ \mathit{xs} = \mathit{withTime}\ (\mathsf{delay}\ (\lambda t \to \mathbf{case}\ \mathsf{select}\ \mathit{ev}\ \mathit{xs}\ \mathbf{of} \\
\qquad \mathit{Snd}\ \ \mathit{ev}' \qquad\ (x' ::: \mathit{xs}') \to \mathit{Nothing}' \qquad\qquad\qquad\qquad ::: \mathsf{run}\ x'\ \mathit{ev}'\ \mathit{xs}' \\
\qquad \mathit{Fst}\ \ (e ::: \mathit{ev}')\ \mathit{xs}' \qquad\ \to \mathit{Just}'\ (\mathsf{unbox}\ f\ e\ (x\ \text{`}\mathit{at}\text{`}\ t))\ ::: \mathsf{run}\ x\ \ \mathit{ev}'\ \mathit{xs}' \\
\qquad \mathit{Both}\ (e ::: \mathit{ev}')\ (x' ::: \mathit{xs}') \to \mathit{Just}'\ (\mathsf{unbox}\ f\ e\ (x'\ \text{`}\mathit{at}\text{`}\ t))\ ::: \mathsf{run}\ x'\ \mathit{ev}'\ \mathit{xs}'))
\end{array}
$$

The event *sample f e b* samples *b* whenever *e* occurs and produces an occurrence of the returned event using the function *f*. The issue is the case for *Snd* where no event occurs, but the behaviour updates. Thus, we are forced to produce a value even though the event has not fired, and therefore we should not sample the behaviour. By making the return type use *Maybe′*, we can use *Nothing′* in the *Snd* case to indicate that we do not produce a sample.

To accommodate the intended semantics of *filter* and *sample*, we revise the type *Ev* so that it can represent events that do not produce a new value each time the clock of the delayed signal ticks:

$$\textbf{data}\ \mathit{Ev}\ a = \mathit{Dense}\ (\bigcirc(\mathit{Sig}\ a)) \mid \mathit{Sparse}\ (\bigcirc(\mathit{Sig}\ (\mathit{Maybe}'\ a)))$$

That is, in addition to the old representation of events as $\bigcirc(\mathit{Sig}\ a)$, we also allow a *sparse* representation, which may not produce a value on each tick of the clock associated with the delayed signal. Having separate dense and sparse representations is not strictly necessary, but it allows for more efficient implementations of events that are constructed using only dense combinators such as all those in Fig. 2. For example, *mapE* is now implemented as follows:

```
mapE :: □(a → b) → Ev a → Ev b
mapE f (Dense sig) = Dense (run sig) where
  run :: ○(Sig a) → ○(Sig b)
  run sig = delay (let x ::: xs = adv sig in unbox f x ::: run xs)
mapE f (Sparse sig) = Sparse (run sig) where
  run :: ○(Sig (Maybe' a)) → ○(Sig (Maybe' b))
  run sig = delay (let x ::: xs = adv sig in (unbox f <$> x) ::: run xs)
```

For a *Dense* event, the implementation is the same as in Sect. 3.2 before, which again yields a *Dense* event. The *Sparse* case works in a similar fashion and produces a *Sparse* event. The other two event combinators *interleave* and *scan* can be generalized in the same way as well.

More importantly, the revised definition of *Ev* allows us to implement filtering of events and sampling of behaviours. For example, filtering can now be implemented as follows:

```
filter :: □(a → Bool) → Ev a → Ev a
filter p (Dense ev) = Sparse (run ev) where
  run ev = delay (let x ::: xs = adv ev
                  in (if unbox p x then Just' x else Nothing') ::: run xs)
filter p (Sparse ev) = Sparse (run ev) where
  run ev = delay (case adv ev of
    Nothing' ::: xs → Nothing' ::: run xs
    Just' x  ::: xs → (if unbox p x then Just' x else Nothing') ::: run xs)
```

4.2 Behaviours

We now turn to behaviour combinators that cannot be expressed using the simple representation from Sect. 3, namely *stop*, *derivative*, and *integral* from Fig. 3.

We account for these combinators by generalizing the definition of *Pull a* so that it no longer represents functions from *Time* to *a* but rather some form of state machine. We will perform this generalization in two steps: The first generalization allows us to halt a behaviour during its pull mode. We then further extend this to more general state machines.

The behaviour *stop p b* first behaves like behaviour *b*, but remains constant as soon as *b* has a value that satisfies the predicate *p*. This is useful for example if *b* is a timer which should stop once it reaches its maximum value. The variant *stopWith p b* is a slight generalization of *stop*: As soon as *p* produces a value *Just′ v*, the behaviour stops and takes on the constant value *v*. To be able to represent behaviours that may stop in between ticks of a clock, we only have to extend *Fun* so that it allows us to return a Boolean flag indicating whether the behaviour has stopped:

$$\textbf{data}\ \mathit{Pull}\ a = K\ a \mid \mathit{Fun}\ (\mathit{Time} \to (a \times \mathit{Bool}))$$

With this representation, we can implement *stop* as follows:

$$\begin{aligned}
&\mathit{stop} :: \Box(a \to \mathit{Bool}) \to \mathit{Beh}\ a \to \mathit{Beh}\ a \\
&\mathit{stop}\ p\ (K\ x \quad ::: xs) = K\ x ::: \textbf{if}\ \mathsf{unbox}\ p\ x\ \textbf{then}\ \mathsf{never}\ \textbf{else}\ \mathsf{delay}\ (\mathit{stop}\ p\ (\mathsf{adv}\ xs)) \\
&\mathit{stop}\ p\ (\mathit{Fun}\ f ::: xs) = \mathit{Fun}\ (\mathsf{box}\ (\lambda t \to \textbf{let}\ (x \times b) = \mathsf{unbox}\ f\ t \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in}\ (x \times (\mathsf{unbox}\ p\ x \vee b)))) : \\
&\qquad\qquad\qquad\qquad :: \mathsf{delay}\ (\mathit{stop}\ p\ (\mathsf{adv}\ xs))
\end{aligned}$$

When *stop* encounters a constant, we stop the behaviour by just producing a delayed behaviour that will never update. In the *Fun* case we update the Boolean component of the return value. The *stopWith* combinator is implemented in a similar fashion. The old behaviour combinators from Fig. 2 have to be updated to respect the Boolean stoppage flag.

We further generalize *Pull* so that it can represent state machines, which allows us to implement combinators to perform derivation and integration. To this end, we replace the *Fun* constructor with an *SM* constructor that takes two arguments: The current state of some type *s* and a function that makes the state machine perform one state transition. The *Pull* type is now defined using the syntax for generalized algebraic data types so that we can existentially quantify the type *s* and restrict it to the *Stable* type class:

$$\begin{aligned}
&\textbf{data}\ \mathit{Pull}\ a\ \textbf{where} \\
&\quad K \quad :: a \to \mathit{Pull}\ a \\
&\quad \mathit{SM} :: \mathit{Stable}\ s \Rightarrow s \to \Box(s \to \mathit{Time} \to (a \times \mathit{Maybe}'\ s)) \to \mathit{Pull}\ a
\end{aligned}$$

Applying the function stored by *SM* to the current state and time, gives us the current value of the behaviour and possibly a new state. If no new state is returned, the behaviour stops. So this definition allows us to implement *stop* and *stopWith* along with the other behaviour combinators from Fig. 2.

With this revised definition of *Pull*, we can still look up the current value of a *Pull* element via the *at* function and construct continuous behaviours:

$at :: Pull\ a \to Time \to a$
$at\ (K\ a)\quad \_ = a$
$at\ (SM\ s\ f)\ t = fst'\ (\mathsf{unbox}\ f\ s\ t)$
$cont :: \Box(Time \to a) \to Beh\ a$
$cont\ f = SM\ ()\ (\mathsf{box}\ (\lambda\_\ t \to (\mathsf{unbox}\ f\ t \times Just'\ ()))) ::: \mathsf{never}$

To implement a continuous behaviour we pick the unit type () as the state space of the state machine, so that the state transition function simply applies the given function f and remains in the () state.

To illustrate how we can make use of the state machine in *Pull*, we consider the implementation of the *derivative* and *integral* combinators shown in Fig. 4. Note that instead of plain behaviours of type *Beh Float* both combinator produce behaviours in the C monad, which we need to read the current time using $\mathsf{time} :: C\ (Time)$. The derivative requires timing information in order to compute a numerical approximation of the derivative: Given an observed value $v :: Float$ at time $t :: Time$ and another value $v' :: Float$ shortly later at time $t' :: Time$, the value of the derivative at time t' can be computed as $(v' - v)\ /\ (t' \ominus t)$ where $\ominus :: Time \to Time \to Float$ computes the time difference between two time points measured in seconds. Here we only consider *Float* behaviours, but the definition can be easily generalized to any type representing a vector space.

The main work of *derivative* is performed by the *der* helper function which takes two additional arguments: the previously observed value v_0 and the time t_0 at which v_0 was observed. The C monad of the *derivative* combinator is only used to obtain the first such time stamp t_0. The behaviour produced by *der* uses the *SM* constructor so that it can use the state to store the last observed value v_0 of the underlying behaviour and the time t_0 at which v_0 was observed. In addition, if the underlying behaviour itself uses a state machine, the state machine produced by *der* also stores the state s of the underlying state machine.

The implementation of *integral* uses a similar idea – with only two notable differences: First, instead of keeping track of the last observed value v_0 from the underlying behaviour, it keeps track of the last *produced* value c of the integral and uses that as the integration constant. Initially, this integration constant c is the first argument provided to the *integrate* combinator. Second, in case the underlying behaviour uses a state machine with state space s, it produces a state machine with state space $Float \times Time \times (s \oplus Float)$, where $\oplus$ is the strict sum type constructor. This is necessary to account for the fact that if the underlying state machine stops and maintains a constant value $v :: Float$, the new state machine needs to keep track of v instead of the state s of the underlying state machine.

5 GUI Programming

Async Rattus provides a library for constructing GUIs using signals [7]. Since this library uses signals, it only supports event-driven reactive programming. This is sufficient for most applications, since GUIs are mostly event driven. But this

```
derivative :: Beh Float → C (Beh Float)
derivative (x ::: xs) = (λt → der (x ::: xs) (x `at` t) t) ⟨$⟩ time where
    der :: Beh Float → Float → Time → Beh Float
    der (K x ::: xs) v0 t0 = SM (v0 × t0) (box fun) ::: next where
      next = withTime (delay (der (adv xs) x))
      fun (v0 × t0) t' = (x − v0) / (t' ⊖ t0) × Just' (x × t')
    der (SM s f ::: xs) v0 t0 = SM (v0 × t0 × s) (box fun) ::: next where
      next = withTime (delay (λt' → der (adv xs) (SM s f `at` t') t'))
      fun (v0 × t0 × s) t' = case unbox f s t0 of
        v × Just' s'   → (v − v0) / (t' ⊖ t0) × Just' (v × t' × s')
        _ × Nothing' → 0 × Nothing'
integral ::       Float → Beh Float → C (Beh Float)
integral c (Beh xs) = Beh ⟨$⟩ int xs c ⟨$⟩ time where
  int :: Sig (Pull Float) → Float → Time → Sig (Pull Float)
  int (K a ::: xs) c t = SM () (box fun) ::: next where
    next = withTime (delay (λt' → int (adv xs) (c + a ∗ (t' ⊖ t)) t'))
    fun s t' = c + a ∗ (t' ⊖ t) × Just' s
  int (SM s f ::: xs) c t = SM (c × t × Left' s) (box fun) ::: next where
    next = withTime (delay (λt' → int (adv xs) (c + (SM s f `at` t') ∗ (t' ⊖ t)) t'))
    fun (c × t × ls) t' =   c' × Just' (c' × t' × s')
      where c' = c + v ∗ (t' ⊖ t)
            v × s' = case ls of Right' v → v × Right' v
                                Left' ls' → case unbox f ls' t of
                                  v × Nothing' → v × Right' v
                                  v × Just s'    → v × Left' s'
```

Fig. 4. Implementation of derivatives and integrals.

event-driven approach lacks means to express inherently continuous behaviour such as smooth animations. A simple example of this is a progress indicator for a timer, which should update smoothly rather than discretely.

5.1 A Simple GUI Library

In this section, we present an alternative GUI library for Async Rattus that is built upon events and behaviours. To this end, we begin by first reviewing the original, event-driven GUI library of Async Rattus, called Widget Rattus.

GUIs are implemented in Widget Rattus as nested widgets. Such widgets include atomic UI elements such as text fields and buttons. In addition, some widgets consists of other widgets such as a stack of widgets that are horizontally aligned on screen. For example, buttons and text fields are represented by the following types:

```
data Button    = Button    { btnContent :: Sig Text, btnClick :: Chan () }
data TextField = TextField { tfContent  :: Sig Text, tfInput  :: Chan Text }
```

That is, a button consists of a signal that describes the text that should be displayed on the button and a channel that will produce a value () every time

the button is pressed. A text field has a similar structure, except that its channel has type *Chan Text* since it produces the text that the user has written into the text field.

Widgets can be combined into compound widgets such as a horizontal stack:

$$\textbf{data}\ \mathit{HStack}\ \textbf{where}\ \mathit{HStack} :: \mathit{IsWidget}\ a \Rightarrow \mathit{Sig}\ (\mathit{List}\ a) \rightarrow \mathit{HStack}$$

The *HStack* type uses an existential type a to allow arbitrary types of widgets to be combined. Importantly, a stack does not merely consist of a list of widgets but rather a *signal* of a list of widgets, which means that the list of widgets may change dynamically over time.

The design of the push-pull variant of the Widget Rattus library is very simple. It replaces signals with behaviours, so that the above widget types become

$$\begin{aligned}
&\textbf{data}\ \mathit{Button} &&= \mathit{Button} &&\{\, \mathit{btnContent} :: \mathit{Beh}\ \mathit{Text}, \mathit{btnClick} :: \mathit{Chan}\ ()\,\}\\
&\textbf{data}\ \mathit{TextField} &&= \mathit{TextField} &&\{\, \mathit{tfContent} :: \mathit{Beh}\ \mathit{Text}, \mathit{tfInput} :: \mathit{Chan}\ \mathit{Text}\,\}\\
&\textbf{data}\ \mathit{HStack}\ \textbf{where}\ \mathit{HStack} :: \mathit{IsWidget}\ a \Rightarrow \mathit{Beh}\ (\mathit{List}\ a) \rightarrow \mathit{HStack}
\end{aligned}$$

That is, the data made up by these widgets is described as time-varying data, i.e., as behaviours. By contrast the channels associated with widgets give rise to corresponding events. For example, each text field has an event that fires each time the text of the text field is changed by the user:

$$\begin{aligned}
&\mathit{tfInputEv} :: \mathit{TextField} \rightarrow \mathit{Ev}\ \mathit{Text}\\
&\mathit{tfInputEv}\ \mathit{tf} = \mathit{chanEv}\ (\mathit{tfInput}\ \mathit{tf})
\end{aligned}$$

Each widget provides smart constructors that allocate fresh channels using the chan primitive of Async Rattus. For example, buttons are constructed as follows:

$$\begin{aligned}
&\mathit{mkButton} :: \mathit{Beh}\ \mathit{Text} \rightarrow \mathit{C}\ \mathit{Button}\\
&\mathit{mkButton}\ t = \textbf{do}\ c \leftarrow \mathsf{chan}\\
&\qquad\qquad\qquad\ \mathit{return}\ (\mathit{Button}\ c\ t)
\end{aligned}$$

As a simple example, consider the implementation of a primitive timer GUI presented in Fig. 5. To aid readability, the let bindings use optional type annotations for the constructed intermediate behaviours and events. The implementation constructs a behaviour *startTime* whose value is the time at which the timer was started. To reset this start time every time the reset button is pressed, we use *sample*, which samples the current time each time the button is pressed. To save this sampled time we use *discr*. The timer value itself is then simply calculated as the difference of the start time and the current time.

5.2 Implementation

The push-pull GUI library can be implemented with relative ease: All widgets are implemented by translating them into a widget of the underlying Widget

```
timeB :: Beh Time
timeB = cont (box id)

window :: C VStack
window = do
  resetBtn ← mkButton (const "Reset timer")
  now ← time
  let resetEv      :: Ev ()      = btnOnClickEv resetBtn
  let startTimeEv :: Ev Time   = sample (box (λ_ t → t)) resetEv timeB
  let startTime    :: Beh Time = discr now startTimeEv
  let timer = zipWith (box (⊖)) timeB startTime
  let txt    = mapB (box (λt → "Current: " <> toText t)) timer
  label ← mkLabel txt
  mkConstVStack (label × resetBtn)
main :: IO ()
main = runApplication window
```

Fig. 5. Simple timer GUI.

Rattus library. In fact, the *IsWidget* type class, which is implemented by all widgets, provides witnesses of such a translation function from push-pull widgets to widgets from the underlying Widget Rattus library:

```
class WidgetRattus.IsWidget (DiscrWidget w) ⇒ IsWidget w where
  type DiscrWidget w
  mkDiscrWidget :: w → C (DiscrWidget w)
```

Each widget type w must provide a translation function to a corresponding 'discretized' widget *DiscrWidget* w. In turn, this discretized widget type must be a widget type from the Widget Rattus library. For example, buttons are implemented as follows:

```
instance IsWidget Button where
  type DiscrWidget Button = WidgetRattus.Button
  mkDiscrWidget (Button click txt) = do
    txt′ ← discretize txt
    return (WidgetRattus.Button txt′ click)
```

The type definition declares *WidgetRattus.Button* the corresponding type from the underlying Widget Rattus library and the translation function performs this translation by discretizing the behaviour that describes the button's text into a discrete signal. The implementation of this discretization is given in Fig. 6.

The *discretize* function samples a given behaviour in order to obtain a discrete signal. To this end, *discretize* uses the C monad to obtain the current time, which is needed to sample any state machine *SM s f* that the given behaviour might produce. To drive the discretization, we also need a delayed computation that ticks at a desired rate, in this case 50 times a second, which is provided by *sampleInterval*.

```
sampleInterval :: ◯()
sampleInterval = timer 20000

discretizeT :: Beh a → Time → Sig a
discretizeT (K x ::: xs) _ = x ::: withTime (delay (discretizeT (adv xs)))
discretizeT (SM s f ::: xs) t = cur ::: next where
  (cur × s') = unbox f s t
  next = case s' of
    Nothing' → withTime (delay (discretizeT (Beh (adv xs))))
    Just' s'' → withTime (delay (case select xs sampleInterval of
      Fst xs' _ → discretizeT xs'
      Both xs' _ → discretizeT xs'
      Snd beh' _ → discretizeT (SM s'' f ::: beh')))

discretize :: Beh a → C (Sig a)
discretize b = discretizeT b ⟨$⟩ time
```

Fig. 6. Discretization of behaviours to signals.

The *discretizeT* function is initialized with the current time and then recursively traverses the given behaviour. The case for K is simple, as no sampling needs to be performed. In the case of a state machine $SM\ s\ f$, we advance the state machine by calling f with the current state s and current time t. If the state machine stops, no further sampling is possible, and we therefore continue with the tail of the behaviour. Otherwise, we need to recursively sample the state machine again by using *sampleInterval*: If *sampleInterval* ticks before the tail of the behaviour ticks, then must recursively sample with the new state s''. Otherwise, we can continue with the tail of the behaviour.

5.3 Extended Example

We have used the GUI library to implement a number of small case studies, including a calculator application and four examples of Kiss' 7 GUIs benchmark [14]. We include one of these case studies – an extended timer application – in abbreviated form in Fig. 7. Similar to the simple timer from Fig. 5, this extended timer has a 'reset' button to reset the timer to zero, and it displays the current time. In addition, the extended timer also has a slider with which to set a maximum time that determines when the timer will stop. This maximum is initially set to 5 seconds, which means that the timer stops after 5 seconds.

The basic behaviour of the timer is defined by the *timeFrom* function which takes two time values (of type *DTime*, which represents time *differences*): the maximum and the starting value for the timer. The slider has an associated event, provided by the $sliderEv :: Slider \to Ev\ Int$ function, which fires every time the value of the slider is changed by the user. We then use $fromSec :: Int \to DTime$ to turn the integer values into corresponding time values in seconds. The resulting event *maxValEv* is turned into an event *maxEv* of type

$$Ev\ (DTime \times DTime \to C\ (Beh\ (DTime \times DTime)))$$

```
stopTimer :: DTime → (DTime × DTime) → Maybe′ (DTime × DTime)
stopTimer max (a × _) | a ⩾ max  = Just′ (max × max)
                      | otherwise = Nothing′

timeFrom :: DTime → DTime → C (Beh (DTime × DTime))
timeFrom d max = do dt ← elapsedTime
                    let addTime = mapB (box (λt → t + d × max)) dt
                    return (stopWith (box (stopTimer max)) addTime)

initialMax :: Int
initialMax = 5

timerGUI :: C VStack
timerGUI = do
    -- Slider
  maxSlider ← mkSlider initialMax (const 1) (const 100)
  let maxBeh    :: Beh Int     = sldCurr maxSlider
  let maxValEv :: Ev DTime = mapE (box fromSec) (sliderEv maxSlider)
    -- Reset button
  resetBtn ← mkButton (const "Reset timer")
  let resetTrigger :: Ev DTime = btnOnClickEv resetBtn
    -- Input events: Ev (DTime × DTime → C (Beh (DTime × DTime)))
  let resetEv = mapE (box (λ_ (_ × max) → timeFrom 0 max)) resetTrigger
  let maxEv  = mapE (box (λmax (cur × _) → timeFrom cur max)) maxValEv
  let updEv  = interleave (box (λ_ m → m)) resetEv maxEv

  elapsedTime :: Beh (DTime × DTime) ← timeFrom 0 (fromSec initialMax)
  let timer :: Beh (DTime × DTime) = switchRC elapsedTime updEv
    -- Output
  text ← mkLabel (mapB (box (λ(t × _) → "Current: " <> toText t)) timer)
  maxText ← mkLabel (mapB (box (λmax → "Max: " <> toText max)) maxBeh)
  mkConstVStack (maxSlider × maxText × text × resetBtn)
```

Fig. 7. Timer GUI from Kiss [14].

which produces a function that takes the current state of the timer (consisting of elapsed and maximum time) and produces the new behaviour of the timer, namely the behaviour that now has a new maximum value. The same happens with the click event of the reset button, for which we produce a new behaviour with the elapsed time set to 0. The two events *maxEv* and *resetEv* are then combined with *interleave*, and the resulting event *updEv* is used to define the global behaviour of the timer using *switchRC*, which a variant of *switchR* that allows the event to produce functions in the C monad.

6 Related Work

Functional Reactive Programming offers a high-level paradigm for building reactive systems, which has been explored in many ways since its original introduction by Elliott and Hudak [8]. Numerous FRP systems have since emerged, which can be split into two evaluation approaches: push- and pull-based.

Previous work has primarily focused on push-based FRP approaches. However, to get the advantages of both approaches, a push-pull approach was proposed by Elliott [9]. To our knowledge there have not been any implementations of Elliott's push-pull model in a modal FRP language. Elliott highlights Lula-FRP [17], which shares many conceptual similarities with the semantics of push-pull based FRP. Nevertheless, Lula-FRP is purely pull-based, making it susceptible to pull-sampling latency issues. The most comparable implementation to a push-pull based approach is Reflex [18], which introduces distinct abstractions: behaviour (pull), event (push), and dynamic (push-pull). The dynamic type in Reflex combines the characteristics of behaviours and events, effectively serving as a tuple that integrates both push- and pull-based functionalities.

The use of modal types in FRP has attracted attention due to its potential to address the issues in traditional FRP, such as space-time leaks and causality [15,3,5,2,4,10]. These issues arise from the high-level abstractions of FRP, which, while powerful, make it challenging to predict and manage resource usage in programs written in this paradigm. To mitigate these challenges, FRP languages require implementation strategies that eliminate space-time leaks [15]. Widget Rattus [7] is based on such an FRP language, called Async Rattus [11], which implements a calculus that guarantees causality, productivity, and the absence of space leaks.

Widget Rattus is an FRP GUI library that extends Async Rattus with two new language features: first-class channels and continuous types. Besides Widget Rattus there is a long history of using the FRP paradigm to implement GUI frameworks in functional languages [10,1,16,6]. One such language is the Elm language [6], which was initially implemented as an embedded language in Haskell for FRP-based GUI programming, but has since abandoned this paradigm in favour of the Elm Architecture.

7 Conclusion

We have explored Elliott's push-pull approach to FRP [9] in the setting of an asynchronous modal FRP language. Much of Elliott's original implementation can be translated into this setting, with only small adjustments to account for the requirement of stable types for some combinators. This results in the addition of *Stable* constraints, e.g., on *zipWith*, and the use of the $\Box$ modality for function arguments.

In addition, we also expanded upon Elliott's work by proposing refinements of the definition of events and behaviours. The addition of sparse events is only needed to overcome one of the limitations imposed by Async Rattus' $\bigcirc$ modality. By contrast, the addition of state machines to behaviours extends the expressiveness of behaviours. This allows us to express finite continuous behaviours, e.g., via the *stop* combinator, and it allows us to implement integration and derivation combinators. While this refinement of the definition of behaviours comes at the expense of the simplicity of the implementation, the user of the resulting combinator library is not burdened by this additional conceptual complexity.

References

1. Apfelmus, H.: Reactive Banana (2011), 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). https://doi.org/10.1017/S0956796822000132, 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). https://doi.org/10.1145/3434283, https://doi.org/10.1145/3434283, 00002
5. 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
6. Czaplicki, E., Chong, S.: Asynchronous functional reactive programming for GUIs **48**(6), 411–422. https://doi.org/10.1145/2499370.2462161, https://doi.org/10.1145/2499370.2462161
7. Disch, J.C., Heegaard, A., Bahr, P.: Functional reactive GUI programming with modal types. In: Gibbons, J. (ed.) Trends in Functional Programming. pp. 93–114. Springer Nature Switzerland (Oct 2025). https://doi.org/10.1007/978-3-031-99751-8_5, https://link.springer.com/chapter/10.1007/978-3-031-99751-8_5
8. 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). https://doi.org/10.1145/258948.258973
9. 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). https://doi.org/10.1145/1596638.1596643, http://doi.acm.org/10.1145/1596638.1596643, 00145 event-place: Edinburgh, Scotland
10. Graulund, C.U., Szamozvancev, D., Krishnaswami, N.: Adjoint reactive GUI programming. In: FoSSaCS. pp. 289–309 (2021)
11. Houlborg, E., Rørdam, G., Bahr, P.: Async Rattus (2023), https://hackage.haskell.org/package/AsyncRattus
12. 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). https://doi.org/10.1145/2103776.2103783
13. 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
14. Kiss, E.: 7GUIs: A GUI programming benchmark. https://eugenkiss.github.io/7guis/tasks (2014), https://eugenkiss.github.io/7guis/tasks
15. Krishnaswami, N.R.: Higher-order Functional Reactive Programming Without Spacetime Leaks. In: Proceedings of the 18th ACM SIGPLAN International Con-

ference on Functional Programming. pp. 221–232. ICFP '13, ACM, Boston, Massachusetts, USA (2013). https://doi.org/10.1145/2500365.2500588
16. Sage, M.: FranTk – a declarative GUI language for Haskell **35**(9), 106–117. https://doi.org/10.1145/357766.351250, https://dl.acm.org/doi/10.1145/357766.351250
17. Sperber, M.: Computer-assisted lighting design and control
18. Trinkle, R.: Reflex (2016), https://reflex-frp.org