Implementation of Push-Pull based FRP in Widget Rattus

IT University of Copenhagen

2nd June, 2025

Lasse Faurby Klausen *lakl@itu.dk*
Philip Flyvholm *phif@itu.dk*

Supervisor Patrick Bahr
Course code KISPECI1SE
Project code S25KISPECI1SE863

Abstract

Functional Reactive Programming (FRP) offers an expressive paradigm for building and interacting with reactive systems, using continuous and discrete values to model change over time. In recent years, several FRP languages have introduced modal types that prevent FRP programs from having implicit space leaks. Traditional FRP implementations rely on either push- or pull-based evaluation, each with distinct trade-offs in performance and applicability.

In this paper, we explore the suitability of a hybrid push-pull evaluation model for FRP, leveraging the advantages of both approaches. We explore implementing a push-pull model in Widget Rattus, which is a push-based modal FRP language for GUI programming embedded in Haskell. Additionally, we propose and evaluate several refinements to the push-pull model. Our findings are demonstrated through two case studies.

Contents

1 Introduction

Traditionally graphical user interfaces (GUIs) rely on an imperative programming model built upon shared mutable state and callback-driven architectures. While efficient, it can be hard to reason about elements such as: mutable state, higher-order functions and concurrency. Functional Reactive Programming (FRP) [1] has emerged as an alternative paradigm for building dynamic and reactive systems using time-varying values, known as *behaviours* or *signals*. Generally FRP can be divided into two evaluation models: push-based (also known as data-driven) and pull-based (also known as demand-driven). Traditionally, many FRP implementations have used pull-based sampling [5] due to:

- Pull-based sampling fits well with the common functional programming style of recursive traversal with parameters.
- Since behaviours might change continuously, idling until the next discrete change is not possible, and therefore pull-based sampling is necessary.

However using pull-based sampling is very inefficient, due to it recomputing values, even when no input change has happened. Another problem with pull-based sampling is that it imposes latency between each pull, which is not ideal in an interactive GUI scenario.

A typically way to evaluate reactive systems, such as GUIs, is using push-based evaluation. At the occurrence of an event, such as a mouse click or keyboard press, the system will react to this event and update the GUI with minimal latency. Such a modern GUI framework is *Widget Rattus* [21], an FRP language built as an extension of *Async Rattus* [20]. Widget Rattus is implemented as an embedded language in Haskell and employs a push-based evaluation. Accordingly, throughout this paper, we use Haskell syntax.

Elliot [5] demonstrates that it is possible to get the efficiency and minimal latency from push-based evaluation, and the simple functional programming style and applicability to sample continuous behaviours from pull-based evaluation. This hybrid approach ensures values are recomputed only when their discrete or continuous inputs change, and immediately when such changes occur.

In this paper, we set out to explore the suitability of implementing such a push-pull based FRP system for GUI programming using Widget Rattus and explore possible improvements to the proposed system. In short, this paper makes the following contributions:

1. We provide an overview of the key background concepts (Section 2)
2. We implement a simple push-pull based FRP in Widget Rattus with continuous and discrete values. Additionally, we update the widget library in Widget Rattus to utilize the push-pull model. (Section 3)
3. We present two case studies to demonstrate the use of our push-pull model. (Section 4)
4. We refine our initial push-pull model with five improvements and reflect on their impact. (Section 5)
5. We compare our push-pull model with related works (Section 6)

2 Background

In this section we will go over the core concepts related to Functional Reactive Programming, such as push- and pull-based evaluation, and the hybrid approach of push-pull based FRP that Elliot [5] suggests. We will present the FRP language Widget Rattus, and outline its core fundamental principles and demonstrate how to construct simple examples using it.

2.1 Functional Reactive Programming

Functional Reactive Programming is a programming paradigm for working with reactive programming in functional languages. Systems using reactive programming have ongoing interactions with the environment. This could be by receiving input, modifying the internal state and producing output. Typically, such systems include GUIs, web frameworks and robotics, since these types of systems generally receive input and must react accordingly.

FRP was introduced by Elliot and Hudak in their work *Functional Reactive Animation* [1], originally presented as a collection of data types and functions to compose interactive multimedia animations. The core abstractions they introduced were:

- **Behaviours:** Time-varying values representing continuous change over time, where discretization happens automatically during rendering. Originally, a $Behaviour\ a$ was defined as a function from time to a value of type a, called a time function, as such:

  $$\textbf{type}\ Behaviour\ a\ =\ Time \to a$$

- **Events:** Streams of discrete occurrences at specific points in time. Originally, an $Event\ a$ was defined as a list of timestamped values of type a, as such:

  $$\textbf{type}\ Event\ a\ =\ [(Time,\ a)]$$

By treating behaviours and events as first-class, composable abstractions, FRP enables the composition of dynamic and reactive systems [5]. Implementations of FRP are generally divided into two variants: push-based and pull-based.

In push-based FRP, the system reacts to incoming values or events as they are pushed from sources, such as GUIs or external data feeds. This approach is typically used in systems that listen for user input, e.g. button clicks or data sources that update periodically. Push-based FRP has the disadvantage that computation only occurs when new input is received, which can pose a problem if there is a lack of input. A simple example of this problem would be a clock in a UI, that shows the current time. To achieve this, we have to push the current time to the renderer constantly to make sure that the time is updated.

In contrast, in pull-based FRP, the system waits for the result to be demanded, and the result is thereby only computed when needed. This approach fits more naturally with the functional paradigm, since values can be evaluated lazily and only when used. However, using pull-based FRP can impose a significant latency due to the delay between the occurrence of an event, and the result being computed because of the sample interval between each pull.

2.2 Push-Pull Based FRP

Ideally, we would like to have an FRP system that combines the responsiveness of push-based evaluation and the on-demand computation of pull-based evaluation. Such a hybrid approach is presented by Elliot [5], which extends the previously mentioned events and behaviours. In this hybrid approach, events represent discrete values and uses push-based evaluation, while behaviours represent continuous time-varying values using both push- and pull-based evaluation. Having both behaviours and events allows the system to handle both discrete and continuous values, making it suitable for interactive applications, such as GUIs.

At the core of his approach are reactive values, which represent values that may change over time in response to future events. These are used to implement push-functionality, where updates happen through a stream of future discrete changes. Reactive values are defined recursively, meaning they contain a current value and a future reactive value. Reactive values are defined in the *Reactive* data type as such:

data $Reactive\ a\ = a\ `Stepper`\ Event\ a$

The stepper function defines a reactive value which remains constant between discrete changes. Here, the a (on the left side of *Stepper*) is the current value and $Event\ a$ represents potential future updates. In this formulation, the *Event* captures discrete changes that may occur at future points in time. Events are defined using future values, $Future\ a$, which contains a value of type a and an associated time, $\hat{T}$, at which the future value occurs. The $Future$ and $Event$ types are defined as follows:

newtype $Future\ a\ =\ Fut\ (\hat{T},\ a)$

newtype $Event\ a\ =\ Ev\ (Future\ (Reactive\ a))$

The *Behaviour* type is defined as a reactive time function. Because of the semantics of reactive values defined earlier, then the *Behaviour* type supports push-functionality [5].

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

The pull-functionality of behaviours are based on time functions which can simply be represented as functions, i.e. $Time \rightarrow a$. However, Elliot [5] notes that certain optimizations are prevented due to functions being opaque at run-time. Such as differentiating between constant functions and time-varying functions. To enable this, a $Fun\ t\ a$ type is proposed to split the time function into a constant and time-varying case. The types of *Behaviour* and Fun are therefore defined as follows:

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

data $Fun\ t\ a = K\ a \mid Fun\ (t \rightarrow a)$

Using the definitions formulated by Elliot, we can now represent discrete values with the type *Event* and continuous values with the type *Behaviour*. Given the reactive nature of behaviours, it is possible to push new values, while still being able to pull the current value by evaluating the Fun type.

Having reactive values that depend on time requires that time is defined. Elliott presents relative and absolute time semantics, both of which can be used in the system with different implementation details.

Relative time is defined as the time relative to the current time. This is useful if you have to run another event after a specific amount of time. However, it increases the complexity to compare two events and determine when both events have occurred, since time is relative to their respective reference points. Given the events have the same reference points then you can easily determine the order of events. However, this becomes more challenging if they do not share the same reference points, since now we also need to determine which reference point occurred first.

Absolute time is defined as a fixed global reference point, such as global timestamps. This is useful if you want to run an event at a specific time. Given two events you can determine the order of events based on their respective timestamps. The problem with this approach is that you have to keep track of a global reference point.

2.3 Introduction to Widget Rattus

This and the following subsections regarding Widget Rattus, relies on examples from the respective paper [21].

Widget Rattus is an extension of the *Async Rattus* language [20], a shallowly embedded language in Haskell, provided as a plugin, based on the *Async Ratt* calculus [19, 25]. This language introduces *signals*, which are a stream of discrete values based on modal types. In the following sections we will introduce these concepts in depth.

There are two major differences between Haskell and Widget Rattus. Firstly, Widget Rattus is eagerly evaluated, in contrast to Haskell that uses lazy evaluation by default. With eager evaluation the value is computed when available, while lazy evaluation computes the value when needed. The reason for this choice is to prevent space leaks, since we do not store already evaluated computations, while still allowing signals to be directly manipulated [20].

Secondly, Widget Rattus introduces two type modalities, $\bigcirc$ and $\Box$, also called the *later* and *box* modalities. The later modality expresses the passage of time at the type level [15]. This makes it possible to differentiate between the type a, which is available now, and the type $\bigcirc a$ which is available in the future. The box modality ensures that types can be safely and efficiently moved across time. These modalities and the type system will be described in more detail in the following sections. Selected typing rules for the Widget Rattus language[1] can be seen in Figure 1.

2.4 Clocks and Delayed Computations

In Widget Rattus, the later modality $\bigcirc$ is used to represent values which are available in the future. A value of type $\bigcirc A$ represents a delayed computation that produces a value of type A, at some point in the future. An exception to Widget Rattus' eager evaluation, is the

[1]The typing rules are based on [21, 20], however the typing rule for *delay* are incorrect in both of these papers. We have revised it and instead use the correct rule defined in [19].

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

$$\frac{\Gamma^\Box \vdash t :: A}{\Gamma \vdash \text{box}\, t :: \Box A} \qquad \frac{\Gamma \vdash s :: \bigcirc A \qquad \Gamma \vdash t :: \bigcirc B \qquad \checkmark \notin \Gamma'}{\Gamma, \checkmark_{\text{cl}(s) \cup \text{cl}(t)}, \Gamma' \vdash \text{select}\, s\, t :: \text{Select}\, A\, B} \qquad \frac{\Gamma \vdash t :: \bigcirc A \qquad \checkmark \notin \Gamma'}{\Gamma, \checkmark_{\text{cl}(t)}, \Gamma' \vdash \text{adv}\, t :: A}$$

$$\frac{}{\Gamma \vdash \text{never} :: \bigcirc A} \qquad \frac{}{\Gamma \vdash \text{chan} :: C(\text{Chan}\, A)} \qquad \frac{\Gamma \vdash t :: \text{Chan}\, A}{\Gamma \vdash \text{wait}\, t :: \bigcirc A}$$

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

Figure 1: Select typing rules for Widget Rattus

later modalities, which are evaluated lazily. This is required to compute the value when the delayed computation is needed, and not before. Widget Rattus uses this type modality to implement signals as such:

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

These signals can easily be manipulated using pattern matching or recursions. We have to ensure productivity of recursive functions, i.e. ensure the the program does not hang or enter an infinite loop. Therefore it is required that the recursive calls are guarded by a *delay* in Widget Rattus, as seen in Figure 1. A concrete example of this is shown later in this section.

Bahr et. al. [20] presents a combinator library for creating and interacting with signals. The combinator library presented will be used as the base for our implementation.

In Widget Rattus [21] each delayed computations has a local clock, due to the asynchronous nature of FRP systems. A clock θ is a set of input channels, and when the clock receives data on one of the channels, we call this a *tick* on the clock θ. These temporal dependencies are kept track of with the $\checkmark_\theta$ token as seen in the typing rules in Figure 1. Input channels could be button presses or text field changes, with the values referred to as *input values*.

Concretely this is implemented such that an element of type $\bigcirc a$ consists of a pair (θ, f) which is the local clock θ and a delayed computation f. When the clock θ ticks, f computes a value of type a. Accessing the delayed value is possible with the $adv :: \bigcirc a \to a$ (advance) function, which takes a delayed computation and returns the value produced. We can also construct a value of type $\bigcirc a$ by using the function $delay :: Clock \to a \to \bigcirc a$. This is however, an oversimplification, since *adv* and *delay* have to follow the typing rules presented in Figure 1. These typing rules specify that given a value t and its associated clock θ_t, then delaying t must occur before a tick on the clock θ_t, while advancing must occur after a tick on θ_t.

$increment :: \bigcirc Int \to \bigcirc Int$
$increment\ x = \text{delay}(\text{adv}\ x + 1)$

Above is an example of a function, *increment* x, using these concepts. It takes a delayed integer and increments it. Due to the typing rules of *delay* and *adv*, seen in Figure 1, we can not delay without advancing, and vice versa. Since x is a delayed computation of type $\bigcirc Int$, *increment* x will only produce a result when x is evaluated in a future time step. Let us look at advancing and delaying multiple integers. If we were to add two delayed integers, then we would assume it would look like this:

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

However, this would only work, if the two delayed integers x and y are guaranteed to arrive at the same time. This would mean using a global clock, which would be highly inefficient in e.g. GUIs and concurrent systems. The global clock would need to run as fast as the fastest input signal, to ensure that the GUI application shows the correct output. Widget Rattus fixes this by implementing dynamic local clocks [20], however this removes the guarantee that the clocks of x and y tick at the same time. To process more than one delayed computation, Widget Rattus implements a *select* primitive, as seen below, which handles the problem of multiple local clocks by forming a union of two clocks θ and θ', denoted as $\theta \sqcup \theta'$, which ticks whenever θ or θ' ticks.

data $Select\ a\ b = Fst\ a\ (\bigcirc b) \mid Snd\ (\bigcirc a)\ b \mid Both\ a\ b$

Given two delayed computations $x :: \bigcirc A$ and $y :: \bigcirc B$, then the primitive *select* $x\ y$ returns a value of type $Select\ A\ B$, i.e. a constructor $Fst\ A\ (\bigcirc B)$, $Snd\ (\bigcirc A)\ B$ or $Both\ A\ B$ depending on if the value of x arrives before, after or at the same time as y, respectively.

Another use case of the *select* primitive is to implement a combinator that allows us to switch from one signal to another signal:

$switch :: Sig\ a \to \bigcirc(Sig\ a) \to Sig\ a$
$switch\ (x ::: xs)\ d = x ::: delay\ (\textbf{case}\ select\ xs\ d\ \textbf{of}$
$\quad Fst\ xs'\ d' \to switch\ xs'\ d'$
$\quad Snd\ \_\ d' \to d'$
$\quad Both\ xs'\ d' \to d')$

The *switch* $xs\ ys$ function produces a signal that first behaves like xs, until ys arrives where it will behave like ys.

2.5 Stable Types and Box Modality

In Widget Rattus, there is a concept of time-dependent and time-independent data. An example of time-dependent data is delayed computations of type $\bigcirc A$, since they arrive in the future and thereby contain temporal dependencies. Such data requires the system to keep its computational values in memory, until the computation that references it is performed. If there is no system to restrict which values are kept in memory, then they may cause space-leaks.

To prevent this *stable* types are introduced. Widget Rattus restricts the programmers to only move data across time steps if they are a *stable* type. All types that can not carry temporal dependencies, such as Int and $Bool$, are stable, while types of the form $\bigcirc A$ and $A \to B$ are not stable. Delayed computations are per definition time-dependent, and are therefore not stable. Since functions can contain references to arbitrary time-dependent data in their closure, functions are time-dependent as well. All base types like Int and $Bool$ are stable. All algebraic data types and record types, that only contain stable types, are also stable.

Let's look at an example with time-dependent data. The following definition would fail with the Widget Rattus plugin, due to the variable scoping rules.

$$mapLaterBad :: (a \to b) \to \bigcirc a \to \bigcirc b$$
$$mapLaterBad\ f\ x = \text{delay}\ (f\ (\text{adv}(x))$$

These rules specify that types inside of a *delay* scope must be stable types. Therefore the function f is no longer in scope, since the type $(a \to b)$ is not stable.

Widget Rattus provides the box modality $\square$ to turn a non-stable type A into a stable type $\square A$, with some restrictions. Boxing of a type can be done using the *box t* primitive, which enforces restrictions to make sure that the boxed values are actually time-independent. Similarly to the *delay* primitive, *box* evaluates its arguments lazily, such that its argument t is only evaluated when the boxed value is forced using *unbox*.

Given the box modality, we can fix the *mapLater* function by requiring a boxed type for f, and then unboxing the value of f in the scope of *delay*. This can be seen below:

$$mapLater :: \square(a \to b) \to \bigcirc a \to \bigcirc b$$
$$mapLater\ f\ x = \text{delay}\ (unbox\ f\ (\text{adv}(x))$$

2.6 Continuous Types

Generally programs with signals of basic types are based on a linear structure, which means each value comes directly after each other. An example of this is with the signal of type $Sig\ Int$. This signals has the form $v_0 ::: (\theta_0, f_0)$, where v_0 is an integer, and f_0 is a delayed computation that produces a new signal of the form $v_1 ::: (\theta_1, f_1)$ when a tick occurs on the clock θ_0. This means that given ticking of the clocks, then the signal produces a linear sequence of values $v_0, v_1, ..., v_i$, triggered by the corresponding clocks.

Now imagine we have a signal of type $Sig\ (Sig\ Int)$, which will have the form:

$$Sig\ (Sig\ Int) = \underbrace{(v_0 ::: (\theta_0, f_0))}_{\text{inner signal}} ::: \underbrace{(\theta_1, f_1)}_{\text{outer signal}}$$

This means that the clock of the signal is $\theta_0 \sqcup \theta_1$ i.e. either the inner or outer signal can tick. This means that the produced output of this signal is a tree-shaped structure. This, however, introduces a problem of updating the signal, when the inner signal ticks, because then the outer signals has to update its current value as well.

This is one of the problems Widget Rattus tries to solve, since GUI programs generally have a tree-shaped structure. An example of this is if we had a $Widget$ type, that could describe a GUI widget, then we could represent the whole GUI as a signal with the type $Sig\ Widget$. Since widgets might recursively contain other widgets, then representing GUIs would be in the form of a tree-shaped structure. For example, we could define two widgets, *Label* and *LabelList*. *Label* defines a simple label that takes a signal of text and *LabelList* takes a signal consisting of list of labels, as well as a signal of a colour.

$$\textbf{data}\ Label = Label\ (Sig\ Text)$$
$$\textbf{data}\ LabelList = LabelList\ (Sig\ Colour)\ (Sig\ (List\ Label))$$

The *LabelList* widget describes the colour of the container, but also a signal of a list of labels. These labels also consist of signals and therefore the language needs to handle nested signals as well.

To solve this problem of updating signals in a tree-shaped structure, Widget Rattus introduces continuous types [21], which are types of values that can be updated over time.

A continuous type A has two functions defined. The clock function $clock :: A \to Clock$ and the update function $update :: InputValue \to A \to A$. Given we have a continuous type A, the value v of type A can be updated with $update\ i\ v$ when we receive an input value i on a channel $c \in clock(v)$.

With signals of type $Sig\ A$, where A is continuous, it has the form $v_0 ::: (\theta_0, f_0)$. The clock for this signal, is the union of $clock(v_0)$ with θ_0 i.e. $clock(v_0) \sqcup \theta_0$. When updating the signal the result depends on which clock ticks. If we receive an input value i on a channel $c \in clock(v_0) \sqcup \theta_0$ then it could either be the clock of A ticking or the clock of the signal ticking. The signal gets updated depending on the channel c, with the following cases:

- If $c \in \theta_0$, then it is the clock of the signal ticking. The signal is updated by computing the delayed computation f_0 resulting in a signal of the form: $v_1 ::: (\theta_1, f_1)$.
- If $c \notin \theta_0$, then it is the clock of A ticking. The signal is updated with $update\ i\ v_0 ::: (\theta_0, f_0)$.

Any basic type or stable type is a continuous type, since these types are time-independent. Continuous types are closed under product, sum and recursive types, however unlike stable types, continuous types are also closed under forming signal types. For example, if A is a continuous type then $Sig\ A$ is also continuous.

2.7 Widgets and Channels in WidgetRattus

Widget Rattus uses Monomer [14], a GUI library for writing user interfaces in Haskell. Monomer works by representing GUIs as models, generally one top-level model called $AppModel$, with widgets (or GUI components) containing the structure of the program. The GUI is then updated via events from button presses, input changes, etc. A new app model is rendered based on the occurrences of events.

Widgets must be able to produce data obtained from user interaction. Such an interaction could be keyboard input, mouse clicks, etc. To do this, Widget Rattus introduces two new primitives, *chan* and *wait*, to construct and interact with channels. This can also be seen in Figure 1. Data produced by widgets are used in channels, where *chan* creates a new channel of type $Chan\ A$ that can send data of type A. This is an effectful operation, as it allocates a fresh channel, which is indicated with the use of the C monad. The C monad allows executions of a set of effectful operations such as creating channels and getting the current time. To evaluate the C monad in a delayed context, one can use the function $delayC :: \bigcirc\ (C\ a) \to \bigcirc\ a$, provided by Widget Rattus.

Since we do not know when a channel receives data, we need to delay computations based on data from channels. To achieve this, the function $wait :: Chan\ A \to \bigcirc A$ can be used. These delayed computations will produce a value of type A when a value has been sent on the channel $Chan\ A$.

```
data SimpleButton = SimpleButton (Sig Text) (Chan ())
mkSimpleButton :: C SimpleButton
mkSimpleButton = do c ← chan
                    return (SimpleButtion (const "OK") c)
```

Above is a simplified example to show a use case of widgets using channels. We define a simple button which displays a constant signal of the text "OK". On each button press, the assigned channel receives a unit value.

To display different types in widgets, Widget Rattus has implemented the *Displayable* typeclass. This is used to define the *display* function, which is used to convert a datatype into a text representation.

```
class (Stable a) ⇒ Displayable a where
    display :: a → Text
```

Using this *Displayable* typeclass Widget Rattus defines a more abstract button which can display any *Displayable* type and not only of type $Text$. In Widget Rattus each GUI element has their own uniquely defined data-structure. For example, *Button* consists of two fields *btnContent*, which is a signal of the button's content and *btnClick*, an input channel triggered by clicking the button.

```
data Button where
    Button :: Displayable a
        ⇒ {btnContent :: Sig a, btnClick :: Chan ()} → Button
```

Widget Rattus generalizes widget construction via the $IsWidget$ typeclass. Given an instance of $IsWidget$ on a type A, it is possible to convert a value of type A into a Monomer widget.

This is done by defining the *mkWidgetNode* function which given a value of type A converts it to a *WidgetNode* type from Monomer.

```
class (Continuous a) ⇒ IsWidget a where
    mkWidgetNode :: a → Monomer.WidgetNode AppModel AppEvent
```

With the *IsWidget* typeclass, it is now possible to define a button instance as follows:

```
instance IsWidget Button where
    mkWidgetNode Button {btnContent = (val ::: _), btnClick = click} =
        Monomer.button (display val) (AppEvent click ())
```

This definition renders a Monomer button where the current value of the signal is displayed using the *display* function and an *AppEvent*. The *AppEvent* data type contains data about both the channel associated with the button, and the value it would produce when the button is pressed. See below for the reference of the *AppEvent* definition:

```
data AppEvent where
    AppEvent :: Chan a → a → AppEvent
```

Disch et al. [21] also presents a combinator library for building widgets and running applications. The library provides combinators for creating widgets, including buttons and sliders, using the corresponding *mkButton* and *mkSlider* functions.

The *mkButton* function is implemented as follows:

```
mkButton :: Displayable a ⇒ Sig a → C Button
mkButton t = do c ← chan
                return Button {btnContent = t, btnClick = c}
```

This function constructs a *Button* with the signal t assigned and allocates a new channel using the *chan* function.

It is then possible to listens for events on these widgets using the corresponding provided functions such as *btnOnClickSig* and *sldCurr*. These functions return signals that tick when the associated events are triggered.

The *btnOnClickSig* is defined by waiting for the *btnClick* channel to produce a value using the *wait* function. This is converted to a signal using the *mkSig* function, which recursively converts a boxed delayed computation into signal.

For reference these implementations can be found below:

```
mkSig :: □ (◯ a) → ◯ (Sig a)
mkSig b = delay (adv (unbox b) ::: mkSig b)

btnOnClick :: Button → □ (◯ ())
btnOnClick btn =
  let ch = btnClick btn
   in box (wait ch)

btnOnClickSig :: Button → Sig ()
btnOnClickSig b = mkSig (btnOnClick b)
```

An example use case of this library can be seen in Figure 2 of a simple counter GUI. Widget Rattus provides example GUIs for a set of Kiss' 7GUIs [12]. All following GUI examples presented will also be from 7GUIs, to make it easier to compare a push-pull based approach with Widget Rattus' push-based approach. Our push-pull examples of the GUIs can all be seen in Appendix D.

In this example, the *mkButton* and *mkLabel* functions are used for creating a button and a label. Pressing the button increments the value displayed on the label. The *btnOnClickSig* function produces a signal that ticks on each button click, which accumulates the count by incrementing it with each tick using the *scanAwait* function. The *scanAwait* function is similar to the *scan* function in the Haskell base library [23], which applies a given function f to a signal.

These two widgets are then displayed in a vertical stack using the *mkConstVStack* function.

The *runApplication* function starts the GUI application using Widget Rattus, which renders the widget given. This function sets up a Monomer application with a builder capable of rendering the current model via the *mkWidgetNode* implementations for each *IsWidget* instance. An event handler is also defined to process incoming events and update the widgets accordingly.

```
counter :: C VStack
counter = do
    btn ← mkButton (const "Increment")
    let clicks = btnOnClickSig btn
    let counts = scanAwait (box(λn () → n + 1 )) 0 clicks
    lbl ← mkLabel counts
    mkConstVStack (lbl :* btn)

main :: IO()
main = runApplication counter
```

Figure 2: Counter GUI implementation

3 Simple Push-Pull in Widget Rattus

In this section we will show an initial implementation of push- and pull-functionality in an FRP language. In section 4 we will do a case study on a GUI example using this library we have created. In section 5 we will refine this simple push-pull model with more advanced concepts.

We have chosen to work in Widget Rattus, due to it providing first-class support for both later modality and stable types, making it a suitable language for us to explore and implement push-pull based FRP.

Our approach is inspired by the semantics presented by Elliot in his work on push-pull FRP [5]. We implement these semantics using type representations for events and behaviours as abstractions over the *Sig* type already defined in Widget Rattus. Behaviours will provide both push- and pull-functionality, while events only provide push-functionality. We will implement corresponding combinator libraries for the types, which we will explore in more detail in the following sections. Since Widget Rattus is strictly evaluated language, it is important that we define the push-pull model based on this. This can be done in Haskell using the ! operator [22]. Widget Rattus also provides a strict version of $Maybe$, defined as $Maybe'$, as well as the *:** infix operator for making strict pairs.

3.1 Events

To implement the event semantics described in section 2.2, we start by defining an event type from Elliot's definition:

```
newtype Event a = Ev (Future (Reactive a))
```

An event contains a future reactive value using the *Future* type. This type simply defines that the value will be available in the future. In Widget Rattus, the ◯ modality has a similar purpose of defining delayed computations.

The *Reactive* type models discrete value changes and is structurally similar to the *Sig* type used in Widget Rattus:

```
data Reactive a = a 'Stepper' Future (Reactive a)

data Sig a      = a ::: !(◯ (Sig a))
```

Given this similarity we can use *Sig* in place of *Reactive* in the event data constructor. We can represent future values using the ◯ modality. We decided to use the name *Ev* instead of *Event* for the type in the implementation:

```
newtype Ev a    = Ev(◯(Sig a))
```

An event is just a delayed signal, with the key difference being that, unlike regular signals, events do not assume an initial value is available. This aligns with the intuition of e.g., a keyboard press event; no value is present until a key is actually pressed.

Delaying the signal, allows us to "wait" for the event to occur. Once this value is available, the rest of the stream is represented by delayed signals $\bigcirc(Sig\ a)$. This means, that future occurrences of the event is also not known in advance.

Making $Ev\ a$ continuous is fairly straightforward given that a continuous instance requires a *clock* function, for when the type updates, and an *update* function for updating the value. An event is simply a delayed computation, which has the form (θ_0, f_0), where θ_0 is the clock and f_0 is a delayed computation that produces a signal. The clock of an event is therefore θ_0. The update function is not directly the f_0 function, since this returns a signal of the form $v_1 ::: (\theta_1, f_1)$. Since we need to return the same form, we can ignore v_1 and simply return (θ_1, f_1).

We have implemented an event library with a set of combinators for interacting events. The event library is shown in Figure 3

$$
\begin{aligned}
&\mathit{map} && :: \Box(a \to b) \to \mathit{Ev}\ a \to \mathit{Ev}\ b \\
&\mathit{stepper} && :: a \to \mathit{Ev}\ a \to \mathit{Beh}\ a \\
&\mathit{trigger} && :: \mathit{Stable}\ b \Rightarrow \Box(a \to b\ \to c) \to \mathit{Ev}\ a \to \mathit{Beh}\ b \to \mathit{Ev}\ (\mathit{Maybe}'\ c) \\
&\mathit{interleave} && :: \Box(a \to a \to a) \to \mathit{Ev}\ a \to \mathit{Ev}\ a \to \mathit{Ev}\ a \\
&\mathit{scan} && :: \mathit{Stable}\ b \Rightarrow \Box(b \to a \to b) \to b \to \mathit{Ev}\ a \to \mathit{Ev}\ b
\end{aligned}
$$

Figure 3: Event library

A common interaction would be to map an event with a function. The *map* combinator is fairly straightforward to implement as seen below:

```
map :: □ (a → b) → Ev a → Ev b
map f (Ev sig) = Ev (aux f sig)
  where
    aux :: □ (a → b) → ◯ (Sig a) → ◯ (Sig b)
    aux f xs = delay (let (x ::: xs') = adv xs in unbox f x ::: aux f xs')
```

The primary challenge of implementing map, is mapping in a delayed context. This is achieved by advancing the delayed signal xs, which has type $\bigcirc\ \mathit{Sig}\ a$. When advancing, it is crucial to ensure that the event stream is in a delayed computation, to maintain the correct semantics of the language. To avoid repeatedly wrapping and unwrapping the result with the Ev type, we introduce an auxiliary function which maps directly on the underlying signal. This can, however, be simplified by using the existing signal combinator library as such:

```
map :: □ (a → b) → Ev a → Ev b
map f (Ev sig) = Ev (Signal.mapAwait f sig)
```

In the above case we use the *mapAwait* function, which maps a delayed signal. To keep the code snippets readable and concise, we will utilize functions from the existing library in the future.

The *map* combinator can be used by giving a boxed function f and an event $Ev\ a$. For example incrementing a stream of natural numbers (nat) as such:

```
map (box (λx → x+1) nat
```

In situations where data is collected from several independent sources, such as multiple sensors operating, it could be useful to combine these event streams into a single stream. This is possible by using the *interleave* function. Consider having two event streams e_1 and e_2 and interleaving them, then a new event stream e_3 is created. When either e_1 or e_2 ticks, then e_3 ticks with the produced value. An important consideration arises in cases where two events occur at exactly the same time. In these scenarios, a user-defined function is required to determine how the values should be merged.

```
interleave :: □ (a → a → a) → Ev a → Ev a → Ev a
interleave f (Ev xs) (Ev ys) = Ev (Signal.interleave f xs ys)
```

An example of a use case for the *interleave* function could be merging two integer events, using the addition operator, like so:

```
                        x: 1  2   4 5  6 7 ...
                        y:    8 6 0    4   ...
interleave (box(+)) x y: 1 10 6 4 5 10 7 ...
```

Lastly a *scan* function similar to *scanl* [23] in Haskell is implemented for events. Given an initial value v and an event e_1, then a new event e_2 is created. When e_1 ticks then e_2 ticks as well with a value produced by a binary operator of the accumulator and the current value of e_1.

```
scan :: (Stable b) ⇒ □ (b → a → b) → b → Ev a → Ev b
scan f acc (Ev as) = Ev $ delay (Signal.scan f acc (adv as))
```

An example of a use case for the *scan* function could be to create an event stream that increments a number each time an event, *ev*, ticks.

```
scan (box (λn _ → n + 1)) 0 ev: 1 2 3 4 5 6...
```

The *stepper* and *trigger* combinators use behaviours, and thus will be implemented in the next section, after the *Beh* type has been defined.

3.2 Behaviours

A behaviour should have both push- and pull-functionality, meaning that a new value can be pushed, like signals and events, as well as having the ability to always pull the current value. These values are time-varying values, and therefore pulling current value requires the current time. As described by Elliot, we can optimize the pull-functionality by supporting constant values as well. These values are always the same no matter the time.

To implement the pull-functionality we introduce the $Fun\,a$ type, where a is the type of the values produced and is defined as follows:

```
data Fun a = K !a | Fun !(□ (Time → a))
```

The type is pretty similar to the type defined by Elliot; it contains a constant case $K\,a$ and a time-varying case $Fun\ \square(Time \rightarrow a)$. In contrast to Elliot, we decided to remove the type parameter for time and instead hardcode it to be the $Time$ type defined in Widget Rattus. $Time$ is a strict version of $UTCTime$ [18] defined in Haskell. UTC time is a global reference point, and therefore we use absolute time in our implementation.

We might require the type a to be stable, therefore we box the entire function $Time \rightarrow a$. If the function is not boxed then we would not be able to guarantee that the result of type a is stable.

The $Fun\ a$ type is continuous given a is continuous. We can prove this since, in the K case we simply have constant a, i.e. it updates when a updates. In the time-varying Fun case, the function is boxed, and therefore stable. As previous mentioned, all stable types are continuous as well.

To simplify the use of the Fun type we have created two functions, *apply* and $mapF$, as seen below. The *apply* function was presented in Elliot's paper and simplifies accessing the value of Fun types by treating both the constant and time-varying case the same way, i.e. using a function of type $Time \rightarrow a$ [5]. The $mapF$ function is used for mapping Fun types, which is useful for applying a function on a Fun type without applying time.

```
apply :: Fun a → (Time → a)
apply (K a) = (λ_ → a)
apply (Fun f) = unbox f

mapF :: □ (a → b) → Fun a → Fun b
mapF f (K a) = K (unbox f a)
mapF f (Fun t) = Fun (box (unbox f . unbox t))
```

After introducing the $Fun\ a$ type, we can define a behaviour as a signal of $Fun\ a$, as seen below. The $Fun\ a$ type enables pull-functionality by representing behaviours as either constant values or as time functions. This allows the system to evaluate the current value at any given moment, by applying the time function when needed. Since the behaviour type is implemented as a signal, this also enables push-functionality. Therefore, combining the Fun type and signals enables both push- and pull-functionality. Our definition is very similar to Elliot's, as seen in section 2.2, with the differences being that it uses the Sig type instead of *Reactive*, for similar reasons as events. The type can be seen here:

```
newtype Beh a = Beh (Sig (Fun a))
```

Given a continuous type a, then $Beh\ a$ is also continuous since it is equivalent to $Sig\ (Fun\ a)$. Both signals and $Fun\ a$ are continuous when a is continuous.

Given this definition of Beh we can create the simplest behaviour, the *Time Behaviour*. This behaviour returns the current time by simply using the Fun type with the identity function.

```
timeBehaviour :: Beh Time
timeBehaviour = Beh (Fun (box id) ::: never)
```

We can define the future values as *never*, which is presented in Widget Rattus as a way of saying that a signal never ticks again. Using this function we can always pull the current value of the behaviour and get the current time.

To generalize this concept of behaviours that only support pull-functionality of constant values, we created the *const* function. This function produces constant behaviours that never tick, by assigning *never* as the later value:

```
const :: a → Beh a
const x = Beh (K x ::: never)
```

To safely access time, we need to use the C monad, due to the *time* function having side-effects. One such case is with the *elapsedTime* function, which returns a behaviour based on a start time. At each pull we get the difference between the current time and the start time.

```
elapsedTime :: C (Beh NominalDiffTime)
elapsedTime = do
  startTime ← time
  return $
    Beh (Fun (box (λcurrentTime → diffTime currentTime startTime)) ::: never)
```

From the definition of *Beh* we can create an initial behaviour combinator library as seen in Figure 4.

$$
\begin{array}{ll}
const & :: Fun\ a \to Beh\ a \\
constK & :: a \to Beh\ a \\
map & :: \Box(a \to b) \to Beh\ a \to Beh\ b \\
zipWith & :: (Stable\ a,\ Stable\ b) \Rightarrow \Box(a \to b \to c) \to Beh\ a \to Beh\ b \to Beh\ c \\
zipWith3 & :: (Stable\ a,\ Stable\ b,\ Stable\ c) \Rightarrow \\
 & \quad \Box(a \to b \to c \to d) \to Beh\ a \to Beh\ b \to Beh\ c \to Beh\ d \\
switch & :: Beh\ a \to \bigcirc(Beh\ a) \to Beh\ a \\
switchS & :: (Stable\ a) \Rightarrow Beh\ a \to \bigcirc(a \to Beh\ a) \to Beh\ a \\
switchR & :: (Stable\ a) \Rightarrow Beh\ a \to Ev\ (a \to Beh\ a) \to Beh\ a \\
timeBehaviour & ::\ Beh\ Time \\
elapsedTime & :: C\ (Beh\ NominalDiffTime) \\
withTime & :: \bigcirc(Time \to a) \to \bigcirc a
\end{array}
$$

Figure 4: Behaviour library

Mapping of behaviours is similar to mapping of events and signals, as seen below. The *map* function simply applies a function f using the *mapF* function on the current value of type Fun.

```
map :: □ (a → b) → Beh a → Beh b
map f (Beh sig) = Beh $ Signal.map (box (λa → mapF f a)) sig
```

In some cases it would be useful to combine two behaviours. For example, to create a behaviour of the sum of two behaviours' values. This can be solved using the $zipWith\ f\ a\ b$ combinator, which applies a zip function f on the behaviours a and b. The implementation of $zipWith$ for behaviours is similar to the implementation of $zipWith$ for signals in Widget Rattus, with the major difference being that behaviours use the Fun type for the current value. To zip two values of the Fun type, then we use an auxiliary function $app\ x\ y$ which takes two Fun types and for each combinator zips them together using the function f, given by the $zipWith$ function.

```
zipWith :: (Stable a, Stable b) ⇒ □ (a → b → c) → Beh a → Beh b → Beh c
zipWith f (Beh (x ::: xs)) (Beh (y ::: ys)) =
  Beh $ app x y
        ::: delay
          ( let (Beh rest) =
                  ( case select xs ys of
                      Fst xs' lys → zipWith f (Beh xs') (Beh (y ::: lys))
                      Snd lxs ys' → zipWith f (Beh (x ::: lxs)) (Beh ys')
                      Both xs' ys' → zipWith f (Beh xs') (Beh ys')
                  )
             in rest
          )
  where
    app :: Fun a → Fun b → Fun c
    app (K x') (K y') = K (unbox f x' y')
    app (Fun x') (Fun y') = Fun (box (λt → unbox f (unbox x' t) (unbox y' t)))
    app (Fun x') (K y') = Fun (box (λt → unbox f (unbox x' t) y'))
    app (K x') (Fun y') = Fun (box (unbox f x' . unbox y'))
```

The *Stable* constraints are necessary in $zipWith$, since values of type $Fun\ a$ or $Fun\ b$ have to be moved into the future (whenever only one of the behaviours ticks). Due to this behaviour, the box modality is needed in the definition of Fun. The function f is also boxed due to the delayed context. If we omitted the box modality, the function would not be in scope in the delayed context.

A variant of the $zipWith$ function for three behaviours is also available, called $zipWith3$. This variant firstly zips behaviours a and b. The result of this zip is then zipped with the c behaviour. The $zipWith3$ function is useful for creating behaviours that are similar to if-else statements, since a becomes the condition, b is the true case and c is the false case.

```
zipWith3 :: forall a b c d. (Stable a, Stable b, Stable c) ⇒
            □ (a → b → c → d) → Beh a → Beh b → Beh c → Beh d
zipWith3 f as bs cs = zipWith (box (λf' x → unbox f' x)) cds cs
  where
    cds :: Beh (□ (c → d))
    cds = zipWith (box (λa b → box (λc → unbox f a b c))) as bs
```

Consider the scenario, where we have a behaviour y, which is in a delayed context, meaning that is is not currently available. Instead of waiting for the behaviour to arrive, we would instead want to use an initial behaviour x, until the behaviour y arrives. Once y arrives it will take over. In Widget Rattus, this is done with signals using the *switch* function. Using the *switch* function provided for signals, we can create a *switch* function for behaviours as well, as seen below:

```
switch :: Beh a → ◯ (Beh a) → Beh a
switch (Beh s) d = Beh $ Signal.switch s $ mapLater (box (λ(Beh a) → a)) d
```

This makes it possible to achieve the example described above simply by writing $switch\ x\ y$.

This *switch* function is a bit limiting in it is functionality since it does not support creating behaviours based on the previous state, and only allows switching once. In Widget Rattus, these limitations are removed using the $switchS$ and $switchR$ functions.

The *switchS* function is a stateful switch function, which means that it allows switching from an initial behaviour b_1 to a future behaviour b_2. The b_2 behaviour is produced depending on the current value of b_1.

An example use case of *switchS* is seen below. Here we use *switchS* to return a behaviour that shows the current time, until the *clickEv* ticks. When this event ticks it switches over to a constant behaviour which stores the current time.

```
stopWatch :: Beh Time
stopWatch = switchS timeBehaviour
    (box (delay (λa → let _ = adv (unbox clickEv) in const a))
```

Similarly to *switch*, we can create the *switchS* function for behaviours based on the implementation of *switchS* for signals. The implementation can be seen below:

```
switchS :: (Stable a) ⇒ Beh a → ◯ (a → Beh a) → Beh a
switchS (Beh s) d =
  Beh $
    Signal.switchS s $
      withTime $
        mapLater (box (λf a t → let (Beh s') = f (apply t a) in s')) d
```

The *withTime* function, used within the *switchS* function, is a helper function for getting the current time at a point in the future. It applies the current time to a delayed computation. It takes a delayed value of type $\bigcirc(Time \to a)$ and produces a delayed result of type $\bigcirc a$. This is done by advancing the delayed function, retrieving the current time from the C monad, and applying the time to the function. The use of *delayC* eliminates the C monad, yielding a delayed value.

```
withTime :: ◯ (Time → a) → O a
withTime delayed =
  delayC $ delay (let f = adv delayed in do f <$> time)
```

The *switchR* function is a recursive version of the *switchS* function. Instead of taking an argument of type $\bigcirc(a \to Beh\ a)$, it takes an event of type $Ev\ (a \to Beh\ a)$. This means that every time the event produces a new function, a new behaviour is computed and switched to.

```
switchR :: (Stable a) ⇒ Beh a → Ev (a → Beh a) → Beh a
switchR beh (Ev steps) =
    switchS beh (delay (let step ::: steps' = adv steps in
    (λx → switchR (step x) (Ev steps'))))
```

Now that the type *Beh* is defined, we can finish implementing our event combinators, that use behaviours.

Events are generally used for discrete values at some point in the future, and are therefore useful for button presses, input changes, etc. Due to the discrete nature of events, only push-functionality is needed. However this makes it challenging to get the value after it has ticked. For example, if you have a keyboard press event, and you want to access it at a later time, you would need pull-functionality. This is solved using the *stepper* function which converts an event into a behaviour. Since event values are discrete then we can convert the event into a behaviour made up of constant values.

Due to events being delayed computations, we do not have an initial value for the behaviour. We solve this by assigning an initial value when creating a behaviour. Alternatively we could return a delayed behaviour. However this would not make sense in our implementation, since our combinators do not produce any delayed behaviours.

```
stepper :: (Stable a) ⇒ a → Ev a → Beh a
stepper initial (Ev ev) = Beh (K initial ::: Signal.mapAwait (box K) ev)
```

Using the *trigger* function, it is possible to sample the current value from a behaviour each time an event ticks. In this context, we are working with the *select* function, which allows us to distinguish whether it is the event, the behaviour, or both that have ticked. In the case where the behaviour ticks, we do not want to produce a value. However, due to signals having to produce a value at every tick of its clock, then we also have to produce a value to the output. Therefore we can make use of the $Maybe'$ type and return $Nothing'$. As a result of this our return type is $Ev\ (Maybe'\ c)$. Since the values are discrete, we represent them as an event as well. We will explain this problem more and improve the solution in section 5.2. The implementation of *trigger* is as follows:

```
trigger :: (Stable b) ⇒ □ (a → b → c) → Ev a → Beh b → Ev (Maybe' c)
trigger f (Ev event) (Beh behaviour) = Ev (trig f event behaviour)
  where
    trig :: (Stable b) ⇒ □ (a → b → c) → ◯ (Sig a) → Sig (Fun b) → ◯ (Sig (Maybe' c))
    trig f' as (b ::: bs) =
      withTime $
        delay
          ( let d = select as bs
             in ( λt →
                    ( case d of
                        Fst (a' ::: as') bs' →
                            Just' (unbox f' a' (apply b t))
                            ::: trig f' as' (b ::: bs')
                        Snd as' bs' →
                            Nothing' ::: trig f' as' bs'
                        Both (a' ::: as') (b' ::: bs') →
                            Just' (unbox f' a' (apply b' t))
                            ::: trig f' as' (b' ::: bs')
                    )
                )
          )
```

Consider the following example: Given a behaviour *beh*, that contains the current time, and a button which emits an event *ev* on button click. If we want to sample the current time at the moment the button is clicked, we can use the trigger function to sample the behaviour in response to the event. This is shown in the example:

```
lastClickTime :: Ev Time = trigger (box (λ_ t) → t) ev beh
```

For an alternative example see Figure 5 for a more visual approach of the *trigger* function. Here we see how the event ticks at different times (e.g. every time you click on a button) and the value on the behaviour is sampled.

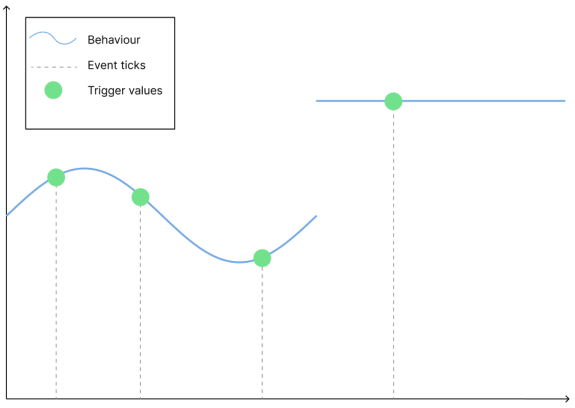

Figure 5: A visual example of the trigger function. The y axis is an arbitrary value that the behaviour produces, and the x axis is the time passed. The behaviour consists of two functions. The initial function is a time-varying sinus curve, while after ticking it becomes a constant function.

3.3 Push-Pull GUI Widgets

As previously mentioned Widget Rattus provides a library for creating and updating widgets using signals. Since this library uses signals, it only supports push-functionality. Our goal is to make it support push- and pull-functionality by using behaviours and events instead of signals.

A path we explored was to update the underlying data structures of widgets from Widget Rattus to use behaviours and events directly, instead of signals. For example, updating the *Button* widget would look like the following:

```
data Button' where
  Button' :: (Displayable a) ⇒
    {btnContent :: !(Beh a), btnClick :: !(Chan ())} → Button'
```

In Widgets Rattus, updates to the GUI are handled via the *continuous* instances of widgets. *Continuous* types define when a value is updated, which optimally should lead to a re-render of the value. This leads to a problem with the current definition of the *continuous* instance for the behaviour type, since it is defined by the signals instance of continuous. Since signals are push-based, that means that we are only notified about updates of behaviours when a new value is pushed, and thus ignoring the pull-functionality.

Because of this problem, we need to introduce some kind of pull-sampling to re-render the values in the GUI, which Elliot [5] also noted as the need for behaviours to be discretized when rendered. Using pull-sampling, we sample the value of a behaviour at a fixed interval. This approach, however, increases the latency between value updates and renders. Therefore, we need to minimize the sampling interval while still avoiding unnecessary re-computations of the same value. We will explain the implementation of the pull-sampling functionality later.

First we tried to implement the pull-sampling directly in the continuous instance of the behaviour type. However, this would not be ideal if you use behaviours outside of widgets. Since we want our push-pull implementation to be used generally, and not only in the context of widgets in Widget Rattus, we want the user to have the freedom of choosing their own pull-sampling method instead of us building it directly into the continuous instance of a behaviour.

Therefore we need to discretize outside of the continuous instance of behaviours. To achieve this, we suggest modifying only the widget library functions such as *mkButton*, *mkSlider*, etc., so these functions take a behaviour or an event as input parameter and internally discretize to convert the behaviours into signals and construct the signal-based widgets. The helper functions like *btnOnClick* would then convert the resulting signals and delayed computations back into behaviours and events. The resulting behaviours would then be represented as discrete. The fully updated widget library can be seen in Figure 6.

$$
\begin{array}{ll}
mkButton & :: (Displayable\ a) \Rightarrow Beh\ a \to C\ Button \\
btnOnClickEv & :: Button \to Ev\ () \\
mkTextField & :: Text \to C\ TextField \\
textFieldOnInput & :: TextField \to Ev\ Text \\
textFieldContent & :: TextField \to Beh\ Text \\
mkLabel & :: (Displayable\ a) \Rightarrow Beh\ a \to C\ Label \\
mkConstText & :: String \to Beh\ Text \\
mkHStack & :: (IsWidget\ a) \Rightarrow Beh\ (List\ a) \to CHStack \\
mkConstHStack & :: (Widgets\ ws) \Rightarrow ws \to C\ HStack \\
mkVStack & :: (IsWidget\ a) \Rightarrow Beh\ (List\ a) \to C\ VStack \\
mkConstVStack & :: (Widgets\ ws) \Rightarrow ws \to C\ VStack \\
mkTextDropdown & :: Beh\ (List\ Text) \to Text \to C\ TextDropdown \\
textDropdownCurrent & :: TextDropdown \to Beh\ Text \\
mkPopup & :: Ev\ Bool \to Beh\ Widget \to C\ Popup \\
mkSlider & :: Int \to Beh\ Int \to Beh\ Int \to C\ Slider \\
sliderOnChange & :: Slider \to Ev\ Int \\
sliderCurrent & :: Slider \to Beh\ Int \\
mkProgressBar & :: Beh\ Int \to Beh\ Int \to Beh\ Int \to C\ Slider \\
runApplication & :: (IsWidget\ a) \Rightarrow C\ a \to IO\ ()
\end{array}
$$

Figure 6: The updated GUI library. Notice the type *Widgets*, which is essentially a list of the *Widget* type.

The pull-sampling functionality has been implemented in the *discretize* function given below, which is part of the behaviour combinator library. This function takes a behaviour and converts it into a discrete signal using pull-sampling. The C monad is required since we need the time to compute the initial value of the behaviour.

The discretize functions works by recursively calling itself either when the behaviour ticks or when our sample interval ticks. Thereby both push- and pull-functionality is preserved. The function has been optimized further by having a case for constant cases. When a value is constant, then we can guarantee that the next update will be when a new value is pushed onto the behaviour, since the current value will not change.

```
discretize :: Beh a → C (Sig a)
discretize (Beh sig) = time >>= (pure . discr sig)
  where
    discr :: Sig (Fun a) → Time → Sig a
    discr (K x ::: xs) _ = x ::: withTime (delay (discr (adv xs)))
    discr (Fun f ::: xs) t =
      let cur = unbox f t
          rest =
            withTime $
              delay
                ( case select xs sampleInterval of
                    Fst x _ → discr x
                    Snd beh' _ → discr (Fun f ::: beh')
                    Both x _ → discr x
                )
      in (cur ::: rest)
```

Now that we have implemented a function for enabling pull-sampling, we can update the widget creation functions, *mkButton*, *mkSlider*, etc. to discretize the given behaviours and use them as signal based widgets in Widget Rattus. As an example, let's look at the updated *mkButton* function as seen below. The function allocates a channel using the *chan* function like the previous signal-based version, but now it also discretizes the given behaviour. The resulting channel and signal are then passed to the *Button* data structure.

```
mkButton :: (Displayable a) ⇒ Beh a → C Button
mkButton t = do
    c ← chan
    t' ← discretize t
    return Button {btnContent = t', btnClick = c}
```

Since events are discrete by nature, they do not need to be discretized like behaviours. To ease the use of creating events based on produced values from channels, we created a *mkEv* function similar to *mkSig*. This function recursively converts a box delayed computation into an event stream

```
mkEv :: □ (◯ a) → Ev a
mkEv a =
  Ev (delay (adv (unbox a) ::: mkSig a))
```

An example use case for events in the context of widgets is listening to button press events. In our GUI library we include a *btnOnClickEv* function that given a *Button* widget returns an *Ev* () type for button clicks. This is done by waiting for the channel *btnClick b*, of a

button b, to produce a value using the *wait* function. We must box it before passing it into the *mkEv* function. The *btnOnClickEv* function can be seen below:

```
btnOnClickEv :: Button → Ev ()
btnOnClickEv b =
  let ch = btnClick b
   in mkEv (box (wait ch))
```

We continue this pattern of updating the widget library, so all functions that creates widgets now take behaviours and events, and discretizes them into signals used by Widget Rattus.

```
updatedCounter :: C VStack
updatedCounter = do
    btn ← mkButton $ mkConstText "Increment"
    let clicks = btnOnClickEv btn
    let counts = scan (box (λn _ → n + 1 :: Int)) 0 clicks

    lbl ← mkLabel $ stepper 0 counts
    mkConstVStack (lbl :⋆ btn)

main :: IO()
main = runApplication updatedCounter
```

Figure 7: Counter GUI implementation using Events and Behaviours

The previously mentioned *counter* example seen in Figure 2 which used signals, can now be updated to use events and behaviours as seen in Figure 7. The new implementation is fairly similar to the previous implementation with some minor changes. Firstly, the *clicks* variable is now of type $Ev\ ()$ instead of $\bigcirc(Sig\ ())$ meaning that it is also now delayed by default. This means that the *scanAwait* function is replaced with a similar *scan* function that requires an event instead of a delayed signal. Secondly, the label created uses a stepped version of the *counts* variable to convert it from an $Ev\ Int$ type to $Beh\ Int$ type. Lastly, we use a new utility method *mkConstText* which simply creates a behaviour with constant text.

Due to the discrete nature of the *counter* example, updating the example to use behaviours and events does not affect the implementation that much nor make it more efficient. In the following section we will give an example of a continuous use case.

4 Case Study: A Simple Timer GUI

In this section we will look at a more interesting example that uses continuous data. It includes a simple timer, where a user can reset the timer using a button. We'll then expand the example with more complex features, such as, stopping the timer when it reaches a user-defined maximum value. The timer GUIs are shown in Figure 8

The core idea of this example is a resettable timer, which counts up until a user presses the reset button. This will produce a reset event, which captures the time of the press. The time is stored and later used to compute the elapsed time by comparing it to the current time.

(a) Simple Timer GUI (b) Improved Timer GUI

Figure 8: Example Timer GUIs

```
window :: C VStack
window = do
  resetBtn ← mkButton $ mkConstText "Reset timer"
  let resetTrigger = btnOnClickEv resetBtn

  startTime ← time
  let resetTime = Event.stepper startTime
        (Event.trigger (box (λ_ t → t)) resetTrigger timeBehaviour)

  let timer = zipWith (box diffTime) timeBehaviour resetTime

  text ← mkLabel (Behaviour.map
        (box (λt → "Current: " <> toText (floor $ toRational t))) timer)

  mkConstVStack $ text :* resetBtn

main :: IO()
main = runApplication' window
```

Figure 9: The implementation of our simple timer example.

To sample the time at which the reset event occurs, we use the *trigger* function[2]. An example of this can be seen below, where *trigger* samples the value of the *timeBehaviour*, when the *eventTrigger* event ticks.

```
Event.trigger (box (λ_ t → t)) resetTrigger timeBehaviour
```

We want a behaviour which tracks the most recent reset time. This can be done using the *stepper* function, given an initial start time and our sampled times using the above code.

With this behaviour in place, we can compute the elapsed time since the last reset by applying the *diffTime* function to the current time and the last reset time. We can achieve this by using the *zipWith* function over the two behaviours, resulting in a new behaviour that represents the timers value in seconds. The implementation of the example is given in Figure 9.

[2]This and following examples, use a revised version of the trigger function: $trigger :: (Stable\ b) \Rightarrow \square(a \rightarrow b \rightarrow c) \rightarrow Ev\ a \rightarrow Beh\ b \rightarrow Ev\ c$. This will be explained in detail in section 5.

4.1 Improving the Timer Example

Now we want to introduce a timer example with more features. This timer should stop counting when it reaches a user-defined maximum value. The timer should start again, when the maximum value is changed or the timer is reset by the user.

The timer example shown in Figure 10 assumes that we have an implementation of a $stopWith$ function, for stopping a behaviour and choosing the value after stopping it. How we created this function is described in section 5.1.

Just like in the simple timer example, we have a reset button with a corresponding event reference: $resetTrigger$. In addition to this, we create a slider which changes a maximum value, providing a $maxBeh$ behaviour and $maxChangeEv$ event. This behaviour and event are useful for retrieving the current value and also listening for changes in the value.

To combine $resetTrigger$ and $maxChangeEv$ into one event, that determines when a reset should occur, we use the $interleave$ function. Whenever the combined event ticks, we reset our timer and change the maximum value to the selected value.

To do this we use the $trigger$ function like earlier, but with a new $timeWithMax$ behaviour. We need to be able to access both the current time and maximum value, which we can get from the $timeBehaviour$ and $maxBeh$ behaviours respectively. We use $zipWith$ to zip the values of the two behaviours, into one behaviour with a tuple of ($time$, max).

Using the $trigger$ function on the interleaved events and the $timeWithMax$ behaviour, we sample a tuple of type ($time$, max) every time the interleaved events ticks. Using these values, we can now produce a new behaviour, that should count up to the max value. We create a function $timeFrom$, which constructs a behaviour that computes the time passed since a start time. When the elapsed time has reached a maximum value it stops using the $stopWith$ function. It uses a function $intToNominal$ to convert an Int to a $NominalDiffTime$. The function can be seen below:

```
timeFrom :: Int → Time → Beh NominalDiffTime
timeFrom max startTime =
  stopWith
    (box (λt → if t >= intToNominal max then Just' (intToNominal max) else Nothing'))
    (map (box ('diffTime` startTime)) timeBehaviour)
```

We can also use the $timeFrom$ function to create our initial behaviour that counts up when opening the application. To switch between this initial behaviour and subsequent behaviours, triggered either by the reset button or changing the max slider, we use the $switchR$ function. The full example can be seen in Figure 10.

```
nominalToInt :: NominalDiffTime → Int
nominalToInt x = floor $ toRational x

intToNominal :: Int → NominalDiffTime
intToNominal x = fromInteger (toInteger x)

timeFrom :: Int → Time → Beh NominalDiffTime
timeFrom max startTime =
  stopWith
    (box (λt → if t >= intToNominal max then Just' (intToNominal max) else Nothing'))
    (map (box (`diffTime` startTime)) timeBehaviour)


window :: C VStack
window = do
  let initialMax = 5

  resetBtn ← mkButton $ mkConstText "Reset timer"
  let resetTrigger = btnOnClickEv resetBtn

  maxSlider ← mkSlider initialMax (constK 1) (constK 100)
  let maxBeh = sldCurr maxSlider
  let maxChangeEv =
    map (box (Prelude.const ())) $ sliderOnChange maxSlider

  startTime ← time
  let timeWithMax = zipWith (box (:*)) timeBehaviour maxBeh
  let timer =
    trigger (box (λ_ (t :* max) _ → timeFrom max t))
        (interleave (box (λ_ _ → ())) resetTrigger maxChangeEv)
        timeWithMax

  let timer' = switchR (timeFrom initialMax startTime) timer

  text ← mkLabel (map (box (λt → "Current: " <> toText (nominalToInt t))) timer')
  maxText ← mkLabel (map (box (λmax → "Max: " <> toText max)) maxBeh)
  mkConstVStack $ maxSlider :* maxText :* text :* resetBtn

main :: IO()
main = runApplication window
```

Figure 10: The implementation of our advanced timer example.

Utilizing both push- and pull-functionality for the timer example, over only push-functionality has the benefit of not having to propagate a value every second to update the timer, instead we can pull the time when needed. We can still utilize the push-functionality by pushing events from buttons and sliders to trigger an update with minimal latency.

5 Refining Push-Pull in Widget Rattus

The current implementation provides all the core features of push-pull based functional reactive programming. In addition to the core features, there are four possible ideas which we refine our current implementation with. In the following section we will show how we have implemented the following ideas and reflect on their applicability:

- Allow stopping of behaviours
- Filtering of events
- Implementing integral and derivative functions on behaviours
- Demonstrating a possible way of avoid the use of the C monad
- Optimizing the combinator libraries using Haskell compiler

5.1 Stopping a Behaviour

There are many cases where it would be useful to stop a behaviour. In section 4.1 we want a timer to stop updating, when it reaches a maximum value.

In Widget Rattus they stop a signal using the *stop* and *jump* functions. Stopping works by checking the value x on a given signal $x ::: xs$ at every tick. If the predicate applied with x is true, then they return *const* x, which stops future ticks of the signal. Otherwise they return the signal without modification. The implementation from Widget Rattus can be seen below:

```
jump :: □ (a → Maybe' (Sig a)) → Sig a → Sig a
jump f (x ::: xs) = case unbox f x of
                        Just' xs' → xs'
                        Nothing' → x ::: delay (jump f (adv xs))


stop :: □ (a → Bool) → Sig a →  Sig a
stop p = jump (box (λ x → if unbox p x then Just' (const x) else Nothing'))
```

This implementation would not work in our push-pull based approach. The reason for this is, that the *jump* function only checks the predicate every time a new value is pushed on the signal. This would work for events, but not for behaviours where the pull-functionality is also used. Imagine the *timeBehaviour* from earlier, it is simply a identity function that will never tick. This means that there would not be pushed a new value on it. The predicate would only check on the initial value of the behaviour, and then never check again.

A possible solution could be to replace the Fun type when a predicate is true, and thereby mimic the semantics of *jump* with only the pull-functionality. Our first attempt of adding the stop functionality, was to add a $Switch$ constructor to the Fun type. This constructor would contain a function outputting either a value of type a or a new behaviour to switch to, depending on the time.

```
data Fun a =
    K a |
    Fun ( □ (Time → a)) |
    Switch ( □ (Time → Either' a (Sig (Fun a))))
```

As seen in the implementation above, we use the type $Sig(Fun\ a)$ instead of $Beh\ a$ to avoid circular dependency. We also used the $Either'$ type to reflect that the $Switch$ statement could either be a value a or a new signal of type $Sig(Fun\ a)$. The $Either'$ type is a strict version of the $Either$ type provided by Haskell.

Below is a simplified example of the $Switch$ constructor. In this example the behaviour counts to five and then switches over to a constant behaviour when it reaches five.

```
countToFive :: Time → Either' a (Sig (Fun a))
countToFive t = if t > 5 then Right' (K 5) else Left' t

countToFiveBeh = Beh (Switch (□ countToFive ::: never)
```

Due to the new $Switch$ case, we ran into some problems in the apply function. The purpose of *apply* is to generalize time functions, by returning a function of the type $Time \rightarrow a$. Implementing *apply* for both $K\ a$ and $Fun\ \square(Time \rightarrow a)$ is pretty straightforward. However, introducing the $Switch\ (Time \rightarrow Either'\ a\ (Sig\ (Fun\ a)))$ constructor complicates this, since the *apply* function might end up in unbounded recursion unless we restrict it. An example of this problem would be if the Fun type returned a $Switch$ case, which would then be recursively evaluated, resulting in a new $Switch$ case. This could continue for an unknown amount of time, potentially resulting in a stack overflow. A possible fix would be to run the recursive cases of *apply* in a delayed context to stop the evaluation, however this would eliminate the purpose of *apply*. Our attempt of this idea can be seen below.

```
-- Would not compile, due to recursive use of apply.
apply :: Fun a → Time → a
apply (K a) =  Prelude.const a
apply (Fun f) = unbox f
apply (Switch f) = λt →
    case unbox f t of
        Left' a → a
        Right' (a ::: as) → apply a t
```

Due to the increasing levels of complexity and the problems mentioned previously, we decided to focus on a more simple approach. Instead of having a $Switch$ constructor in the Fun type, we instead modify the Fun constructor to have type $\square(Time \rightarrow (a :* Bool))$ as seen below. With this boolean embedded in the result, we can check if the behaviour is supposed to be stopped, and thus return a constant value instead of the function.

```
data Fun a where
  K :: !a → Fun a
  Fun :: !( □ (Time → (a :* Bool))) → Fun a
```

With this new Fun type in place, we can implement the *stop* function. The stop function essentially changes the existing function to include the predicate logic, which handles the stop boolean. Since we sample at an interval, we can not ensure to stop at the exact correct value we intend to. The $stopWith$ helper function is similar to the *stop* function, and is created to counter this problem. It uses a $Maybe'$ instead of a $Bool$ in the predicate, which allows us to modify the final value. This is useful for rounding down the emitted value or similar purposes. The *stop* and $stopWith$ functions can be seen below. The implementation details have been omitted from $stopWith$, since it is almost identical to *stop*.

```
stop :: □ (a → Bool) → Beh a → Beh a
stop p (Beh b) = Beh (run b)
  where
    run (K x ::: xs) =
        K x ::: if unbox p x then never else delay (run (adv xs))
    run (Fun f ::: xs) =
        Fun (box (λt →
            let (a :* b) = unbox f t
            in (a :* (unbox p a || b)))) ::: delay (run (adv xs))

stopWith :: □ (a → Maybe' a) → Beh a → Beh a
...
```

Going back to the previous *countToFiveBeh* example, it would look like the following with the *stopWith* function and the *timeBehaviour* behaviour:

```
countToFive :: a → Maybe' a
countToFive t = if t > 5 then Just' 5 else Nothing'

countToFiveBeh = stopWith (box countToFive) timeBehaviour
```

Compared to the previous version of *countToFiveBeh*, this version adds stopping functionality to an already defined behaviour. While the *Switch* example had the stopping functionality as part of the behaviour.

Now that the predicate logic is included in the *Fun* constructor, we need a way to actually check the predicate at runtime. To solve this we added a check in the *discretize* function which can be seen below. We simply check if the behaviour is supposed to stop according to the boolean logic. If this is the case, we handle it by stopping pull-sampling and wait for the behaviour to tick. If the behaviour is not supposed to stop, we handle it like we normally would in *discretize*.

```
discretize :: Beh a → C (Sig a)
discretize (Beh sig) = time >>= (pure . discr sig)
  where
    discr :: Sig (Fun a) → Time → Sig a
    discr (K x ::: xs) _ = x ::: withTime (delay (discr (adv xs)))
    discr (Fun f ::: xs) t =
      let (cur :* b) = unbox f t
          rest =
            if b
              then withTime $ delay (discr (adv xs))
              else
                withTime $
                  delay
                    ( case select xs sampleInterval of
                        Fst x _ → discr x
                        Snd beh' _ → discr (Fun f ::: beh')
                        Both x _ → discr x
                    )
      in (cur ::: rest)
```

This refinement improves behaviours by adding stop functionality, which is a necessity for implementing the previously mentioned timer example. This comes with the cost of modifying the resulting Fun type to include a boolean value.

Since we can only stop a behaviour through our pull-sampling function *discretize*, this limits the usability of behaviours in external contexts. If a programmer wants to implement their own pull-sampling method, they must also support the stop functionality.

Since this stop functionality is a necessity of our examples, we have adopted this functionality in our final implementation, however in a different variation as mentioned in section 5.3.

5.2 Filtering of Events

Consider the scenario, where you listen for keyboard inputs, but you are only interested in keyboard inputs that produce numbers. In this scenario, a way to filter events based on a predicate would be helpful. However in the current implementation of events, there is no straightforward way of filtering events. As previously observed by Bahr et. al. [20], the Sig type does not support a filter function such as $filter :: \square(a \to Bool) \to Sig\ a \to Sig\ a$, since a signal has to produce a value of type a at every tick. This problem extends to the Ev type, since it is based on the Sig type. A proposed solution is to use the $Maybe'$ constructor to return $Ev\ (Maybe'\ a)$ when filtering as such:

```
filter :: □ (a → Bool) → Ev a → Ev (Maybe' a)
filter p = map (box (λx → if unbox p x then Just' x else Nothing'))
```

While this is useable it is also not optimal, since programs that work with events of type $Ev\ (Maybe'\ a)$ would now need to check for $Nothing'$ at every tick. To solve this we extend the definition of the Ev type to contain a dense ($EvDense$) and sparse ($EvSparse$) case as seen below. Given an event is using the $EvDense$ case, then we assume that all values are valid and we can re-use the existing Ev type and the current combinators without modifications. If however an event is using the $EvSparse$ case, then we can not assume that all values are valid, and have to check in the combinators if it is the $Nothing'$ case, since the type of $EvSparse$ will be $\bigcirc(Sig\ (Maybe'\ a))$.

```
data Ev a
  = EvDense  (○ (Sig a))
  | EvSparse (○ (Sig (Maybe' a)))
```

Given this change to the definition of Ev we set out to update the combinators for events to support both cases. Since both $EvDense$ and $EvSparse$ are cases of Ev, then it is only necessary to update the function bodies and not the function signatures. If a function previously returned an Ev, then it should now return either an $EvDense$ or $EvSparse$, depending on the function semantics. In most cases, functions return the same type as their input. However, filtering and triggering do not always produce a valid value, and therefore return sparse outputs. A list of updated outputs types can be seen in Table 1.

Function	Outputs
mkEv	*EvDense*
filterMap	*EvSparse*
filter	*EvSparse*
trigger	*EvSparse*
map	Same as input
scan	Same as input
interleave	Depends on input

Table 1: A list of functions that return events and the type of constructor they output.

When updating the combinator we need to handle both the dense and sparse cases. An example of this is the *map* function:

```
map :: □ (a → b) → Ev a → Ev b
map f (EvDense sig) = EvDense (Signal.mapAwait f sig)
map f (EvSparse sig) =
  EvSparse (Signal.mapAwait f' sig)
  where f' = box (fmap (unbox f))
```

The dense case in the *map* function stays the same, however, an extra sparse case is added. In this case, we have to map on the $Maybe'$ type as well.

In some cases we have to make a choice on how to handle $Nothing'$ cases. Some functions, such as, *stepper* requires a valid value to be given at every tick, since it returns a behaviour. A solution to this is to repeat the previous last valid value, if a non-valid value is produced. A valid value in this case would be all values from *EvDense* case and all $Just'$ values from the *EvSparse* case. This is a bit inefficient, since it would be preferred to only produce a new value when it is a new valid value, however as previously mentioned this is an unsolved problem. The following is a version of the *stepper* function that uses the proposed solution.

```
stepper :: (Stable a) ⇒ a → Ev a → Beh a
stepper initial ev =
  Beh (K initial ::: delay (adv (aux initial ev)))
  where
    aux :: (Stable a) ⇒ a → Ev a → ◯ (Sig (Fun a))
    aux _ (EvDense ev) = stepperDense ev
    aux initial (EvSparse ev) = stepperSparse initial ev

stepperDense :: (Stable a) ⇒ ◯ (Sig a) → ◯ (Sig (Fun a))
stepperDense ev =
    delay (let (x ::: xs) = adv ev in K x ::: delay (adv (stepperDense xs)))

stepperSparse :: (Stable a) ⇒ a → ◯ (Sig (Maybe' a)) → ◯ (Sig (Fun a))
stepperSparse initial ev =
  delay
    ( let (x ::: xs) = adv ev
        in case x of
            Just' x' →
              K x' ::: delay (adv (stepperSparse x' xs))
            Nothing' → K initial ::: delay (adv (stepperSparse initial xs))
    )
```

The introduction of two event data constructors increase the complexity of *interleave*, as seen below, since the function requires a case for all event pair combinations. The return type of *interleave f x y* depends on the input types x and y. If both inputs are $EvDense$, the result is $EvDense$. In all other cases the result is $EvSparse$.

If both events are dense, then we can do the same as the previous implementation of *interleave*. If both events are sparse, additional pattern matching is needed after the *select xs ys* call. The Fst and Snd branches of the select statement behave as before. However, if both events tick simultaneously ($Both$), we must consider their values:

1. If both values are $Just'$, they are combined using the provided operator into a new $Just'$ value.
2. If only one is $Just'$, that value becomes the result.
3. If both are $Nothing'$, the result is $Nothing'$.

Lastly, there are the mixed case, where one is dense and the other is sparse. These cases are handled by converting the dense case into a sparse case of $Just'$ values. This adds some overhead since the dense case has to be converted. However, we determine that the advantage of readability outweighs the disadvantage in the context of this project. For a more efficient implementation the last two cases could be implemented similar to the two sparse case with only checking the sparse event for $Nothing'$. The code can be seen below:

```
interleave :: □(a → a → a) → Ev a → Ev a → Ev a
interleave f (EvDense xs) (EvDense ys) = EvDense (Signal.interleave f xs ys)
interleave f (EvSparse xs) (EvSparse ys) = EvSparse (aux f xs ys)
  where
    aux f xs ys =
      delay
        ( case select xs ys of
          Fst (x ::: xs') ys' → (x ::: aux f xs' ys')
          Snd xs' (y ::: ys') → (y ::: aux f xs' ys')
          Both (Just' x ::: xs') (Just' y ::: ys') →
            Just' (unbox f x y) ::: aux f xs' ys'
          Both (Just' x ::: xs') (Nothing' ::: ys') → Just' x ::: aux f xs' ys'
          Both (Nothing' ::: xs') (Just' y ::: ys') → Just' y ::: aux f xs' ys'
          Both (_ ::: xs') (_ ::: ys') → Nothing' ::: aux f xs' ys'
      )
interleave f (EvSparse xs) (EvDense ys) =
    interleave f (EvSparse xs) (EvSparse (Signal.mapAwait (box Just') ys))
interleave f (EvDense xs) (EvSparse ys) =
    Event.interleave f (EvSparse(Signal.mapAwait (box Just') xs)) (EvSparse ys)
```

A function that can be improved by changing the type signature is the *trigger* function. Recall that the trigger function samples a behaviour based on the updates of an event. Previously it returned an $Ev\ (Maybe'\ a)$, however, now it is possible to simply return $Ev\ a$, since the returned event is sparse. Once again the dense case works similarly as the previous implementation of trigger. In the sparse case some extra pattern matching is needed, such that if either the Fst or $Both$ case is reached in the *select* statement, then we have to check if the current value of the event is $Nothing'$. If the value is $Nothing'$ then we produce a $Nothing'$ value, otherwise we produce a $Just'$ value with the given combination operator

applied on the current event and behaviour values. See below for the implementation.

```
trigger :: (Stable b) ⇒ □ (a → b → c) → Ev a → Beh b → Ev c
trigger f event behaviour = EvSparse (trig f event behaviour)
  where
    trig :: (Stable b) ⇒ □ (a → b → c) → Ev a → Beh b → ◯ (Sig (Maybe' c))
    trig f' (EvDense as) (Beh (b ::: bs)) =
      ...
    trig f' (EvSparse as) (Beh (b ::: bs)) =
      withTime $
        delay
          ( let choice = select as bs in
              (λt → case choice of
                  Fst (a' ::: as') bs' →
                    let rest = trig f' (EvSparse as') (Beh (b ::: bs'))
                     in fmap (λa'' → unbox f' a'' (apply b t)) a' ::: rest
                  Snd as' bs' → Nothing' ::: trig f' (EvSparse as') (Beh bs')
                  Both (a' ::: as') (b' ::: bs') →
                    let rest = trig f' (EvSparse as') (Beh (b' ::: bs'))
                     in fmap (λa'' → unbox f' a'' (apply b' t)) a' ::: rest
              )
            )
```

After updating all the combinators to use the dense and sparse versions of events, we can now implement a function to actually filter the events, which can be seen below. The filter function should always return a *EvSparse*, since we filter out values. If the values are already $Nothing'$ then $Nothing'$ is returned, while if the value x is valid and it fulfils the given predicate, then we return $Just'\ x$

```
filter :: □ (a → Bool) → Ev a → Ev a
filter f (EvDense ev) =
  EvSparse (aux f ev)
  where
    aux :: □ (a → Bool) → ◯ (Sig a) → ◯ (Sig (Maybe' a))
    aux f ev =
      ( delay
          ( let (x ::: xs) = adv ev
                rest = aux f xs
            in (if unbox f x then Just' x else Nothing') ::: rest
          )
      )
filter f (EvSparse ev) =
  EvSparse (aux f ev)
  where
    aux :: □ (a → Bool) → ◯ (Sig (Maybe' a)) → ◯ (Sig (Maybe' a))
    aux f ev = (
        delay
         ( let (x ::: xs) = adv ev
               rest = aux f xs
           in case x of
               Just' x' → (if unbox f x' then Just' x' else Nothing') ::: rest
               _ → Nothing' ::: rest
         )
      )
```

With filtering implemented it is now possible to implement the example from the beginning of the section. Given an event stream of keyboard inputs called *keyboardPresses*, then we use the *filter* function to filter the stream with the *isDigit* function as the predicate. The *isDigit* function is a built-in function in Haskell that returns *True* if the input is a number. Filtering the event is done as follows:

```
numberKeys = filter (box isDigit) keyboardPresses
```

The refinement of supporting filtering of events is generally useful as it reduces complexity for the programmers using the library, since they do not have to handle invalid values. The trade-off for this is that it can lead to code explosion in our combinators, due to the *Ev* type having both an *EvDense* and *EvSparse* constructors. One such case is in the *interleave* function, which went from one case to four cases with the introduction of these constructors.

We have chosen to adopt the filtering functionality in our final implementation, since it enables programmers to focus on the logic that is relevant to them and making our combinators handle invalid inputs.

5.3 Integral and Derivative

Previously Bahr and Møgelberg [19] have proposed *integral* and *derivative* combinators using signals. The implementation of *integral* works by using simple approximation by sampling the value of the underlying signal each time the sample channel produces a value. The area of the rectangle formed by the current value and the time passed since last sample is then added. This can be optimized when the value of the underlying signal is 0. If this is the case, the value of the integral will not change, and we simply wait for the underlying signal to tick again.

Similarly to the integral function, a derivative functions was also implemented to return the derivative of a given signal. The implementation of derivative works by sampling the value of the underlying signal and finding the difference over time from the previously sampled valued. If the computed derivative is 0, then no further sampling is performed until the underlying signal ticks. As soon as it does, we emulate that the sample has ticked, and provide an update of the derivative, taking time into account.

Based on the ideas by Bahr and Møgelberg a similar version can be implemented that supports both push- and pull-functionality using behaviours. Most parts of the implementation can stay the same due to behaviours being built on top of signals.

Initial support for pull-based functionality is illustrated in Figure 11, which shows the core idea behind our first implementation of the *integral* function. When a pull or push event occurs, we compute the difference between the current time and the last push.

To do this, we return a behaviour constructed with the *Fun* constructor. This allows the current value to be computed dynamically when a pull occurs, using the time difference and the latest available signal value.

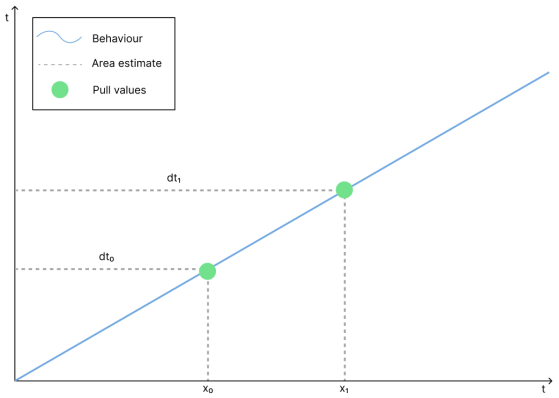

Figure 11: Illustration of our initial integral function on the *timeBehaviour*. It calculates the integral based on the time at a pull, with the time from the last push. The illustration contains two pulls, with the values x_0 and x_1 and the corresponding delta times dt_0 and dt_1. The integral at the second pull in this example would be $dt_1 \cdot x_1$, since it does not know the value or time of the previous pull.

Similar changes are made to the *derivative* function, which now calculates the rate of change since last push. Both functions handle the constant and time-varying cases slightly differently, which will be explained in depth later. The *integral* and *derivative* functions can be seen in Figure 12, and the helper functions they use can be found in Appendix B.

When using behaviours with our combinator library, in most cases the behaviour gets passed into the *discretize* function, which handles sampling. Therefore to avoid re-sampling we did not implement sampling as part of the integral and derivative functions as Bahr and Møgelberg did [19].

With the current versions of integral and derivative, we always calculate the value based on the last push, since there currently is no reference to when the previous pull was nor the value returned from the previous pull. This is generally a good thing, since storing all previous pull values in the hope that it will be used in the future is not ideal. However, in some cases, such as in the integral and derivate functions, using information from the last pull would result in calculating a more precise result.

To achieve this, our idea is to update the Fun constructor with an internal state, that will enable us to store the last pull value and timestamp. Since this state should persist into the future, it must be stable. By storing the last pull value and timestamp, we can calculate only the difference since the last pull, avoiding unnecessary re-computation.

Currently the function returned by the Fun constructor outputs a pair $(a,\ Bool)$. One way of including the state in the Fun constructor would be to simply add a state value to the result of the returned function, i.e. returning $(a,\ s,\ Bool)$.

Since stopping is handled by converting the current value into a K constructor, then the state will not be preserved after stopping the behaviour. This means that we could simply represent the state and boolean together using the $Maybe'$ type. Thereby returning $(a,\ Maybe'\ s)$.

```
integral :: Float → Beh Float → C (Beh Float)
integral cur (Beh s) = time >>= (λt → pure (Beh (int cur s t)))
  where
    int :: Float → Sig (Fun Float) → Time → Sig (Fun Float)
    int cur (x ::: xs) t =
      let curF =
            case x of
              K 0 → K cur
              x → Fun $ box (λt' → calcIntegral cur x t t' :* False)
          rest =
            withTime
              (delay (λt' → int (calcIntegral cur x t t') (adv xs) t'))
       in curF ::: rest

derivative :: Beh Float → C (Beh Float)
derivative (Beh (x ::: xs)) =
  time >>= (λt → pure (Beh (der (apply x t) (x ::: xs) t)))
  where
    der :: Float → Sig (Fun Float) → Time → Sig (Fun Float)
    der last (x ::: xs) t =
      let curF =
            case x of
              K a → calcDerivative (a - last) t (a == last)
              x → calcDerivative (apply x t - last) t False
          rest =
            withTime
              (delay (λt' → der (apply x t') (adv xs) t'))
       in (curF ::: rest)
```

Figure 12: Our initial implementation of integral and derivative.

This design preserves support for stopping behaviours, while allowing us to keep an internal state.

There result is then a *Fun* constructor that accepts an initial state and a boxed function that uses this state along with the current time to compute the output.

```
data Fun a where
  K :: !a → Fun a
  Fun :: (Stable s) ⇒ !s → !( □ (s → Time → (a :* Maybe' s))) → Fun a
```

This change to the *Fun* type requires all functions making use of the *Fun* type to be updated to correctly pass the state. This is particularly important in the *discretize* function which must pass forward the updated state every time *discretize* samples the behaviour.

```
discretize :: Beh a → C (Sig a)
discretize (Beh sig) = time >>= (pure . discr sig)
  where
    discr :: Sig (Fun a) → Time → Sig a
    discr (K x ::: xs) _ = x ::: (withTime $ delay (discr (adv xs)))
    discr (Fun s f ::: xs) t =
      let (cur :* s') = unbox f s t
          rest =
            case s' of
              Just' s'' →
                withTime $
                  delay
                    ( case select xs sampleInterval of
                        Fst x _ → discr x
                        Snd beh' _ → discr (Fun s'' f ::: beh')
                        Both x _ → discr x
                    )
              Nothing' → withTime $ delay (discr (adv xs))
      in (cur ::: rest)
```

In the updated version of *discretize* seen above, the $Nothing'$ case represents the stopping of the behaviour and was previously the $True$ case in the previous implementation.

Having updated the *Fun* type and *discretize* function to support internal states, it is now possible to optimize the *integral* and *derivative* functions making use of state. To illustrate how the *integral* function would approximate the integral, see Figure 13.

The updated integral function, seen in Figure 14, now utilizes the state in the pull case, to only estimate the new area since last pull. The implementation distinguishes between the two *Fun* type cases:

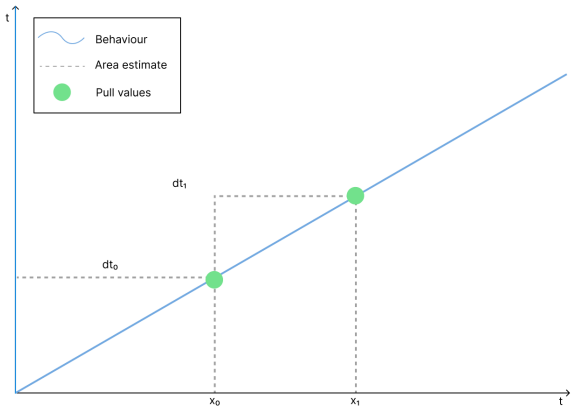

Figure 13: Updated integral function, that makes use of the new state in the pull case. The illustrations contains two pulls, with the values x_0 and x_1 and the corresponding delta times dt_0 and dt_1. The integral at the second pull in this example would be $dt_0 \cdot x_0 + dt_1 \cdot x_1$.

- **Constant K case:** On every pull the integral is calculated based on the time passed since last push and the constant value. If the constant value is 0, then we return a constant K case with the value provided by the function and wait for the behaviour to tick until we calculate the integral of the newly-pushed value.
- **Time-varying Fun case:** On each pull, the Fun constructor maintains a state of the form (v, t, s), representing the last integral value v, the last pull timestamp t, and the state of the underlying Fun type s. Using this, we can calculate the integral of the behaviour since the last pull, denoted as v'. The state is then updated to (v', t', s'), which corresponds to the new integral value, the new timestamp and the new underlying state. This approach enables us to calculate the integral based on the most recent pull rather than the the most recent push, improving accuracy for pull-based evaluation.

In the updated derivative function, seen in Figure 14, we now utilize the state to estimate the rate of change since the last pull, instead of the last push. The implementation, which can be seen below, distinguishes between two cases:

- **Constant K case:** On the first pull we return the difference from the previous value to the new constant value. Since the value is now constant, all subsequent pulls would have a rate of change of 0. To optimize this, we stop pull sampling when the rate of change is 0, until a new value is pushed. This is achieved by setting the state to be $Nothing'$.
- **Time-varying Fun case:** Similarly to the Fun case in the *integral* function, the *derivative* function utilizes state to calculate the rate of change between each pull.

After implementing the *integral* and *derivative* functions with behaviours, we create two tests for assessing the correctness of the functions. Both tests can be found in Appendix A. From these tests we can conclude that there is some variability in the results. Our implementations use simple approximation and also only sample every 20 millisecond. Lowering the sample interval would improve the result, however you will get diminishing returns the faster you sample, while reducing performance due to increasing renders of the GUI.

```
integral :: Float → Beh Float → C (Beh Float)
integral cur (Beh s) = time >>= (λt → pure (Beh (int cur s t)))
  where
    int :: Float → Sig (Fun Float) → Time → Sig (Fun Float)
    int cur (K a ::: xs) t =
      let rest = withTime $ delay
                 (λt' → int (calcIntegral cur a t t') (adv xs) t')
          curF =
            case a of
              0 → K cur
              a → Fun () $ box (λs t' → calcIntegral cur a t t' :* Just' s)
       in curF ::: rest
    int cur (Fun s f ::: xs) t =
      let rest = withTime $ delay ( λt' →
                       let (v :* _) = unbox f s t'
                       in int (calcIntegral cur v t t') (adv xs) t'
                   )
          curF =
            Fun (cur :* t :* s)
                (box(λ(last :* t :* s) t' →
                    let (v :* s') = unbox f s t'
                    in case s' of
                        Just' s'' →
                            let v' = calcIntegral last v t t'
                            in v' :* Just' (v' :* t' :* s'')
                        _ → calcIntegral v v t t' :* Nothing'
                  ))
       in curF ::: rest

derivative :: Beh Float → C (Beh Float)
derivative (Beh (x ::: xs)) = time >>= (λt → pure (Beh (der (apply x t) (x ::: xs) t)))
  where
    der :: Float → Sig (Fun Float) → Time → Sig (Fun Float)
    der last (Fun s f ::: xs) t =
      let rest = withTime $ delay
                       (λt' → let (v :* _) = unbox f s t' in der v (adv xs) t')
          curF =
            Fun (last :* t :* s) $
              box(λ(last :* t :* s) t' →
                    let (v :* s') = unbox f s t
                     in calcDerivative v last t t' :* fmap (λs'' → v :* t' :* s'') s'
                )
       in curF ::: rest
    der last (K x ::: xs) t =
      let rest = withTime (delay (der x (adv xs)))
          curF = Fun (last :* t) $ box
                (λ(last :* t) t' →
                    if x /= last
                    then calcDerivative x last t t' :* Just' (x :* t')
                    else 0 :* Nothing'
                )
       in curF ::: rest
```

Figure 14: The final version of our integral and derivative function, utilizing push- and pull-functionality to calculate a more precise result.

The implementations of the *integral* and *derivative* functions demonstrates how the use of an internal state can optimize such functions by utilizing the push- and pull-functionality of behaviours. However, similarly to the refinement of the stopping functionality in section 5.1, this limits the usability of behaviours in external contexts, since state needs to be handled as part of the pull-sampling. Since our implementation of state can replace the previously embedded boolean, while still maintaining the stopping functionality, we have chosen to adopt this in the final implementation. While state has few use cases in our functions, we include it to highlight the possible optimizations of utilizing a push-pull based model.

5.4 Avoiding the C Monad

One of the challenges we have run into is the use of the C monad, which gives us access to the current time. While it is definitely useful in some cases, it can also make things more complicated, especially when we try to combine different behaviours or reason about them more abstractly. So, we started exploring whether we could avoid using C altogether by restructuring our definitions of behaviours and events.

The idea was to make time an explicit argument, i.e. $Time \to Sig\ (Fun\ a)$, This would enable writing simpler types like $elapsedTime :: Beh\ Time$ instead of $elapsedTime :: C\ (Beh\ Time)$. This would make it easier to define combinators without having to work in a monadic context.

To try this out, we introduced a simplified behaviour type, and an updated behaviour type, where the initial time is provided:

```
type SBeh a = Sig (Fun a)
newtype Beh' a = Beh' (Time → SBeh a)
```

Only the initial time needs to be provided, since we previously defined the $withTime$ function, which can be used to provide the current time in a delayed context.

This way, a behaviour is just a function from time to an $SBeh$, and we hoped this would let us avoid C entirely. Using the newly defined behaviour type we can update the $elapsedTime$ function to return a behaviour which takes a start time as an argument. This is in contrast to the old implementation which uses the C monad to get the start time.

```
-- Old implementation
elapsedTime :: C (Beh NominalDiffTime)
elapsedTime = do
  startTime ← time
  return (Beh (
    Fun () (box (λs currentTime → diffTime currentTime startTime :* Just' s))
    ::: never))

-- New implementation
elapsedTime :: Beh NominalDiffTime
elapsedTime = Beh (
    λstartTime →
    Fun () (box (λs currentTime → diffTime currentTime startTime :* Just' s))
    ::: never)
```

This proves that we can create behaviours that require a time without using the C monad. Updating the functions to avoid the the use of C monad can be split up into two groups:

- **Functions returning a behaviour**: Functions such as *map*, *elapsedTime* or similar return a new behaviour based on the arguments provided. Since these functions return a new behaviour, then the current time is given from the time parameter when defining the behaviour. An example of this can be seen below with the *map* function which returns a new behaviour with time passed on to the original behaviour.

```
map :: □ (a → b) → Beh' a → Beh' b
map f (Beh' as) =
 Beh' (λt → Signal.map (box (λa → mapF f a)) (as t)
```

- **Functions evaluating a behaviour**: Functions such as *discretize*, *trigger* or similar evaluate a behaviour to produce another type. To do so, they require the current time to evaluate the behaviour into a $SBeh$ type. As a result, these functions generally require the C monad. This is fine for the *discretize* function since it already uses the C monad. However, the trigger as previously defined, did not require the use of C before. This was possible under the old definition of Beh, since we could unpack the signal without requiring a time parameter, unlike with *Beh'* where we require a time parameter. This means we have to get the current time in the *trigger* function. There are generally two ways of getting the current time. The first way is to use the C monad to get the time. Alternatively, one could get the time in a delayed context using the *withTime* function, however this would require to make *Beh'* stable, which is not possible with the current definition. Therefore the only possible way of getting the time is with the C monad. The updated *trigger* function is defined as follows:

```
trigger :: (Stable b) ⇒ □ (a → b → c) → Ev a → Beh' b → C (Ev c)
trigger f e (Beh' b) = do
  t ← time
  let b' = b t
  return $ EvSparse (trig f e b')
  where
    trig :: (Stable b) ⇒ □ (a → b → c) → Ev a → SBeh b → ◯ (Sig (Maybe' c))
    trig f' ev (b ::: bs) =
      ...
```

A problem we encounter when avoiding the C monad is that we still require $SBeh\ a$ and $Beh'\ a$ to be continuous. The old definition of $Beh\ a$ was continuous due to the underlying signal in the type definition. Specifically, since $Sig\ (Fun\ a)$ is continuous, then $Beh\ a$ would also be continuous.

Proving $Beh'\ a$ is continuous is a bit more challenging, due to the use of the function $Time \rightarrow SBeh\ a$ in the type definition. Functions generally are not continuous, since they are harder to argue for when an update occurs.

A possible workaround would be to unsafely retrieve the current time t and apply it to the behaviour $Beh'\ a$. This yields a value of type $SBeh\ a$, which is continuous and can therefore be updated. However, using this approach raises the concern regarding the correctness of this simple behaviour $SBeh$. Since the behaviour is being evaluated using the current time

in an uncontrolled manner, we can not guarantee that the resulting $SBeh$ truly corresponds to the intended time context.

In other words, even though the resulting value is continuous and usable, it might not be the correct value. So even though this is a possible workaround, we believe that more work should be done to validate whether avoiding the C monad gives a greater benefit than the drawbacks presented. Due to this, we have not chosen to adopt the Beh' type in our final implementation.

5.5 Optimizing With the Haskell Compiler

One of the advantages of using Haskell for this implementation is that it supports *Rewrite rules* [24]. This functionality gives fine-tuned control over how expressions are optimized by the compiler, which can improve performance by avoiding unnecessary intermediate structures.

For example, consider an optimization for the *map* function: mapping a behaviour first with a function f and then with a function g is equivalent to mapping it once with the composition $f \circ g$. With rewrite rules, we can detect such patterns and replace them with a single mapping operation.

Defining rewrite rules is done using the {#- RULES #-} pragma and the map example explained above is defined as follows:

```
{-# NOINLINE [1] map #-}

{-# RULES
"map/map" forall f g xs.
  map f (map g xs) = map (box (unbox f . unbox g)) xs
  #-}
```

The Haskell compiler optimizes calls to small functions by replacing the call with the function's body. This concept in generally referred to as "inlining".

In the example above, we also use the {#- NOINLINE #-} pragma to prevent the *map* function from being inlined too early for the rules defined to be fired. If we omitted this pragma, then there would be a possibility that the optimizer would inline the function before applying the rule.

Widget Rattus already defined some rewrite rules for the signal combinator library, such as optimizing the *map* and *const* functions. Based on this, we have defined similar rewrite rules for behaviours and events, but have additionally defined rules for the filter functionality as well. All rewrite rules can be seen in Appendix C

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 Elliot and Hudak [1]. Numerous FRP systems have since emerged, which can be split up into two evaluation approaches: push- and pull-based.

Previous related works have primarily focused on push-based FRP approaches. However, to get the advantages of both approaches, a push-pull approach was proposed by Elliott [5]. To our knowledge there has not recently been any notable implementations of Elliot's push-pull model. Elliott highlights *Lula-FRP*, 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-FRP* [13]. *Reflex* 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 recently attracted attention due to its potential to address the issues in traditional FRP, such as space-time leaks and causality [15, 6, 11, 17, 9, 7]. 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 well-defined implementation strategies that eliminate space-time leaks [10]. *Widget Rattus* [21] is based on such an FRP language, called *Async Rattus*. *Async Rattus* implements calculus based on *Async RaTT*, which presents the later ($\bigcirc$) and box ($\Box$), as well as *stable* types and signals. All *Async RaTT* programs are casual, productive and do not have space leaks.

Widget Rattus differentiates from *Async Rattus* by focusing on GUI programming using FRP and incorporates two notable extensions: 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 [17, 6, 2, 4, 3, 17]. One such language is the *Elm* language [8], 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 and Future Work

In this thesis, we present an implementation of Elliot's proposed push-pull based FRP approach [5], combining the efficiency of push-based evaluation, with the applicability of pull-based semantics. This push-pull model is implemented in the Widget Rattus language [21]. We define new event and behaviour abstractions, along with combinators that allow programmers to compose applications utilizing these benefits.

Existing GUI widgets from Widget Rattus were extended to support events and behaviours, using a discretization strategy to convert behaviours into signals. Through GUI examples, we demonstrate the benefits of this hybrid model for handling both continuous and discrete time-varying data. We demonstrate a set of refinements which can be used for improving implementations of push-pull models in FRP.

This work shows the possibilities of a push-pull based model in a practical GUI framework. However there still remains areas for future work, including:

- **Avoiding the C monad**: As previously mentioned in section 5.4, we explored the possibility of avoiding the C monad. However, due to the current definition of $Continuous$ it would be beneficial to research this refinement more.
- **Further research on utilizing the internal state of behaviours**: In section 5.3, we introduced an internal state to the time function. We believe that there could potentially be more applications for optimizing behaviours by incorporating and utilizing an internal state.
- **Improving pull-support in GUI updates**: Currently, using behaviours in Widget Rattus' GUI library requires external discretization. Ideally, widgets would support native continuous rendering. Furthermore, nested widgets could be updated directly, rather than recreating the full widget tree at each render, to improve runtime efficiency.
- **Make box, delay and adv implicit**: At the moment, you have to explicitly *box*, *delay* and *adv* which can result in cumbersome and hard to read code. A possible solution would be to use bidirectional type checking [16] to extend the compiler to insert these functions.

References

[1] Conal Elliott and Paul Hudak. "Functional reactive animation". In: *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming, ICFP '97*. New York, NY, USA: ACM, 1997, pp. 263–273. ISBN: 0-89791-918-1.

[2] M. Sage. "FranTk - a declarative GUI language for Haskell". In: *Proceedings of the International Conference on Functional Programming (ICFP)*. 2000.

[3] A. Courtney and C. Elliott. "Genuinely functional user interfaces". In: *Proceedings of the Haskell Workshop*. 2001.

[4] H. Nilsson, A. Courtney, and J. Peterson. "Functional reactive programming, continued". In: *Proceedings of the Workshop on Haskell*. 2002.

[5] Conal M. Elliott. "Push-pull functional reactive programming". In: *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*. Haskell '09. New York, NY, USA: Association for Computing Machinery, 2009, pp. 25–36. DOI: 10.1145/1596638.1596643.

[6] Heinrich Apfelmus. *Reactive banana*. 2011. URL: https://hackage.haskell.org/package/reactive-banana.

[7] Alan Jeffrey. "LTL Types FRP: Linear-Time Temporal Logic Propositions as Types, Proofs as Functional Reactive Programs". In: *Proceedings of the Sixth Workshop on Programming Languages Meets Program Verification*. Ed. by Koen Claessen and Nikhil Swamy. PLPV 2012. Philadelphia: ACM, 2012, pp. 49–60. ISBN: 978-1-4503-1125-0.

[8] E. Czaplicki and S. Chong. "Asynchronous functional reactive programming for GUIs". In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2013.

[9] Wolfgang Jeltsch. "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*. PLPV '13. New York: ACM, 2013, pp. 69–78. ISBN: 978-1-4503-1860-0.

[10] Neel R. Krishnaswami. "Higher-Order Functional Reactive Programming Without Space-Time Leaks". In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13. Boston: ACM, 2013, pp. 221–232. ISBN: 978-1-4503-2326-0.

[11] Andrew Cave et al. "Fair Reactive Programming". In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. San Diego: ACM, 2014, pp. 361–372. ISBN: 978-1-4503-2544-8.

[12] Eugen Kiss. *7GUIs: A GUI Programming Benchmark*. https://eugenkiss.github.io/7guis/. Accessed: 2025-05-12. 2014.

[13] Obsidian Systems. *Reflex: Functional Reactive Programming for Haskell*. https://reflex-frp.org. Accessed: 2024-12-07. 2016.

[14] F. Vallarino. *Monomer*. Accessed: 2025-04-15. 2018. URL: https://hackage.haskell.org/package/monomer.

[15] Patrick Bahr, Christian Uldal Graulund, and Rasmus Ejlers Møgelberg. "Diamonds are not forever: liveness in reactive programming with guarded recursion". In: *Proceedings of the ACM on Programming Languages* 5.POPL (2021), pp. 1–28.

[16] Jana Dunfield and Neel Krishnaswami. "Bidirectional Typing". In: *ACM Comput. Surv.* 54.5 (May 2021). ISSN: 0360-0300. DOI: 10.1145/3450952. URL: https://doi.org/10.1145/3450952.

[17] Christian Uldal Graulund, Dmitri Szamozvancev, and Neel Krishnaswami. "Adjoint reactive GUI programming". In: *Foundations of Software Science and Computation Structures (FoSSaCS)*. 2021, pp. 289–309.

[18] The Haskell Time Library Maintainers. *Data.Time.Clock - Time library for Haskell*. Accessed: 2025-05-06. 2021. URL: https://hackage.haskell.org/package/time-1.14/docs/Data-Time-Clock.html#g:3.

[19] Patrick Bahr and Rasmus Ejlers Møgelberg. "Asynchronous Modal FRP". In: *Proc. ACM Program. Lang.* 7.ICFP (Aug. 2023). DOI: 10.1145/3607847. URL: https://doi.org/10.1145/3607847.

[20] Patrick Bahr, Emil Houlborg, and Gregers Thomas Skat Rørdam. "Asynchronous Reactive Programming with Modal Types in Haskell". In: *Practical Aspects of Declarative Languages: 26th International Symposium, PADL 2024, London, UK, January 15–16, 2024, Proceedings*. Vol. 13825. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, 2024, pp. 18–36. DOI: 10.1007/978-3-031-52038-9_2.

[21] Jean-Claude Disch, Asger Heegaard, and Patrick Bahr. "Functional Reactive GUI Programming with Modal Types". Submitted for peer review November 2024. Copenhagen, Denmark, 2024.

[22] The GHC Team. *Strictness: The Strict Language Extension*. Accessed: 2024-12-12. Glasgow Haskell Compiler (GHC), 2024. URL: https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/strict.html.

[23] The GHC Team. *Prelude - scanl*. Accessed: 2025-05-06. 2024. URL: https://hackage.haskell.org/package/base-4.21.0.0/docs/Prelude.html#v:scanl.

[24] The GHC Team. *GHC User's Guide: Rewrite Rules*. Accessed: 2025-05-16. Glasgow Haskell Compiler. 2025. URL: https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/rewrite_rules.html.

[25] *WidgetRattus*. [Online; accessed 24. May 2025]. May 2025. URL: https://hackage.haskell.org/package/WidgetRattus.

Appendix

A Integral and derivative test

Content of *IntegralTests.hs*:

```
module Main where

import WidgetRattus
import WidgetRattus.Widgets
import WidgetRattus.Behaviour

import Prelude hiding (const, filter, getLine, map, null, putStrLn, zip, zipWith)

integralTests :: C VStack
integralTests = do
  time ← elapsedTime
  time' ← integral 0 (WidgetRattus.Behaviour.map (box realToFrac) time)
  derivativeTime ← derivative time'

  let shouldBe =
    WidgetRattus.Behaviour.map
        (box (λt → fromRational ((toRational t)^2)/2)) time
  originalLbl ←
    mkLabel (WidgetRattus.Behaviour.map
        (box (λt → "Input (Time): " <> toText t)) time)
  resultLbl ←
    mkLabel (WidgetRattus.Behaviour.map
        (box (λt → "Result (Integral): " <> toText t)) time')
  shouldLbl ←
    mkLabel (WidgetRattus.Behaviour.map
        (box (λt → "Should be: " <> toText t)) shouldBe)

  -- Re-derivative
  derivativeLbl ←
    mkLabel (WidgetRattus.Behaviour.map
        (box (λt → "Integral → Derivative: " <> toText t)) derivativeTime)
  -- UI
  mkConstVStack $ originalLbl :* resultLbl :* shouldLbl :* derivativeLbl

main :: IO()
main = runApplication integralTests
```

Content of *DerivativeTests.hs*:

```
module Main where

import WidgetRattus
import WidgetRattus.Widgets
import WidgetRattus.Behaviour
import Prelude hiding (const, filter, getLine, map, null, putStrLn, zip, zipWith)

derivativeTests :: C VStack
derivativeTests = do
  time ← elapsedTime
  time' ← derivative (WidgetRattus.Behaviour.map (box realToFrac) time)
```

```
  let shouldBe = constK 1
  originalLbl ← mkLabel time
  resultLbl ← mkLabel (WidgetRattus.Behaviour.map (box toText) time')

  shouldLbl ← mkLabel (WidgetRattus.Behaviour.map (box toText) shouldBe)

  let constantTest :: (Beh Float) = constK 514
  constantTest' ← derivative constantTest
  constantResultLbl ← mkLabel (WidgetRattus.Behaviour.map (box toText) constantTest')
  constantShouldLbl ← mkLabel (WidgetRattus.Behaviour.map (box toText) (constK 0))


  -- UI
  mkConstVStack $
    originalLbl :* resultLbl :* shouldLbl :* constantResultLbl :* constantShouldLbl

main :: IO()
main = runApplication derivativeTests
```

B Integral and Derivative helper functions

Part of *Behaviour.hs*:

```
getDiff :: Time → Time → Float
getDiff t' t = fromRational (toRational (diffTime t' t))
```

Initial version:

```
calcIntegral ::  Float → Fun Float → Time → Time → Float
calcIntegral cur a t t' = cur + apply a t' * getDiff t' t

calcDerivative ::  Float → Time → Bool → Fun Float
calcDerivative a t b = Fun $ box (λt' → a / getDiff t' t :* b)
```

Final version:

```
calcIntegral :: Float → Float → Time → Time → Float
calcIntegral cur a t t' = cur + a * getDiff t' t

calcDerivative :: Float → Float → Time → Time → Float
calcDerivative a b t t' = (a - b) / getDiff t' t
```

C Rewrite rules for behaviours and events

Rewrite rules for behaviours:

```
{-# NOINLINE [1] map #-}

{-# NOINLINE [1] const #-}

{-# NOINLINE [1] constK #-}

{-# NOINLINE [1] switch #-}

{-# RULES
"beh.map/beh.map" forall f g xs.
  WidgetRattus.Behaviour.map f (WidgetRattus.Behaviour.map g xs) =
    WidgetRattus.Behaviour.map (box (unbox f . unbox g)) xs
"beh.constK/beh.map" forall (f :: (Stable b) ⇒ □ (a → b)) x.
  WidgetRattus.Behaviour.map f (constK x) =
    let x' = unbox f x in constK x'
"beh.const/beh.switch" forall x xs.
  switch (const x) xs =
    Beh (x ::: delay (unwrap (adv xs)))
"beh.constK/beh.switch" forall x xs.
  switch (constK x) xs =
    Beh (K x ::: delay (unwrap (adv xs)))
  #-}
```

Rewrite rules for Events:

```
{-# NOINLINE [1] map #-}

{-# NOINLINE [1] filter #-}

{-# RULES
"ev.map/ev.map" forall f g xs.
  map f (map g xs) =
    map (box (unbox f . unbox g)) xs
"ev.map/ev.filter" forall f g xs.
  map f (filter g xs) =
    filterMap (box (λx → if unbox g x then Just' (unbox f x) else Nothing')) xs
"ev.filter/ev.map" forall f g xs.
  filter f (map g xs) =
    filterMap (box (λx → if (unbox f . unbox g) x then Just' $ unbox g x else Nothing')) xs
"ev.filter/ev.filter" forall f g xs.
  filter f (filter g xs) =
    filterMap (box (λx → if unbox f x && unbox g x then Just' x else Nothing')) xs
  #-}
```

D 7GUIs code examples

D.1 Calculator

```
module Main where

import WidgetRattus
import WidgetRattus.Widgets
import WidgetRattus.Behaviour
import WidgetRattus.Event
nums :: List Int
nums = [0 .. 9]

data Op = Plus | Minus | Equals | Reset

compute :: (Int :* Op :* Bool → Maybe' (Int :* Op) → Int :* Op :* Bool)
compute (n :* op     :* _)  Nothing'            = n :* op :* False
compute  _                  (Just' (_ :* Reset)) = 0 :* Reset :* True
compute (n :* Plus   :* _) (Just' (m :* op))    = (n + m) :* op :* True
compute (n :* Minus  :* _) (Just' (m :* op))    = (n - m) :* op :* True
compute (_ :* Equals :* _) (Just' (m :* op))    = m :* op :* True
compute (_ :* Reset  :* _) (Just' (m :* op))    = m :* op :* True

calculatorExample :: C VStack
calculatorExample = do
  -- Buttons
  numBtns :: List Button ←
    mapM (mkButton . constK) nums

  let [b0, b1, b2, b3, b4, b5, b6, b7, b8, b9] = numBtns
  resetBut ← mkButton (mkConstText "C")
  addBut ← mkButton (mkConstText "+")
  subBut ← mkButton (mkConstText "-")
  eqBut ← mkButton (mkConstText "=")

  let numClicks :: List (Ev (Int → Int)) =
        zipWith' (λb n → WidgetRattus.Event.map (box (λ_ x → x * 10 + n))
            (btnOnClickEv b)) numBtns nums

  let resetEv :: Ev (Int → Int) =
        WidgetRattus.Event.map (box (λ_ _ → 0)) $
          interleaveAll (box (λa _ → a)) $
            map' btnOnClickEv [addBut, subBut, eqBut, resetBut]

  let evList = resetEv :! numClicks :: List (Ev (Int → Int))
  let combinedEvs = interleaveAll (box (λa _ → a)) evList
constructed)
  let numberEv :: Ev Int =
        scan (box (λa f → f a)) 0 combinedEvs

  let opEv :: Ev Op =
          interleaveAll (box (λa _ → a)) $
            map'
              (λ(op :* btn) → WidgetRattus.Event.map (box (λ_ → op)) (btnOnClickEv btn))
              [Plus :* addBut, Minus :* subBut, Equals :* eqBut, Reset :* resetBut]
```

```
  let operand :: Ev (Maybe' (Int :* Op)) =
        triggerM (box (λop n → Just' (n :* op))) opEv (stepper 0 (buffer 0 numberEv))

  let resultEv :: Ev (Int :* Op :* Bool) =
        scan (box compute) (0 :* Plus :* True) operand

  let displayBeh :: Beh Int =
        WidgetRattus.Behaviour.zipWith (box (λ(n :* _ :* b) m → if b then n else m))
            (WidgetRattus.Event.stepper (0 :* Plus :* False) resultEv)
            (WidgetRattus.Event.stepper 0 numberEv)

  -- UI
  result ← mkLabel displayBeh
  operators ← mkConstVStack (resetBut :* addBut :* subBut :* eqBut)
  row1 ← mkConstHStack (b7 :* b8 :* b9)
  row2 ← mkConstHStack (b4 :* b5 :* b6)
  row3 ← mkConstHStack (b1 :* b2 :* b3)

  numbers ← mkConstVStack (row1 :* row2 :* row3 :* b0)

  input ← mkConstHStack (numbers :* operators)
  mkConstVStack (result :* input)

main :: IO()
main = runApplication calculatorExample
```

D.2 Counter

```
module Main where
import WidgetRattus
import WidgetRattus.Widgets
import WidgetRattus.Event
import Prelude hiding (const, filter, getLine, map, null, putStrLn, zip, zipWith)
counterAndTimer :: C VStack
counterAndTimer = do
  -- Button
  counterBtn ← mkButton $ mkConstText "Increment"
  let counterEv = scan (box (λn _ → n + 1 :: Int)) 0 $ btnOnClickEv counterBtn

  -- UI
  lbl ← mkLabel $ stepper 0 counterEv
  mkConstVStack $ lbl :* counterBtn

main :: IO()
main = runApplication counterAndTimer
```

D.3 Stopwatch

```
module Main where

import WidgetRattus
import WidgetRattus.Signal (Sig ((:::)))
import WidgetRattus.Widgets
import WidgetRattus.Behaviour
```

```
import WidgetRattus.Event
import Prelude hiding (const, filter, getLine, map, null, putStrLn, zip, zipWith)

elapsedTime' :: C (NominalDiffTime → Beh NominalDiffTime)
elapsedTime' =
  do
    startTime ← time
    return (λf → Beh (Fun ()
        (box (λ_ currentTime → (f + diffTime currentTime startTime) :⋆  Just' ())) ::: never))

timerExample :: C VStack
timerExample = do
  -- Time
  startElapsedTime ← elapsedTime

  -- Buttons
  startBtn ← mkButton (mkConstText "Start")
  let startEv = btnOnClick startBtn
  stopBtn ← mkButton (mkConstText "Stop")
  let stopEv = btnOnClick stopBtn

  let startTime :: Ev (NominalDiffTime → Beh NominalDiffTime) =
        mkEv' (box (delay (let _ = adv (unbox startEv) in elapsedTime')))
  let stopTime :: Ev (NominalDiffTime → Beh NominalDiffTime) =
        mkEv (box (delay (let _ = adv (unbox stopEv) in constK)))

  let combinedInput = WidgetRattus.Event.interleave (box (λx _ → x)) startTime stopTime
  let stopWatchSig = switchR (constK 0) combinedInput

  -- UI
  timeLabName ← mkLabel (mkConstText "Current Time:")
  swLabName ← mkLabel (mkConstText "Elapsed Time:")

  timeLab ← mkLabel startElapsedTime
  stopWatchLab ← mkLabel stopWatchSig

  time ← mkConstHStack (timeLabName :⋆ timeLab)
  sw ← mkConstHStack (swLabName :⋆ stopWatchLab)
  buttons ← mkConstHStack (startBtn :⋆ stopBtn)
  mkConstVStack (time :⋆ sw :⋆ buttons)

main :: IO()
main = runApplication timerExample
```

D.4 Temperature Converter

```
import WidgetRattus
import Prelude hiding (map, const, zipWith, zip, filter, getLine, putStrLn,null)
import Data.Text hiding (zipWith, filter, map, all)
import Data.Text.Read
import WidgetRattus.Widgets
import WidgetRattus.Behaviour
import WidgetRattus.Event

celsiusToFahrenheit :: Int → Int
```

```
celsiusToFahrenheit t = t * 9 `div` 5 + 32

fahrenheitToCelsius :: Int → Int
fahrenheitToCelsius t = (t - 32) * 5 `div` 9

isNumber :: Text → Maybe' Int
isNumber "" = Just' 0
isNumber t =
    case signed decimal t of
        Right (t', "") → Just' t'
        _ → Nothing'

window :: C HStack
window = do
    -- TextFields
    tfF1 ← mkTextField "32"
    tfC1 ← mkTextField "0"

    let fEvent = WidgetRattus.Event.filterMap (box isNumber) (textFieldOnInput tfF1)
    let cEvent = WidgetRattus.Event.filterMap (box isNumber) (textFieldOnInput tfC1)

    let convertFtoC = WidgetRattus.Event.map (box fahrenheitToCelsius) fEvent
    let convertCtoF = WidgetRattus.Event.map (box celsiusToFahrenheit) cEvent

    let c = stepper 0 (interleave (box (λx _ → x)) cEvent convertFtoC)
    let f = stepper 32 (interleave (box (λx _ → x)) fEvent convertCtoF)

    tfF2 ← setInputBehTF tfF1 (WidgetRattus.Behaviour.map (box toText) f)
    tfC2 ← setInputBehTF tfC1 (WidgetRattus.Behaviour.map (box toText) c)

    -- UI
    fLabel ← mkLabel $ mkConstText "Fahrenheit"
    cLabel ← mkLabel $ mkConstText "Celsius"

    fStack ← mkConstVStack (tfF2 :* fLabel)
    cStack ← mkConstVStack (tfC2 :* cLabel)
    mkConstHStack (fStack :* cStack)


main :: IO()
main = runApplication window
```

D.5 Timer

```
module Main where
import WidgetRattus.Widgets
import WidgetRattus
import WidgetRattus.Behaviour
import WidgetRattus.Event

nominalToInt :: NominalDiffTime → Int
nominalToInt x = floor $ toRational x

intToNominal :: Int → NominalDiffTime
intToNominal x = fromInteger (toInteger x)
```

```
timeFrom :: Int → Time → Beh NominalDiffTime
timeFrom max startTime =
  stopWith (box (λt → if t >= intToNominal max then Just' (intToNominal max) else Nothing'))
    (WidgetRattus.Behaviour.map (box (`diffTime` startTime)) timeBehaviour)


window :: C VStack
window = do
  let initialMax = 5
  -- Reset button
  resetBtn ← mkButton $ mkConstText "Reset timer"
  let resetTrigger = btnOnClickEv resetBtn

  -- Slider
  maxSlider ← mkSlider initialMax (constK 1) (constK 100)
  let maxBeh = sliderCurrent maxSlider
  let maxChangeEv = WidgetRattus.Event.map (box (Prelude.const ())) $
                        sliderOnChange maxSlider

  startTime ← time
  let timeWithMax = WidgetRattus.Behaviour.zipWith (box (:*)) timeBehaviour maxBeh
  let timer = WidgetRattus.Event.trigger
    (box (λ_ (t :* max) _ → timeFrom max t))
    (WidgetRattus.Event.interleave (box (λ_ _ → ())) resetTrigger maxChangeEv) timeWithMax

  let timer' = switchR (timeFrom initialMax startTime) timer

  text ← mkLabel
    (WidgetRattus.Behaviour.map (box (λt → "Current: " <> toText (nominalToInt t))) timer')
  maxText ← mkLabel
    (WidgetRattus.Behaviour.map (box (λmax → "Max: " <> toText max)) maxBeh)
  mkConstVStack $ maxSlider :* maxText :* text :* resetBtn

main :: IO()
main = runApplication window
```