Functional Reactive GUI Programming with Modal Types

Jean-Claude Disch, Asger Heegaard, and Patrick Bahr

IT University of Copenhagen, Copenhagen, Denmark
jdis@itu.dk, asgerheegaard@gmail.com, paba@itu.dk

Abstract.

Functional reactive programming (FRP) is a programming paradigm for implementing reactive systems, i.e. programs that continuously interact with their environments. While FRP allows for a functional, high-level programming style, FRP programs are prone to undesirable operational behaviours such as *space leaks*. To ensure favourable operational properties of FRP programs, *modal type systems* have been introduced, which – among other things – make it impossible to write FRP programs with implicit space leaks. In a recent development, several modal FRP languages have been introduced that are able to handle *asynchronous* events and behaviours – motivated by the goal to use such languages for GUI programming.

This paper explores the suitability of one such asynchronous modal FRP language – called Async Rattus – for GUI programming in *practice*. To this end, we have implemented a mild extension of the Async Rattus language and used it to implement a small GUI framework. We demonstrate the language and its GUI framework by a number of case studies.

1 Introduction

Interactive applications, especially those with *graphical user interfaces* (GUIs), form a cornerstone of contemporary software systems. Most modern GUI frameworks are based on an imperative programming model that is based on shared mutable state that is read and updated via callback functions. While computationally efficient, this model combines features that are notoriously difficult to reason about, namely mutable state, higher-order functions, and concurrency. This often leads to error-prone code that is difficult to maintain and debug.

Functional reactive programming (FRP) has arisen as an alternative high-level programming paradigm to implement such interactive systems [24]. The fundamental idea is to represent dynamic behaviours using a type of time varying values, called *signals* or *behaviours*:

$$\textbf{type}\ \mathit{Sig}\ a = \mathit{Time} \to a$$

However, while conceptually simple and easier to reason about, functional reactive programs directly based on this (denotational) notion of signals are impossible to implement efficiently in general – with early implementations suffering from space and time leaks. This has led to two lines of work on devising FRP languages that carefully reign in the expressive power of the language in order to avoid such pathologic performance behaviour: One based on a carefully restricted set of combinators that are made available to programmers to construct signals [22,23], and one based on modal type systems to keep track of temporal dependencies [14,15,21,18].

In this paper, we focus on the latter approach, which retains more of the expressive power of the denotational notion of signals – they can still be manipulated directly – at the cost of a more complicated type system. To this end, these modal FRP languages feature a modal type operator $\bigcirc$, so that a value of type $\bigcirc A$ is the promise of a value of type A that is available in the next time step. Using this *later modality*, (discrete) signals can be represented by the recursive type[1]:

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

That is, a signal of type *Sig a* consists of a value of type a and the promise of a new signal of type *Sig a* in the next time step.

However, these early modal FRP languages are inherently *synchronous* in nature, i.e. each delayed value of type $\bigcirc A$ arrives at the same time, namely *the* next time step according to some global clock. This is an unrealistic and fundamentally inefficient notion of time for GUI applications where different signals may receive updates at different, independent times. In response to this, the asynchronous modal FRP calculi $\lambda_{\mathsf{Widget}}$ [12] and Async RaTT [6] have been proposed recently.

In this paper, we present *Widget Rattus*, an FRP language based on the Async RaTT calculus and implemented as an embedded language in Haskell for the purpose of GUI programming. Widget Rattus consists of a small extension of the Async Rattus language [5], which implements Async RaTT as an embedded language in Haskell, along with a small GUI framework.

In short, this paper makes the following contributions:

- We extend the Async Rattus language with a first-class notion of *channels* and a generalised notion of output channels (section 2).
- We present a purely functional GUI framework implemented in Widget Rattus and demonstrate its use on two extended examples (section 3).
- We give an overview of the implementation of the GUI framework based on the principles of FRP (section 4).
- We compare Widget Rattus to $\lambda_{\mathsf{Widget}}$ and other related work (section 5).

The Widget Rattus language and GUI framework is available as a Haskell package [13], which also contains further examples beyond those presented in section 3.

[1] This definition uses ::: as an infix operator, similarly to : for Haskell lists.

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

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

$$
\frac{}{\Gamma \vdash \mathsf{chan} :: C(\mathsf{Chan}\ A)}
\qquad
\frac{\Gamma \vdash t :: \mathsf{Chan}\ A}{\Gamma \vdash \mathsf{wait}\, t :: \bigcirc A}
\qquad
\frac{\Gamma \vdash t :: A \qquad A \text{ continuous}}{\Gamma \vdash \mathsf{promote}\, t :: \Box A}
$$

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

Fig. 1: Select typing rules for Widget Rattus.

2 From Async Rattus to Widget Rattus

In this section, we give a brief introduction to the Async Rattus language [5] (sections 2.1 to 2.5) and then present two extensions to the language that will enable GUI programming (section 2.6).

2.1 Introduction to Async Rattus

Async Rattus is an implementation of the *Async RaTT* calculus [6] as a shallowly embedded language in Haskell. By virtue of being shallowly embedded, Async Rattus has access to Haskell's extensive library ecosystem. However, Async Rattus differs from Haskell in two major ways.

The first difference is that Async Rattus is eagerly evaluated while Haskell uses lazy evaluation by default. The choice of eager evaluation is an important part of how Async Rattus prevents space leaks while still allowing the programmer to manipulate signals directly [6].

The second fundamental difference introduced by Async Rattus is an extension of the type system. Async Rattus introduces two type modalities, $\bigcirc$ and $\Box$, called the *later* and *box* modalities. The later modality $\bigcirc$ represents values that will be available in the future, whereas the box modality $\Box$ is used for computations that remain stable across time and can be executed when needed. Figure 1 presents the most important typing rules of Async Rattus and its extension Widget Rattus. We describe the type modalities and the type system that enables them in more detail below.

To account for these fundamental differences in semantics and type system, Async Rattus is implemented by a combination of a Haskell library, which implements the basic primitives and types of the language, and a compiler plugin. This compiler plugin transforms the code so that it matches the eager evaluation semantics of Async Rattus, and it performs an additional typechecking pass to enforce the stricter typing rules of the language.

2.2 Later modality and clocks

The later modality $\bigcirc$ indicates values that are not immediately available but expected in the future, contingent on the occurrence of some event. A value of type $\bigcirc A$ represents a delayed computation that will produce a value of type A in the future. The timing of when the value will become available is determined by a *clock*, which is a record of data dependencies: An Async Rattus program may receive data from several input channels such as the keyboard or a button on a GUI. A clock θ is a set of such input channels, e.g. $\theta = \{c_{keyboard}, c_{ok\_button}\}$, and a *tick* on θ means that data has been received on some input channel $c \in \theta$.

Any value of type $\bigcirc A$ is effectively a pair (θ, f) consisting of a clock θ and a computation f that will produce a value of type A when executed. The computation f remains dormant until the clock θ *ticks*, which signals the occurrence of the anticipated event. This mechanism ensures that delayed computations respect temporal causality. That is, delayed computations are performed only when their time comes, according to the ticking of their associated clocks.

Conceptually, we can think of the two components θ and f of a delayed computation of type $\bigcirc A$ to be accessible via two functions $\mathsf{cl} :: \bigcirc A \to \mathrm{Clock}$ and $\mathsf{adv} :: \bigcirc A \to A$, respectively. However, as we shall see, cl is not directly accessible to the programmer, and adv is subject to the typing rule in Figure 1, which we discuss shortly. Conversely, to construct a delayed computation, one can use the delay function, which – for now – we can think of as having type $\mathsf{delay} :: \mathrm{Clock} \to A \to \bigcirc A$. That is, it takes a clock θ and a computation producing A and returns a delayed computation (θ, f) that will yield the value of type A once θ ticks. This conceptual representation suffices for the moment, but we will refine this oversimplification when presenting the typing rules for adv and delay. Also note that while Async Rattus is eagerly evaluated by default, delay does not evaluate its argument of type A eagerly, since it represents a delayed computation that may only be performed once the associated clock θ ticks.

An illustrative example of the interplay between the later modality and clocks is the following increment function [5]:

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

The function *incr* delays the increment operation until the clock $\mathsf{cl}(x)$ associated with x ticks. This schedules the increment to happen in the future, thereby enforcing causality of the system.

To keep track of temporal dependencies, typing contexts contain tokens of the form $\checkmark_\theta$ to indicate that a tick on some clock θ has occurred. These tokens $\checkmark_\theta$ are also called *ticks*. An example context in Async Rattus could look like this:

$$x :: \mathit{Int}, \checkmark_\theta, y :: \mathit{Int}, z :: \mathit{Text}, \checkmark_{\theta'}$$

Here the ticks represent the passage of time according to the clocks θ and θ'. Intuitively speaking, this context expresses the fact that the value assigned to variable x is available one time step (on clock θ) before the values assigned to y and z, which in turn are one time step (on clock θ') old. With this understanding in mind, we can see from the typing rules in Figure 1 that adv can only be used to advance a delayed computation $t :: \bigcirc A$ if a tick on the clock of t has occurred, indicated by the token $\checkmark_{\mathsf{cl}(x)}$. Moreover, this tick must be the most recent tick, i.e. there are no other ticks occurring to the right of it, which is indicated by the condition $\checkmark \notin \Gamma'$. And finally, t itself must be typeable using only using the part of the context to the left of the tick $\checkmark_{\mathsf{cl}(x)}$, i.e. t must come from the time before its clock ticked. This typing rule encodes the intuition that the presence of the tick $\checkmark_{\mathsf{cl}(t)}$ in the typing context signifies that the clock $\mathsf{cl}(t)$ has responded to an event making the value computed by t *presently* available.

Consider the following example of a function for eliminating the later modality that is not causal and indeed does not typecheck:

$$\begin{aligned}&\mathit{advBad} :: \bigcirc\ \mathit{Int} \rightarrow \mathit{Int}\\&\mathit{advBad}\ x = \mathsf{adv}\ x\end{aligned}$$

The *advBad* function tries to use adv to extract a future value of the delayed computation x without waiting for its clock $\mathsf{cl}(x)$ to tick. In other words, *advBad* is not causal as it tries to look up data now that is only available later.

While adv is used for eliminating the later modality and executing delayed computations in response to events, delay is used to construct delayed computations. To safely construct a delayed computation one needs to associate it with a clock to signify when it can be executed. The delay function allows us to delay a computation $t :: A$ until the time at which a certain clock θ ticks. The type system keeps track of that by making sure that t type checks in a context that includes $\checkmark_\theta$. The new context $\Gamma, \checkmark_\theta$ not only indicates that θ has ticked by the time t gets evaluated, but also that all variables that were previously available in Γ, are now one time step older according to the clock θ.

As an example, consider the following ill-typed definition using delay:

$$\begin{aligned}&\mathit{delayBad} :: \mathit{Int} \rightarrow \bigcirc\ \mathit{Int}\\&\mathit{delayBad}\ x = \mathsf{delay}_{\theta}\ x \quad \text{-- no clock } \theta \text{ available}\end{aligned}$$

In this example, *delayBad* tries to delay an integer x, but in order to do so it needs to specify a clock θ. Clocks cannot be constructed directly, but can only be extracted from other delayed computations. Since no such delayed computation is in context, we cannot use delay here.

Together, adv and delay enable precise control over when computations are carried out in Async Rattus. This ensures that values are only accessed or modified at appropriate times, often in response to user generated input. The following is a typical example of how adv and delay interact:

$$\begin{aligned}&\mathit{doubleLater} :: \bigcirc\ \mathit{Int} \rightarrow \bigcirc\ \mathit{Int}\\&\mathit{doubleLater}\ x = \mathsf{delay}_{\mathsf{cl}(x)}\ (2 * \mathsf{adv}\ x)\end{aligned}$$

In this example, *doubleLater* takes a delayed integer and returns a new delayed integer that is twice the original value. To this end, the new delayed computation has the same clock as the incoming delayed computation x. That means, the term $2 * \mathsf{adv}\ x$ is type checked in the context $x :: \bigcirc Int, \checkmark_{\mathsf{cl}(x)}$.

The clock argument to delay can typically be inferred from the context in which it appears. In the above example, the clock has to be $\mathsf{cl}(x)$, as we need the tick $\checkmark_{\mathsf{cl}(x)}$ in context so that we can advance x. We therefore elide the clock argument from now on, and indeed the Async Rattus type checker will infer the clock argument. In practice, the above example would thus be written as follows in Async Rattus:

$$\begin{aligned}&\mathit{doubleLater} :: \bigcirc\ \mathit{Int} \to \bigcirc\ \mathit{Int}\\&\mathit{doubleLater}\ x = \mathsf{delay}\ (2 * \mathsf{adv}\ x)\end{aligned}$$

2.3 The Box Modality and Stable Types

Computations in Async Rattus may contain references to time-dependent data as for example the reference to x in the computation performed by $2 * \mathsf{adv}\, x$ in the definition of *doubleLater* above. Such references to time-dependent values may cause space leaks in FRP programs as these values have to be kept in memory until the computation that references it is performed. To prevent this, Async Rattus does not allow programmers to move arbitrary data across time steps. Only data of certain types can be moved across time. Such types are called *stable types*, and they include base types like *Int* and *Bool*, which do not carry any temporal dependency and can, therefore, be referenced safely at any point in a program's execution. Types of the form $\bigcirc A$ and $A \to B$ are not stable, since delayed computations are by definition time-dependent and since functions may contain references to arbitrary data in their closure – including time-dependent data. All other types are stable, including algebraic data types and record types that don't contain types of the form $\bigcirc A$ or $A \to B$.

The typing rule for variables enforces that only values of stable types can be moved across time: A variable that occurs to the left of a tick – and is thus from the past – can only be used if it is of a stable type. The following example illustrates this:

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

This definition does not typecheck since f is no longer in scope when it is used under delay. The typing context to typecheck $f\ (\mathsf{adv}\ x)$ is $f :: a \to b, x :: a, \checkmark_{\mathsf{cl}(x)}$ and according to the typing rule for variables, we can see that f typechecks in this context only if its type $a \to b$ was stable, which it is not.

To safely and efficiently move values of such non-stable types across time, Async Rattus provides the box modality $\Box$, which turns any type A into a stable type $\Box A$. However, when constructing values of type $\Box A$ using the box primitive, the type system enforces restrictions that makes sure that such boxed values are indeed time-independent. The typing rule for box requires that its argument t typecheck in a context $\Gamma^{\Box}$, which is obtained from Γ by removing all ticks and all variable bindings $x :: A$ where A is not stable. This ensures that $\mathsf{box}\, t$ is time-independent and can thus be moved across time. Similarly to delay, also box evaluates its argument lazily. That is, the argument t is only evaluated when the boxed value is forced using unbox.

Using the box modality, we can revise the *mapLaterBad* function so that it takes a boxed function instead:

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

Now f is in scope because, while it is still occurring to the left of a tick, it is of a stable type, namely $\Box\,(a \to b)$.

2.4 Signals

Signals can be defined in Async Rattus by the following definition:

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

That is, a signal of type $\mathit{Sig}\, a$ consists of a current value of type a and a future update to the signal of type $\bigcirc(\mathit{Sig}\, a)$. Such signals can be easily manipulated using pattern matching and recursion. For example, we can define a *map* function for signals, but similarly to the *mapLater* function on the $\bigcirc$ modality, the function argument has to be boxed:

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

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

The following example shows the use of never to introduce delayed computations that will never be triggered:

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

This function allows us to construct signals whose update component will never be triggered and thus maintain a constant value.

2.5 Asynchronous computations

Unlike synchronous modal FRP languages, each delayed computation in Async Rattus comes with its own local clock. That means we cannot easily combine two delayed computations since they may not be triggered simultaneously. For example, we cannot implement a function like this:

$$\begin{array}{l} addBad :: \bigcirc\, Int \to \bigcirc\, Int \to \bigcirc\, Int \\ addBad\ x\ y = \mathsf{delay}\ (\mathsf{adv}\ x + \mathsf{adv}\ y) \end{array}$$

The problem is that we cannot annotate delay with a single clock θ that will allow us to advance on both x and y since they may have different clocks.

To process more than one delayed computation, Async Rattus allows us to form the union $\theta \sqcup \theta'$ of two clocks θ and θ', which ticks whenever θ or θ' ticks. We can use such union clocks via the select primitive, which takes two delayed computations $s :: \bigcirc A$ and $t :: \bigcirc B$ as arguments and requires that a tick on the clock $\mathsf{cl}(s) \sqcup \mathsf{cl}(t)$ is in the context. In return, select produces a value of type *Select A B*:

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

A term select s t produces a value with constructor *Fst*, *Snd*, or *Both* if $\mathsf{cl}(s)$ ticks before, after, or at the same time as $\mathsf{cl}(t)$, respectively. Using select, we can implement a combinator that allows us to dynamically switch from one signal to another one:

$$\begin{array}{l} switch :: Sig\ a \to \bigcirc\ (Sig\ a) \to Sig\ a \\ switch\ (x ::: xs)\ d = x ::: \mathsf{delay}\ (\mathbf{case}\ \mathsf{select}\ xs\ d\ \mathbf{of} \\ \qquad\qquad\qquad\qquad\qquad\qquad Fst\quad xs'\ d' \to switch\ xs'\ d' \\ \qquad\qquad\qquad\qquad\qquad\qquad Snd\ \ \_\ \ d' \to d' \\ \qquad\qquad\qquad\qquad\qquad\qquad Both\ \_\ \ d' \to d') \end{array}$$

The signal produced by *switch xs ys*, first behaves like the signal *xs*, but it will start behaving like the delayed signal *ys* as soon as it arrives, i.e. when $\mathsf{cl}(ys)$ ticks.

2.6 Widget Rattus

Widget Rattus is a small extension of the Async Rattus language with two features that enable GUI programming: first-class channels and continuous types.

First-class channels. To implement GUIs we need to be able to dynamically create GUI components, or *widgets*, which in turn must be able to produce data obtained from user interaction, e.g. button press events or keyboard inputs from a text field. To this end, Widget Rattus introduces two primitives – shown in Figure 1 – to construct and interact with channels, which can then be used by widgets to send their data: chan creates a new channel of type Chan A, which can send data of type A, and wait turns such a channel into a delayed computation that produces a value of type A as soon as such a value is sent on the channel. Since chan is an effectful operation – it allocates a fresh channel – it uses a monad C to indicate its effectful nature.

Below we use a channel to construct a simple button that consists of a signal that describes its text and a channel of type Chan () that is intended to produce a unit value whenever it is pressed

$$\begin{array}{l}\textbf{data}\ \mathit{SimpleButton} = \mathit{SimpleButton}\ (\mathit{Sig}\ \mathit{Text})\ (\mathsf{Chan}\ ())\\ \mathit{simpleButton} :: C\ \mathit{SimpleButton}\\ \mathit{simpleButton} = \textbf{do}\ c \leftarrow \mathsf{chan}\\ \qquad\qquad\qquad\qquad \mathit{return}\ (\mathit{Button}\ (\mathit{const}\ \texttt{"OK"})\ c)\end{array}$$

Here the button just displays a constant signal of the text "OK". In the example below we make a more dynamic button that changes its text from "OK" to "Clicked" as soon as the button is clicked

$$\begin{array}{l}\mathit{respond} :: C\ \mathit{SimpleButton}\\ \mathit{respond} = \textbf{do}\\ \quad c \leftarrow \mathsf{chan}\\ \quad \textbf{let}\ \mathit{sig} = \mathit{switch}\ (\mathit{const}\ \texttt{"OK"})\\ \qquad\qquad\qquad\qquad (\mathit{mapLater}\ (\mathsf{box}\ (\lambda() \to \mathit{const}\ \texttt{"Clicked"}))\ (\mathsf{wait}\ c))\\ \quad \mathit{return}\ (\mathit{Button}\ \mathit{sig}\ c)\end{array}$$

The dynamic behaviour is achieved by the switch function we implemented in section 2.5. In turn, *mapLater* is used to turn the delayed computation $\mathsf{wait}\,c :: \bigcirc()$ into a delayed signal of type $\bigcirc(\mathit{Sig}\ \mathit{Text})$.

The channel type $\mathsf{Chan}\,a$ is stable and can thus be moved across time, which allows us to implement the following combinator to turn channels into signals:

$$\begin{array}{l}\mathit{chanSig} :: \mathsf{Chan}\ a \to \bigcirc\ (\mathit{Sig}\ a)\\ \mathit{chanSig}\ c = \mathsf{delay}\ (\mathsf{adv}\ (\mathsf{wait}\ c) ::: \mathit{chanSig}\ c)\end{array}$$

Since the channel c is of a stable type, we can move it across the tick introduced by delay and then pass it to the recursive call of *chanSig*.

Continuous types. The second extension provided by Widget Rattus generalises the outputs that a reactive program can produce. An Async Rattus program can produce output in the form of signals of type $\mathit{Sig}\ A$ for basic types A. A value of type $\mathit{Sig}\ A$ is of the form $v_0 ::: (\theta_0, f_0)$, where $v_0 :: A$ and f is a delayed computation that produces a new signal $v_1 ::: (\theta_1, f_1)$ as soon as θ_0 ticks. That is, a signal produces a sequence of values $v_0, v_1, \ldots$, each triggered by a corresponding clock.

The output produced by GUI programs does not have this linear structure but is instead tree-shaped. We can imagine a GUI being represented as a signal of type *Sig Widget*, where *Widget* is a type that describes the top-level widget of the GUI, e.g. a container widget that contains other widgets, which in turn may consist of other widgets. That is, GUIs are described by a tree structure. For example, we may define a widget that consists of several buttons:

$$\textbf{data}\ \mathit{Buttons} = \mathit{Buttons}\ (\mathit{Sig}\ \mathit{Color})\ (\mathit{Sig}\ (\mathit{List}\ \mathit{SimpleButton}))$$

This container widget consists of a colour signal but also of a signal of a list of buttons. Since buttons themselves consist of signals, the output mechanism of the language needs to handle nested signals.

To this end, Widget Rattus introduces the notion of continuous types. These are types whose values may dynamically change over time. Each value of a continuous type has a current clock, which tells us when the value needs to be updated, and an update function that updates the value whenever its clock ticks. For example, a signal $v_0 ::: (\theta_0, f_0)$ has the clock θ_0 and is updated once θ_0 ticks by performing the delayed computation f_0 which yields a new signal $v_1 ::: (\theta_1, f_1)$.

This allows us to deal with signals of type *Sig A* where A is not a basic type, but may in general be a continuous type. In that case, the clock of a signal $v_0 ::: (\theta_0, f_0)$ is the union of the clock of v_0 and θ_0, i.e. $\mathsf{clock}(v_0) \sqcup \theta_0$. This clock ticks when either $\mathsf{clock}(v_0)$ or θ_0 ticks. If θ_0 ticks then the new signal is produced by performing the delayed computation f_0. Otherwise, if $\mathsf{clock}(v_0)$ ticks, the new signal is $v_0' ::: (\theta_0, f_0)$, where v_0' is obtained by (recursively) updating v_0.

Any basic type is a continuous type, and – like stable types – continuous types are closed under product, sum types and recursive types. However, unlike stable types, continuous types are closed under forming signal types, i.e. if A is a continuous type, then so is *Sig A*. Since continuous types can be updated over time, Widget Rattus comes with a primitive promote that promotes a continuous type to a boxed type as shown in Figure 1. In particular, any widget type that we implement in Widget Rattus is a continuous type. That means, it can be used by Widget Rattus' runtime system to render it as a GUI on screen and update it in response to GUI events. Moreover, we can promote any widget so that we can safely move it into the future. So a button that is constructed at some point in time can safely be moved into the future by promoting it to a boxed type. The semantics of promote makes sure that the boxed widget is updated in response to events that make its clock tick, e.g. if it is the *respond* button defined above, it will change its label when it is clicked.

3 GUIs in Widget Rattus

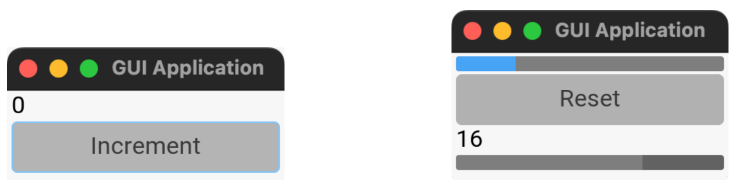
This section will cover two examples of GUIs implemented using Widget Rattus[2]. The GUIs are based on the *counter* and *timer* benchmarks from Kiss's *7GUIs* benchmark [17]. The GUIs are constructed using the Widget Rattus libraries shown in Figure 2 and Figure 5. Figure 2 shows a selection of combinators to construct and manipulate signals, and Figure 5 contains functions to construct widgets. Most of the signal combinators shown in Figure 2 are taken from the original Async Rattus work [5]. But we have added a few additional combinators in response to the GUI examples, and we discuss these in this section. The implementation of widgets and their constructor functions will be covered in more detail in section 4.

---

[2] Further examples are included in the Widget Rattus package: `https://github.com/pa-ba/AsyncRattus/tree/WidgetRattus/examples/gui`

$$
\begin{aligned}
&map :: \Box\,(a \to b) \to Sig\ a \to Sig\ b\\
&mkSig :: \Box\,(\bigcirc\ a) \to \bigcirc\,(Sig\ a)\\
&const :: a \to Sig\ a\\
&scan :: (Stable\ b) \Rightarrow \Box\,(b \to a \to b) \to b \to Sig\ a \to Sig\ b\\
&scanAwait :: (Stable\ b) \Rightarrow \Box\,(b \to a \to b) \to b \to \bigcirc\,(Sig\ a) \to Sig\ b\\
&switch :: Sig\ a \to \bigcirc\,(Sig\ a) \to Sig\ a\\
&switchS :: Stable\ a \Rightarrow Sig\ a \to \bigcirc\,(a \to Sig\ a) \to Sig\ a\\
&switchR :: Stable\ a \Rightarrow Sig\ a \to \bigcirc\,(Sig\ (a \to Sig\ a)) \to Sig\ a\\
&interleave :: \Box\,(a \to a \to a) \to \bigcirc\,(Sig\ a) \to \bigcirc\,(Sig\ a) \to \bigcirc\,(Sig\ a)\\
&zipWith :: (Stable\ a, Stable\ b) \Rightarrow \Box\,(a \to b \to c) \to Sig\ a \to Sig\ b \to Sig\ c\\
&stop :: \Box\,(a \to Bool) \to Sig\ a \to Sig\ a\\
&timer :: Int \to \Box\,(\bigcirc\,())\\
&mkSig :: \Box\,(\bigcirc\ a) \to \bigcirc\,(Sig\ a)
\end{aligned}
$$

Fig. 2: Signal combinator library.



(a) Counter. (b) Timer.

Fig. 3: Example GUIs.

3.1 Counter

We begin with an example of a simple counter GUI depicted in Figure 3a. It contains a button that increments a label whenever it is pressed [17]. The implementation of this GUI in Widget Rattus is shown in Figure 4.

To start a GUI application using Widget Rattus, we have to pass the (compound) widget that is to be rendered in the application to the *runApplication* function. Widget Rattus provides a number of functions to create widgets. For the widgets relevant to the examples in this section, the type signatures can be seen in Figure 5.

For the counter GUI it is necessary to make a button and a label. The *mkButtons* function takes a signal that determines what text to display on the button. The signal may be of any type a that implements the *Displayable* type class, which includes *Text* and *Int*. For the counter we simply pass our button a constant signal with the *Text* value "Increment".

To add functionality to a button the *btnOnClickSig* function is used. This function takes as input a button and returns a signal that ticks – producing a unit – whenever the button is pressed. This signal can be turned into an

```
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
  mkVStack (const [mkWidget lbl, mkWidget btn])
main :: IO ()
main = runApplication counter
```

Fig. 4: Counter GUI Implementation

```
mkButton :: Displayable a ⇒ Sig a → C Button
btnOnClickSig :: Button → ◯ (Sig ())
mkTextField :: Text → C TextField
textFieldOnInputSig :: TextField → ◯ (Sig Text)
mkLabel :: Displayable a ⇒ Sig a → C Label
mkVStack :: IsWidget a ⇒ Sig (List a) → C VStack
mkConstVStack :: Widgets ws ⇒ ws → C VStack
mkSlider :: Int → Sig Int → Sig Int → C Slider
sldCurr :: Slider → Sig Int
mkProgressBar :: Sig Int → Sig Int → Sig Int → C Slider
mkWidget :: IsWidget a ⇒ a → Widget
runApplication :: IsWidget a ⇒ C a → IO ()
```

Fig. 5: GUI combinator Library

integer signal that produces the intended value of the counter. To this end, we use the *scanAwait* combinator, which similarly to Haskell's *scanl* combinator on lists applies a given function f to a signal. An input signal that produces values $v_1, v_2, \ldots$ is thus transformed into a signal producing values $x, f\ x\ v_1, f\ (f\ x\ v_1)\ v_2, \ldots$, where x is the starting value provided to *scanAwait*. In this case, we start with 0 and increment the previous value by one at each tick of the input signal *clicks*. The resulting signal *counts* is then used to create a label that always displays the current value of *counts*, which is exactly the number of times that the button has been pressed.

Finally, *runApplication* only takes as input a single widget, but in most GUIs it is necessary to display multiple widgets. For this purpose Widget Rattus provides horizontal and vertical stacks. Stacks are widgets that take as input a list of other widgets, allowing users of Widget Rattus to create their GUI as a tree structure composed of widgets. In the counter GUI a single vertical stack is made using the *mkVStack* function to contain both the button and label.

Note that *lbl* and *btn* are of two different types, namely *Label* and *Button*, but we have to construct a list of a single widget type in order to pass it to *mkVStack*. To this end, the library provides the *mkWidget* function to turn any widget type into the type *Widget*. Since we often find ourselves passing a constant signal of a list of differently-typed widgets, the library also provides a *mkConstVStack* function that takes any tuple consisting of widget types, e.g. $Label \times Button$. That means, the last line of *counter* can be more compactly written as $mkConstVStack\ (lbl \times btn)$ instead.

3.2 Timer

As a second example, we consider an interactive timer that ticks up every second and whose value is displayed in both a progress bar and numerically on a label [17]. User input comes in the form of a slider that determines the maximum value of the timer, as well as a button that resets the timer. Figure 3b shows our Widget Rattus implementation of this GUI application: The grey bar is the progress bar which – like the text label above it – increments towards the maximum value determined by the blue slider at the top. However, the progress bar also changes in response to inputs to the slider, since changing the maximum timer value changes the percentage of how much time has passed relative to the maximum. Pressing the reset button sets the timer (and thus both the label and the progress bar) to zero. The primary challenge of the timer GUI is concurrency, since user input competes with the state of the timer.

The full Widget Rattus code for the timer GUI is shown in Figure 6. At the top, *mkTimerSig* defines the base signal that increments the timer value until we reach the end of the timer. The state of the timer consists of two integer values: the number of seconds elapsed since the start of the timer and the maximum value of the timer, i.e. the number of seconds at which the timer will stop. This state is represented by the type $Int \times Int$, where $\times$ denotes the strict pair type of Widget Rattus.

The *mkTimerSig* function takes the initial state as input and produces a new timer signal that starts with that state. To this end, we use *timer* and *mkSig* from Figure 2 to produce a signal of type $\bigcirc(Sig\ ())$ that ticks every second (= 1000000 microseconds). Using *scanAwait* we turn this into a signal that advances the state of the timer, by incrementing the first component (elapsed seconds) but leaving the second component (the maximum timer value) untouched. To stop the timer when the maximum value is reached, we finally apply the *stop* combinator. This combinator takes a predicate and a signal as argument, and produces a new signal that behaves as the old signal, but stops as soon as the predicate is satisfied. We can implement *stop* as follows:

$$
\begin{aligned}
&stop :: \Box\,(a \to Bool) \to Sig\ a \to Sig\ a\\
&stop\ f\ (x ::: xs) = \mathbf{if}\ \mathsf{unbox}\ f\ x\ \mathbf{then}\ const\ x\\
&\qquad\qquad\qquad\quad\ \mathbf{else}\ x ::: \mathsf{delay}\ (stop\ f\ (\mathsf{adv}\ xs))
\end{aligned}
$$

The below example shows how the *stop* function works:

$$
\begin{array}{l}
\mathit{everySecondSig} :: \bigcirc\ (\mathit{Sig}\ ())\\
\mathit{everySecondSig} = \mathit{mkSig}\ (\mathit{timer}\ 1000000)\\
\mathit{mkTimerSig} :: \mathit{Int} \times \mathit{Int} \to \mathit{Sig}\ (\mathit{Int} \times \mathit{Int})\\
\mathit{mkTimerSig}\ \mathit{startState} = \mathit{stop}\ (\mathsf{box}\ (\lambda(n \times \mathit{nMax}) \to n \geqslant \mathit{nMax}))\ \mathit{timerSig}\\
\quad \textbf{where}\ \mathit{timerSig} :: \mathit{Sig}\ (\mathit{Int} \times \mathit{Int})\\
\qquad\qquad \mathit{timerSig} = \mathit{scanAwait}\ (\mathsf{box}\ (\lambda(n \times \mathit{nMax})\ () \to (n+1) \times \mathit{nMax}))\\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{startState}\ \mathit{everySecondSig}\\
\mathit{reset} :: () \to (\mathit{Int} \times \mathit{Int}) \to (\mathit{Int} \times \mathit{Int})\\
\mathit{reset}\ ()\ (n \times \mathit{nMax}) = (0 \times \mathit{nMax})\\
\mathit{setMax} :: \mathit{Int} \to (\mathit{Int} \times \mathit{Int}) \to (\mathit{Int} \times \mathit{Int})\\
\mathit{setMax}\ \mathit{nMax}'\ (n \times \mathit{nMax}) = \mathit{min}\ n\ \mathit{nMax}' \times \mathit{nMax}'\\
\mathit{window} :: C\ \mathit{VStack}\\
\mathit{window} = \textbf{do}\\
\quad \mathit{slider} \leftarrow \mathit{mkSlider}\ 50\ (\mathit{const}\ 1)\ (\mathit{const}\ 100)\\
\quad \mathit{resetBtn} \leftarrow \mathit{mkButton}\ (\mathit{const}\ (\texttt{"Reset"} :: \mathit{Text}))\\
\quad \textbf{let}\ \mathit{resetSig} = \mathit{mapAwait}\ (\mathsf{box}\ \mathit{reset})\ (\mathit{btnOnClickSig}\ \mathit{resetBtn})\\
\quad \textbf{let}\ \mathit{setMaxSig} = \mathit{mapAwait}\ (\mathsf{box}\ \mathit{setMax})\ (\mathit{future}\ (\mathit{sldCurr}\ \mathit{slider}))\\
\quad \textbf{let}\ \mathit{inputSig} :: \bigcirc\ (\mathit{Sig}\ (\mathit{Int} \times \mathit{Int} \to \mathit{Int} \times \mathit{Int}))\\
\qquad = \mathit{interleave}\ (\mathsf{box}\ (\circ))\ \mathit{resetSig}\ \mathit{setMaxSig}\\
\quad \textbf{let}\ \mathit{inputSig}' :: \bigcirc\ (\mathit{Sig}\ (\mathit{Int} \times \mathit{Int} \to \mathit{Sig}\ (\mathit{Int} \times \mathit{Int})))\\
\qquad = \mathit{mapAwait}\ (\mathsf{box}\ (\lambda f \to \mathit{mkTimerSig} \circ f))\ \mathit{inputSig}\\
\quad \textbf{let}\ \mathit{currentMax} = \mathit{current}\ (\mathit{sldCurr}\ \mathit{slider})\\
\quad \textbf{let}\ \mathit{counterSig} = \mathit{switchR}\ (\mathit{mkTimerSig}\ (0 \times \mathit{currentMax}))\ \mathit{inputSig}'\\
\quad \textbf{let}\ \mathit{currentSig} = \mathit{map}\ (\mathsf{box}\ \mathit{fst}')\ \mathit{counterSig}\\
\quad \textbf{let}\ \mathit{maxSig} = \mathit{map}\ (\mathsf{box}\ \mathit{snd}')\ \mathit{counterSig}\\
\quad \mathit{label} \leftarrow \mathit{mkLabel}\ \mathit{currentSig}\\
\quad \mathit{pb} \leftarrow \mathit{mkProgressBar}\ (\mathit{const}\ 0)\ \mathit{maxSig}\ \mathit{currentSig}\\
\quad \mathit{mkConstVStack}\ (\mathit{slider} \times \mathit{resetBtn} \times \mathit{label} \times \mathit{pb})
\end{array}
$$

Fig. 6: Timer GUI implementation

$$
\begin{array}{rl}
\mathit{xs} : & 1\ 2\ 3\ 4\ 5\ 6\ \ldots\\
\mathit{stop}\ (\mathsf{box}\ (\geqslant 3))\ \mathit{xs} : & 1\ 2\ 3
\end{array}
$$

Every time the input signal ticks it applies the function *f*. If the result is true the signal progression is stopped by returning *const* x, which is a constant signal that never ticks. As can be seen in the example, this could be used to stop an incrementing signal at a specified value.

The GUI constructed by *window* in Figure 6 first constructs the slider to adjust the timer and the reset button. The *mkSlider* function constructs a slider with 50 as its initial value and two constant signals that determine the minimum and maximum value of the slider to be 0 and 100, respectively.

Both the reset button and the slider change the state of the timer as described by the *reset* and *setMax* functions defined above *window*. These two

Table 1: Example table for *counterSig*

Events		**1 second**	**Max set to 10**	**1 second**	**Reset pressed**
inputSig			setMax 10		reset
counterSig	(0,50)	(1,50)	(2,10)	(3,10)	(0,10)

functions are applied to the two signals produced by the reset button and the slider using *mapAwait*, to obtain the signals *resetSig* and *setMaxSig*, both of type $\bigcirc(Sig((Int \times Int) \to (Int \times Int)))$. Both signals produce functions that are meant to manipulate the timer state, and we combine the two with the *interleave* combinator to create a signal that responds to both types of user input. The resulting signal *inputSig* ticks whenever either input signal ticks. When both signals tick, then the values of the two signals are combined – in this case using function composition $\circ$, so that both functions are applied – one after the other.

As an example of how *interleave* works consider the following integer signals *xs* and *ys* and how their interleaving looks like with the addition operator.

$$\begin{aligned} xs &: 1\ 3\quad 5\ 3\ 1\ 3 \ldots \\ ys &: \quad 0\ 2\quad 4 \qquad \ldots \\ interleave\ (\mathsf{box}\ (+))\ xs\ ys &: 1\ 3\ 2\ 5\ 7\ 1\ 3 \ldots \end{aligned}$$

Next, we need to combine the *inputSig* signal with the timer signal produced by *mkTimerSig*. To do so, we make use of a *switch* combinator. The signal combinator library in Figure 2 has three such combinators: The simplest, *switch*, we have already seen in section 2.5. It takes a signal *xs* and a delayed signal *ys* as arguments and produces a signal that first behaves like *xs* and switches to behaving like *ys* as soon as it arrives. The stateful version, *switchS*, works similarly but *ys* is now a delayed function that produces a signal depending on the previous value of the signal, rather than just a signal that is independent of the previous value of the signal. Finally, *switchR* is a repeating version of *switchS*. Instead of a single delayed function *ys*, it takes a delayed signal of such functions, and each time this signal produces a new function, that function is used to change the behaviour of the signal. We can implement it by repeatedly calling *switchS*:

$$\begin{aligned} &switchR :: Stable\ a \Rightarrow Sig\ a \to \bigcirc\ (Sig\ (a \to Sig\ a)) \to Sig\ a \\ &switchR\ sig\ steps = switchS\ sig \\ &\quad (\mathsf{delay}\ (\mathbf{let}\ step ::: steps' = \mathsf{adv}\ steps\ \mathbf{in} \lambda x \to switchR\ (step\ x)\ steps')) \end{aligned}$$

We use *switchS* to construct a signal that first behaves like *sig*. We further give *switchS* a delayed function that takes as argument the previous value x of the signal and produces a new signal. As soon as a function $step :: a \to Sig\ a$ arrives on the *steps* signal, our delayed function will apply *step* to x to obtain a new signal on which to recursively continue.

In the case of the timer GUI, *switchR* is used to create *counterSig*, a signal representing a capped timer, whose maximum and current value can be affected by user input. The signal first starts as *mkTimerSig* $(0 \times \mathit{currentMax})$, i.e. it simply ticks up every second. This signal then dynamically switches according to the signal *inputSig′*, which simply takes the functions of *inputSig* and composes them with *mkTimerSig*. In other words, for every user input, which produces a function f on *inputSig*, we apply that function f to the current timer state, and then we pass the resulting new state to *mkTimerSig* to resume the timer with this new state.

Table 1 illustrates an example of how *counterSig* behaves for a given sequence of user input: *counterSig* initially behaves like *mkTimerSig* applied to (0×50). After every second the first part of *counterSig* increments. When the user sets the maximum value to 10 by dragging the slider, *setMax 10* is applied to the current value of *counterSig*. This returns (2×10) and *counterSig* switches its behaviour to reflect *mkTimerSig* (2×10). This increments to (3×10) after another second. Once the reset button is pressed the *reset* function is called with the current value of *counterSig*. This returns (0×10) and *counterSig* now behaves like *mkTimerSig* (0×10).

Finally, *counterSig* is split into its two components with the help of the two projection functions $\mathit{fst}' :: a \times b \to a$ and $\mathit{snd}' :: a \times b \to b$. The resulting signals are passed on to the label and the progress bar, which along with the other widgets are grouped into a vertical stack.

4 GUI Library Implementation

After demonstrating the use of the Widget Rattus GUI library in the previous section, we turn to the implementation of the GUI library in this section. To simplify the implementation of the GUI library, we have built it on top of *Monomer* [25], a Haskell library for writing GUI applications using pure functional code in a style pioneered by Elm [9].

GUI elements in Widget Rattus GUIs are called *widgets* and the GUI is represented as a tree structure. Conceptually speaking, widgets are simply GUI elements that are defined as unique data structures. A button, for example, consists of two fields: a signal of text *btnContent* and an input channel *btnClick*:

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

The *btnContent* field represents the value displayed on the button. Any widget that takes user input needs to instantiate a channel of the corresponding type as described in section 2.6. In the case of a button this is a channel of type (), since the click event does not carry any data.

The *Displayable* type class is a variant of the standard type class *Show*:

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

It provides a method to render a value as text. The difference to *Show* is that: (1) it is a subclass of *Stable*, (2) it produces text of type *Text*, which unlike *String* is a strict type, and (3) its instance for *Text* implements *display* as the identity function rather than embedding the text in quotation marks.

To render widgets using Monomer, we need a way to turn them into corresponding data structures in Monomer, called *widget nodes*. This is achieved by the *IsWidget* type class:

$$\begin{array}{l}\textbf{class } \mathit{Continuous}\ a \Rightarrow \mathit{IsWidget}\ a\ \textbf{where}\\ \quad \mathit{mkWidgetNode} :: a \to \mathit{Monomer.WidgetNode}\ \mathit{AppModel}\ \mathit{AppEvent}\end{array}$$

The *mkWidgetNode* method translates a Wiget Rattus widget of type a into a widget node in Monomer, which can then be rendered by the monomer library. Every widget node in monomer has a model and event type. We will return to the definition of these types shortly.

Note that *IsWidget* is a subclass of *Continuous*, which we introduced in section 2.6. This ensures that we can use widgets in a way that is similar to signals: Widgets have a current state, which is rendered on screen, and they can be updated in response to events. Recall that any stable type is continuous, and that continuous types are closed under forming product types, sum types, recursive types, and signals. Widget Rattus provides Template Haskell code that automatically generates *Continuous* instance declarations for such types, so we can focus on defining instances of *IsWidget*. For buttons, the instance declaration looks as follows:

$$\begin{array}{l}\textbf{instance } \mathit{IsWidget}\ \mathit{Button}\ \textbf{where}\\ \quad \mathit{mkWidgetNode}\ \mathit{Button}\,\{\mathit{btnContent} = \mathit{val} ::: \_, \mathit{btnClick} = \mathit{click}\} =\\ \quad\quad \mathit{Monomer.button}\ (\mathit{display}\ \mathit{val})\ (\mathit{AppEvent}\ \mathit{click}\ ())\end{array}$$

The Monomer button constructor takes as input the text to be shown on the button and the event that should be produced when clicking the button. For the first argument, we use the current value of the *btnContent* signal and render it as text. For the second argument, we want the event to contain both the channel associated with the button, and the value it produces when clicking the button – in this case (). To this end, we define an *AppEvent* data type to contain exactly this data:

$$\begin{array}{l}\textbf{data } \mathit{AppEvent}\ \textbf{where}\\ \quad \mathit{AppEvent} :: \mathsf{Chan}\ a \to\ a \to \mathit{AppEvent}\end{array}$$

This data type consists of an input a channel of type a and a value of type a. Button click events are of type (), since they do not contain any information.

As shown in Figure 5, Widget Rattus provides functions to make the process of constructing GUI elements simpler. The *mkButton* function is implemented as follows:

$$\begin{array}{l}\mathit{mkButton} :: \mathit{Displayable}\ a \Rightarrow \mathit{Sig}\ a \to C\ \mathit{Button}\\ \mathit{mkButton}\ t = \textbf{do}\ c \leftarrow \mathsf{chan}\\ \qquad\qquad\qquad\quad \mathit{return}\ \mathit{Button}\,\{\mathit{btnContent} = t, \mathit{btnClick} = c\}\end{array}$$

The *mkButton* function only takes a signal as input. The input channel required for *btnClick* is allocated by the call to *chan*. This simple widget only consists of a single signal and a single channel. A more full-featured library would include additional signals (e.g. signals describing colour, styling etc.) and channels (e.g. a channel to indicate that the button received focus).

Widgets can be nested so that they form a tree structure. Vertical stacks provide the simplest example of this nested structure:

$$\begin{array}{l}\textbf{data}\ \mathit{VStack}\ \textbf{where}\\ \quad \mathit{VStack} :: \mathit{IsWidget}\ a \Rightarrow\ \mathit{Sig}\ (\mathit{List}\ a) \rightarrow \mathit{VStack}\end{array}$$

As observed before, the list of widgets that the signal of a stack produces only allows lists of the *same* widget type (e.g. only a list of buttons). To allow widgets of different types, we also have a wrapper type *Widget*, so that we can turn a widget of any type into a widget of type *Widget*:

$$\begin{array}{l}\textbf{data}\ \mathit{Widget}\ \textbf{where}\\ \quad \mathit{Widget} :: \mathit{IsWidget}\ a \Rightarrow\ a \rightarrow\ \mathit{Sig}\ \mathit{Bool} \rightarrow \mathit{Widget}\end{array}$$

Such a wrapper widget type can contain any additional data that applies to all widgets. For this simple example, we include a signal of type *Bool* that is used to determine whether the widget is enabled at any given time. If a *Widget* never needs to be disabled, one can use the *mkWidget* function:

$$\begin{array}{l}\mathit{mkWidget} :: \mathit{IsWidget}\ a \Rightarrow a \rightarrow \mathit{Widget}\\ \mathit{mkWidget}\ w = \mathit{Widget}\ w\ (\mathit{const}\ \mathit{True})\end{array}$$

Finally, we turn to to the implementation of *runApplication*, which takes a computation of type $C\,w$ that produces a widget of type w and runs an application that renders this widget. To this end, *runApplication* uses Monomer's *startApp* function, which requires four components, which we present here in simplified form. First, we have the event type *AppEvent*, which we have introduced above. Second, we have the *AppModel* type, which describes the current state of the application:

$$\begin{array}{l}\textbf{data}\ \mathit{AppModel}\ \textbf{where}\\ \quad \mathit{AppModel} :: \mathit{IsWidget}\ a \Rightarrow\ a \rightarrow\ \mathit{Clock} \rightarrow \mathit{AppModel}\end{array}$$

That is, the state of a Widget Rattus application is solely described by the widget that is to be rendered and a clock. The latter is simply the clock that represents all timers that are currently running, i.e. delayed computations produced by *timer* from Figure 2. The *runApplication* function initialises the *AppModel* data structure with the widget it takes as argument and with the empty clock.

The last two components for Monomer's *startApp* are functions that interact with these two state and event types. The first is a function $\mathit{build} :: \mathit{AppModel} \rightarrow \mathit{WidgetNode}\ \mathit{AppModel}\ \mathit{AppEvent}$ that constructs a new widget node from the current state of the GUI application. This is simply implemented by invoking the *mkWidgetNode* on the widget stored in the state.

The second function is an event handler that is essentially of type $handle :: AppModel \to AppEvent \to AppModel$. The implementation of this function simply updates the widget in the current state $s :: AppModel$ using the data in the event $e :: AppEvent$. For example, if the widget in s contains a signal $v_0 ::: (\theta_0, f_0)$ where $v_0 :: a$ and (θ_0, f_0) is the delayed computation of type $\bigcirc(Sig\ a)$, then we update this signal to $v_1 ::: (\theta_1, f_1)$ as described in section 2.6 if the channel c from the event e is contained in θ_0.

5 Related Work

There is a long history of using the FRP paradigm to implement GUI frameworks in a functional programming language: Notable examples are the Haskell libraries *FranTk* [24] (based on the original *Functional Reactive Animation* framework *Fran* [10]), *Fruit* [8] (based on the arrowized FRP library *Yampa* [22]), and *Threepenny GUI* [2] (based on *Reactive Banana* [1], a traditional FRP library with a carefully selected set of combinators to avoid time leaks). The prominent *Elm* language [9] was initially also implemented as an embedded language in Haskell designed for FRP-based GUI programming but has later abandoned the FRP paradigm in favour of the *Elm Architecture*.

More recently, FRP languages have emerged that use modal types with the goal of avoiding issues of traditional FRP [20,14,15,21,18,16,7,3,4,12,6], namely space/time leaks and causality, while still maintaining its conceptual simplicity, i.e. manipulation of first-class signals using functional programming. The first foray into modal FRP for GUI programming was undertaken by Krishnaswami & Benton [19] using linear types to describe dynamically updating GUIs. However, this language is synchronous, which is at odds with the asynchronous nature of GUI applications. Graulund et al. [12] introduced the $\lambda_{\mathsf{Widget}}$ calculus which also combines modal types and linear types, but its temporal modalities are asynchronous. A crucial difference between Widget Rattus and $\lambda_{\mathsf{Widget}}$ is that the latter uses destructive updates to dynamically change GUI components, e.g. via a `setColor` function. By contrast Widget Rattus uses the core FRP idea of signals to *declaratively describe* the dynamic behaviour of GUI elements. Moreover, to our knowledge Widget Rattus is the first implementation of an asynchronous modal FRP language for GUI programming.

6 Conclusions and Future Work

We have demonstrated how the modal FRP language Async Rattus can support purely functional GUI programming with two mild extensions to the language: first-class channels and continuous types. We consider this as a proof of concept that demonstrates the language's expressiveness for GUI programming. There are several avenues for future research to make it easier to implement GUIs (e.g. by devising a more expressive signal library that incorporates ideas from push-pull FRP [11]) and to further improve runtime efficiency (e.g. by updating nested widgets in-place rather that producing a new full widget tree).

References

1. Apfelmus, H.: Reactive Banana. `https://wiki.haskell.org/Reactive-banana` (2011)
2. Apfelmus, H.: Threepenny GUI. `https://wiki.haskell.org/Threepenny-gui` (2013)
3. Bahr, P., Graulund, C.U., Møgelberg, R.E.: Simply RaTT: A Fitch-style modal calculus for reactive programming without space leaks. PACMPL **3**(ICFP) (2019)
4. Bahr, P., Graulund, C.U., Møgelberg, R.E.: Diamonds are not forever: liveness in reactive programming with guarded recursion. PACMPL **5**(POPL) (2021)
5. Bahr, P., Houlborg, E., Rørdam, G.T.S.: Asynchronous reactive programming with modal types in Haskell. In: Gebser, M., Sergey, I. (eds.) Practical Aspects of Declarative Languages (2024)
6. Bahr, P., Møgelberg, R.E.: Asynchronous modal FRP. Proc. ACM Program. Lang. **7**(ICFP) (Aug 2023)
7. Cave, A., Ferreira, F., Panangaden, P., Pientka, B.: Fair Reactive Programming. In: POPL (2014)
8. Courtney, A., Elliott, C.: Genuinely functional user interfaces. In: Haskell Workshop (2001)
9. Czaplicki, E., Chong, S.: Asynchronous functional reactive programming for GUIs. In: PLDI (2013)
10. Elliott, C., Hudak, P.: Functional reactive animation. In: ICFP (1997)
11. Elliott, C.M.: Push-pull Functional Reactive Programming. In: Symposium on Haskell (2009)
12. Graulund, C.U., Szamozvancev, D., Krishnaswami, N.: Adjoint reactive gui programming. In: FoSSaCS (2021)
13. Houlborg, E., Rørdam, G., Bahr, P., Disch, J.C., Heegaard, A.: WidgetRattus: An asynchronous modal FRP language. `https://hackage.haskell.org/package/WidgetRattus` (2024)
14. Jeffrey, A.: LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs. In: PLPV (2012)
15. Jeltsch, W.: Towards a common categorical semantics for linear-time temporal logic and functional reactive programming. ENTCS **286** (2012)
16. Jeltsch, W.: Temporal logic with "until", functional reactive programming with processes, and concrete process categories. In: PLPV (2013)
17. Kiss, E.: 7GUIs: A GUI programming benchmark. `https://eugenkiss.github.io/7guis/tasks` (2014)
18. Krishnaswami, N.R.: Higher-order Functional Reactive Programming Without Spacetime Leaks. In: ICFP (2013)
19. Krishnaswami, N.R., Benton, N.: A semantic model for graphical user interfaces. In: ICFP (2011)
20. Krishnaswami, N.R., Benton, N.: Ultrametric semantics of reactive programs. In: LICS (2011)
21. Krishnaswami, N.R., Benton, N., Hoffmann, J.: Higher-order functional reactive programming in bounded space. In: POPL (2012)
22. Nilsson, H., Courtney, A., Peterson, J.: Functional reactive programming, continued. In: Workshop on Haskell (2002)
23. Ploeg, A.v.d., Claessen, K.: Practical principled FRP: forget the past, change the future, FRPNow! In: ICFP (2015)
24. Sage, M.: FranTk - a declarative GUI language for Haskell. In: ICFP (2000)
25. Vallarino, F.: Monomer. `https://hackage.haskell.org/package/monomer` (2018)