

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 accommodate *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 use 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 [27]. The fundamental idea is to represent dynamic behaviours using a type of time varying values, called *signals* or *behaviours*. Using Haskell syntax, we can express this idea by defining a type of signals as follows:

type *Sig* *a* = *Time* → *a*

We can then implement combinators that manipulate such signals, which allows us to describe complex dynamic behaviours. For example, we can implement a

mapping function $map :: (a \rightarrow b) \rightarrow Sig\ a \rightarrow Sig\ b$ to apply a function to a signal, or a switching function $switch :: Sig\ a \rightarrow Event\ (Sig\ a) \rightarrow Sig\ a$, which constructs a signal that behaves as the first argument but switches its behaviour to another signal as soon as it arrives as an event at a later time.

However, while conceptually simple and relatively easy 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 deficiency has led to two prominent 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 identifying a carefully restricted set of combinators that is made available to programmers to construct signals [25,26], and one based on modal type systems to keep track of temporal dependencies [17,18,23,21].

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 captured by the following recursive type definition in Haskell notation:

$$\mathbf{data}\ Sig\ a = Cons\ a\ (\bigcirc\ (Sig\ a))$$

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 λ_{Widget} [15] and Async RaTT [8] 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 [7], 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 λ_{Widget} and other related work (section 5).

$$\begin{array}{c}
\frac{\Gamma, \check{\text{cl}}(t) \vdash t :: A}{\Gamma \vdash \text{delay}_{\text{cl}(t)} t :: \bigcirc A} \quad \frac{\check{\not\in} \Gamma' \text{ or } A \text{ stable}}{\Gamma, x :: A, \Gamma' \vdash x :: A} \quad \frac{\Gamma \vdash t :: \Box A}{\Gamma \vdash \text{unbox } t :: A} \quad \frac{\Gamma^\Box \vdash t :: A}{\Gamma \vdash \text{box } t :: \Box A} \\
\\
\frac{\Gamma \vdash s :: \bigcirc A \quad \Gamma \vdash t :: \bigcirc B \quad \check{\not\in} \Gamma'}{\Gamma, \check{\text{cl}}(s) \sqcup \text{cl}(t), \Gamma' \vdash \text{select } st :: \text{Select } AB} \quad \frac{\Gamma \vdash t :: \bigcirc A \quad \check{\not\in} \Gamma'}{\Gamma, \check{\text{cl}}(t), \Gamma' \vdash \text{adv } t :: A} \quad \frac{}{\Gamma \vdash \text{never} :: \bigcirc A} \\
\\
\frac{}{\Gamma \vdash \text{chan} :: C(\text{Chan } A)}^\dagger \quad \frac{\Gamma \vdash t :: \text{Chan } A}{\Gamma \vdash \text{wait } t :: \bigcirc A}^\dagger \quad \frac{\Gamma \vdash t :: A \quad A \text{ continuous}}{\Gamma \vdash \text{promote } t :: \Box A}^\dagger \\
\\
\text{where} \quad \begin{array}{l} \cdot^\Box = \cdot \\ (\Gamma, \check{\not\in})^\Box = \Gamma^\Box \end{array} \quad (\Gamma, x :: A)^\Box = \begin{cases} \Gamma^\Box, x :: A & \text{if } A \text{ stable} \\ \Gamma^\Box & \text{otherwise} \end{cases}
\end{array}$$

Fig. 1: Select typing rules for Async Rattus and its extension Widget Rattus. New typing rules introduced by Widget Rattus are marked by $\dagger$.

The Widget Rattus language and GUI framework is available as a Haskell package [16], which also contains further examples beyond those presented in section 3. Throughout this paper, we use Haskell syntax.

2 From Async Rattus to Widget Rattus

In this section, we give a brief introduction to the Async Rattus language [7] (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

The purpose of Async Rattus is to provide a language that allows us to implement the type of signals, *Sig*, along with a rich library of signal combinators such as *map* and *switch* from section 1, while maintaining crucial operational properties such as avoiding space leaks and ensuring causality.

Async Rattus is an implementation of the *Async RaTT* calculus [8] 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 [8]. In a nutshell, the eager evaluation semantics of Async Rattus ensures that computations that wait for future inputs will be performed as soon as these future inputs arrive. As a consequence these inputs do not need to be kept in memory since they are consumed immediately.

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, and written in ASCII syntax as `0` and `Box`, respectively. 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 thus can be executed at any time. Figure 1 presents the most important typing rules of Async Rattus and its extension Widget Rattus (indicated by $\dagger$). 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 as described by Bahr et al. [7].

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. $\theta = \{c_{\text{keyboard}}, c_{\text{ok_button}}\}$, 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 $\text{cl} :: \bigcirc A \rightarrow \text{Clock}$ and $\text{adv} :: \bigcirc A \rightarrow A$, 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 delay , which for now we can think of as having type $\text{delay} :: \text{Clock} \rightarrow A \rightarrow \bigcirc A$. That is, it takes a clock θ and a computation f producing A and returns a delayed computation (θ, f) that will yield the value of type A once θ ticks. 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:

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

The function *incr* delays the increment operation until the clock $\text{cl}(x)$ associated with x ticks. This schedules the increment to happen when the underlying value of x becomes available, thereby enforcing causality of the system.

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

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

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 $\checkmark_{\text{cl}(t)}$. 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 the part of the context to the left of the tick $\checkmark_{\text{cl}(t)}$, i.e. t must come from the time before its clock ticked. This typing rule encodes the intuition that the presence of the tick $\checkmark_{\text{cl}(t)}$ in the typing context indicates that the clock $\text{cl}(t)$ has ticked in response to an event making the value computed by t available *now*.

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

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

The *advBad* function tries to use *adv* to extract a future value of the delayed computation x without waiting for its clock $\text{cl}(x)$ 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 indicate 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*:

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

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 and delayed computations are only accessed at appropriate times, often in response to user generated input. The following is a typical example of how `adv` and `delay` interact:

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

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 $2 * \text{adv } x$ is type checked in the context $x :: \bigcirc \text{Int}, \checkmark_{\text{cl}(x)}$.

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 $\text{cl}(x)$, as we need the tick $\checkmark_{\text{cl}(x)}$ 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:

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

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 $2 * \text{adv } x$ 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. We call such types *stable types*, and they include all types that cannot carry any temporal dependency. Types of the form $\bigcirc A$ and $A \rightarrow B$ 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. Stable types include all base types like *Int* and *Bool* as well as all algebraic data types and record types that in turn only contain stable types, e.g. lists of integers.

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:

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

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

$$\begin{aligned} \text{mapLater} &:: \Box (a \rightarrow b) \rightarrow \bigcirc a \rightarrow \bigcirc b \\ \text{mapLater } f \ x &= \text{delay}_{\text{cl}(x)} \ (\text{unbox } f \ (\text{adv } x)) \end{aligned}$$

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

2.4 Signals

Signals can be defined in Async Rattus by the following definition, which uses infix operator `:::` as the constructor:

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

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

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

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

Similarly to `mapLater`, the type system of Async Rattus prevents us from implementing a `map` function that takes an unboxed function of type $a \rightarrow b$

as argument. Such an unboxed function would no longer be in scope under the `delay`, where it is needed for the recursive call. This restriction, enforced by the type system, is crucial for avoiding space leaks. A function f of type $a \rightarrow b$ is allowed to have values of non-stable types such as signals in its closure. Such values would need to be kept in memory indefinitely so that we can apply the function f at any point in time in the future as required by the `map` function.

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

```
const :: a → Sig a
const x = x :: never
```

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:

```
addBad :: Ⓞ Int → Ⓞ Int → Ⓞ Int
addBad x y = delay (adv x + adv y)
```

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 $\text{cl}(s) \sqcup \text{cl}(t)$ is in the context. In return, `select` produces a value of type `Select A B` defined as follows:

```
data Select a b = Fst a (Ⓞ b) | Snd (Ⓞ a) b | Both a b
```

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

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

The signal `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 $\text{cl}(ys)$ 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:

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

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:

```
respond :: C SimpleButton
respond = do
  c ← chan
  let sig = switch (const "OK")
                (mapLater (box (λ() → const "Clicked")) (wait c))
  return (Button sig c)
```

The dynamic behaviour is achieved by the *switch* function from section 2.5. In turn, *mapLater* is used to turn the delayed computation `wait c :: ()` into a delayed signal of type `○(Sig Text)`.

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

```
chanSig :: Chan a → ○ (Sig a)
chanSig c = delay (adv (wait c) :: chanSig c)
```

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 $\text{Sig } A$ for basic types A . A value of type $\text{Sig } A$ 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 $v_0, v_1, \dots$, 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 $\text{Sig } \text{Widget}$, 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:

```
data Buttons = Buttons (Sig Color) (Sig (List 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. For each continuous type A , the runtime system of Widget Rattus has a clock function $\text{clock} :: A \rightarrow \text{Clock}$ and an update function $\text{update} :: \text{InputValue} \rightarrow A \rightarrow A$. If we have a value v of a continuous type A , and we receive an input i on a channel $c \in \text{clock}(v)$, the value v is updated to $\text{update } i \ v :: A$. For example, $\text{Sig } \text{Int}$ is a continuous type: An integer signal value $v_0 :: (\theta_0, f_0)$ has the clock θ_0 and is updated once an input i is received on a channel $c \in \theta_0$ by performing the delayed computation f_0 which yields a new signal $v_1 :: (\theta_1, f_1)$.

Continuous types allow us to deal with signals of type $\text{Sig } A$ 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. $\text{clock}(v_0) \sqcup \theta_0$. If we receive input i on channel $c \in \text{clock}(v_0) \sqcup \theta_0$, there are two possible outcomes for the updated signal. If $c \in \theta_0$, then the new signal is produced by performing the delayed computation f_0 , which produces a new signal $v_1 :: (\theta_1, f_1)$. Otherwise, if $c \notin \theta_0$ (and thus $c \in \text{clock}(v_0)$), then the new signal is $\text{update } i \ v_0 :: (\theta_0, f_0)$.

Any basic type is a continuous type, and – like stable types – continuous types are closed under product, sum 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 $\text{Sig } A$. 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

```

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

```

Fig. 2: Signal combinator library.

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

To demonstrate the expressiveness of Widget Rattus, we have implemented the first four tasks from Kiss’s *7GUIs* benchmark [20] as well as a simple calculator. The full implementation of these example GUIs can be found in the Widget Rattus package [16]. In this section, we take a closer look at two of these examples: a counter and a timer. 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 [7]. But we have added a few combinators that have become handy for implementing the GUI examples, and we discuss their implementation in this section. The implementation of widgets and their constructor functions will be covered in more detail in section 4.

Since Widget Rattus uses eager evaluation, it also uses strict variants of common data types, e.g. strict pair types $A \times B$ instead of the non-strict version (A, B) and the strict list type $\text{List } A$ instead of the non-strict version $[B]$ normally used in Haskell. We will encounter these types in the examples below.

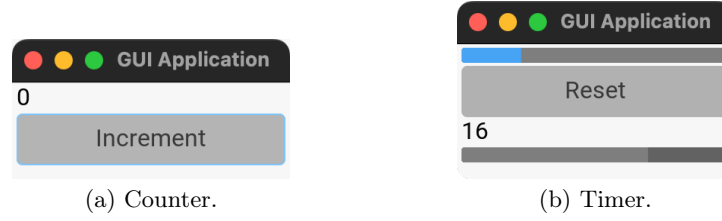


Fig. 3: Example GUIs.

```

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

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 [20]. 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 give our button a constant signal with the *Text* value "Increment".

To add functionality to a button, we use the *btnOnClickSig* function. This function takes as input a button and returns a signal that ticks – producing a unit value – whenever the button is pressed. This signal can be turned into an 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 $v_1, v_2, \dots$ is thus transformed into a signal producing values $x, f\ x\ v_1, f\ (f\ x\ v_1)\ v_2, \dots$, 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

```

mkButton :: Displayable a ⇒ Sig a → C Button
btnOnClickSig :: Button →  $\bigcirc$  (Sig ())
mkTextField :: Text → C TextField
textFieldOnInputSig :: TextField →  $\bigcirc$  (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

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* × *Button*. That means, the last line of *counter* can be more compactly written as *mkConstVStack (lbl × 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 [20]. 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

```

everySecondSig ::  $\bigcirc$  (Sig ())
everySecondSig = mkSig (timer 1000000)

mkTimerSig :: Int  $\times$  Int  $\rightarrow$  Sig (Int  $\times$  Int)
mkTimerSig startState = stop (box ( $\lambda(n \times nMax) \rightarrow n \geq nMax$ )) timerSig
  where timerSig :: Sig (Int  $\times$  Int)
        timerSig = scanAwait (box ( $\lambda(n \times nMax) () \rightarrow (n + 1) \times nMax$ ))
                           startState everySecondSig

reset :: ()  $\rightarrow$  (Int  $\times$  Int)  $\rightarrow$  (Int  $\times$  Int)
reset () (n  $\times$  nMax) = (0  $\times$  nMax)

setMax :: Int  $\rightarrow$  (Int  $\times$  Int)  $\rightarrow$  (Int  $\times$  Int)
setMax nMax' (n  $\times$  nMax) = min n nMax'  $\times$  nMax'

window :: C VStack
window = do
  slider  $\leftarrow$  mkSlider 50 (const 1) (const 100)
  resetBtn  $\leftarrow$  mkButton (const ("Reset" :: Text))

  let resetSig = mapAwait (box reset) (btnOnClickSig resetBtn)
  let setMaxSig = mapAwait (box setMax) (future (sldCurr slider))

  let inputSig ::  $\bigcirc$  (Sig (Int  $\times$  Int  $\rightarrow$  Int  $\times$  Int))
      = interleave (box ( $\circ$ )) resetSig setMaxSig
  let inputSig' ::  $\bigcirc$  (Sig (Int  $\times$  Int  $\rightarrow$  Sig (Int  $\times$  Int)))
      = mapAwait (box ( $\lambda f \rightarrow mkTimerSig \circ f$ )) inputSig

  let currentMax = current (sldCurr slider)
  let counterSig = switchR (mkTimerSig (0  $\times$  currentMax)) inputSig'

  let currentSig = map (box fst') counterSig
  let maxSig = map (box snd') counterSig

  label  $\leftarrow$  mkLabel currentSig
  pb  $\leftarrow$  mkProgressBar (const 0) maxSig currentSig
  mkConstVStack (slider  $\times$  resetBtn  $\times$  label  $\times$  pb)

```

Fig. 6: Timer GUI implementation

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*.

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:

$$\begin{aligned} \text{stop} &:: \square (a \rightarrow \text{Bool}) \rightarrow \text{Sig } a \rightarrow \text{Sig } a \\ \text{stop } f (x :: xs) &= \text{if unbox } f \ x \ \text{then } \text{const } x \\ &\quad \text{else } x :: \text{delay } (\text{stop } f \ (\text{adv } xs)) \end{aligned}$$

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

$$\begin{aligned} xs &: 1\ 2\ 3\ 4\ 5\ 6\ \dots \\ \text{stop } (\text{box } (\geq 3)) \ xs &: 1\ 2\ 3 \end{aligned}$$

To illustrate a modular programming style, we have used two separate combinators – *scanAwait* and *stop* – to first define a signal and then modify it so that it stops at a certain point. Of course, we could have also used a combinator that combines the functionality of *scanAwait* and *stop*.

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 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 $\bigcirc(\text{Sig}((\text{Int} \times \text{Int}) \rightarrow (\text{Int} \times \text{Int})))$. 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.

$$\begin{aligned} xs &: 1\ 3\ 5\ 3\ 1\ 3\ \dots \\ ys &: 0\ 2\ 4\ \dots \\ \text{interleave } (\text{box } (+)) \ xs \ ys &: 1\ 3\ 2\ 5\ 7\ 1\ 3\ \dots \end{aligned}$$

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

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)	(1,10)	(2,10)	(0,10)

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*:

$$\begin{aligned}
& \text{switchR} :: \text{Stable } a \Rightarrow \text{Sig } a \rightarrow \bigcirc (\text{Sig } (a \rightarrow \text{Sig } a)) \rightarrow \text{Sig } a \\
& \text{switchR } xs \text{ } ys = \text{switchS } xs \\
& \quad (\text{delay } (\text{let } step :: ys' = \text{adv } ys \text{ in } \lambda x \rightarrow \text{switchR } (step \ x) \ ys'))
\end{aligned}$$

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* → *Sig a* is received 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* ($0 \times \text{currentMax}$), i.e. it simply ticks up every second. This signal then dynamically switches according to the signal *inputSig'*, which 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 (1×10) and *counterSig* switches its behaviour to reflect *mkTimerSig* (1×10) . This increments to (2×10) 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 *fst'* :: *a* × *b* → *a* and *snd'* :: *a* × *b* → *b* of the strict pair

type. 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* [28], a Haskell library for writing GUI applications using pure functional code in a style pioneered by Elm [11].

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 displayable content and an input channel:

```
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:

```
class Continuous a => IsWidget a where
  mkWidgetNode :: a -> Monomer.WidgetNode AppModel AppEvent
```

The *mkWidgetNode* method translates a Widget 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:

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

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:

```
data AppEvent where
  AppEvent ::  $\text{Chan } a \rightarrow a \rightarrow \text{AppEvent}$ 
```

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:

```
mkButton :: Displayable a  $\Rightarrow$  Sig a  $\rightarrow$  C Button
mkButton t = do c  $\leftarrow$  chan
               return Button { btnContent = t, btnClick = c }
```

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:

```
data VStack where
  VStack :: IsWidget a  $\Rightarrow$  Sig (List a)  $\rightarrow$  VStack
```

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*:

data Widget where

Widget :: *IsWidget* *a* $\Rightarrow$ *a* $\rightarrow$ *Sig Bool* $\rightarrow$ *Widget*

Such a wrapper widget type can contain any additional data that applies to all widgets. For our simple library, 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:

mkWidget :: *IsWidget* *a* $\Rightarrow$ *a* $\rightarrow$ *Widget*
mkWidget *w* = *Widget* *w* (*const True*)

Finally, we turn 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:

data AppModel where

AppModel :: *IsWidget* *a* $\Rightarrow$ *a* $\rightarrow$ *Clock* $\rightarrow$ *AppModel*

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 the state and event types. The first is a function that constructs a new widget node from the current state of the GUI application:

build :: *AppModel* $\rightarrow$ *WidgetNode* *AppModel* *AppEvent*
build (*AppModel* *w cl*) = *mkWidgetNode* *w*

The second function is an event handler that updates the model. Below we present a simplified version that ignores timers (the *cl* component of the model):

handle :: *AppModel* $\rightarrow$ *AppEvent* $\rightarrow$ *AppModel*
handle (*AppModel* *w cl*) (*AppEvent* *c i*) =
 if *c* $\in$ *clock w* then *AppModel* (*update i w*) *cl* else *AppModel* *w cl*

Since the widget *w* is of a continuous type, it can be updated using the clock and *update* functions as described in section 2.6.

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 [27] (based on the original *Functional Reactive Animation* framework *Fran* [12]), *Fruit* [10] (based on the arrowized FRP library *Yampa* [25]), 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 [11] 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 [24,17,18,23,21,19,9,5,6,15,8], 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 [22] using linear types to describe dynamically updating GUIs. However, this language is synchronous, which is at odds with the asynchronous nature of GUI applications and thus leads to inefficiencies. Graulund et al. [15] introduced the λ_{Widget} calculus which also combines modal types and linear types, but its temporal modalities are asynchronous. A crucial difference between Widget Rattus and λ_{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.

To date, most mainstream GUI frameworks are still based on an object-oriented model with a shared mutable state at its core. Nonetheless, recent developments strive towards APIs that encourage a more declarative programming style that reduces the reliance on callbacks to manipulate the state such as in Apple’s *SwiftUI* [4] and Google’s *Jackpack Compose* [14] libraries. However, the ideas of FRP have been adopted by Apple’s *Combine* [3] library, which allows programmers to construct and manipulate signals for SwiftUI GUIs using higher-order functions, not unlike the *Sig* type we presented in this paper.

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 work 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 [13]) and to further improve runtime efficiency (e.g. by updating nested widgets in-place rather than producing a new full widget tree). Finally, we conjecture that the type soundness result of the Async RaTT calculus [8], which undergirds Async Rattus, can be extended to account for the two language extensions presented here, so that we can formally establish the same operational guarantees: productivity, causality, and absence of implicit space leaks.

References

1. Apfeldmus, H.: Reactive Banana. <https://wiki.haskell.org/Reactive-banana> (2011)
2. Apfeldmus, H.: Threepenny GUI. <https://wiki.haskell.org/Threepenny-gui> (2013)
3. Apple Inc.: Combine. <https://developer.apple.com/documentation/Combine>
4. Apple Inc.: SwiftUI. <https://developer.apple.com/documentation/SwiftUI>
5. Bahr, P., Graulund, C.U., Møgelberg, R.E.: Simply RaTT: A Fitch-style Modal Calculus for Reactive Programming Without Space Leaks. *Proc. ACM Program. Lang.* **3**(ICFP), 109:1–109:27 (2019)
6. Bahr, P., Graulund, C.U., Møgelberg, R.E.: Diamonds are not forever: liveness in reactive programming with guarded recursion. *Proceedings of the ACM on Programming Languages* **5**(POPL), 2:1–2:28 (2021)
7. 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)
8. Bahr, P., Møgelberg, R.E.: Asynchronous Modal FRP. *Proceedings of the ACM on Programming Languages* **7**(ICFP), 205:476–205:510 (2023)
9. Cave, A., Ferreira, F., Panangaden, P., Pientka, B.: Fair Reactive Programming. In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. pp. 361–372. ACM (2014)
10. Courtney, A., Elliott, C.: Genuinely functional user interfaces. In: *Haskell Workshop* (2001)
11. Czaplicki, E., Chong, S.: Asynchronous functional reactive programming for GUIs. *ACM SIGPLAN Notices* **48**(6), 411–422 (2013)
12. Elliott, C., Hudak, P.: Functional reactive animation. In: *Proceedings of the second ACM SIGPLAN International Conference on Functional Programming*. pp. 263–273. Association for Computing Machinery (1997)
13. Elliott, C.M.: Push-pull Functional Reactive Programming. In: *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*. pp. 25–36. ACM (2009)
14. Google: Jetpack Compose. <https://developer.android.com/compose>
15. Graulund, C.U., Szamozvancev, D., Krishnaswami, N.: Adjoint Reactive GUI Programming. In: Kiefer, S., Tasson, C. (eds.) *Foundations of Software Science and Computation Structures. Lecture Notes in Computer Science*, Springer International Publishing (2021)
16. Houlborg, E., Rørdam, G., Bahr, P., Disch, J.C., Heegaard, A.: WidgetRattus. <https://hackage.haskell.org/package/WidgetRattus> (2024)
17. Jeffrey, A.: 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*. ACM (2012)
18. Jeltsch, W.: Towards a Common Categorical Semantics for Linear-Time Temporal Logic and Functional Reactive Programming. *Electronic Notes in Theoretical Computer Science* **286**, 229–242 (2012)
19. Jeltsch, W.: Temporal logic with "Until", functional reactive programming with processes, and concrete process categories. In: *Proceedings of the 7th workshop on Programming languages meets program verification*. pp. 69–78. Association for Computing Machinery (2013)
20. Kiss, E.: 7GUIs: A GUI programming benchmark. <https://eugenkiss.github.io/7guis/tasks> (2014)

21. Krishnaswami, N.R.: Higher-order Functional Reactive Programming Without Spacetime Leaks. In: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming. pp. 221–232. ACM (2013)
22. Krishnaswami, N.R., Benton, N.: A semantic model for graphical user interfaces. In: Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming. pp. 45–57. Association for Computing Machinery (2011)
23. Krishnaswami, N.R., Benton, N., Hoffmann, J.: Higher-order Functional Reactive Programming in Bounded Space. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 45–58. ACM (2012)
24. Krishnaswami, N., Benton, N.: Ultrametric Semantics of Reactive Programs. In: 26th Annual IEEE Symposium on Logic in Computer Science (LICS). pp. 257–266 (2011)
25. Nilsson, H., Courtney, A., Peterson, J.: Functional reactive programming, continued. In: Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell. pp. 51–64. ACM (2002)
26. Ploeg, A.v.d., Claessen, K.: Practical principled FRP: forget the past, change the future, FRPNow! In: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming. pp. 302–314. Association for Computing Machinery (2015)
27. Sage, M.: FranTk - a declarative GUI language for Haskell. SIGPLAN Not. **35**(9), 106–117 (2000)
28. Vallarino, F.: Monomer. <https://hackage.haskell.org/package/monomer> (2018)