Asynchronous FRP for Implementing GUIs

STADS code: KISPECI1SE

Jean-Claude S. Disch
jdis@itu.dk

Asger L. Heegaard
ahee@itu.dk

Supervisor: Patrick Bahr
paba@itu.dk

June 3rd 2024

Abstract

Interactive applications with Graphical User Interfaces (GUIs) are fundamental to modern software systems. GUIs demand efficient management of asynchronous operations to provide seamless user experiences. Traditional imperative programming languages often face challenges in handling concurrency, such as race conditions and deadlocks. Functional Reactive Programming (FRP) offers a robust alternative. Async Rattus [2], a research language embedded in Haskell, leverages FRP principles and presents a way to avoid space leaks, preserve causality and ensure productivity, while addressing these concurrency issues.

This thesis investigates the implementation and effectiveness of Async Rattus for developing GUIs. We explore the language's core concepts and sophisticated type system designed to manage asynchronous operations through modal types. A comparative study with other declarative frameworks, specifically React and Monomer, highlights the similarities and differences of using Async Rattus for GUI development.

Given the absence of an existing GUI library for Async Rattus, we develop a new library and evaluate its performance using the 7GUIs: A GUI Programming Benchmark [8]. Our implementation demonstrates how Async Rattus can handle typical concurrency-related tasks and showcases its potential for building interactive applications. The findings contribute to a deeper understanding of Async Rattus' capabilities in GUI development. We conclude that Async Rattus can compactly write performant GUIs of some complexity. However, reasoning about data flow in Async Rattus GUIs is harder than model based declarative frameworks like Monomer and React.

Contents

1 Introduction

Interactive applications, especially those with *Graphical User Interfaces* (GUIs), form a cornerstone of contemporary software systems. The user experience these applications provide relies on robust asynchronous programming paradigms, that manage concurrent tasks effectively. Traditional development of such applications often employs imperative programming languages. Imperative languages, while powerful, can be prone to errors and challenging to manage due to the complexity of shared state management [2].

Developing interactive applications with imperative languages frequently involves handling issues such as *race conditions*, *deadlocks*, and other concurrency-related bugs[2]. These challenges necessitate advanced approaches to manage concurrency and synchronization Effectively. *Functional Reactive Programming* (FRP) has emerged as a promising paradigm to address these issues, offering a declarative and robust approach to handle asynchronous operations.

This thesis explores the implementation and effectiveness of *Async Rattus*, a functional programming language embedded in *Haskell*, proposed by Bahr, Houlborg, and Rørdam (2023) [2]. This investigation will utilize an experimental extension of Async Rattus that provides utilities for handling user input. Async Rattus leverages a sophisticated type system and specialized modal types to manage asynchronous operations, aiming to avoid the pitfalls of concurrency and synchronization in interactive applications[2].

Our objective is to investigate the core concepts and type system of Async Rattus to demonstrate its utility in creating GUIs. This exploration involves a comparison between existing frameworks for GUI development, focusing on *React*, a widely-used JavaScript library, and *Monomer*, a Haskell-based GUI library. React and Monomer will be used when discussing the implementation of the Async Rattus GUI library.

Given that Async Rattus is a research language with no existing GUI library, this thesis aims to showcase how such a library can be implemented. The Async Rattus GUI library is built using Monomer's infrastructure. We will demonstrate the library's effectiveness in handling typical concurrency-related tasks essential for creating interactive applications. Our approach includes solving some of the GUIs presented in the *7GUIs: A GUI Programming Benchmark* [8], using the challenges presented by the benchmarks to form the features of the Async Rattus GUI library.

By the end of this thesis, we aim to provide a comprehensive evaluation of Async Rattus for GUI development. This includes offering valuable insights into the programming language's strengths and limitations, and comparing GUI development in Async Rattus to React and Monomer.

2 GUI Frameworks

Before investigating the intricacies of FRP and Async Rattus, this thesis will examine two frameworks that exemplify diverse approaches to GUI development: React and Monomer. To articulate the differences and capabilities of React and Monomer, we will reference 7GUIs: A GUI Programming Benchmark [8]. Specifically, our discussion will be anchored by an in-depth look at the benchmark's first challenge which is an incrementing counter [8] as showcased in Figure 1.

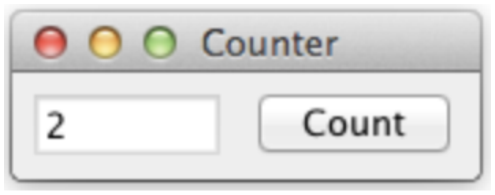

Figure 1: Image of the counter Benchmark from 7GUIs: A GUI Programming Benchmark [8].

React is a declarative JavaScript library widely recognized for its efficiency and flexibility in building interactive GUIs[11]. Developed by Facebook, React has become a cornerstone in web development, offering a component-based architecture that emphasizes reusability and the reactive updating of the GUI in response to data changes[11].

Monomer is a Haskell-based library that, similarly to React, is inherently declarative. Monomer leverages Haskell's powerful type system and functional purity, providing a robust framework for constructing GUIs in a purely functional language [14].

2.1 React

React provides a novel programming model centered around a declarative approach to GUI development. At its core, React abstracts away direct manipulation of the *Document Object Model* (DOM), instead favoring a component-based architecture [9]. The DOM is a structured representation of the HTML elements comprising a webpage's GUI [9]. The DOM constructs a tree-like data structure where each node corresponds to a GUI element within the webpage [9]. This design promotes reusability and encapsulation, allowing developers to think about GUIs in isolated, manageable pieces.

React's efficiency in rendering and updating the GUI stems from its use of a *virtual DOM*, a lightweight representation of the actual DOM [9]. This approach enables React to minimize direct interactions with the browser's DOM, addressing performance bottlenecks and enhancing the user experience[9].

In React, each component maintains its state and properties. The state represents data that can change over time, influencing how the component is rendered and behaves [9]. *Properties* (props) are immutable data passed to a component, defining how that component should appear or behave [10].

React's declarative approach allows developers to focus on defining what the UI should look like for a given state, rather than how to update the UI when the state changes. This means developers describe the desired outcome, and React takes care of the details of updating the DOM to match the current state. For example, if a component's state changes, React will automatically re-render the component to reflect those changes. To exemplify some of the core functionalites of React, consider the following counter component:

```
function Counter() {
const [count, setCount] = useState(0);

function incrementCount = {
setCount(prevCount => prevCount + 1);
};

return (
<div>
<p>Count: {count}</p>
<button onClick={incrementCount}>Increment</button>
</div>
);
}
```

Note that boilerplate code such as import statements has been removed. A key concept within the React framework are *hooks*. React hooks allow functional components to manage state and access other React features [11]. The *useState* hook, as seen in the above code, initializes and manages the components' state:

```
const [count, setCount] = useState(0);
```

The *useState* hook returns a pair: the current state value *count*, and a function that updates it *setCount*. Whenever *setCount* is called, React will re-render the component with the updated *count* value. *setCount* can be called directly to update the state, or as can be see in the above code, can be passed a function to determine the new state based on the previous state.

Additionally, React utilizes callback functions to manage responses to user interactions. In React callback functions are passed to functions or components and used for handling events and effects. For example, a user action could affect a button component and trigger a callback function which only then returns a result. In the above code, the *incrementCount* function exemplifies this:

```
const incrementCount = () => {
setCount(prevCount => prevCount + 1);
};
```

Within the component's return statement, the *onClick* attribute of the button is set to the *incrementCount* callback function.

```
<button onClick={incrementCount}>Increment</button>
```

By tying the *incrementCount* function to the *onClick* button event, React ensures that clicking the button calls the *incrementCount* function. This function in turn calls the *setCount* setter function, which updates the current state based on the previous state.

2.2 Monomer

Monomer is a GUI library for writing native Haskell applications using pure functional code. Monomer is inspired by the *Elm architecture*, keeping the application state in a model that can be updated by user generated events [4][14]. Monomer aims to be an accessible and extensible library for writing GUIs [14].

In Monomer each part of the GUI, for example a button or textfield, is called a *widget*. Similarly to React, Monomer keeps every GUI element in a tree structure [14]. Unlike React which utilizes *css* to define its GUI layout, the placement of GUI elements in Monomer applications is determined by their position in the tree. So some care has to be taken by the programmer to ensure that the layout is as expected when constructing the tree [13].

Below is one possible Monomer implementation of the Counter GUI, where the type annotations for the *buildUI* and *handleEvent* functions have been left out for the sake of brevity. This version is made using *lenses* as it improves readability. A lens is a first class citizen that can be used to reference subparts of a data type[1]. One example is the _2 lens which focuses on the second part of a pair[1]. Monomer also provides widgets that do not require the use of lenses [13].

```haskell
newtype AppModel =
AppModel {
    _clickCount :: Int
} deriving (Eq, Show)

data AppEvent =
  AppInit | AppIncrease
  deriving (Eq, Show)

makeLenses 'AppModel

buildUI wenv model = widgetTree where
  widgetTree = hstack [
        label $ "Click count: " <> showt (model ^. clickCount),
        button "Increase count" AppIncrease
        ] `styleBasic` [padding 10]

handleEvent wenv node model evt = case evt of
  AppInit -> []
  AppIncrease -> [Model (model & clickCount +~ 1)]

main :: IO ()
main = do
  startApp model handleEvent buildUI config
  where
    config = [
      appWindowTitle "Benchmark 01 - Counter",
      appWindowIcon "./assets/images/icon.png",
      appTheme darkTheme,
      appFontDef "Regular" "./assets/fonts/Roboto-Regular.ttf",
      appInitEvent AppInit
      ]
    model = AppModel 0
```

In Monomer applications, all mutable state is described in a *Model* data type [13]. Parts of the Model can then be passed to a widget's constructor to create stateful GUI elements. In the example above this Model is called *AppModel* and it contains a single integer value *_clickcount*. *_clickcount* represents the number of times the button widget gets pressed. The *_clickcount* variable is passed as input to the label constructor, in order to make a label that displays the value.

Monomer uses events whenever information needs to be passed between widgets. For this purpose, Monomer uses an algebraic data type, representing the different events that the application's event handler can respond to[13]. The counter example uses the type *AppEvent* to contain the events that the application will handle.

The *AppEvent* type only specifies which events can occur and what information they carry, it does not define any functionality, nor determine when the events are created. The counter GUI defines two events as part of the *AppEvent* type. *AppInit* for when the application launches and *AppIncrease* for whenever the button is pressed.

The functionality of events is described in an event handler, such as the *handleEvent* function above. The event handler must take as input a value e of the *AppEvent* type [13]. *handleEvent* matches each part of the *AppEvent* type to the desired functionality. In the case of the *AppIncrease* event, this means fetching the current value of $\_clickCount$. Then incrementing it and updating the model with the new state.

Monomer's events and event handlers are one part of how it adheres to the Elm architecture [4][14]. User input is translated into events by widgets and the event handler uses this information to update the state of the application [13]. The second part of the Elm architecture, displaying the program state to the user, is handled by the *buildUI* function. The *buildUI* function constructs the *widgetTree* to be displayed. *buildUI* is called whenever an event changes the state of the program's model [13].

The *buildUI* function takes an *AppModel* as input and constructs a *widgetTree* composed of a number of *widgetNodes* each of which are a GUI element [13]. These *widgetNodes* are constructed by passing pieces of the *AppModel* and *AppEvent* types to widget constructors [13]. For example the button constructor take as input an event to raise whenever the button is pressed. In the counter GUI a button for incrementing the counter is constructed by passing the *AppIncrease* event to the button constructor.

Once state, layout, and events are defined, Monomer GUIs are created by calling the *startApp* function. The *startApp* function takes several inputs[13]:

- A model describing the initial state of the GUI.
- An event handler that describes how the model is updated in response to user generated events.
- A function to build the GUI based on the current state of the model.
- Configuration options for things like window size and title.

When the model, events, event handler and *buildUI* function are defined the counter GUI can be started with a call to the *startApp* function. It receives four arguments: An *Appmodel*, the *handleEvent* function, the *buildUI* function and *config* which is defined during the call itself. See line 22 and onwards in the code above.

2.2.1 Comparison

React and Monomer take similar approaches to GUI development. They both leverage the advantages of the declarative programming paradigm and both use a tree structure to represent the GUI elements.

Similarly to React, Monomer's *widgetTree* is optimized not to recalculate the entire GUI in response to every single user input [15]. Monomer applications always call a builder function, like *buildUI* in the example, in response to user input that changes the *AppModel*. This creates a new *widgetTree* that can be recursively merged with the existing *widgetTree* [15].

Still there are notable distinctions between the two, React uses imperative tools such as callbacks, whereas Monomer prioritizes pure functional code. It is also worth noting that React distributes the functionality of events among individual GUI elements, whereas Monomer applications describe all event functionality in a single event handler. The React and Monomer frameworks will be used for comparison with the Async Rattus GUI library in section 6.

With a fundamental understanding of React's and Monomer's approaches to GUI development, we now delve into the realm of FRP. This exploration will investigate core FRP principles necessary to understand Async Rattus and our GUI library.

3 Background

In this section we investigate the foundational principles of *Functional Reactive Programming* (FRP). The FRP paradigm describes pivotal concepts, notably *signals* and *events*, for handling asynchronous data flows using functional programming.

Being able to manage time-varying information smoothly across time is essential for building any GUI. Often when working with GUIs, one part of the GUI needs to affect another. This is exemplified by the counter GUI, where a label needs to change in response to a button press. FRP presents ways to manage these time dependent variables in functional code, by utilizing signals and events.

3.1 Functional Reactive Programming

Elliott & Hudak (1997) first introduced FRP, as a tool for making reactive animations using functional programming [5]. In their paper, Elliot & Hudak (1997) introduced two important concepts for managing time dependent values; *behaviours* and *events*[5].

Behaviours are time-varying values central to the FRP framework. Semantically an $\alpha-$behaviour is a function from time to α [5]. As such an $\alpha-$behaviour associates each point in time with a value of type α. An event represents an occurrence at a singular point in time. Semantically an $\alpha-$event is a non-strict time-value pair[5].

In their paper, Elliot & Hudak (1997) introduce the semantics of behaviours and events, along with functions and combinators to manipulate them[5]. Using these functions, FRP can be used to model complex time dependent applications using functional programming.

This could for example be used to model the counter GUI. Such an application would consist of a button and a label to represent the number of times the button was clicked. The label is best represented by an $Int - Behaviour$ since it has a value at all times that should be displayed to the user. On the other hand, the button should produce an *Event* whenever clicked, that changes the current value of the $Int - Behaviour$.

The button is unlikely to be clicked continuously, and therefore is not well represented by a behaviour, since it does not need a value at all points in time. But at some point in time, when it gets clicked, an event associated with that particular point in time should be created.

The concepts and semantics introduced by Elliot & Hudak (1997) provide a conceptual way to integrate time flow into functional programming. Unfortunately they also introduce a number of issues that must be considered when making concrete implementations.

The ability to look up the value of a behaviour at any time gives vast flexibility when using them. However, this will be prone to implicit *space leaks* when every past point in time has to be associated with a stored value. Additionally the semantics presented by Elliot & Hudak (1997) do not follow *temporal causality.* This is because the semantics do not prevent looking up future values.

Therefore concretising the abstract semantics of Elliot & Hudak (1997) is no simple task. This has led to several implementations of FRP's core ideas in fields ranging from user interfaces through simulation to robotics [6]. This thesis will present how one such implementation, Async Rattus, can be used for the purpose of constructing GUIs. Note that the following section makes extensive reference to the Async Rattus paper [2], as well as the Async Rattus hackage page [7] and individual references will therefore not be provided throughout the section.

3.2 Async Rattus

Async Rattus is an FRP language using *modal logic* to represent time dependent values. In Async Rattus time varying values are called *signals* rather than behaviours and that terminology will be used from this point onwards. Async Rattus is a Haskell embedded language built on the *Async RaTT* framework [2].

Async RaTT uses modal types to ensure *causality*, meaning that any output produced by a reactive program, only depends on previously received or current input[3]. Further, Async RaTT uses aggressive garbage collection between time steps[3]. This prevents data from being kept in memory longer than necessary, therefore Async RaTT prevents implicit space leaks[3]. Async RaTT also has the property of *productivity*, so every recursive call produces a value[3]. Hence ensuring that each computational step terminates [3]. Since Async Rattus is built on the Async RaTT framework, it also has these properties.

3.2.1 Introduction to Async Rattus

As described by Bahr, Houlborg & Rørdam (2023), Async Rattus is shallowly embedded in Haskell. Hence Haskell's extensive library ecosystem is fully available and compatible with Async Rattus. This provides the programmer with access to Haskell's rich tool set. However, Async Rattus differs from Haskell in two major ways:

The first difference, is that Async Rattus is eagerly evaluated. The choice to evaluate eagerly is an important part of how Async Rattus prevents implicit space leaks in an asynchronous environment. Haskell is more error prone to issues such as space leaks due to its lazy evaluation.

The second fundamental difference introduced by Async Rattus is an extension of the type system. Async Rattus introduces two modalities; $\bigcirc$ and $\Box$ called the *later* and *box* modalities. The later modality $\bigcirc$ represents values that will be available in a future time step. While the box modality $\Box$ is used for computations that remain stable across time steps and can be forced when needed. To better describe the advantages of Async Rattus, these modalities will be expanded on below.

3.2.2 Later modality and clocks

The later modality $\bigcirc$ indicates values that are not immediately available but expected in the future, contingent on some event. Thus a value of type $\bigcirc a$ represents a delayed computation that will produce a value of type a in the future. This modality is vital for asynchronous computations where the exact timing of values is unknown, and their arrival may be scattered over time.

The later modality in Async Rattus is intrinsically linked with the concept of *clocks*. Any value v of type $\bigcirc a$ consists of a pair such that $v = (\theta, f)$. Where θ is a clock and f is some computation that will produce a value of type a when executed. The computation f remains dormant until the clock θ *ticks*, signaling the occurrence of the anticipated event. This mechanism ensures that delayed computations adhere to temporal causality. Values are computed only when their time comes, according to the ticking of their associated clocks.

Functions such as $cl :: \bigcirc a \to \text{Clock}$ and $adv :: \bigcirc a \to a$ are pivotal, as they allow the extraction of a clock θ from a delayed value (θ, f) and the advancement of the computation f to produce a value of type a, respectively. To create a delayed value, one can use the *delay* function. Conceptually *delay* can be described as a function $delay :: \text{Clock} \to a \to \bigcirc a$, which takes a clock θ and a value and returns a computation f that will yield the value once the clock ticks. This conceptual representation suffices for the moment, but will be specified when presenting the typing rules for *adv* and *delay*.

An illustrative example of the interplay between the later modality and clocks is the following increment function [2]:

```
incr :: O Int -> O Int
incr x  = delay (adv x + 1)
```

In this function, *incr* delays the increment operation until the clock associated with x ticks. This schedules the increment to happen in the future, thereby preserving the order of operations and the causality of the system.

Clocks are the backbone of Async Rattus's ability to manage asynchronicity. They allow the language to control when computations happen, ensuring that programs behave predictably even in the face of concurrent and delayed operations. The later modality, combined with the clock system, provides a powerful abstraction for programmers to handle values that unfold over time.

By integrating clocks and the later modality into the type system, Async Rattus can uphold the causality principle essential for correct program behavior. To illustrate this, consider the below example of a function for eliminating the later modality that is not causal and therefore does not typecheck:

```
incorrectAdv :: O Int -> Int
incorrectAdv x = adv x
```

In the above example, the *incorrectAdv* function tries to use the *adv* function to extract a future value x of type $\bigcirc Int$ without ensuring the associated clock has ticked. This operation defies Async Rattus' typing rules, as it attempts to access a future value prematurely, disregarding the causality and synchronization requirements determined by the language's type system. This results in the *incorrectAdv* function not typechecking. This is caused by the typing rule for the *adv* function[2]. Typing rules in Async Rattus use $\checkmark_\theta$ to signify a tick on some clock θ. In Async Rattus, a context Γ consists of variable bindings as well as ticks. An example could be:

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

Here the ticks represent the passage of time according to the clocks θ and θ'. The value x will be available one time step before the values y and z. With this in mind, here is the typing rule for *adv*:

$$\frac{\Gamma \vdash x :: \bigcirc A \quad \checkmark \notin \Gamma'}{\Gamma, \checkmark_{\mathrm{cl(x)}}, \Gamma' \vdash \mathrm{adv}\ x :: A}$$

This rule states: Let Γ be a context with variable x of type $\bigcirc A$. If Γ is to the left of a tick $\checkmark_{cl(x)}$ for the clock associated with x, then it is safe to 'advance' x using the *adv* function. Consequently *adv* can be used as an eliminator for the later modality, given the context Γ. The presence of the tick $\checkmark_{\mathrm{cl}(x)}$ in the typing rule signifies that the clock $cl(x)$ has responded to an event making the value of x presently available. The constraint on the resulting context Γ' means that calling *adv* must produce a new tick free context by carrying out the delayed computation. The *adv* function serves to advance the state of a delayed computation, effectively 'unlocking' the value it contains once the corresponding clock has ticked.

Where *adv* is used for eliminating the later modality and executing delayed computations in response to events, the delay function is used to construct delayed computations. To safely construct a delayed computation one needs to associate it with a clock to signify when it will execute, therefore the delay function has the following typing rule [2]:

$$\frac{\Gamma, \checkmark_{\mathrm{cl}(x)} \vdash x :: A}{\Gamma \vdash \mathrm{delay}_{\mathrm{cl}(x)} x :: \bigcirc A}$$

In this rule, if the context Γ produces a value x of type A after a tick on the clock associated with x, then *delay (x)* will have type $\bigcirc A$ in the context Γ. The presence of the tick, $\checkmark_{\mathrm{cl}(x)}$, denotes the passage of one time step according to the clock $\mathrm{cl}(x)$, ensuring that the delay respects the flow of time and the causality of the system. Then, $\Gamma \vdash \mathrm{delay}_{\mathrm{cl}(x)} x :: \bigcirc A$ indicates that in the context Γ, the delayed value x will be of type $\bigcirc A$.

The *delay* function introduced above is used to postpone a computation until a certain point in the future, determined by the ticking of a clock. For this reason the hypothesis of the typing rule requires the presence of a tick on some clock θ, since this will be the prompt for the execution of the delayed computation.

Similarly to the *incorrectAdv* function showcased above, consider the below example of a function which incorrectly tries to use the *delay* function. Note how unlike the notation in the typing rule, the clock is not explicitly described when using the *delay* function, this will be the case going forward.

```
incorrectDelay :: Int -> O Int
incorrectDelay x = delay x
```

In this example, *incorrectDelay* tries to delay an integer x without specifying when the computation will occur. However, as stated the *delay* function requires more than just a value to delay - it also requires a clock that dictates the timing of the delayed computation. The above function will thus result in a type error because the type system cannot determine when the delayed computation should occur.

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. This careful management of time-dependent computations is critical for preventing the premature use of values that have yet to be determined. An example of utilising *delay* and *adv* correctly could be:

```
doubleLater :: O Int -> O Int
doubleLater x = delay (2 * adv x)
```

In this example, *doubleLater* takes a delayed integer and returns a new delayed integer that is twice the original value. This exemplifies how *delay* and *adv* are used to manage the time-varying aspects of computations within Async Rattus. As is the case in the above example, *delay* and *adv* will often be used in conjunction. This is because *delay's* typing rule ensures that there will be a tick on an associated clock, whereas *adv* produces a value from a delayed computation in response to a tick on the associated clock.

3.2.3 The Box Modality and Stable Types

The box modality, denoted by $\Box$, complements the later modality in Async Rattus by providing a mechanism for managing time-independent computations. A boxed type, is a thunk that can be evaluated at any time, either now or in the future. Therefore a value of type $\Box a$ can be forced at any time using the *unbox* function, to produce a value of type a:

```
unbox ::  Box a → a
```

As such *unbox* serves as the eliminator of the box modality. Examples of how to use the box modality and the unbox function will be presented in Section 3.2.4.

General values in Async Rattus can contain references to time-dependent data, such as delayed computations stored in the heap. This could cause issues if references are kept after the time dependent data has been altered or deleted. To prevent this, Async Rattus does not keep arbitrary data across time steps. There are some types that cannot contain references to time-dependent data, in Async Rattus these are called *stable* types.

Stable types form a core part of the Async Rattus type system. 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. General function types are notably not stable by default, additionally any value of type $\bigcirc a$, is by definition time dependent and therefore not stable. To keep such types across time steps we need the box modality $\Box$, whose typing rules ensure that $\Box a$ is kept in a stable context. Stable types are thus crucial for ensuring that certain values remain constant and unaffected by the flow of time within an asynchronous system. In the Async Rattus paper [2], this is formalized by the following type rule:

$$\frac{\Gamma^{\square} \vdash x :: A}{\Gamma \vdash \text{box x} :: \square A}$$

This rule stipulates that if a value x is of a type A in the context $\Gamma^{\square}$, then box x has type $\square A$ within the context Γ. Here $\Gamma^{\square}$ is the context derived from Γ by removing all values that are not stable as well as ticks $\checkmark_{\theta}$ on any clock θ.

The rule's premise is that x will be stable when it occurs in a context without time-dependent values or ticks on any clock. This allows it to be treated as a stable type.

The later $\bigcirc$ and box $\square$ modalities in Async Rattus are used for distinguishing between time-dependent and time-independent computations, enabling robust asynchronous programming. The later modality facilitates deferring computations until specified events occur. The box modality ensures that stable values are kept across time steps, while allowing aggressive garbage collection, which prevents space leaks[3]. Together, these modalities enhance Async Rattus's ability to model asynchronous systems, making it a powerful tool for FRP.

3.2.4 Signals

The later and box modalities described above are a central part of the functions used to construct and manipulate signals in Async Rattus. With these concepts in place it is now possible to describe the higher order Async Rattus functions for manipulating user input. First and foremost, signals in Async Rattus can be constructed using the following definition [2]:

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

As such, signals are recursively defined as a value of type a coupled with a delayed computation producing a new value of type *Sig a*. In addition to the constructor, Async Rattus has numerous functions that can be used to manipulate the values of signals. One example is the *map* function, that applies a function f to all current and future values of a signal [2]:

$$\texttt{map} :: \square(a \to b) \to \text{Sig}\ a \to \text{Sig}\ b$$
$$\texttt{map}\ f\ (x ::: xs) = \texttt{unbox}\ f\ x ::: \texttt{delay}\ (\texttt{map}\ f\ (\texttt{adv}\ xs))$$

Here, *map* takes a function f of type $\square(a \to b)$ and a signal of type *Sig a*. The boxed function f is time-independent due to the $\square$ modality, ensuring that f is stable. This is crucial because f will be used to transform all future values carried by the signal stream *xs*. Had the function instead been defined as [2]:

$$\texttt{mapX} :: (a \to b) \to \text{Sig}\ a \to \text{Sig}\ b$$
$$\texttt{mapX}\ f\ (x ::: xs) = f\ x ::: \texttt{delay}\ (\texttt{mapX}\ {\color{red}f}\ (\texttt{adv}\ xs))$$

The highlighted reference to f would be out of scope. The closure of f may contain time dependent data, so it cannot simply be postponed to a future time step without causing potential space leaks. Therefore any function that is not guarded by the box operator will be garbage collected between time steps and the definition of *mapX* will not pass the type checker.

Note how, in the definition of *map*, the function f cannot be applied directly due to the box operator. While a function is guarded by box the computation will not be carried out. To force the computation we can use the *unbox* function [2]. This eliminates the $\Box$ modality, hence allowing us to apply f to the current value.

The recursive part of the definition, *delay* (*map* f (*adv xs*)), exemplifies how to work with the $\bigcirc$ modality. The expression *adv xs* will advance the signal *xs* to its next state, eliminating the $\bigcirc$ modality and allowing access to the signal's next value. The *map* function will then be recursively called on this advanced signal. The *delay* function ensures that the entire computation is deferred until the next tick of the signal's clock, which is required by the typing rule of *adv.*

This use of *adv* and *delay* is necessary for maintaining the causality of the program, while ensuring that each recursive call is productive. As the recursive call *map f (adv xs)* only occurs once those future values are actually available.

Asynchronous environments also demand the ability to manage concurrent events. Therefore the capacity to effectively handle simultaneous input is essential. For this purpose Async Rattus provides the *Select* primitive, defined by the following typing rule[2]:

$$\frac{\Gamma \vdash s :: \bigcirc A \quad \Gamma \vdash t :: \bigcirc B \quad \checkmark \notin \Gamma'}{\Gamma, \checkmark_{cl(s) \sqcup cl(t)}, \Gamma' \vdash \mathit{select}\ s\ t :: \mathit{Select}\ A\ B}$$

Given a context Γ with two different delayed computations $s :: \bigcirc(A)$ and $t :: \bigcirc(B)$. When Γ is to the left of a tick on the *union clock* $cl(s) \sqcup cl(t)$, it results in a new tick-free context Γ' wherein *select s t* has type *Select A B.* The union clock is defined as the clock that ticks whenever $cl(s)$ or $cl(t)$ ticks. The type for Select A B is defined below [2]:

$$\textbf{data}\ \text{Select a b } = \text{Fst a } (\bigcirc b) \quad | \quad \text{Snd } (\bigcirc a) \text{ b} \quad | \quad \text{Both a b}$$

By returning *Fst*, *Snd* or *Both Select* identifies which part of the union clock has produced a tick, and returns the corresponding value. Thus any delayed computations with potential overlap can be managed without causing any data races or similar issues.

Since *Select* concerns itself with delayed computations in general rather than signals, Async Rattus utilises *select* to define functions that handle cases where two signals might tick at the same time. One example of such is the *switch* function which is implemented as follows [2]:

```
switch :: Sig $a$ → $\bigcirc$(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 *switch* function allows dynamic behavior in asynchronous programming by creating a signal whose behaviour changes in response to clock ticks. Initially, the signal *xs* dictates the output. However, when the delayed signal *d* ticks, *switch* transitions the output to follow *d*. Thus the behaviour of signals can be changed dynamically during program execution. *select* ensures that even if both *xs* and *d* tick simultaneously, the output will properly transition to follow the behaviour of *d*

The *select* and *switch* functions in Async Rattus provide a powerful mechanism for managing concurrent events in an asynchronous environment. They allow for complex behaviors based on multiple sources of input, and to dynamically switch the behaviours of signals based on these inputs. This is vital in a GUI context where events such as clicks and keystrokes might arrive simultaneously.

In summary, Async Rattus implements the concept of FRP with a focus on type safety. The typing rules of the Async RaTT framework that Async Rattus implements ensure that Async Rattus programs are causal, free of implicit space leaks, and that every recursive call is productive. In addition to this, Async Rattus provides the programmer with a number of powerful tools for handling concurrent input and manipulating signals. Async Rattus also allows dynamically changing signal functionality during program execution.

By combining the capabilities provided by Async Rattus with the GUI framework Monomer, the next section will cover a library implementation for constructing compact GUIs.

4 Design and Implementation

In this section, an FRP library implemented in the Haskell embedded language Async Rattus will be presented. While Async Rattus provides powerful tools for type safety in asynchronous environments, it currently does not have any tools for constructing GUIs. For that purpose we have written an Async Rattus library that leverages Monomer's ability to construct GUIs. We will present the overall programming model of our library, and provide a high-level perspective alongside examples throughout the section. As previously stated, the 7GUIs: A GUI Programming Benchmark [8] were utilised as a starting point for the GUI library we have implemented. In Section 5 the benchmarks themselves will be covered in detail during our case studies. Note that we only showcase and discuss the implementation of the first four of the seven benchmarks. The source code for these four benchmarks can be found in Appendix 2.

Throughout the implementation of the GUI library there will be extensive reference to Async Rattus functions that manipulate signals. Not all of these have been covered in detail throughout Section 3 due to space limitations, but an overview of them can be seen in Table 1:

Function Name	**Type**	**Description**
`const`	$a \to \text{Sig } a$	Constructs a constant signal that never updates.
`map`	$\Box(a \to b) \to \text{Sig } a \to \text{Sig } b$	Applies a boxed function to all current and future values of a signal, creating a new signal.
`mapAwait`	$\Box(a \to b) \to \bigcirc(\text{Sig } a) \to \bigcirc(\text{Sig } b)$	Applies a boxed function to a delayed signal, creating a new delayed signal.
`scan`	$\Box(b \to a \to b) \to b \to \text{Sig } a \to \text{Sig } b$	Accumulates results over time by applying a binary function to a signal and an initial value.
`scanAwait`	$\Box(b \to a \to b) \to b \to \bigcirc(\text{Sig } a) \to \text{Sig } b$	Similar to scan, but works on a delayed signal
`interleave`	$\Box(a \to a \to a) \to \bigcirc(\text{Sig } a) \to \bigcirc(\text{Sig } a) \to \bigcirc(\text{Sig } a)$	Combines two signals such that the resulting signal ticks whenever either input signal ticks, applying a function if both tick simultaneously.
`stop*`	$\Box(a \to \text{Bool}) \to \text{Sig } a \to \text{Sig } a$	Stops the progression of a signal when a condition is met.
`addInputSigTF*`	`TextField` $\to \bigcirc(\text{Sig Text}) \to \text{TextField}$	Interleaves a signal with the contents of a textfield, allowing it to update its display based on the signal.
`switch`	$\text{Sig } a \to \bigcirc(\text{Sig } a) \to \text{Sig } a$	Creates a signal that switches its behavior based on a delayed signal.
`switchS`	$\text{Sig } a \to \bigcirc(a \to \text{Sig } a) \to \text{Sig } a$	Like switch but the second signal can depend on the final value of the first signal
`switchB*`	$\bigcirc(\text{Sig } (a \to a)) \to \Box(a \to \text{Sig } a) \to a \to \text{Sig } a$	Variant of switchS that switches signal behaviour whenever the delayed input signal ticks
`zipWith`	$\Box(a \to b \to c) \to \text{Sig } a \to \text{Sig } b \to \text{Sig } c$	Combines two signals by applying a function to their current values whenever either ticks.
`buffer*`	$a \to \text{Sig } a \to \text{Sig } a$	Creates a signal that is always one tick behind the input signal.
`triggerStable*`	$\Box(a \to b \to c) \to c \to \bigcirc(\text{Sig } a) \to \text{Sig } b \to \text{Sig } c$	Creates a signal that updates based on a function applied to two input signals, but only changes its value when the delayed signal ticks.

Table 1: Async Rattus library functions
(Functions marked with $*$ were developed over the course of this project)

This is only meant to provide a quick overview, more detailed explanations of relevant signal functions will be provided in the following sections. A number of the functions in the table above are not part of the Async Rattus library, but have been developed throughout this project.

To effectively grasp the structure of our Async Rattus GUI library, it is crucial to recognize how it incorporates the Monomer library. As described previously Monomer GUIs are constructed by calling the *startApp* function. Our GUI library uses the function *runApplication* to call Monomer's *startApp* function which renders the GUI.

Like Monomer, elements in our GUI library 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*. A button is thus a widget and can be defined as follows:

```
data Button where
    Button :: Displayable a, Stable a) =>
        {btnContent :: !(Sig a), btnClick :: !(Chan ())} -> Button
```

btnContent represents the value displayed on the button. Note that bang is used to force the compiler to evaluate the arguments eagerly, since Async Rattus uses eager evaluation. The channel is Async Rattus' way of handling user input. Any widget that takes user input needs to instantiate a channel of the corresponding type. In the case of a button this is a channel of type *unit*, since the click event does not carry any data.

Channels are defined by the following constructor[7]:

```
chan ::  C (Chan a)
```

chan is a function with a side effect. The function produces a new channel and keeps a record of it. To allow this side effect the function uses the *C monad*. The C monad is an IO monad made for this purpose.

In order to pass the button constructor to Monomer, a widget must fulfill certain criteria based on the corresponding Monomer constructor. We want to create custom data types and pass them to existing Monomer constructors. To do this, we need to ensure that these custom data types can indeed be interpreted as Monomer widgets. For this purpose the *IsWidget* typeclass is defined.

All widgets must be instances of the *IsWidget* typeclass, that requires the implementation of a *mkWidget* function. The *mkWidget* function ensures that a datatype can be interpreted as a Monomer widget. This is because its return type is a *widgetNode*, which is the type Monomer uses for GUI elements. The *IsWidget* type class is a sub class of *Continuous* [7].

The *Continuous* type class is a definition from Async Rattus that represents a stream of data that can be moved forward in time [7]. In Async Rattus any datatype that can change over time must be an instance of *Continuous*. Therefore it is vital that widgets are instances of *Continuous* since they may need to change their state in response to user generated events. Note that any stable type is trivially *Continuous*.

The importance of widgets being *Continuous* is apparent in the *runApplication* function. In order to change the state of a widget it needs to call the Async Rattus *progressAndNext* function, which is part of the *Continuous* class [7]. This function progresses the state of a given widget in response to input on some channel. It returns the new state of the widget alongside the clock that determines when next to advance the widget.

With this in mind, the definition of *IsWidget* and the *IsWidget* instance declaration for a button can be seen below. *continuous ''Button* is Template Haskell for creating a *Continuous* instance of the button type:

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

continuous ''Button

instance IsWidget Button where
      mkWidget :: Button -> Monomer.WidgetNode AppModel AppEvent
      mkWidget Button{btnContent = txt ::: _ , btnClick = click} =
            Monomer.button (display txt) (AppEvent click ())
```

The *mkWidget* function needs to return a Monomer *widgetNode*, which is exactly the type returned by Monomer's widget constructors. So as stated above the *mkWidget* functions must call the constructor for a Monomer widget.

For example, the Monomer button constructor takes as input some text and an event. In order to call the Monomer button constructor, we use the current value of the *btnContent* signal as text input. The event input is constructed from the *btnClick* channel. To convert channels into events, we define the *AppEvent* data type:

```
data AppEvent where
     AppEvent :: !(Chan a) -> !a -> AppEvent
```

The AppEvent data type takes as input a channel of type *a* and a value of type *a*. Button click events are of type *unit*, since they do not contain any information. *btnClick* is a channel of type *unit* and by calling *AppEvent click ()* an *AppEvent* is constructed and passed to Monomer.

To make the process of constructing GUI elements simpler, the GUI library provides some helper functions, such as *mkButton*:

```
mkButton :: (Displayable a, Stable a) => Sig a -> C Button
mkButton t = do
    c <- chan
    return Button{btnContent = t, btnClick = c}
```

The *mkButton* function only takes a signal as input. The input channel required for *btnClick* is constructed and assigned by the call to *mkButton*. It is worth noting that *mkButton* operates within the C monad, since it works with IO operations in the form of channels. Hence *mkButton* returns C Button.

With the *IsWidget* typeclass and constructors in place it is possible to instantiate widgets and pass them to the *runApplication* function. However, most widgets represent only a single GUI element, so it is necessary to have a container widget when showing multiple elements on screen. Similarly to Monomer, horizontal and vertical *stacks* of widgets are provided for this purpose. A stack is a widget consisting of a list of other widgets. In Async Rattus we define a vertical stack by:

```
data VStack = VStack {vGrp :: !(Sig (List Widget))}
```

Since stacks should be able to contain multiple different widget types it is defined as a signal of a list of *Widgets*. *Widget* is itself a data type made from any instance of the *IsWidget* typeclass and a signal of type *Bool*.

```
data Widget where
     Widget :: IsWidget a => !a -> !(Sig Bool) -> Widget
```

Being able to enable and disable widgets becomes significant in some of the later benchmarks, but is in fact part of the *Widget* constructor, as can be seen above. If a *Widget* never needs to be disabled, one can use the *enabledWidget* function:

```
enabledWidget :: IsWidget a => a -> Widget
enabledWidget w = Widget w (AsyncRattus.Signal.const True)
```

enabledWidget takes as input some type *a* that is an instance of *IsWidget* and returns a *Widget*. Since we never want to disable such widgets, we can use the Async Rattus *const* function to pass it an unchanging signal of *True* and create a *Widget* that is enabled for display.

The GUI library is not meant to be functionally complete and does not implement every Monomer widget. But a number of different widgets have been implemented with the goal of creating the first four benchmarks. The implemented widgets are shown with a short description in the Table 2:

Widget Name	**Fields**	**Description**
`Button`	*btnContent* :: Sig a *btnClick* :: Chan ()	A button with a content signal, supporting any instances of displayable, and a channel registering clicks
`TextField`	*tfContent* :: Sig Text *tfInput* :: Chan Text	A textfield widget with a text signal and an input channel.
`Label`	*labText* :: Sig a	A label widget displaying a signal of a stable displayable instance.
`HStack`	*hGrp* :: Sig (List Widget)	A horizontal stack of widgets, represented by a signal of Widget lists.
`VStack`	*vGrp* :: Sig (List Widget)	A vertical stack of widgets, represented by a signal of Widget lists.
`TextDropdown`	*tddCurr* :: Sig Text *tddEvent* :: Chan Text *tddList* :: Sig (List Text)	A dropdown menu with a current text signal, channel for onChange events, and a list of options.
`Popup`	*popCurr* :: Sig Bool *popEvent* :: Chan Bool *popChild* :: Sig Widget	A popup widget with a signal determining visibility, event channel, and a child widget signal.
`Slider`	*sldCurr* :: Sig Int *sldEvent* :: Chan Int *sldMin* :: Sig Int *sldMax* :: Sig Int	A slider widget with signals for current value, minimum, and maximum and a channel for raising events when changed.
`Widget`	IsWidget a => a Sig Bool	General Widget constructor where a satisfies IsWidget. Boolean signal used for enabling/disabling the widget

Table 2: *Widgets* in the GUI Library

This is again to provide an overview of the widgets currently supported by our GUI library and used throughout the case studies. The following sections will provide in-depth explanations for relevant widgets.

5 Case Studies

Similar to Section 2, we have used 7GUIS: A GUI Programming Benchmark [8] to inform the features needed in our library, and assess what its capable of. This section will cover the first four of the seven benchmarks as well as a calculator GUI. It will detail the code and showcase how the library was adapted to overcome issues presented by the different cases.

5.1 Counter

The first benchmark is a simple test to verify that the library provides a toolkit for constructing user interfaces. This is the same GUI that was presented in Section 2. Figure 2 shows how one such benchmark could look:

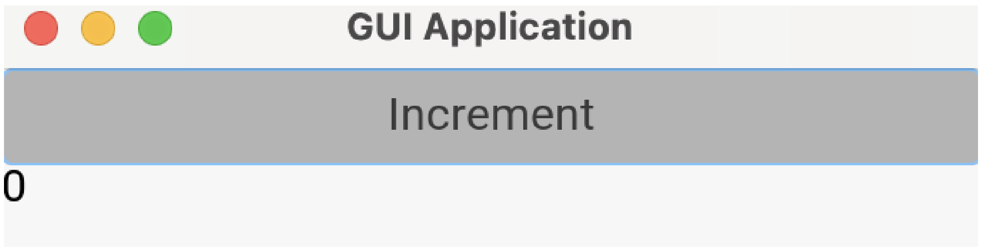

Figure 2: Screenshot of the counter benchmark:
With an Increment button and a label displaying the current value

As described in Section 4, GUIs in our library are made by passing a widget, typically a stack containing more widgets, to the *runApplication* function. A stack describing the counter GUI can be constructed as follows:

```
benchmark1 :: C VStack
benchmark1 = do
    btn <- mkButton (const ("Increment" :: Text))
    let sig = btnOnClickSig btn
    let sig' = scanAwait (box (\n _ -> n + 1 :: Int)) 0 sig
    lbl <- mkLabel sig'
    mkVStack (const [enabledWidget lbl, enabledWidget btn])
```

Here the GUI is defined as a vertical stack consisting of a button and a label. The button is constructed using the *mkButton* function, which takes a signal of some *a* that is an instance of *Displayable* and *Stable*, then returns a *C* button. To be an instance of the *Displayable* typeclass *a* must have a function *display* that converts it into a *Text*, so it can be shown on screen.

In our case we pass the button constructor a constant signal of text that always has the value "Increment". The Async Rattus *const* function makes a constant signal from any value[7]. The helper function *btnOnClickSig* takes a button and returns a delayed signal of type *unit*. This signal ticks every time the button is pressed.

A tick on *sig* corresponds to a user generated event. Thus signals constructed from *sig* can be used to add functionality to this event. The Async Rattus *scanAwait* function behaves similarly to Haskells *scanl* function, but for signals rather than lists [7].

Here is an example of how *scanAwait* could be used:

```
scanAwait box(+) 0 O(1:::2:::3:::...) == (0:::1:::3:::6:::...)
```

Here the sum function is applied whenever the input signal ticks, which returns a new signal consisting of the accumulating sum. The signal notation above is not a valid constructor, but the notation is a convenient way to show the value of a signal across time steps. In the code for the counter, *scanAwait* is used to apply an increment function whenever *sig* ticks. The resulting signal *sig'* has an initial value of 0 and increments whenever *sig* ticks.

Therefore the value of *sig'* increments whenever the button is pressed. Calling *mkLabel sig'* then returns a label that always displays the current value of *sig'*. Between the increment button and this label the functionality of the counter GUI is attained.

In the case of the counter GUI, we create a vertical stack *Widget* for passing to the *runApplication* function. This stack needs to contain the button and the label. Since neither *Widget* should be disabled, the *enabledWidget* function is used to create two elements of type *Widget*. The two widgets are inserted into a list then passed to *mkVStack*; the function for constructing vertical stacks. Note that we once again use the *const* function to make the list into a constant signal of *List Widget*.

5.2 Temperature Converter

The temperature converter GUI requires creating a bidirectional temperature converter from Celsius to Fahrenheit [8]. The GUI should contain two textfields representing Fahrenheit and Celsius respectively[8]. When a number is typed into either textfield the other should update to show the converted value. This benchmark introduces two new challenges; handling user generated text input and introducing bidirectional data flow [8]. Figure 3 illustrates how the temperature converter might look:

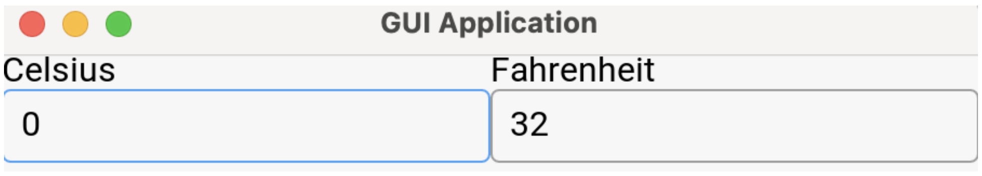

Figure 3: Screenshot of the Temperature Converter Benchmark
Shows the two textfields containing Celsius and Fahrenheit

To handle user generated input, we define the *TextField* widget. A textfield consists of a signal of type *text* and a channel of type *text*:

```
data TextField =
     TextField {tfContent :: !(Sig Text), tfInput :: !(Chan Text)}
```

The channel *tfInput* contains any input the user types into the textfield. On the other hand the signal *tfContent* produces the values that the textfield displays on screen. By default *tfContent* has no intrinsic connection to *tfInput*. However, displaying the user input is usually desired, so the *mkTextfield* function is defined as follows:

```
mkTextField :: Text -> C TextField
mkTextField txt = do
      c <- chan
      let sig = txt ::: mkSig (box (wait c))
      return TextField{tfContent = sig, tfInput = c}
```

When calling *mkTextField*, *tfContent* is constructed from the input channel *tfInput*. This ensures the display is updated when the user types in the textfield.

While the *mkTextField* function provides a simple way to handle user generated text input, it presents difficulties in regards to creating a bidirectional data flow. To achieve this we need to create a Celsius textfield that can be written to, but also converts values written in the Fahrenheit textfield. So the *tfContent* signal has to tick in response to the Celsius textfield's *tfInput* and whenever the Fahrenheit signal ticks. However, *mkTextField* defines *tfContent* as a signal that responds only to the *tfInput* channel, so it cannot be used in this case. The temperature converter can still be completed by using the textfield constructor directly:

```
    cC <- chan
    cF <- chan
    let sigC = mkSig (box (wait cC))
    let sigF = mkSig (box (wait cF))
    let convertFtoC = mapAwait (box fahrenheitToCelsius) sigF
    let convertCtoF = mapAwait (box celsiusToFahrenheit) sigC
    let sigC' = "0":::interleave (box (\ x y -> x)) FtoC sigC
    let sigF' = "32":::interleave (box (\ x y -> x)) CtoF sigF
    let tfC = TextField {tfContent = sigC', tfInput = cC}
    let tfF = TextField {tfContent = sigF', tfInput = cF}
```

In the code above, we initially construct two input channels, one for each textfield. From these, two signals reflecting the user input are instantiated.

mapAwait is an Async Rattus function that constructs a signal by applying a function to any value produced by a delayed signal[2]. An example usage could be:

```
mapAwait box(+1) (1:::2:::3:::...) == (2:::3:::4:::...)
```

In the temperature converter *mapAwait* is used to apply functions that convert between Celsius and Fahrenheit. Thus, when *sigC* produces an integer value *convertCtoF* produces the corresponding value in Fahrenheit and vice versa.

Next the Async Rattus function *interleave* is used to combine the values produced by *sigC* and *convertFtoC*. *interleave* is a function that takes two delayed signals and produces a combined delayed signal. The combined signal ticks whenever one of the input signals tick, producing the same value. Given signals *xs* and *ys*, an example could be[7]:

```
xs: 1 3   5 3 1 3
ys:   0 2   4
```

```
interleave (box (+)) xs ys: 1 3 2 5 7 1 3
```

interleave takes as input a binary function that is applied when both signals tick at once . In the temperature converter, we ensure that if the input signal ticks simultaneously with the conversion signal, it is the value of the conversion signal that gets displayed.

Having interleaved the signals representing user input with signals applying the conversion functions, it is possible to call the textfield constructor directly and get the desired functionality.

Though the above solution works, it does not leverage much of the GUI library, instead requiring the programmer to manipulate input channels directly. *mkTextfield* cannot be used in this solution, since it defines *tfContent* prematurely and the value cannot be changed. To provide better helper functions for this problem we define *addInputSigTF*:

```
addInputSigTF :: TextField -> O (Sig Text) -> TextField
addInputSigTF tf sig =
      let tfContent' = current (tfText tf) :::
          interleave (box (\x y -> x))
                     (future (tfContent' tf)) sig
      in tf{tfContent = tfContent', tfInput = tfInput tf}
```

The function *addInputSigTF* takes a textfield *tf* and a delayed signal *sig* as input. It then defines a new signal *tfContent'* by interleaving the *tfContent* value of *tf* with *sig*. This new signal retains the behaviour of the original textfield, but also ticks whenever *sig* does. Thus a new textfield can be returned that keeps the original input channel, but also displays any value produced by *sig*. This can be used to implement an alternative solution to the temperature converter GUI, which does not require the programmer to manipulate input channels:

```
benchmark2 :: C HStack
benchmark2 = do
    tfF1 <- mkTextField "32"
    tfC1 <- mkTextField "0"

    let convertFtoC = map (box fahrenheitToCelsius) (tfContent tfF1)
    let convertCtoF = map (box celsiusToFahrenheit) (tfContent tfC1)

    let tfF2 = addInputSigTF tfF1 (future convertCtoF)
    let tfC2 = addInputSigTF tfC1 (future convertFtoC)

    fLabel <- mkLabel (const ("Fahrenheit" :: Text))
    cLabel <- mkLabel (const ("Celsius" :: Text))
    fStack <- mkVStack
                (const [enabledWidget tfF2, enabledWidget fLabel])
    cStack <- mkVStack
                (const [enabledWidget tfC2, enabledWidget cLabel])

    mkHStack (const [enabledWidget fStack, enabledWidget cStack])
```

This implementation simplifies creating bidirectional data flows by using the helper functions provided in our GUI library. Thus, it is shown that Async Rattus can be used for creating GUIs, combining multiple widgets and handling non-trivial data flows.

5.3 Flight Booker

The primary challenge behind the flight booker GUI comes in the form of constraints [8]. The task is to build a flight booker where widgets disable or enable one another[8]. Figure 4 illustrates an implementation of the flight booker:

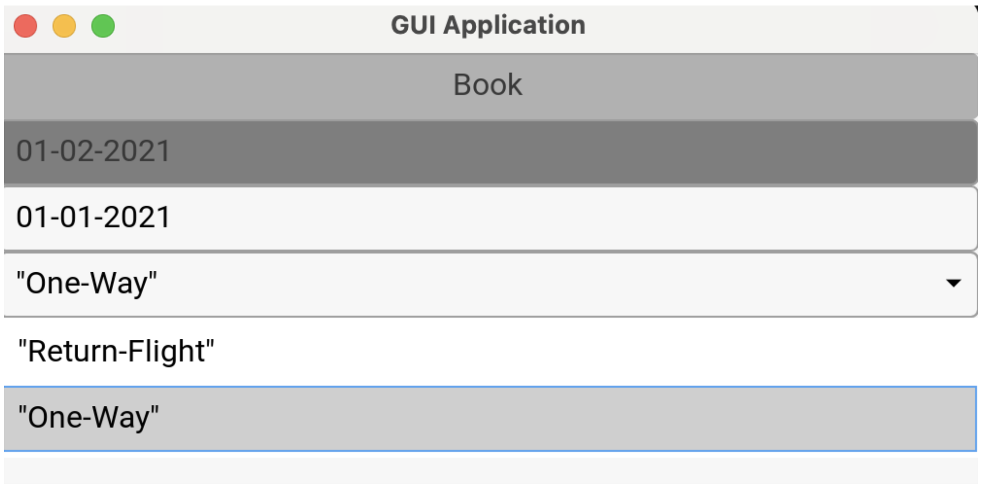

Figure 4: Screenshot of the Flight Booker:
At the bottom, an open textdropdown showing the flight options.
Above, the two textfields for dates and a booking button

The benchmark requires the implementation of a dropdown menu, wherein you can select whether you are booking a return flight or a one-way ticket. For this purpose a *textDropdown* is defined as:

```
data TextDropdown = TextDropdown {
                    tddCurr ::!(Sig Text),
                    tddEvent :: !(Chan Text),
                    tddList :: !(Sig (List Text))}
```

In the case where a one-way ticket is being selected, the textfield containing the return date gets disabled. Additionally the button for booking a flight is disabled whenever the dates are incorrectly formatted.

Monomer has a function for enabling or disabling widgets, called *nodeEnabled*. *nodeEnabled* is called with a widget and a boolean value, and returns a widget that is enabled when the boolean value is true. To include this functionality in our Async Rattus library we call *nodeEnabled* when constructing any *Widget*:

```
instance IsWidget Widget where
    mkWidget :: Widget -> Monomer.WidgetNode AppModel AppEvent
    mkWidget (Widget w (e ::: _)) =
             Monomer.nodeEnabled (mkWidget w) e
```

Any call to the *Widget* constructor must include a signal of type *Bool*. The value of this signal then determines when the *Widget* is enabled. As mentioned in Section 4 the *enabledWidget* function can be used to construct *Widgets* that are always enabled.

In addition to the aforementioned textdropdown, the flight booker requires the implementation of a popup for signifying a correct booking. Popups are defined by:

```
data Popup = Popup {popCurr :: !(Sig Bool),
                    popEvent :: !(Chan Bool),
                    popChild :: !(Sig Widget)}
```

Here the *popChild* is a signal of *Widget*. *popChild's* value indicates the widget to display whenever *popCurr* is true.

The code for the flight booker GUI (Appendix 2.3) is more extensive than the previous benchmarks. It will not be included here, since it consists mostly of constructor calls and comparison logic written in plain Haskell.

5.4 Timer

The timer GUI entails building an interactive timer[8]. The timer ticks up every second and its value is displayed in both a progressbar and numerically on a label [8]. 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[8]. Figure 5 has an image of our solution:

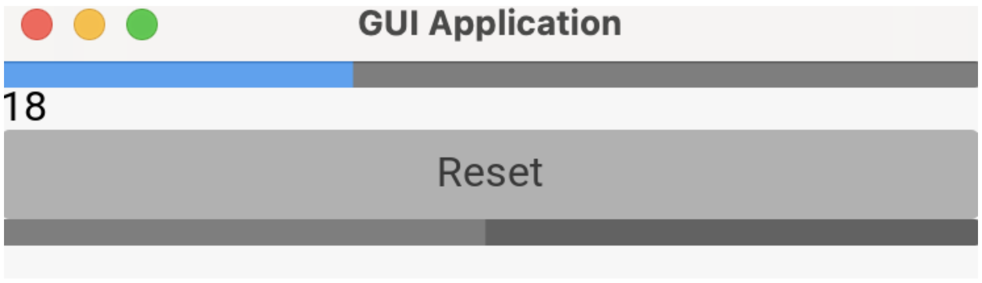

Figure 5: Screenshot of the Timer Benchmark:
Showcasing the slider, progressbar and reset button

In the above GUI, the blue bar is the progress bar which increments towards the maximum value of the grey slider. Pressing the reset button sets both the numerical value and the progressbar to zero.

This benchmark required the implementation of a slider and a progressbar. A slider can be represented by the following data type:

```
data Slider = Slider {sldCurr :: !(Sig Int),
                      sldEvent :: !(Chan Int),
                      sldMin :: !(Sig Int),
                      sldMax :: !(Sig Int)}
```

The slider uses integer signals for its maximum, minimum and current values. It also has an integer channel that is used to receive user input when the bar is dragged, setting the current value. This is implemented in the *mkSlider* function:

```
mkSlider :: Int -> Sig Int -> Sig Int -> C Slider
mkSlider start min max = do
      c <- chan
      let curr = start ::: mkSig (box (wait c))
      return Slider{
                sldCurr = curr, sldEvent = c,
                sldMin = min, sldMax = max}
```

As can be seen in the above definition, the current value of the slider is a signal made from the input channel *c* which is used for the *sldEvent*. Since the min and max values are given by signals, these can be changed during program execution, giving much flexibility to the widget.

Unfortunately Monomer does not currently have a progressbar widget [12]. However, a serviceable alternative can be constructed from the slider widget:

```
mkProgressBar :: Sig Int -> Sig Int -> Sig Int -> C Slider
mkProgressBar min max curr = do
      c <- chan
      let boundedCurrent =
        zipWith (box Prelude.min) curr max
      return Slider{sldCurr = boundedCurrent,
                    sldEvent = c,
                    sldMin = min,
                    sldMax = max}
```

Instead of letting the current value *sldCurr* depend on the input channel, the above implementation sets the current value based on the input signal *curr*. The slider constructor is still passed a channel to satisfy type constraints, but it is given no functionality.

mkProgressBar uses the Async Rattus function *zipWith*, to ensure that the current value of the progressbar never exceeds its maximum value. *zipWith* takes as input two signals and returns a new signal that ticks whenever either of the input signals tick[2]. An example usage of *zipWith* could be [7]:

```
                     xs:  1 2 3     2
                     ys:  1     0 5 2

zipWith (box (+)) xs ys:  2 3 4 3 8 4
```

The returned signal is the result of applying the input function, (+) to the current values of both input signals. This creates a new signal that is at all times equivalent to the sum of the current values of *xs* and *ys*.

In the case of the timer, *zipWith* is used to apply the *min* function to *curr* and *max*. This ensures that the current value of the progressbar never exceeds its current maximum value.

The primary challenge of the timer GUI is concurrency, since user input competes with the state of the timer[8]. This is a challenge that Async Rattus is well suited for, since the *Select* primitive handles simultaneous ticks elegantly. However, constructing a signal representing the current value of the progressbar requires some more advanced helper functions.

First, it is necessary to create a signal that ticks periodically. For this purpose we define *everySecondSig*, which is signal that ticks every second:

```
everySecond :: Box (O())
everySecond = timer 1000000

everySecondSig :: Sig ()
everySecondSig = () ::: mkSig everySecond
```

This signal can be used to create another signal that increments every second using the Async Rattus *scan* function. *scan* works similarly to *scanAwait*, but applied to signals that are not delayed (See example in Section 5.1.). However, such a signal would not be capped by the maximum value of the slider. Hence a function that can stop the increments is necessary:

```
stop :: Box (a -> Bool) -> Sig a -> Sig a
stop f (x:::xs) = if unbox f x then x:::never
                  else x:::delay (stop f (adv xs))
```

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

```
stop box (>=3) 1:::2:::3:::4 == 1:::2:::3:::never
```

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

```
nats :: (Int :* Int) -> Sig (Int :* Int)
nats (n :* max) =
    stop (box (\ (n :* max) -> n >= max))
         (scan (box (\ (n :* max) _ -> min (n + 1) max :* max))
               (n :* max)
               everySecondSig)
```

nats takes as input a pair of integers $(n : *max)$ and returns a signal of the same type. The resulting signal will tick every second, incrementing n, until it reaches max. An example of how *nats* behaves can be seen below:

```
nats(0:*3) == (0:*3):::(1:*3):::(2:*3):::(3:*3):::never
```

This is achieved by passing the input tuple to *scan* and using a lambda function that compares the current value of $(n + 1)$ with max every tick. Using the *nats* function it is possible to implement the timer GUI:

```
reset :: (Int :* Int) -> (Int :* Int)
reset (n :* max) = (0 :* max)

setMax :: Int -> (Int :* Int) -> (Int :* Int)
setMax max' (n :* max) = min n max' :* max'

benchmark4' :: C VStack
benchmark4' = do
    slider <- mkSlider 50 (const 1) (const 100)
    resetBtn <- mkButton (const "Reset")

    let resSig = mkSig (btnOnClick resetBtn)
    let resetSig = mapAwait (box (\ _ -> reset)) resSig

    let currentMax = current (sldCurr slider)
    let setMaxSig = mapAwait (box setMax) (future (sldCurr slider))

    let inputSig = interleave (box (.)) resetSig setMaxSig

    let counterSig = switchB inputSig (box nats) (0 :* currentMax)
    let currentSig = map (box first) counterSig
    let maxSig = map (box second) counterSig

    label <- mkLabel (map (box display) currentSig)
    pb <- mkProgressBar (const 0) maxSig currentSig

    mkVStack (const [enabledWidget slider,
                     enabledWidget resetBtn,
                     enabledWidget label,
                     enabledWidget pb])
```

To make the timer GUI the progressbar must be constructed using a signal that represents a capped timer. Such a timer must consist of a current and a maximum value. It needs to respond to two different types of user input: The reset button that alters its current value, and the slider that determines its maximum value. For this purpose two helper functions are defined, *reset* and *setMax*. From these we create two signals: *resetSig* which returns the *reset* function any time the reset button is pressed, and *setMaxSig* which partially applies *setMax* whenever the slider is changed. *setMax* always gets partially applied to the current value of the slider.

resetSig and *setMaxSig* are then interleaved using function composition. This creates the signal *inputSig* that ticks in response to either type of input producing the corresponding helper function.

The signal representing the timer needs to dynamically update its behaviour in response to user input, by applying the functions produced by *inputSig*. For this purpose the *switchB* library function is defined:

```
switchB :: Continuous a =>
           O (Sig (a -> a)) -> Box (a -> Sig a)-> a -> Sig a
switchB steps f st = switchS ((unbox f) st)
      (delay (let step ::: steps' = adv steps
              in (switchB steps' (f . step))))
```

Like *switch* (see Section 3.2.4), *switchB* is a function that changes the behaviour of a signal during program execution . Whereas *switch* works by changing from one input signal to another, *switchB* takes as input a delayed signal *steps* and updates the behaviour of the returned signal whenever *steps* ticks. This is done by recursively calling *switchB* in a call to the Async Rattus function *switchS*:

$$\texttt{switchS} :: \text{Stable } a \Rightarrow \text{Sig } a \to \bigcirc(a \to \text{Sig } a) \to \text{Sig } a$$

The *switchS* function is similar to *switch*, but the second signal may depend on the last value of the first signal[7]. *switchS* needs two inputs: a signal of type *a* to dictate the initial behaviour, and a delayed function that creates signals of type *a* from values of type *a*. *switchS* returns a signal that initially behaves like its input signal and changes behaviour when the delayed function becomes available. The new behaviour of the signal will be the result of applying the delayed function to the current value of the input signal.

switchB takes as input a function *f* that constructs signals from values of type *a*. This function is applied to an initial value *st* to create a signal that determines the initial behaviour of *switchB's* output signal. This initial signal is passed to *switchS* alongside a delayed recursive call to *switchB*. The delayed call is constructed by calling *adv* on the input signal *steps*. *delay* and *adv* ensure that *steps* has ticked and produced a current value *step* when the recursive call executes. *step* is a function of type $a \to a$. *step* composed with *f* gives a function of type $a \to$ *Sig (a)*. By partially applying *switchB* to *steps'*, the tail of *steps* after it has ticked, and *f* composed with *step* we get a function of type $a \to$ *Sig a* which is the exact argument that *switchS* expects.

In the case of the timer GUI *switchB* is used to create *counterSig*, a signal representing a capped timer. Table 3 contains an example of how *counterSig* behaves for a given sequence of user input:

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)

Table 3: Example table for *counterSig*

counterSig initially behaves like *nats* 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 *nats (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 *nats (0:*10)*.

The rest of the benchmark code is a matter of passing the values from *counterSig* to a label and a progressbar. These details can be seen in the code (Appendix 2), but will not be discussed here. Although Async Rattus handles concurrency well, in the timer GUI it becomes challenging to construct signals representing the needed data flow, as it has become more advanced than in previous benchmarks. This is something that will be further discussed in Section 6.

5.5 Calculator

Following the implementation of the initial four benchmarks with Async Rattus, this section introduces the development of a basic calculator. By using Async Rattus to create a calculator GUI, we get to see how Async Rattus can handle input from a multitude of different widgets. This serves as a test for evaluating Async Rattus' capabilities in managing dynamic user interactions in a more involved GUI. Figure 6 showcases our calculator GUI:

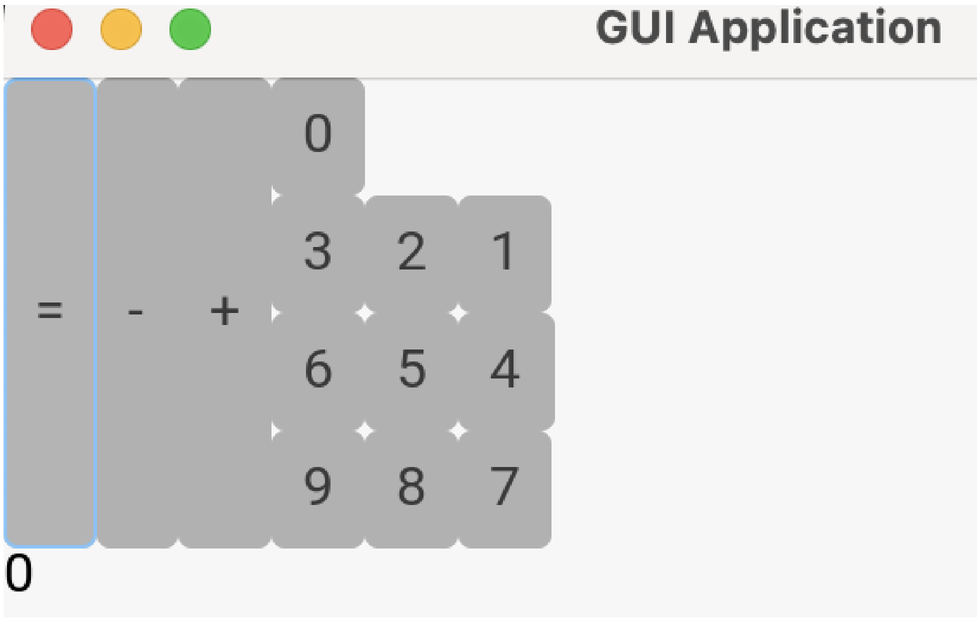

Figure 6: Screenshot of the Calculator GUI
Containing a 13 button numpad and a display label

The full source code for the calculator implementation can be found in Appendix 2. The user interface consists of 13 buttons and a label. One button for each digit, as well as plus minus and equality. The label represents the display of the calculator. It should reflect any number typed in by the user and the result after calculation. The calculator implementation uses several signals and many signal combinators, a diagram describing the data flow can be seen here:

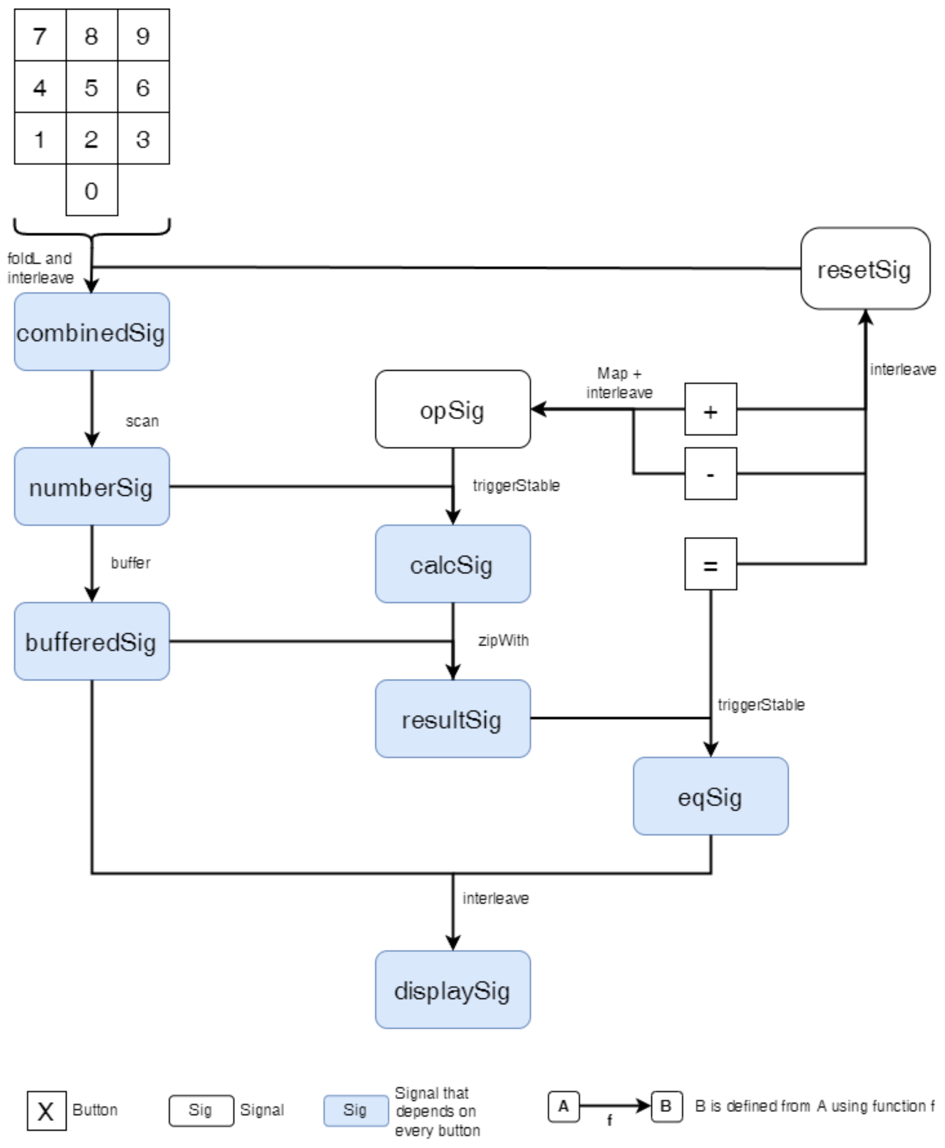

Figure 7: Data flow in the calculator.
Shows how signals are defined throughout the calculator

Figure 7 describes how data flows from user input to the *displaySig* which is used to define the aforementioned label. The diagram uses arrows to show how user input flows from the buttons through signals, to the *displaySig*. Written alongside the arrows are the functions used to define each signal. The definitions of each signal will be expanded on below.

The main difficulty presented by the calculator lies in controlling the data shown in the label. To better understand what the calculator should do, the following requirements are defined:

- When the user presses a digit button, the corresponding digit should appear as the last digit in the number on the display.
- Pressing either the plus or minus buttons resets the display to zero in preparation for a new entry.
- When the user presses the equality button, the calculator should display the result of applying the most recent operation to the number on the display.
- If a numerical button is pressed following the equality button, the display will reset to show just the newly pressed digit.

To show any combination of numbers typed by the user one can use *interleave* to combine a signal made from each of the numpad buttons:

```
let sigList = [onclick0,
               onclick1,
               onclick2,
               onclick3,
               onclick4,
               onclick5,
               onclick6,
               onclick7,
               onclick8,
               onclick9, resetSig] :: [O (Sig (Int->Int))]

let combinedSig = Prelude.foldl1
                      (interleave (box (\ a b -> a)))
                      sigList
```

Here the *onClick* values are signals designed to produce functions when the corresponding digit button is pressed. Each *onClick* signal ticks with a function of the form: $f_n(x) = 10x + n$ where n is the relevant digit.

To create a signal satisfying the first requirement, all the *onClick* signals are combined into a single signal. This signal ticks in response to a user pressing any digit button. This is achieved by folding through the *onClick* signals using *interleave*.

The combined signal also uses the *interleave* function with *resetSig*, which is defined from the operators and equality buttons:

```
let resetSig =
         mapAwait (box (\ _ _ -> 0))
             (interleave (box (\ a b -> a))
                 (interleave (box (\ a b -> a))
                     (btnOnClickSig addBut)
                     (btnOnClickSig subBut))
                 (btnOnClickSig eqBut))
```

resetSig responds to the user pressing an operator or equality and returns a function that resets any integer to zero. Since *combinedSig* interleaves this and signals from every digit button, it ticks in response to any button on the numpad. This produces a function to update the displayed value. The equality button must be included in *resetSig* to ensure that the calculator is well behaved after carrying out the first calculation, as specified in the fourth requirement.

What remains to do is apply the functions produced by *combinedSig* to a value. This is done in the *numberSig* signal:

```
let numberSig = scanAwait (box (\ a f-> f a)) 0 combinedSig
```

numberSig uses *scanAwait* to apply any functions produced by *combinedSig*, starting from an initial value of zero. *numberSig* is therefore a signal satisfying the first two requirements.

Before handling the equality button and the third requirement, it is necessary to implement a signal that saves the calculation when a user presses an operator button. This is done with the *calcSig*:

```
    let buffered = buffer 0 numberSig
    let addSig = mapAwait
                    (box (\ _ -> box (+))) (btnOnClickSig addBut)
    let subSig = mapAwait
                    (box (\ _ -> box (-))) (btnOnClickSig subBut)
    let opSig = interleave (box (\ a b -> a)) addSig subSig
    let calcSig = triggerStable
                    (box (\ op x ->box (unbox op x))) (box (0 +))
                    opSig buffered
```

Here the order of operations becomes a challenge. When the user presses an operator the *numberSig* immediately resets to zero. However, the value of *numberSig* prior to the user pressing an operator is vital for defining *calcSig*. Hence the *buffer* function is defined:

```
buffer :: Stable a => a -> Sig a -> Sig a
buffer x (y ::: ys) = x ::: delay (buffer y (adv ys))
```

This function takes an initial value and a signal as input, and returns a signal that is always one tick behind the input signal. An example using buffer could look like this:

```
         xs     1  2  7     5  6
buffer 0 xs     0  1  2     7  5
```

Using the *buffer* function *bufferedSig* is defined as the signal that is always one tick behind *numberSig*. When the user presses an operator, *numberSig* gets the value zero and *bufferedSig* has the integer typed in by the user prior to pressing an operator. Some sample input could produce a sequence like the following:

```
input              3  +  4  5
numberSig       0  3  0  4  45
bufferSig       0  0  3  0  4
```

This makes it possible to define *calcSig*. *calcSig* applies the most recently pressed operator to the current value of *bufferedSig*, producing a partially applied function. This is done using the *triggerStable* function:

```
triggerStable :: (Stable b, Stable c) =>
    Box (a -> b -> c) -> c -> O (Sig a) -> Sig b -> Sig c
triggerStable f c as (b ::: bs) = c :::
delay (case select as bs of
        Fst (a' ::: as') bs' ->
            triggerStable f (unbox f a' b) as' (b ::: bs')
        Snd as' bs' -> triggerStable f c as' bs'
        Both (a' ::: as') (b' ::: bs') ->
             triggerStable f (unbox f a' b') as' (b' ::: bs'))
```

triggerStable is a variant of the Async Rattus *trigger* function. *triggerStable* creates a signal that only updates its value in response to the delayed input signal. *trigger-Stable* takes as input a function *f*. Whenever the delayed input signal *as* ticks, it applies *f* to the current values of both input signals. When the other input signal *bs* ticks, *triggerStable's* value remains unchanged. Note that *trigger* uses the *Maybe* monad and therefore returns an IO type, whereas *triggerStable* returns a signal. An example usage of how *triggerStable* reacts to a sequence of user input can be seen in Table 4:

User Input	4	0	+	3	
numberSig	4	40	0	3	0
bufferedSig	0	4	40	0	3
opSig			(+)		
calcSig	(0+)	(0+)	(40+)	(40+)	(40+)

Table 4: Signal values according to user input

Since *calcSig* is defined using *triggerStable* its value only changes when the user presses one of the operator buttons. In the example, *calcSig* changes when + is pressed, returning the partially applied function (40+). This now lets us define a signal that applies the current value of *calcSig* to *bufferedSig* like so:

```
let resultSig = AsyncRattus.Signal.zipWith
                 (box (\ f x -> unbox f x)) calcSig buffered
```

resultSig is implemented using *zipWith*. Hence it ticks in response to any tick on either *bufferedSig* or *calcSig*. This signal computes the number used for the third calculator requirement. What remains is to display the value of *resultSig* when the equality button is pressed, for this purpose we use *triggerStable* again:

```
let eqSig = triggerStable
   (box (\ _ x -> x)) 0 (btnOnClickSig eqBut) resultSig
```

Because of *triggerStable eqSig* only produces a new value when the equality button is pressed. This value is the result from *resultSig*. *numberSig's* value corresponds to the numbers typed in by the user, and *eqSig* produces the results of any typed calculation when equality is pressed. These two signals combine to form the *displaySig*:

```
let displaySig = 0 ::: interleave (box (\ a b -> b))
                       (future numberSig) (future eqSig)
result <- mkLabel displaySig
```

By this construction the label result should satisfy the four requirements set out in the beginning of this section. Note that the function passed to *interleave* prioritizes *eqSig* over *numberSig* in the case that both signals tick simultaneously, which should happen only when the equality button is pressed.

Unfortunately *displaySig* does not behave as intended, due to a peculiarity of *triggerStable*. Table 5 shows the ticks produced by each signal in response to a given sequence of user input:

User Input	4	0	+	3	=
numberSig	4	40	0	3	0
bufferedSig	0	4	40	0	3
opSig			(+)		
calcSig	(0+)	(0+)	(40+)	(40+)	(40+)
resultSig	0	4	80	40	43
eqSig	0	0	0	0	43
displaySig	0	0	0	0	43
idealDisplaySig	4	40	0	3	43

Table 5: Signal values according to user input
displaySig is what is shown in the label
idealDisplaySig are the desired values

The values marked with red are the result of implementing *triggerStable* without the Maybe monad. As can be seen in the definition of *triggerStable*, signals made using *triggerStable* will tick whenever either of the input signals tick, although the current value only updates if the delayed input signal ticks. In the Async Rattus function *trigger*, the Maybe monad is used to filter any ticks on the input signal, so that the returned signal does not tick.

Although signals made from *triggerStable* will have the intended value, the extra ticks that are not filtered interfere with the data flow when interleaving *eqSig* and *numberSig*. Since *eqSig* is defined from *numberSig* it ticks in response to every button, not just the equality button. This means the function passed to *interleave* is called every time the display updates, and hence the *numberSig* is always overruled in the display.

This behaviour is a consequence of signals made using *triggerStable* ticking even when the value does not update. The Async Rattus function *trigger* would not cause this issue, but is incompatible with the GUI library as of now, since the IO and C monads cannot be used in conjunction. This inability to use *trigger* and other filter functions in the GUI library restricts the programmer significantly. This makes tasks such as the calculator more difficult to implement, and representing some complex data flows may not be possible.

6 Discussion

Having implemented four out of the seven benchmarks from 7GUIs: A GUI Programming Benchmark[8] and attempted a calculator, we now discuss the implementation of our GUI library. While this thesis has demonstrated the effectiveness of Async Rattus as a programming language, it has not yet explored alternative solutions to some of the problems we have addressed. It also remains to discuss the challenges encountered during the project. This section will therefore outline the hurdles faced during the implementation of the GUI library, how these obstacles were managed, and identify areas where Async Rattus needs further development.

6.1 Challenges and Alternative Solutions

The calculator GUI tested the limits of what we could achieve using Async Rattus and our GUI library. As concluded in Section 5.5, the current version using *trigger-Stable* does not behave as intended. While Async Rattus provides several alternative approaches to solving the calculator problem, we chose to extensively use Async Rattus' signals and associated functions. This decision was made to test the current limits of Async Rattus as a tool for building GUIs.

One alternative implementation could have treated the display as a signal of text. Using text concatenation to update after each button press. Then parsing it into a calculation whenever the equality button is pressed. This would be a valid solution using Async Rattus, but it would make minimal use of signal combinators and simplify state management by shifting most functionality into a parsing function. We did not choose this solution because the purpose of the thesis was to explore the capabilities of Async Rattus as a language for implementing GUIs, rather than to build the most optimal GUIs.

Additionally, it is worth noting that Async Rattus has a type that could have been used to implement the calculator. This is the concept of futures from Async Rattus, defined in the *SigF* data type [2]. Futures work similarly to signals but implement a call to *Maybe* whenever they tick [2]. *SigF* implements a number of filter functions, including trigger which would solve the issue we encountered when programming the calculator. However, *SigF* is less efficient than the *sig* data type, because it has to check for *Nothing* every time it ticks [2]. Therefore using *sig* is preferable when possible. Since the entire GUI library was set up to work with the *sig* data type, making this implementation could be the focus of another project.

6.2 Working with Async Rattus

The GUI library we have developed for Async Rattus enables the creation of widgets and GUI layouts with simple, compact code. This is evident in the counter GUI, where the entire user interface is implemented in just seven lines of code. Even in the later benchmarks, the Async Rattus library requires less code compared to its Monomer counterpart (see Appendix 4). Additionally, Async Rattus excels at handling concurrent events. In the timer case study, we observed how Async Rattus elegantly manages conflicting events. Even when three events compete to modify the same state, Async Rattus functions provide built-in features to resolve such conflicts seamlessly.

Async Rattus differs significantly from both Monomer and React in state management. While Monomer and React use explicit model types to contain an application's state, the Async Rattus GUI library defines all state in terms of signals. In Monomer and React, state changes are explicitly triggered by functions that modify parts of the model. In contrast, Async Rattus signals define their own state and how they change in response to user input. In React and Monomer, events call specific functions when they occur updating the state of the application. On the other hand events in Async Rattus are not defined as part of an event handler. Instead event handling in Async Rattus involves mapping and combining signals, as well as filtering them when necessary.

This approach to state management and event handling is very different from traditional methods. As a result, reasoning about state and event handling in Async Rattus can become challenging as the complexity of data flows increase. This difficulty was particularly apparent in the timer GUI, where dynamically switching the behavior of signals during execution was needed. Although Async Rattus provides functions for switching signal behaviors, they can be difficult to use and often require custom modifications for specific purposes. The calculator GUI also proved that complex data flows increases the number of signals and combinators needed to achieve the desired functionality.

6.3 Qualitative vs. Quantitative Metrics

The GUI library we present is a showcase of how Async Rattus can utilize the Monomer library to create GUIs. By carrying out the case studies, we show that the GUIs can reach a certain level of complexity. As seen in the project, this can require further tool development in the GUI library, or Async Rattus when the available signal combinators do not suffice.

These are, however, mostly qualitative metrics. Testing the performance of GUIs in Async Rattus quantitatively has been beyond the scope of this project. Every GUI described above has run smoothly without input lag. We did note that the compilation time of the calculator noticeably exceeds that of the other GUIs, possibly due to the increased number of widgets that need to be constructed. Still, the performance of Async Rattus compared to other programming languages has not been tested. The merits of such testing is discussed in Section 6.6.

6.4 Limitations and Areas for Improvement

Async Rattus provides powerful tools for handling concurrent input, but as a research language, its implementation is not without flaws. Like most software under development, occasional unintended behavior and bugs are bound to occur and will only be revealed through extensive testing.

This project has been the first extensive exploration and test of Async Rattus, particularly in regards to GUI development. Throughout the course of this project several bugs and limitations of Async Rattus have been discovered. This has in turn helped the development of Async Rattus.

When first attempting the temperature converter, we encountered an unusual issue. Trying to define a *mkTextField* function that properly updated its display in response to user input, caused the compiler to panic. This put a temporary stop to development since the five-page error message concluded that something impossible had happened, and the issue was not in the GUI library. Once discovered, the issue was quickly resolved by an update to the Async Rattus implementation. Improving Async Rattus and allowing development to continue unhindered.

Additionally the *interleave* function, which is a key signal combinator, exhibited unintended behaviour when this project began. *interleave's* input function meant to handle simultaneous input was not properly applied when simultaneous ticks occurred. This bug was fixed by an update to Async Rattus. Similar issues can only be found through extensively testing and using the Async Rattus library, whether for GUI development or otherwise.

Async Rattus has another feature that proved problematic for the development of this GUI library. In addition to the many signal combinators presented in this thesis, Async Rattus has functions for filtering any input given to signals. Unfortunately, these filter functions make use of the IO monad and hence have IO return types. Since the GUI library makes extensive use of the C monad, the filter functions cannot currently be used when constructing GUIs.

This severely restricts options for constructing useful signals in GUIs. For example, in the temperature converter, two text fields should contain valid temperatures. Using an Async Rattus filter function to create a signal that only responds to numbers would be ideal, but the conflict between the two monads prevents this.

While this is not an insurmountable issue, it will require a rework of either the GUI library or the filter functions in Async Rattus to make the two tools compatible. Creating alternative filter functions that work without the use of the IO monad is also an option, but may cause unforeseen issues, such as those we experienced with *triggerStable*.

6.5 The Complexities of a Research Language

Having engaged extensively with Async Rattus throughout this thesis, we now present our subjective reflections on the experience. We will articulate our perspectives on working with Async Rattus and compare them with our prior experiences using other frameworks, such as React and Monomer.

During the course of this thesis, numerous challenges emerged, which is expected given that Async Rattus is still under development as a research language. Notably, some of these difficulties significantly influenced the implementation of our GUI library and are worth discussing.

One recurrent issue was the disabling of IntelliSense in our IDEs. This absence meant that the only way to detect code errors was through compilation, complicating the process of testing and debugging. This challenge is particularly severe with a research language that lacks extensive documentation. Additionally, we faced issues with inadequate error messaging. For example, we encountered the following misleading error message:

```
No instance for (Stable Text)   arising from a use of 'mkButton'
```

This message was generated in relation to this segment of code:

```
btn <- mkButton (const ("Increment"::Text))
let sig = btnOnClickSig btn
let sig' = scanAwait (box (\ n (_ :: b) -> n+1)) 0 sig
btf <- mkLabel sig'
mkVStack (const [enabledWidget btf, enabledWidget btn])
```

It turns out that the issue was related to a missing type annotation in the following line:

```
let sig' = scanAwait (box (\ n (_ :: b) -> n+1 :: Int)) 0 sig
```

Dealing with such misleading error messages proved to be a challenging aspect of developing our GUI library. There were also instances where IntelliSense was operational within our IDEs but failed to provide any warnings. An example of this occurred with the following line of code:

```
let adds = mapAwait (\ _ -> (+)
                    (current number))
                    (btnOnClickSig add)
```

Here, *number* is a signal of type *Int*. Thus, calling the Async Rattus function *current* should raise an error, since signals are inherently unstable. However, this error was only caught during compilation; the IDE provided no prior warnings.

This highlights that working in an environment that either provides misleading or no error messages at all (before compile-time) can be particularly challenging, especially for inexperienced programmers. Developers looking to explore Async Rattus should be aware of these potential pitfalls.

6.6 Future Research

This paper serves as an initial investigation into GUI programming with Async Rattus. Async Rattus shows promise as a language capable of producing GUIs of considerable complexity and size. However, further research is necessary to explore its full potential. Some areas prime for further exploration include the following topics:

Although the benchmarks highlight several useful qualities and capabilities of the language, its scalability has not been thoroughly tested. In more complex GUIs, data flow quickly became intricate and harder to manage. It would be valuable to explore whether this poses additional issues as GUIs increase in scale. Similarly, the performance of large GUIs might differ from the small benchmark examples. Although no performance issues were noted throughout this paper, this is not guaranteed to hold for larger-scale programs.

It might be worth investigating Async Rattus GUIs more quantitatively. This paper has shown qualitatively that Async Rattus can build small, performant GUIs using compact code. It has not covered any quantitative comparisons between Async Rattus and other tools for building GUIs. Such an exploration could focus on whether the asynchronous nature of Async Rattus offers any significant benefits in terms of efficiency and maintainability. This could be compared to traditional GUI programming languages. Specifically comparing complex GUIs that have to handle various input channels with a greater degree of asynchronicity.

The GUI library presented in this thesis is not functionally complete. Expanding the library with additional widgets and tools would uncover more of Async Rattus' potential for GUI development. Such development would also help find potential limitations of Async Rattus. A significant issue encountered in this paper is the inability to use filter functions in the current iteration of the GUI library. This problem is caused by incompatibility between the C and IO monads. Solving this issue, or finding a suitable workaround, could increase the flexibility of the library and is certainly worth pursuing.

7 Conclusion

This thesis explored the potential of Async Rattus, a Functional Reactive Programming (FRP) language, for creating interactive applications with a focus on Graphical User Interfaces (GUIs). GUI programming inherently requires handling asynchronous operations. Async Rattus' ability to handle such challenges was assessed using 7GUIs: A GUI Programming Benchmark [8].

Our findings highlight several key strengths of Async Rattus:

Async Rattus excels in managing concurrent events, ensuring that GUIs remain responsive and free from common issues such as race conditions and deadlocks. This is achieved through its sophisticated type system and the use of the modalities: $\bigcirc$ (later) and $\Box$ (box) to manage time-dependent computations and stable types.

The implementation of the first four benchmarks demonstrated that Async Rattus could achieve the desired functionality with relatively concise code. The language's ability to dynamically switch signal behaviors during execution underscores its suitability for reactive programming tasks.

Despite these strengths, the development process revealed significant challenges:

The lack of robust IntelliSense support and the prevalence of misleading or absent error messages posed some difficulties. These issues highlight the need for improved development tools and more comprehensive error reporting to enhance the programmer's experience and productivity.

The incompatibility between Async Rattus' C monad and the IO monad restricted the implementation options for GUIs, by preventing the use of filter functions. This limitation necessitates further development of Async Rattus' library functions to expand the functionality of the GUI library.

Async Rattus is still a research language under active development. As this thesis has showcased, several implementation bugs were encountered which resulted in direct development of the language. Similar issues may be encountered as Async Rattus is explored further.

While Async Rattus' approach to state management through signals is powerful, it can be challenging to reason about as the complexity of data flows increases. This complexity was particularly evident in the implementation of the timer GUI, where dynamically switching signal behaviors proved intricate and necessitated custom solutions. This presents a contrast to React and Monomer's traditional model based approaches, where state can be manipulated more straightforwardly.

Given the promising yet challenging nature of developing GUIs with Async Rattus, several avenues for future research emerge:

Future research could investigate how Async Rattus scales with more complex applications, particularly in terms of efficiency and maintainability compared to traditional GUI frameworks.

Addressing the current limitations in library functions and development tools is crucial. The GUI library would be much enhanced by finding a way to utilize Async Rattus' filter functions and providing additional widgets and helper functions. Enhancements to Async Rattus could include better IntelliSense support and clearer error messages.

By addressing these challenges and exploring these research directions, Async Rattus can continue to evolve as a powerful tool for reactive programming offering robust solutions for developing interactive applications in an asynchronous environment.

References

[1] Joseph Abrahamson. *A Little Lens Starter Tutorial.* Accessed: 2024-05-29. 2018. URL: https://www.schoolofhaskell.com/school/to-infinity-and-beyond/pick-of-the-week/a-little-lens-starter-tutorial.

[2] Patrick Bahr, Emil Houlborg, and Gregers Thomas Skat Rørdam. *Asynchronous Reactive Programming with Modal Types in Haskell.* IT University of Copenhagen, Copenhagen, Denmark. Available online at https://bahr.io/pubs/files/asyncrattus-paper.pdf. 2024.

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

[4] Evan Czaplicki. *The Elm Architecture.* https://guide.elm-lang.org/architecture/. Accessed: 2024-05-15. 2024.

[5] Conal Elliott and Paul Hudak. "Functional Reactive Animation". In: *International Conference on Functional Programming.* 1997. URL: http://conal.net/papers/icfp97/.

[6] HaskellWiki. *Functional Reactive Programming — HaskellWiki.* [Online; accessed 28-March-2024]. 2022. URL: https://wiki.haskell.org/index.php?title=Functional_Reactive_Programming&oldid=65422.

[7] Emil Houlborg, Gregers Rørdam, and Patrick Bahr. *WidgetRattus: An asynchronous modal FRP language.* https://hackage.haskell.org/package/WidgetRattus. Version 0.2. 2024. URL: https://hackage.haskell.org/package/WidgetRattus.

[8] Eugen Kiss. *7GUIs: A GUI Programming Benchmark.* 2024. URL: https://eugenkiss.github.io/7guis/tasks (visited on 03/21/2024).

[9] Magnus Madsen, Ondrej Lhotak, and Frank Tip. "A Semantics for the Essence of React". In: *European Conference on Object-Oriented Programming* (). URL: https://par.nsf.gov/biblio/10157540.

[10] Pranav. *What is the Difference Between Props and State in React.* https://codedamn.com/news/reactjs/what-is-the-difference-between-props-and-state-in-react. Accessed: 2024-04-17. 2023.

[11] React. *Getting Started with React.* https://legacy.reactjs.org/docs/getting-started.html. Accessed: 2024-04-17.

[12] Francisco Vallarino. *Monomer.* https://hackage.haskell.org/package/monomer. Accessed: 2024-05-12. 2024.

[13] Francisco Vallarino. *Monomer Basics Tutorial.* https://github.com/fjvallarino/monomer/blob/main/docs/tutorials/01-basics.md. Accessed: 2024-05-12. 2024.

[14] Francisco Vallarino. *Monomer GitHub Repository.* https://github.com/fjvallarino/monomer/tree/main. Accessed: 2024-05-12. 2024.

[15] Francisco Vallarino. *Monomer Life Cycle Tutorial.* https://github.com/fjvallarino/monomer/blob/main/docs/tutorials/03-life-cycle.md#merge-process. Accessed: 2024-05-12. 2024.