Asynchronous Rattus: A Functional Reactive Programming Language with Multiple Clocks

Bachelor Thesis - Bsc Software Development

STADS code: BIBAPRO1PE

IT University of Copenhagen

May 15, 2023

Emil Houlborg	ehou@itu.dk
Gregers Skat Rørdam	grtr@itu.dk

Supervisor

Patrick Bahr	paba@itu.dk

Abstract

Many computer systems are designed to operate continuously; they are *reactive* in that they wait for some form of external input and produce an output in response. An imperative approach often leads to unnatural modeling and the use of complex shared states. Functional Reactive Programming (FRP) promises to solve this in the functional paradigm by operating on time-varying values. Various efforts to implement this in practice have been either inefficient or restrictive for the programmer. In recent years, the "Modal FRP" family of languages has appeared. Using modal types, they introduce a temporal aspect into the type system, allowing both programmer freedom and efficient implementation. One of them, Rattus, allows manipulating time-varying values with proven operational guarantees. It has a global update rate at which all the outputs are updated as a function of the inputs. In programs where some values are updated more frequently than others, this model is unnatural and inefficient. A new calculus, Async RaTT, allows output values to be recomputed only when the input they rely on has changed.

This paper introduces the Asynchronous Rattus programming language which implements most of the Async RaTT calculus. The language is implemented as a package for the Glasgow Haskell Compiler (GHC). The package includes a plugin for GHC, Haskell types representing the language primitives, a standard library, and an interface between Haskell and Asynchronous Rattus. The language allows easy development of reactive applications. This is demonstrated through useful standard library combinators and example programs.

Contents

1 Introduction

When building a software solution, software engineers employ modeling to understand their application and its relation to the real world before it has been built [3, p. 7]. They do so because building software is expensive, and because any non-trivial application needs to be built with the surrounding environment in mind. Thus, modeling plays a central role in software engineering. The Object Oriented Programming paradigm promises to aid this process by allowing a natural representation of real-world phenomena, but it is not always the best and most natural way to model these phenomena.

Reactive applications are a class of applications that react to external events. They wait for input and provide output in response. For such applications, it can be more natural to express the output as a function of the input than to use Object Oriented or imperative techniques. This is the idea behind Functional Reactive Programming (FRP), which was introduced by Conal Elliot and Paul Hudak with the *Functional Reactive Animation (FRAN)* [1] library for Hugs [1].

FRAN introduces two main concepts, *Behaviours* and *Events*. *Behaviours* are time-varying values, and *Events* could be external events such as mouse clicks or keyboard clicks, and internal events such as predicates being satisfied. Both are first-class values, allowing the programmer to freely combine and manipulate them. This results in an expressive language where programmers can easily model complex systems by using primitive values and accompanying combinators. However, the library suffered from *implicit space leaks*. A space leak is a condition in which the program holds on to memory while allocating more, such that it gradually uses more resources. Eventually, the program halts due to a lack of available resources. An *implicit space leak* is a space leak caused by the implementation of the language or library rather than allocations by the user of the language.

Several libraries based on the concepts from FRAN try to mitigate this problem. One of them is Yampa [2], which belongs to the "arrowized FRP" family of languages that are structured by *Arrows*, which are a generalization of Monads. Yampa only allows signals to be manipulated through operations on the *Signal Function* Arrow. This approach reduces the risk of implicitly introducing space leaks but reduces the simplicity because signals are no longer first-class values.

In recent years, the "modal FRP" family of languages has appeared. They expand the type system with a temporal aspect through *modal type theory*. This allows the programmer full access to primitives while preventing the implementation issues that plagued FRAN. Rattus [4] is one of the newest members of the family, and it has been embedded in Haskell to demonstrate that it is useful in practice. Additionally, it is proven free of implicit space leaks. However, the language has a core assumption that all input data is updated at the same rate, making it synchronous by nature.

[1] Hugs is a discontinued Haskell dialect.

When most of the application's state needs to be recomputed continuously, this makes sense. But for applications where inputs are updated at different intervals, we need to recompute every part of the state even if it the data it relies upon has not changed.

Async RaTT [5] is a calculus that lifts this restriction while retaining Rattus' guarantees. It allows recomputing output only when the input it depends on is updated. This works by dividing both input and output into subsections called *channels*, and by allowing programs to describe which output channels depend on which input channels.

This project aims to build a prototype of an asynchronous modal FRP language, based on the Async RaTT calculus. In addition, we aim to provide a standard library and example programs to demonstrate that the language is useful in practice.

2 Background

Asynchronous Rattus is built on top of Rattus [4] by implementing most of the Async RaTT [5] calculus. These languages incorporate the temporal aspect of reactive applications into the language itself. This section introduces Asynchronous Rattus, how it builds on Async RaTT, and briefly how the language differs from Rattus.

2.1 A basic Rattus program

Rattus introduces the modal type $\bigcirc$, which are delayed computations that will produce a piece of data once the necessary input is available. The *clock* of a delayed computation states when the input data necessary in order to compute its value arrives. The synchronous nature of Rattus means that all delayed computations have the same clock since all input arrives *in the next time step*. A value of type $\bigcirc A$ means *the delayed computation will produce a value of type A in the next time step*. Using this definition, the function below can be interpreted as: "produce a delayed computation, that will calculate the sum of two numbers when they become available in the next time step".

```
add :: O Int -> O Int -> O Int
add li lk = delay (adv li + adv lk)
```

As seen from this example, there is a notion of a global clock that controls "the next time step", in which the entire program state is recomputed. For some applications, such as simulations and games, this is natural, since there is usually already a notion of a global update rate. In these types

of applications, it is often necessary to recompute most of the state in each time step. For other applications, like GUIs, some state should be recomputed more frequently than other, leading to unnecessary resource consumption. This is seen in the way events are represented in the Haskell embedding of Rattus; they are `Maybe` values, containing a value only if the event occurred. But we still need to calculate the `Nothing` case when nothing happened.

2.2 Introducing Asynchronous Rattus

We introduce Asynchronous Rattus, a new programming language that aims to implement the Async RaTT calculus [5] based on Patrick Bahr's existing embedding of Rattus in Haskell [4]. Asynchronous Rattus is somewhat similar to Rattus, but removes the assumption of a global clock. To do this, every delayed computation must have its own clock that specifies which input values allows it to be computed. Moving from the synchronous nature of Rattus to an asynchronous nature, a new modality $\exists$ is introduced by Async RaTT. It corresponds to $\bigcirc$, but is generalized to be connected to *any* clock. The new modality type $\exists$ means *the result of the delayed computation is available when the necessary input arrives*, such that a value of type $\exists A$ means "the result of the delayed computation will produce a value of type A when the necessary input arrives".

Looking back at the example of `add`, we can conclude that it is not a valid Asynchronous Rattus program. The replacement of $\bigcirc$ with $\exists$ invalidates this program. Consequently, the function should now be interpreted as: "produce a delayed computation that computes the sum of the two numbers, when the input necessary to compute each of them is provided". However, unlike in Rattus, it is not guaranteed that the input necessary to compute the two numbers will arrive at the same time.

A basic program which *is* valid in Asynchronous Rattus is the function `plusK`. It produces a delayed computation, that will add an integer `k` to the result of the delayed computation `laterI`, when a value arrives at some point in the future.

```
plusK :: Int -> O Int -> O Int
plusK k laterI = delay (adv laterI + k)
```

Here, the delayed computation that is produced by the function knows exactly when the value it's waiting for will arrive in the future, since it is specified by the clock of the single delayed computation it gets as an argument. The delayed computation will simply inherit the clock of `laterI`.

Being unable to produce delayed computations that depend on more than one delayed computation would be a severe restriction of the expressiveness of the language. This would mean that a function

equivalent to `add` could not be expressed. Asynchronous Rattus features the `select` primitive from Async RaTT [5] for this purpose. The following is a simple Asynchronous Rattus program that creates a delayed computation that produces the maximum of two integers.

```
maxLater :: O Int -> O Int -> O Int
maxLater first second =
  delay (
    case select first second of
      Both first' second' -> max first' second'
      Left first' _ -> first'
      Right _ second' -> second'
  )
```

The `select` primitive is used for synchronizing the two delayed computations. It determines whether the input value that arrived can be used to compute one or both of the delayed computations and provides the computed values. An output for `maxLater` can be computed when either of the two given delayed computations can be computed, thus it can be said to have the union of the clocks of the two delayed computations. When only one input can be computed, it is used as the maximum value. The `select` primitive is even more useful when working with streams, as we shall see in section 2.5.

2.3 Input channels and Clocks

In all the examples, the notion of clocks is mentioned in an abstract manner, but it has not been properly defined. To do that, input channels must be introduced.

In Asynchronous Rattus, *input channels* consist of an identifier and a delayed computation producing values received on that identifier. Other delayed computations build on top of them, such that they wait for an input to arrive on one of the channels they build on. The set of input channels on which a given delayed computation waits for input is said to be its clock, and when input arrives on one of them, the clock is said to "tick" [5]. The clock of an input channel's delayed computation is the singleton set of its identifier.

Async RaTT [5] distinguishes between three different input channels, each with different characteristics; *push-only*, *buffered-only* and *buffered-push*. The input channels explained above are *push-only* input channels, and that is the only form of input channels Asynchronous Rattus implements. Moving forward, all input channels can be assumed to be *push-only* input channels.

2.4 Typing rules

The intuition of Asynchronous Rattus comes from the formal type rules of Async RaTT [5], of which a subset is seen below.

$$\frac{\Gamma \vdash_\Delta \theta : \mathsf{Clock} \qquad \Gamma \vdash_\Delta \theta' : \mathsf{Clock}}{\Gamma \vdash_\Delta \theta \sqcup \theta' : \mathsf{Clock}}$$

$$\frac{\Gamma \vdash_\Delta v : \textcircled{\exists} A}{\Gamma \vdash_\Delta \mathsf{cl}\,(v) : \mathsf{Clock}} \qquad \frac{\Gamma' \text{ tick-free or } A \text{ stable} \qquad \Gamma, x : A, \Gamma' \vdash_\Delta}{\Gamma, x : A, \Gamma' \vdash_\Delta x : A}$$

$$\frac{\Gamma, \checkmark_\theta \vdash_\Delta t : A \qquad \Gamma \vdash_\Delta \theta : \mathsf{Clock}}{\Gamma \vdash_\Delta \mathsf{delay}_\theta\, t : \textcircled{\exists} A}$$

$$\frac{\Gamma \vdash_\Delta v : \textcircled{\exists} A \qquad \Gamma, \checkmark_{\mathsf{cl}(v)}, \Gamma' \vdash_\Delta}{\Gamma, \checkmark_{\mathsf{cl}(v)}, \Gamma' \vdash_\Delta \mathsf{adv}\, v : A}$$

$$\frac{\Gamma \vdash_\Delta v_1 : \textcircled{\exists} A_1 \qquad \Gamma \vdash_\Delta v_2 : \textcircled{\exists} A_2 \qquad \vdash \theta_1 \sqcup \theta_2 = \mathsf{cl}\,(v_1) \sqcup \mathsf{cl}\,(v_2) \qquad \Gamma, \checkmark_{\theta_1 \sqcup \theta_2}, \Gamma' \vdash_\Delta}{\Gamma, \checkmark_{\theta_1 \sqcup \theta_2}, \Gamma' \vdash_\Delta \mathsf{select}\, v_1\, v_2 : ((A_1 \times \textcircled{\exists} A_2) + (\textcircled{\exists} A_1 \times A_2)) + (A_1 \times A_2)}$$

$$\frac{\Gamma^\square \vdash_\Delta t : A}{\Gamma \vdash_\Delta \mathsf{box}\, t : \square A} \qquad \frac{\Gamma \vdash_\Delta t : \square A}{\Gamma \vdash_\Delta \mathsf{unbox}\, t : A}$$

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

Figure 1: A subset of the typing rules from the Async RaTT [5] calculus.

Before any of the typing rules can be interpreted, two concepts have to be introduced; typing contexts and ticks. A typing context ordinarily keeps track of variables in scope and their types. In Async RaTT, it is extended with ticks that mark a step into the future. Ticks are introduced into the typing context by `delay`, and they indicate which variables are in scope at the current time.

2.4.1 Typing rules for example programs

Going back to the example for `plusK`, the typing rules can indeed confirm that this is a valid Asynchronous Rattus program. To type the delayed computation that is produced in `plusK` to $\textcircled{\exists} Int$, three of the typing rules have to be applied - *adv*, *delay* and *cl*(v). From these rules, we

can deduce that the delay introduces a tick on the clock of `laterI`, resulting in a correctly typed program.

From the typing rules, it also becomes clear that the example for `add` would never be a valid program in Asynchronous Rattus. Again, the delayed computation being produced must have the same clock as the `adv` within, but which one of them? In either case, `delay` only introduces one tick, which can only correspond to the clock of one of the variables into the typing context. Regardless of which clock is chosen the other `adv` application will not be typeable under the resulting typing context.

The `maxLater` function, which also operates on two delayed computations, is well-typed. This is because it uses `select`, which means that the clocks of `first` and `second` are combined to form a union clock. This is then the clock for the enclosing `delay`.

2.4.2 The "Stable" modality

Given the temporal aspect of Asynchronous Rattus, there is a distinction between two kinds of types, which is also present in Rattus[4] and Async RaTT [5]; stable and non-stable.

In order for a type to be stable, it must be strict and independent of time. This allows moving it arbitrarily far into the future with no risk of space leaks. Hence, stable types will be in scope even when there is a tick in the typing context. In Rattus[4] as well as in the Async RaTT [5] calculus, the Haskell primitives Int, Bool and the like are considered stable types. Note that Haskell Strings are not stable since they are lazy character lists.

Non-stable types are either non-strict or time-dependent, i.e. have $\textcircled{\exists}$ in them. Furthermore, functions are also considered non-stable types, since it is possible that they have types in their closure that are time-dependent. Non-stable types to the left of a tick cannot be accessed. The reason is that non-stable types could be dependent on variables that are defined in an earlier time step, which is indicated by everything in the typing context to the left of a tick. This would lead to implicit space leaks since the data defined in an earlier typing context must reside in memory until a given delayed computation or function needs it. There is no guarantee that a delayed computation will receive the value it's missing or that the given function is called in the foreseeable future, which means that resource consumption will rise over time, such that the application eventually runs out of memory.

However, not allowing functions to be used for more than one time step is quite restrictive. Rattus[4], and the Async RaTT calculus[5], lift this restriction by introducing the *Box* modality. The `box` primitive introduces the Box modality under the special typing rule that the term it is applied to must be typeable under a stable context. Thus, any type A can be moved into the future as long as it can be guaranteed it does not contain any temporal aspect. Bahr [4] has proved, that this

typing rule ensures that moving the type A into the future will not create an implicit space leak. Hence, `box` allows preserving non-stable types across the passage of time. In the future, the boxed type of A can be unboxed and freely used. This is especially useful for functions.

2.5 Streams

As mentioned, the core abstraction in FRP is that of *signals*, or streams as it is called in Rattus [4]. Essentially, streams represent time-varying values, and they are present in most FRP languages. Rattus and FRAN, have them as first-class values that can be freely composed and manipulated.

In Asynchronous Rattus streams are represented as they are in Rattus[4], but of course with asynchronous delayed computations. The Haskell data type is defined as follows:[2]

```
data Str a = !a ::: !(O (Str a))
```

A stream consists of two components; a current value of the given stream and a delayed computation that produces a new stream. This new stream then has the next value and the next delayed computation. To see how we can work with streams Asynchronous Rattus, consider how the `maxLater` function can be extended to work on streams:

```
maxStr :: O (Stream Int) -> O (Stream Int) -> O (Stream Int)
maxStr as bs = delay (
        case select as bs of
            Both (a ::: as') (b ::: bs') -> max a b ::: maxStr as' bs'
            Left (a ::: as') bs' -> a ::: maxStr as' bs'
            Right as' (b ::: bs') -> b ::: maxStr as' bs'
    )
```

We construct a delayed computation in which we select over the two streams. If a value of both streams can be computed, we compute the maximum of them. Otherwise, we pick the value that arrived as the maximum. The tail of the stream is a delayed computation obtained by a recursive call. To examine a slightly more advanced use case of streams, suppose we are building a simple input field. We want to be able to enter text and clear it by pressing the left mouse button. We represent the keyboard input as a delayed stream of characters, such that the text can be built up as the user types. [3]

[2] In fact it has an additional type parameter, as we shall see in section 3.2

[3] As seen in the code for this example (located in `examples/textwriter/src/TextField.hs`), we cannot actually use lazy lists, but must use strict lists from the standard library.

```
text :: [Char] -> O (Stream Char) -> Stream [Char]
text initial keyboardInput = initial ::: delay (
    let (char ::: chars) = adv keyboardInput
    in text (initial ++ [char]) chars
    )
```

It takes an initial list of characters and a delayed stream of characters and produces a stream where the incoming characters are continuously appended to the list, building up a string. In order to build on the text that has already been written, it is passed as an argument in the recursive call. To finalize the input field, we need to be able to reset the text in response to a mouse click. To do that, we use the `select` primitive to synchronize the text stream and mouse input.

```
resettableText :: Stream (List Char) -> Box (O Input) -> Stream (List Char)
resettableText (txt ::: txts) mouseClick = txt ::: delay (
        case select txts (unbox mouseClick) of
            Left txts' _ -> resettableText txts' mouseClick
            Right txts' _ -> resettableText (text kb) mouseClick
            Both _ _ -> resettableText (text kb) mouseClick
    )
```

When only the clock of the text stream ticks, the new text stream is used to call recursively. However, when we receive a mouse click, the text stream is reconstructed such that it is empty again. The same thing happens if both streams tick at the same time. The representation of the mouse click might seem odd at first as delayed computations usually cannot be boxed. In fact the only delayed computations that can be boxed are input channels. It is necessary for `mouseClick` to be boxed otherwise we could not use it under the scope of delay. Section 3.5 goes into more detail about input channels.

3 Analysis

In order to implement the Async RaTT calculus [5], we choose to build upon the existing embedding of Rattus in Haskell. Reusing the embedded implementation means that we do not need to implement Asynchronous Rattus from scratch, as it allows the reuse of logic from Rattus. It also means that the language can be used with the large Haskell ecosystem. Hence, we have implemented Asynchronous Rattus as a package for the Glasgow Haskell Compiler.

When designing Asynchronous Rattus, we had the following design goals in mind.

1. Provide the Async RaTT guarantees of causality, productivity and freedom of implicit space leaks.
2. Shield the programmer from dealing with clocks.
3. Provide a good interface between Haskell and Asynchronous Rattus.

We attempt to have our implementation provide the guarantees of Async RaTT by reusing and adapting the scope rules of Rattus. Except for the new `select` operator, the scope rules of Async RaTT is a subset of Rattus' scope rules, meaning that the Async RaTT guarantees should hold when using the modified scope rules of Rattus. `select` has the same scoping rules as `adv`, so we conjecture it does not break the guarantees. We shield the programmer from using clocks, since manually handling clocks is error-prone and makes programs more complicated. To shield the programmer from clocks, we infer all clocks automatically using a combination of static analysis and run-time checks. Ensuring a good interface between Asynchronous Rattus and Haskell is important in order to make the language useful in practice. This is implemented in the Channel library, which provides an API such that Haskell code can safely interact with Asynchronous Rattus code.

3.1 Components of the implementation

The implementation consists of several parts. The Asynchronous Rattus primitives are implemented as plain Haskell code. The custom scope rules, the clock inference algorithm, and the pushing of input values through the system are implemented in the compiler plugin. The interface between Haskell and Asynchronous Rattus, the Channel API, is implemented as a Haskell library. Lastly, we have developed a small standard library to demonstrate the usefulness of the language.

All of the components are bundled in a single Haskell package.

3.2 Primitives and their representation

We have already seen that Asynchronous Rattus primitives look like normal Haskell functions to the programmer. As seen from the typing rules, `delay x` produces a value of type $\ominus A$, given that x types to A under a typing context with a tick on the clock of the delay. In Async RaTT, values of this type are described as: "a pair consisting of a clock θ and a computation that can be executed to return data of type A" [5]. Thus we need to decide on a representation for clocks and for delayed computations.

Recall that a clock represents the set of input channels for which a delayed computation is waiting for input. The representation of an input channel is quite arbitrary; there just needs to be a way to distinguish them from each other, so that we can know which input channel received input. Thus we choose simple integers to identify input channels. A natural representation of a clock is then a Haskell Set of integers:

```
type InputChannelIdentifier = Int
type Clock = Set InputChannelIdentifier
```

This is a simple representation of clocks that makes them easy to manipulate.

In order to represent computations, it seems natural to use functions that accept the newly available input data as an argument. However, it is unclear which type the argument should have. This type constrains what can be put into an input channel, so we want to offer the programmer the most flexible type possible.

The simplest solution is to provide a type that can wrap the primitive types as well as some common use cases like tuples. If this data structure were sophisticated enough, it might be able to encode all possible values that programmers would want to provide as input on input channels. However, it seems quite restrictive and cumbersome to work with.

Another solution is to let the input value be an additional type parameter. This is a quite general approach, which allows the programmer to supply their own type for input. It has the drawback that the signatures become more complicated, and that all input must be of the same type. Furthermore, all types deriving from delayed computations must be parameterized as well, or hard code the input types they support.

A third solution is to somehow convert arbitrary types into a common type when inserting it into the input channel and converting it back once the value has been received. Since there is an identifier for each channel, this identifier could be linked to the type of values in that channel. However, we do not see any way to implement this in a type-safe manner.

Asynchronous Rattus implements the second solution, such that delayed computations are parameterized by the type of the input value they accept. Thus, a value of type $\textcircled{\exists} A$ in Async RaTT is implemented as follows in Asynchronous Rattus:

```
type InputValue a = (InputChannelIdentifier, a)
data O v a = Delay Clock (InputValue v -> a)
```

Along with the value, the function receives the identifier of the input channel so delayed computa-

tions know which input channel the input value came from. This type has information about which input channels it is legal to provide input on, and it has a stored computation which computes a value of type A given a new value on one of those input channels. From this representation of $\exists A$, it is quite easy to implement `adv` and `select`. To implement `adv`, we simply accept a delayed computation and an `InputValue` which we use to compute a result of the delayed computation.

```
adv :: O v a -> InputValue v -> a
adv (Delay cl f) inpVal@(chId, _) | chId `elem` cl = f inpVal
adv (Delay cl _) (chId, _) = error "Async Rattus internal error"
```

The function could be shorter and faster if it did not check that the provided input value matches the clock. This should never happen, but we choose to be cautious. This check makes sure that input on an input channel that is not in the clock of the delayed computation fails rather than producing the wrong results.

The implementation of `select` should take an `InputValue` and two delayed computations, and then determine if one or two of the computations can be executed given the input value. This can be determined by inspecting the clocks of the two delayed computations. We then need a data type to represent the return type of the `select` primitive. According to the Async RaTT typing rules from Figure 1, it should be one of three things, depending on which values are available; both the obtained values, the obtained value of the first delayed computation along with the second delayed computation, or vice versa. Thus we can implement `select` as follows:

```
data Select v a b = Left !a !(O v b) | Right !(O v a) !b | Both !a !b

select :: O v a -> O v b -> InputValue v -> Select v a b
select a@(Delay clA inpFA) b@(Delay clB inpFB) inputValue@(chId, _)
  | chId `elem` clA && chId `elem` clB = Both (inpFA inputValue) (inpFB
    inputValue)
  | chId `elem` clA = Left (inpFA inputValue) b
  | chId `elem` clB = Right a (inpFB inputValue)
  | otherwise = error "Tick did not come on correct input channels"
```

We provide the received input to those computations that can use it. In this case, we choose the cautious approach as well.

The primitives presented so far are not the ones that programmers use, however. Recall that a core abstraction in FRP is that of time-varying values, meaning that programmers should be concerned

with the relationship between time-varying values instead of the concrete value. We also see this in Async RaTT, where the `adv` and `select` primitives do not accept arguments; the values are passed implicitly. Thus, the two primitive implementations are instead used behind the scenes as will be described in section 3.3.4. The actual primitives that library clients use are the following:

```
asyncRattusError = error "Did you forget to mark this as an Async Rattus module?"

delay :: a -> O v a
delay _ = asyncRattusError

adv :: O v a -> a
adv _ = asyncRattusError

select :: O v a -> O v b -> Select v a b
select _ _ = asyncRattusError
```

These functions obviously do not do anything useful. Instead, they are replaced with the more useful primitives shown above. The variants of `adv` and `select` which accept values are named `adv'` and `select'`, respectively.

There is an additional primitive which is part of the Async RaTT calculus, and which surprisingly turns out to be useful in practice - the `never` primitive. In Asynchronous Rattus, it is the only way to create a delayed computation that does not expect input on any channel, which means it can never be computed. It is defined simply as:

```
never :: O v a
never = Delay empty (error "Trying to adv on the 'never' delayed computation")
```

Here, "empty" is the empty set, and the computation will cause a runtime error if ever executed. This is fine since it should never be executed. As we will see in section 3.5, this primitive is used to implement some functionality in the standard library for Asynchronous Rattus.

The Asynchronous Rattus primitives presented so far are the foundation of the language. While they can be used manually, they do not allow the abstraction of time-varying values. They force the programmer to calculate clocks, and they allow programs with space leaks. This is remedied by the GHC plugin.

3.3 The GHC plugin

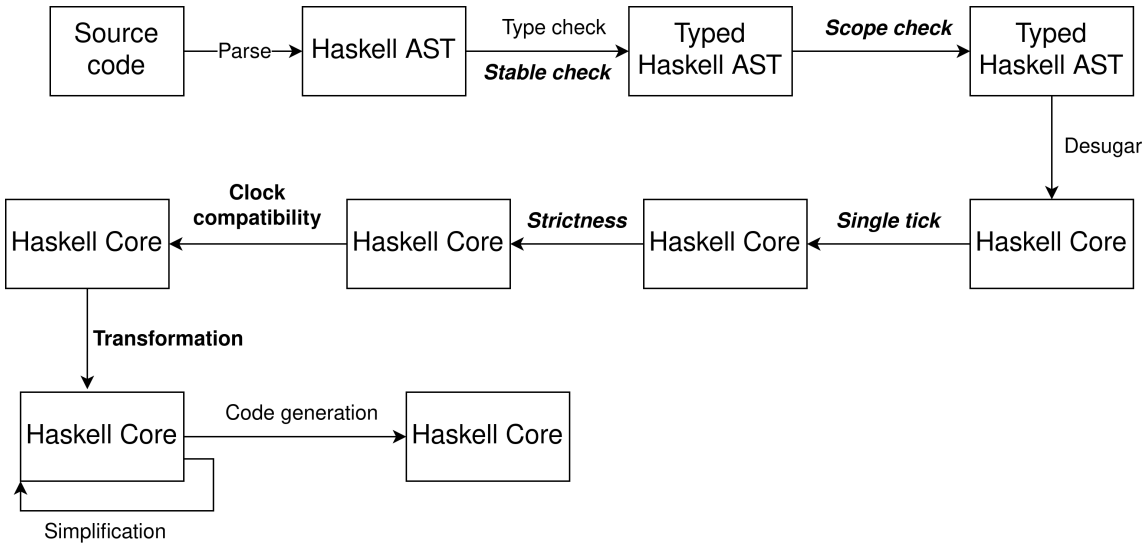

Figure 2: Simplified GHC compilation phases, with phases added by the Asynchronous Rattus compiler plugin in bold. Those phases which are additionally italicized have not been modified, or have been modified minimally from the Rattus implementation. Inspired by a similar figure by Bahr [4].

GHC compiles source code in multiple phases, or passes, as seen in Figure 2. It can also be seen that there are multiple representations of the code from the source code all the way to the executable. The GHC Plugin API allows manipulating representations of the code at different stages of the compilation.

At the type-checking stage, we reuse the stability check from the Rattus source code, which implements custom rules for determining whether a type is a member of the `Stable` type class.

Then we do a pass on the typed Haskell AST where the scope rules of Asynchronous Rattus are checked. Asynchronous Rattus has the same scoping rules as Rattus [4], but with the addition of the `select` primitive. The scope checking pass has been extended to support this.

Next, the AST is desugared into Haskell Core, which is a much simpler representation of the AST with only the most basic constructs. Here, we perform the single-tick transformation from Rattus [4, p. 19], which has been adapted to support `select`. We also reuse the strictness pass from Rattus [4, p. 34] to ensure eager evaluation when necessary.

In the next pass, we do two things; we check that clocks within a delay are compatible with each other, and we double-check the scope and primitive rules. The clock compatibility check is a core

part of our clock inference algorithm, and it allows a simpler implementation of the code generation pass that follows. The double-checking of scope rules is largely inherited from Rattus. Our reason for redoing the scope check is the same as Bahr's [4]; because the Core representation is much simpler to work with than the Typed Haskell AST, it is easier to catch any bugs in the implementation of the scope check on the Typed Haskell AST.

Finally, we transform the Core AST. This pass implements two features of Asynchronous Rattus; automatic calculation of clocks at runtime, and hiding the input values being pushed around in the system from the programmer, allowing them to reason about time-varying values instead.

3.3.1 Enforcing scope rules

Checking that programs conform to the Async RaTT scope rules happens on the typed Haskell AST. Our approach stems from the observation that if the `select` primitive is checked the same way that `adv` is, we can check the scope rules of Async RaTT the same way it is implemented for Rattus. This allows the reuse of the scope-checking logic from Rattus with only minor modifications to support `select`. In this pass, we verify all typing rules of Async RaTT, except that the clocks within a delay are compatible. That check is deferred for the simpler Core representation.

3.3.2 Single-tick and strictness

We retain Rattus' single-tick and strictness passes, which make transformations to prevent implicit space leaks. The single-tick transformation is a technique used by Bahr [4] to allow using `adv` on expressions of type $\textcircled{\exists}$, rather than just variables. We have extended this to include `select`. The strictness transformation is unchanged from the Rattus implementation. It transforms the AST such that arguments of lambda functions and variables bound in let-expressions are evaluated eagerly.

3.3.3 Clock compatibility check

The purpose of the clock compatibility check is to identify delayed computations that have subexpressions with incompatible clocks. As we saw from the typing rules, all occurrences of `adv` and `select` in a given `delay` must require the same clock. We ensure this with *symbolic or static clocks*, as opposed to *concrete or dynamic clocks*. Recall that in the Async RaTT calculus [5], we must delay with respect to a clock. This clock should be the same as that of all applications of `adv`/`select` inside the delayed computation. Thus, a term that will type correctly can be: $\mathsf{delay}_{cl(x)\sqcup cl(y)}(\mathsf{selectxy})$. At compile time, we read this as "the union clock of the variables x and y". Crucially, this is a

legitimate symbolic clock, even though we do not know what that clock will be at run time. At run time, clocks are necessary to determine which of the two delayed computations, or both, can produce a new value. This corresponds to the $cl(v)$ operation, that is "look up the actual clock of this value", and it is implemented by simply saving the clock as part of the delayed computation. Asynchronous Rattus keeps track of symbolic clocks simply as a set of variables that `adv` or `select` is called on. This corresponds to the clock of the variable that `adv` is called on, or the union of the clocks of the two variables `select` is called on.

To check that no subexpressions of a `delay` have incompatible clocks, we find all applications of `adv` or `select` in the `delay`. From each application, we calculate its symbolic clock, which is simply a singleton set for `adv` and a set of the two variables for `select`. For each delay, there must be exactly 1 unique clock in its argument; if there are none, the clock of the delayed computation must be $\emptyset$, meaning that we might as well use `never`. We disallow such a delayed computation since it is most likely a programming error. If there are multiple distinct clocks, they are incompatible. This allows multiple occurrences of `adv`, as long as it is called on the same variable each time, and the same for `select` as long as the same two variables are used. It also allows `adv x` and `select x x` to coexist within a delayed computation (though this is hardly useful).

As mentioned, this pass also double-checks the scope rules of Async RaTT. We retain this check to catch any errors in the scope checking pass on the Typed Haskell AST, since the Core AST is simpler to work with. We check the scoping rules for the primitives, we check that `adv` and `select` are only used within `delay`, and we reject nested delays. We reject nested delays because we have not prioritized implementing support for them.

3.3.4 AST Transformations

We transform the AST for 2 reasons: to generate code that computes clocks of delayed computations, and to pass input values through the delayed computations in the emerging dataflow graph.

Passing input values around in the system only requires that we change the three programmer-exposed primitives `delay`, `adv` and `select` to their internal counterparts which facilitate passing values: `Delay`, `adv'` and `select'`. Given that compilation has not failed yet, we know that all occurrences of `adv` and `select` are nested below a `delay`. This allows a simpler implementation where it is not necessary to handle errors, for instance `adv` and `select` appearing outside a `delay`. Thus, the AST is recursively traversed until a term is met, in which `delay` is applied. Such a term could be the right-hand side of the `plusK` example given earlier:

```
delay (adv laterI + k)
```

We want to transform this term into the following:

```
Delay (clock?) (\inputValue -> adv' laterI inputValue + k)
```

As seen, the `inputValue` is received as an argument to a lambda function, and passed to the delayed computation `laterI` using the `adv'` function, which actually implements advancing, as shown in section section 3.2. Asynchronous Rattus implements this by generating a new variable, and substituting all occurrences of `adv` or `select` with their internal counterparts. This is sufficient to pass around input values implicitly.

The clock of the delayed computation cannot be computed statically, since the clock of the delayed computation for a given variable can vary at runtime. However, the *symbolic* clock can be computed statically. Since clock compatibility has been verified, we can calculate the symbolic clock of the delayed computation from any application of a primitive. Thus we know the set of variables from which we should obtain the concrete clock. As mentioned, we store the clock of a delayed computation within the delayed computation itself, so we just need to extract it. This is done with the following simple function:

```
extractClock :: O v a -> Clock
extractClock (Delay cl _) = cl
```

This function can then be used to calculate clocks at run time. Thus the transformed code for the `plusK` example will be as follows:

```
Delay (extractClock laterI) (\inputValue -> adv' laterI inputValue + k)
```

Since this delayed computation just waits for a value to arrive, and then does something to it, it will have the same exact clock as the value it is waiting for. For terms involving `select`, it is not as simple. In that case, both dynamic clocks must be extracted, and a set union must be performed. Reconsider the `maxLater` example. After the transformation pass, the code would have been transformed to the equivalent of the following.

```
maxLater :: O Int -> O Int -> O Int
maxLater first second =
  Delay (union (extractClock first) (extractClock second)) (\inputValue ->
    case select first second inputValue of
      Both first' second' -> max first' second'
      Left first' _ -> first'
      Right _ second' -> second'
  )
```

Here, `union` is Haskell's union operator for the set datatype, which is part of the standard library. All programmer-exposed primitives have been replaced with their internal counterparts since they cannot be executed. Also, there is now a mechanism for computing and storing run-time clocks. Thus this implementation can run in practice.

It has still not been clarified how clocks and delayed computations are created in the first place. So far, it has only been shown how new delayed computations can be derived from existing delayed computations. This issue is handled by the Channel API.

3.4 Channel API

The Channel API is the component of Asynchronous Rattus which lets Haskell and Asynchronous Rattus code interact with each other. It implements the abstraction of *input channels* and *output channels* from Async RaTT [5] such that Haskell code can provide input on input channels and get output from output channels. For Asynchronous Rattus code, the primary goal with the interface is to declare and obtain input channels from which streams and other logic can be built. For Haskell code, the primary goal is to provide input and get output. When providing input, it is helpful to use meaningful names (as opposed to input channel identifiers, which are just integers). It must also be possible to determine whether a given delayed computation is waiting for input on a given input channel, so that Haskell code can know which delayed computations can produce new output for newly obtained input.

In order to keep the API as simple as possible, input channels are boxed delayed computations that produce values of the same type as the input values, that is they have type `Box (O v v)`. They are boxed because input channels can be boxed under the Async RaTT [5] typing rules, but we have represented them as regular delayed computations, which cannot be boxed. Since they are boxed initially, there is no need for Asynchronous Rattus code to box them, thus there is no need for Asynchronous Rattus to treat them differently. Input channels emit the value which is provided on

their associated input channel identifier. Asynchronous Rattus code can build on them by unboxing them and treating them like any other delayed computations. The Async RaTT concept of output channels is simply any delayed computation in Asynchronous Rattus. A crude way to get output an output channel would be to use `adv'` directly, but it requires that we know in advance the legal input channels for the delayed computation, or an exception will be raised. Thus we check that the input channel is legal to provide input on before calling `adv'`. We also provide a way to query the clock of a delayed computation.

These functions do not expose the input channel identifiers to the programmer. Rather, each channel is given a name when the Asynchronous Rattus code declares the input channels, and a bidirectional mapping between names and input channels is stored in the closure of the functions used by the Haskell code. This ensures that programmers never see the input channel identifiers, only the names assigned to the channels.

3.5 Standard library

The final component that Asynchronous Rattus offers is its standard library written in the language itself. The main purpose of the standard library is to provide programmers with tools that ease the process of working with the first-class values of Asynchronous Rattus, such as ⋿ and streams. Additionally, it demonstrates the practical usefulness of the language. It consists of three modules; `Later`, `Stream`, and `Strict`.

The `Later` module offers utility functions to interact with delayed computations. The most notable function is `selectMany`, which internally calls the function `selectMany'`.

```
selectMany :: List (O v a) -> O v (List (Int :* a))
selectMany = selectMany' 0

{-# ANN selectMany' AllowRecursion #-}
selectMany' :: Int -> List (O v a) -> O v (List (Int :* a))
selectMany' _ Nil = never
selectMany' _ (x :! Nil) = delay ((0 :* adv x) :! Nil)
selectMany' n (x :! y :! Nil) =
    delay (
        case select x y of
            Both a b -> (n :* a) :! (n+1 :* b) :! Nil
            Left a lb -> singleton (n :* a)
            Right la b -> singleton (n+1 :* b)
```

```
    )
selectMany' n (x :! xs) =
    let xs' = selectMany' (n+1) xs in
    delay (
        case select x xs' of
            Both a b -> (n :* a) :! b
            Left a lb -> singleton (n :* a)
            Right la b -> b
    )
```

Given a list of delayed computations, it is possible to create a delayed computation whose result is a list of strict pairs, constructed with the binary infix operator `:*`. These pairs correspond to the indices of the delayed computations that have been computed and their result. The name comes from the fact that `select` is applied to all delayed computations in the input list. `selectMany'` has three base cases. For empty lists, no meaningful output can ever be computed, thus the primitive `never` is returned. If the input list contains a single delayed computation, it is possible to use `adv` on the given delayed computation and return the result. The third is the same as the second but where two delayed computations could be computed when the input was received, so `select` is used to synchronize the delayed computations. If more than two delayed computations are given, the list of delayed computations is synchronized recursively. The `AllowRecursion` annotation is necessary because Asynchronous Rattus ordinarily only allows recursion under `delay`. This is because Async RaTT uses *guarded recursion* [5] such that each computation provably terminates. Additionally, the strict versions of lists and tuples from the `Strict` module are used instead of the lazy counterparts.

When working with streams, programmers need to be able to compose and manipulate them as they wish. Hence, the `Stream` module provides utility functions like the ones Haskell provides for operating on lists. That is, utility functions such as map, filter, and zip.

The `Strict` module contains strict versions of well-known data types in Haskell such as List, Maybe and tuples. Using lazy data types can lead to implicit space leaks, which is why Asynchronous Rattus will generate warnings whenever lazy types are used. Additionally, lazy types are non-stable. Hence, strict types are desirable in Asynchronous Rattus because they are stable and do not introduce implicit space leaks. Furthermore, utility functions for each of the strict types are available.

4 Testing

As described in section section 3.1, Asynchronous Rattus consists of multiple components: the compiler plugin, the channel API, and the standard library. The channel API and standard library can be tested with a standard unit test framework. However, there exists no tooling to test compiler plugins to our knowledge. To test the compiler plugin, we use two Haskell annotations to gain complete control over the compiler phases.

4.1 Compiler plugin

To test that only valid programs can compile, we simply check that a suite of valid programs all can compile. However, to test that specific functions should not be able to compile, we use annotations to mark that a given function is expected to fail. The two annotations `ExpectScopeError` and `ExpectClockError` mark that a function is expected to produce an error in either the scope checking phase or the clock compatibility phase, respectively. We never expect failure in other phases. The annotations can be used via the GHC `ANN` pragma.

There are two reasons we want to be able to differentiate between errors in the two phases, rather than just expecting a failure somewhere in the compilation pipeline. The first is that it allows us to test each phase in isolation. The second is that there are dependencies between compilation phases, which means that it can be unsafe to attempt to continue compilation after a phase has failed. That is, each compiler phase has assumptions about the types of programs it operates on, because it expects earlier phases to have caused compilation to fail if errors occurred. Thus, allowing compilation to proceed to subsequent phases after an expected error has occurred cannot be allowed since it may lead to uncaught errors in subsequent phases. However, GHC provides no way to skip later compilation passes without failing the entire test suite. With the two annotations, the Core passes can simply be disabled if type-checking errors are expected, and the final AST transformation can be disabled if Core errors are expected.

This approach to testing the compilation phases allows the tests to be integrated with Cabal's testing framework. Thus they can be run alongside the unit tests described in section 4.2. An example of a test suite could be the following ill-typed programs, that we want to ensure cannot compile.

```
{-# ANN intPlusOne ExpectTcError #-}
intPlusOne :: O v Int -> Int
intPlusOne laterI = adv laterI + 1
```

```
{-# ANN incompatibleAdv ExpectCoreError #-}
incompatibleAdv :: O v Int -> O v Int -> O v Int
incompatibleAdv li lk = delay (adv li + adv lk)
```

Here, the validity of the two programs is tested. `intPlusOne` is expected to fail during scope-checking because `adv` is used without an enclosing `delay`. When the plugin detects the error, it considers the binding to be successfully compiled, skips the rest of the compilation and starts compiling the program `incompatibleAdv`. This program is expected to fail during the clock compatibility check. Evidently, this is the `add` function we introduced earlier. No scoping rules are violated, but as mentioned earlier the clock of the delayed computation that is being produced cannot be uniquely determined on compile time. Hence, this program fails and the test successfully ends. Ultimately, running this test suite results in success with no unintended behavior.

This approach is used to implement the testing of the two compiler phases. The `illTyped.hs` and `wellTyped.hs` test suites test ill-typed and well-typed programs, respectively.

4.2 Runtime, Channel API & the standard library

Given the testing of the compiler plugin as described in the previous section, we can be reasonably confident that programs can be compiled if, and only if, they are valid. However, we have not yet presented tests of any runtime behavior. This can be done using a standard unit testing framework, since now we need to test program output, not whether it compiles or not. There are no frameworks for testing either Asynchronous Rattus or Rattus code, so we use the Haskell unit testing framework `HUnit`. It allows declaring simple test cases, which is all we need.

In order to test the primitives, we avoid depending on any other Asynchronous Rattus library, and we test that values are propagated correctly. Tests of the Channel API verify that the generated `inputMaybe` and `depends` functions work correctly. We can then use the Channel API when testing the Later and Stream libraries. Finally, the Strict library is tested as a regular Haskell library.

With these tests, we can be reasonably confident that the primitives, the Channel API, and the standard library work as intended at runtime. Thus, we can now be confident that Asynchronous Rattus works as intended.

5 Discussion

Asynchronous Rattus enables us to write reactive programs with ease. Because of the asynchronous nature, we can build isolated components that declare which inputs they depend on to generate a

stream or a delayed computation. We have built a simple spreadsheet application to demonstrate how Asynchronous Rattus can be used to write a simple reactive application. The sheet has four input cells and two output cells. The output cells each compute the sum of two input cells with overlap as shown in Figure 3.

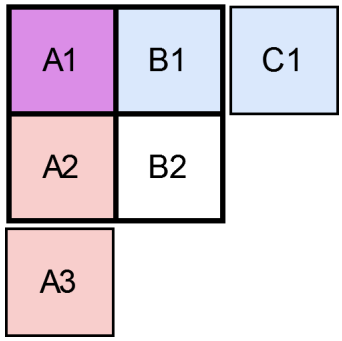

Figure 3: A mini spreadsheet. $A1$, $A2$, $B1$ and $B2$ are input cells. $C1$ and $A3$ are output channels. The coloring signifies which input channels each output channel calculates the sum of.

The input cells are labeled $A1$, $A2$, $B1$ and $B2$, while the output cells are $C1$ and $A3$. $C1$ computes the sum of $A1$ and $B1$, while $A3$ computes the sum of $A1$ and $A2$. That is, $C1$ and $A3$ will be recomputed each time input arrives on either of their input channels. Additionally, no output channel depends on $B2$.

To showcase the application, we have implemented a small parser that allows us to interact with the input cell. The result of a couple of interactions can be seen below.

```
> cabal run
(0 :* 0)

> A1 = 10
(10 :* 10)

> B2 = 40
That cell does not affect the output cells

> B1 = 40
(50 :* 10)
```

Figure 4: Interaction with our small spreadsheet application. The output cells are represented as a strict pair (type from `Strict`) with $C1$ being the left and $A3$ being the right

The output streams have an initial value of 0. Then a value arrives in $A1$, which both $C1$ and $A3$ is

dependent on, causing them to recompute. A value arrives in $B2$, but as stated in Figure 4 neither $C1$ nor $A3$ depends on $B2$. Thus, no computations are needed at all. Lastly, a value arrives on $B1$. $C1$ is the only output channel that depends on $B1$ allowing us to only recompute $C1$.

From this application and its example in Figure 4, we see the benefits of the resulting language Asynchronous Rattus. First of all, one of the goals was to lift the restriction of a global clock. Each delayed computation can have its own clock, which is illustrated clearly by the example. Here, $C1$ and $A3$ share the input channel $A1$, however they each depend another input channel besides $A1$. Thus, Asynchronous Rattus is able to determine precisely which output channels needs to be recomputed whenever an input arrives at one of the input channels, removing unnecessary computations. Secondly, from the source code in `Sheet` we see that we never specify any of these clocks. They are handled by Asynchronous Rattus behind the scenes allowing us to focus on building the application by reasoning about the relationship between time-varying values. That is, we specify the relationship between a certain output channel and the input channels it needs to depend on. Lastly, Asynchronous Rattus offers all its primitives as first class values, which we take advantage of. This allows us to create this simple application where the core logic can be defined with under 10 lines of code. Asynchronous Rattus has allowed writing this program such that the components are easy to reason about, can be composed and can be reused. Expanding this application with further functionality, e.g. new output cells, would be quite simple thanks to the flexibility and abstraction level offered by Asynchronous Rattus.

5.1 Limitations

As mentioned in section 3.1, the core of Asynchronous Rattus is implemented as a compiler plugin for the GHC compiler. This approach has a lot of benefits, such as utilizing the GHC compiler to compile programs into efficient machine code, and not needing to write a compiler from scratch. However, by creating a compiler for Asynchronous Rattus you would have full control over the compiler. Using a purpose-built compiler could potentially reduce the amount of subtle and unexpected bugs resulting from relying on the internals of GHC. Such a bug was discovered shortly before the project deadline.

Whenever Asynchronous Rattus determines clocks statically and generates code to resolve them dynamically, as described in section 3.3.3 and section 3.3.4, it requires access to a specific type class variable. This is no problem for the language itself, since this is always available for Asynchronous Rattus. However, the problem can arise when compiling an application that does not use this type class constraint. In this case, the type class variable will not be available during the compilation process, causing the compilation to fail unexpectedly. It is possible to circumvent this by using

a dummy function that forces the GHC compiler to load this type class variable as well. We see this as a minor inconvenience that does not affect overall usefulness of the language prototype. Nonetheless, it demonstrates the brittleness of the current approach which may be solved with a custom compiler. You would gain full control at the cost of losing the Haskell ecosystem, which is available when Asynchronous Rattus is implemented as a compiler plugin.

Given our testing, described in section 4, we are confident that we have shown that Asynchronous Rattus works as intended. However, it is uncertain how well it scales with increased load. Since the aim of this project was to implement a prototype, there has been no intention to include this form of testing. Nevertheless, you could imagine scenarios that might be troublesome for Asynchronous Rattus. What happens if some excessive amount of input arrives on an input channel (i.e. a sensor that is in an erroneous state) in a short time frame? At the moment, this would force re-computations for each input, but another solution could be to discard or limit the input in some way. Furthermore, since there is no relationship between output channels, parallelism should be an interesting topic in Asynchronous Rattus. In either case, having testing suite specifically for performance would be desirable. This testing suite could also increase our confidence that Asynchronous Rattus is free from implicit space leaks.

6 Future work

The Asynchronous Rattus language provides a foundation on which reactive applications can be built. As mentioned, there has not been much focus on ensuring that Asynchronous Rattus is performant enough for production use. Investigating and possibly improving the performance of Asynchronous Rattus would help the language mature.

Another area of work is the type of input that is currently used. Currently, the user has to use a single type as input to all input channels. We find it most useful to define an algebraic datatype that can hold any type that must be given as input, and pattern match on the correct case in Asynchronous Rattus code. This approach works, but it is inconvenient and it will become unmaintainable in large applications. Given that we have an identifier for each input channel, we should be able to connect the type of values on an input channel to the identifier. This would mean that input channels could produce values of any type. Thus there would be no need for an algebraic wrapper type leading to simpler code. It would also mean that we could get rid of the extra type parameter on delayed computations, which is necessary in the current solution. This extra type parameter can be hidden by using type aliases, except when developing libraries that must work with all input types. We believe that removing the need for the type parameter in the first place would make the language easier to use and more practical.

7 Conclusion

Asynchronous Rattus successfully implements an asynchronous modal FRP language. This means that it provides access to primitives while using an asynchronous update model. Its semantics follows the those of Async RaTT quite closely, but with minor modifications.

The language shields the programmer from clocks. It does so by virtue of a novel clock inference algorithm that detects clock inconsistencies and calculates clocks dynamically at runtime. Furthermore, the abstraction of time-varying values is implemented by streams. This is made possible by the AST transformations that handle input value propagation behind the scenes.

The interface between Haskell and Asynchronous Rattus works well. Programmers use meaningful identifiers for input channels and the API provides a safe way for programmers to interact with input channels and output channels.

The implementation is easily available as a Haskell package. The core language, the compiler plugin, and the provided libraries have been tested such that we are confident that Asynchronous Rattus works as intended. Asynchronous Rattus aims to provide the guarantees of Async RaTT by reusing the working implementation from Rattus, but we have neither proven nor tested it.

It has been demonstrated that Asynchronous Rattus is useful for developing reactive programs, and the standard library provides a useful starting point for doing so. This means that developers now have a tool for developing reactive applications that are asynchronous in nature while staying in the functional paradigm.

8 References

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

[2] Paul Hudak et al. "Arrows, Robots, and Functional Reactive Programming". In: *Advanced Functional Programming: 4th International School, AFP 2002, Oxford, UK, August 19-24, 2002. Revised Lectures*. Ed. by Johan Jeuring and Simon L. Peyton Jones. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 159–187. ISBN: 978-3-540-44833-4. DOI: 10.1007/978-3-540-44833-4_6. URL: https://doi.org/10.1007/978-3-540-44833-4_6.

[3] Bernd Bruegge and Allen H. Dutoit. *Object-Oriented Software Engineering Using UML, Patterns, and Java*. 3rd. Prentice Hall Press, 2009. ISBN: 0136061257.

[4] PATRICK BAHR. "Modal FRP for all: Functional reactive programming without space leaks in Haskell". In: *Journal of Functional Programming* 32 (2022), e15. DOI: 10.1017/S0956796822000132.

[5] Patrick Bahr and Rasmus Ejlers Møgelberg. "Asynchronous Modal FRP". 2023. arXiv: 2303.03170 [cs.PL].

9 Appendix

The code for the project is available on GitHub at `https://github.com/GregersSR/Asynchronous-Rattus`.

Additionally, it can be found in the appendix of the project submission.