Implementation of a Polymorphic Functional Reactive Programming Language with Modal Types.

Natalia Ixchel Rodas Ralda - nrod@itu.dk
Thor Christian Stenbæk - thcs@itu.dk

Spring 2024

Course Code: KISPECI1SE
https://github.itu.dk/thcs/Thesis-code.git

Contents

Abstract

Functional reactive programming is proposed to simplify the development of reactive programs that require handling of data streams and dynamic behavior. However, the existing FRP languages have had some challenges while operating these complex signals and streams, such as the lack of a sufficiently robust type systems, which can lead to non-casual programs and implicit data leaks. Additionally, many FRP implementations are built as libraries within other languages. This means that they have to work with or adapt to the type system of the host language. This can cause problems in creating a dedicated FRP-type checker, which can lead to complications, reduced efficiency, and sometimes, memory leaks. The goal of this project is to implement a faithful version of the FRP language called Rattus introduced by Patrick Bahr (Bahr, 2022) that attempts to deliver an FRP language that uses, extends, and simplifies previous modal types from FRP calculi, while maintaining productivity, causality, and no implicit space leak. Moreover, the language should offer native support for signals, streams, and other FRP constructs. The implementation of the language includes building a parser, a type-checker, and an interpreter. This language should perform an advanced type inference for functional reactive programming in order to handle modal types, thereby avoiding the need for explicit type annotation by the programmer. This feature is crucial in enhancing the usability and accessibility of the language. The language should have reasonable surface syntax supported by the type system and should use some of the Rattus-specific terms implicitly without the programmer having to explicitly write these. This approach aims to simplify the language, so the programmer can effectively focus on the logic of their applications, instead of underlying FRP constructs. Moreover, the development of an interpreter for the language, so that the user can write and execute FRP programs, will provide insights regarding the efficiency and practicality of Rattus.

1 Introduction

Functional reactive programming (FRP) merges the immutable principles of functional programming with the handling of data streams from reactive programs while avoiding shared states. This aims to simplify and optimize the reactive programs that are used in graphical user interfaces (GUIs), robots, and other systems that expect external inputs and handle them as data streams that need to be computed.

Nevertheless, FRP languages present some limitations. For instance, their type systems often allow non-productive, and non-casual programs. Additionally, a major limitation is the presence of space leaks, caused by retaining old data. Patrick Bahr *et. al* proposes to solve these obstacles with Rattus, an FRP language. Rattus is based on existing FRP calculus, and Bahr simplifies it and extends it while preserving productivity, causality, and preventing implicit space leaks.

We created a language based on Rattus, as described in Bahr's work in [1]. Our language contains a typechecker able to type infer polymorphic types and incorporates a parser and an interpreter. The language aims to have a syntax similar to *F#*, so the final user finds some familiarity with it. However, behind the curtains, our type checker will implicitly place some expressions that Rattus introduces to preserve productivity and causality. The language should also be able to infer the type of any expression, including expressions with modal types, so the user doesn't need to annotate them in their language. In summary, our project aims to bridge the gap between the theory of Rattus and its practical application in software development.

In this paper, we will describe the research and how it applies to the development of our language. The first section covers the theory of FRP and Rattus' background. It briefly introduces the concepts and common problems of FRP and how Rattus proposes to solve these. Additionally, in this section, we start to discuss and describe the basic and functional expressions, as well as the types that we use in our program. Then, we describe in more detail how Rattus works, introducing more expressions and types that we include in our language. Finally, we explain in detail how Rattus addresses the common problems of FRP that we introduced before.

The third section describes the scope of our language. First, it describes the syntax that the language will read from the user and then interpret, based on the expressions and types we introduced in the previous section. Then it provides an example of a function that consists of several of our expressions and describes the results that our language should return. Additionally, we describe the importance of some of the expressions introduced by Rattus to make our language work as intended. Finally, we show two examples consisting of simple games that are built using our expressions and we briefly explain how our lan-

guage evaluates them.

In the fourth section, we create a summary of the expressions that we have been introducing. These terms are going to be used to build the abstract syntax tree that our type inference and interpreter will read and compute. In the fifth section, we describe in depth the process of how our language parses the syntax provided by the user to the abstract syntax tree. Besides explaining how our parser translates the syntax, we also included several examples to show how the parser reads different scenarios.

The sixth section explains in depth how the type inference works. It starts by describing polymorphic types and the type inference algorithm that most ML programming languages use. In this part, we introduce concepts, algorithms, and functions that our typechecker applies. Furthermore, it describes how we will extend these algorithms with the modal types of Rattus. Then, it provides a detailed explanation of how we brought this theory to a code implementation that takes an abstract syntax tree and returns the most generalized type for it. Additionally, it explains how this typechecker also "infers" where to place two Rattus terms, so the user doesn't have to include it in their syntax. To wrap this section, it includes examples that illustrate how this term and type inference works.

The seventh section explains the interpreter of our language, following a similar structure as the previous one. First, it describes how the interpreter evaluates expressions and returns values, introducing the structural operational semantics on which we base the interpreter. Secondly, it explains how the Rattus interpreter works according to Bahr in [1] and provides some examples. Then, it continues by explaining in detail how we implemented the theory of the interpreter to our code, while also showing some examples.

Lastly, we give a small explanation of how all of these comes together in our code. Then we discuss how the implementation is proven to solve the obstacles described for FRP. We show how the implementation satisfies causality, productivity, and is absent of space leaks. We argue that these three common FRP problems have been solved by Rattus, and therefore they are not present in our implementation, as it was based on Rattus. This makes our implementation a functional reactive programming language that inherently avoids space-leaks, while maintaining causality and productivity.

2 Background and Theory

We introduce a brief description, benefits, and limitations of functional reactive programs with a short explanation of how Rattus initially approaches common FRP obstacles. Furthermore, we start to explain the theory behind the expressions and modal types that we use for the implementation of Rattus. We name

and describe the basic expressions and types, which can be used for any functional language. Afterward, we go into detail regarding Rattus, how it works, and the terms and types that we shall add to our collection. Along the way, we also introduce typing rules for anything we introduce. Lastly, we explain in detail the problems facing FRP languages and how Rattus solves them.

2.1 Functional Reactive programming

Functional reacting programming (FRP) is a paradigm that combines functional programming with reactive programming. Functional programming is a paradigm of programming where functions are first-class citizens; it emphasizes the avoidance of shared states, mutable data, and side effects. Reactive programming, on the other hand, is centered around handling asynchronous data streams and event streams, such as keyboard inputs or mouse clicks; it then "reacts" to these inputs appropriately. FRP merges these two paradigms, enabling reactivity to data streams and events, while adhering to the principles of functional programming.

Functional reactive programming was originally formulated by Conal Elliot and Paul Hudak [3] on the ICFP 97 paper Functional Reactive Animation. In this paper they formulated two important first-class values for FRP: *Signals* (*behaviors*), and *events*. *Signals* represent values that will change over time continuously; such values can change at any point in time, but a signal can always expect that some value is given at a point in time. A rather mundane example of a *signal* is the position of the mouse on a computer screen. Whenever the signal is accessed, we will get a value associated with the position of the mouse. *Events* represent more specific instances that may occur when something happens. This could be the position of the mouse when the mouse is being clicked. These two elements are now considered first-class values; this means that we can use them as parameters and arguments in functions; we can manipulate and we can even have them return from functions as values.

A straightforward, but naive, representation of signals through time can be represented as a stream of signals:

```
data Str A = A :::  (Str A)
```

Here we have a stream of type *Str A*. It has a head of type *A*, as well as a tail of type *Str A*. The head represents the current value of the signal, meaning what value the signal has in the "present". The tail represents the value of this signal at a specific point in time in the future. [1]

This naive representation comes with a few concerns, however. Once we have a representation like this, it's quite easy to come up with ways of manipulating streams; you could apply a function to each element of the stream. However, applying a function in this manner can lead to having built a program that has present values that depend on future values, namely called: non-causal programs. This is a common FRP problem. Another frequent problem is implicit

space leaks; this means space leaks that are not intended by the programmer but are a mistake in the implementation of the FRP language. The space leak is caused because the language retains old data, and this causes the program to increase its memory usage gradually over its run time; this will eventually lead the program to run out of resources. These problems are very common and have been big hurdles in building FRP languages.

In recent FRP proposals, the use of *modal types* has been suggested to solve these problems. A modal type can be succinctly defined as a type system based on modal logic; a form of formal logic that indicates the "mode" or "manner" in which a statement is true. A new modal FRP language is **Rattus**, as introduced by Patrick Bahr *et al.* [1]; Rattus uses the modal type operator $\bigcirc$ as a way to describe the passage of time. What this essentially means is that a value of type $\bigcirc A$ will in fact be a value of type A in the next time step. The utilization of this modal type $\bigcirc$ to denote time steps within a type ensures causality [5]. Given this newly introduced modal type, the stream of signals, previously defined, ends up looking like this instead:

```
data Str a = a :::(◯(Str a))
```

This way of representing the data of type *Str A*, allows for the head of the stream to have type "now" *A*, defined and accessible in the present, and it allows for the tail to have the type "later" *Str A*, which is only accessible in the future. This modal operator enables guarded recursion for recursive functions; essentially a "delay" modal is placed just before the recursive call and thereby "guards" each recursive call until the next time step. This guarantees causality, since "present" values can no longer depend on future values. It also guarantees productivity, because each element of a stream will be computed in finite time. Rattus also prevents implicit space leaks. This is done via an aggressive garbage collection strategy, where the program effectively forgets values and data that are older than the present. We cover this in more detail in section 7. We also go into detail regarding causality, productivity, and space leaks in section 2.4, after we have introduced the type system of Rattus.

2.2 Expressions and types

In this section, we start to explain the theoretical constructs we need to build the language. In programming languages, expressions are a value or function that can be interpreted to create a new value. Expressions are fundamental for programming languages. In order to start building our language, we need to define the expressions it will evaluate. Such expressions will be part of the abstract syntax tree that the parser will build based on the syntax provided by the user. These expressions have a type, which a type inference function will infer, then an interpreter will evaluate the abstract syntax tree (AST), which will return a value to the user. We will discuss these processes more in-depth in further sections.

To initialize our list of expressions, from now on referred to as *terms*, we started with integer constants, and primary operators (+, -, *, and /).

terms ::=	*Integers*	*CstI - Constant*
	Boolean	*CstB - Constant*
	*+, -, *, /*	*Prim - Primary operators*

For instance, if the user wants to represent a simple addition such as,

```
20+2
```

our parser will translate it to a combination of constants and primary operator represented as abstract syntax. Note that in this language, the term *Prim* can, depending on the operator, evaluate integers and booleans, or other terms that result in integers and booleans when evaluated.

Since our program also handles type inference, we also need a list for *types*, similar to our term list. We will initialize it with the types of the values that we can evaluate at the moment, *int* and *bool*. We add more terms and types until we are able to represent and use our FRP language effectively.

types ::=	*Int, Bool*	*base types*

We can observe that by using these basic expressions, we are not saving the results in values or using functions. In order to accomplish this, we need to extend our list of terms. The lambda calculus gives us a language of expressions in the form of variables and functions [7].

2.2.1 Simply Typed Lambda Calculus

Lamba calculus provides three basic expressions that can be used to write any computable function [9]. We will add these terms to our list of terms so we can create more elaborate expressions in our language.

terms ::=	*x,y,z,...*	*Variables*
	$\lambda x.t$	*Abstraction*
	$t_1 t_2$	*Application*

The term *variable* refers to a string identifier that will be bound to a value in an environment. In this paper, when we refer to an environment, or context, we refer to a map that associates a variable with a value or a type. The term *lambda* represents an anonymous function, for instance,

```
fun x → x + 1
```

Lastly, the term *application* is used to represent a function applied to another term, or in other words, a function that takes an argument. To make use of this expression, we assume that the first term is a function and the second term is its argument. For instance, if we take our previous function and "apply" it to a constant number 5, we would get

```
(fun x → x + 1) 5
```

The simply typed lambda calculus (STLC) is an extension of the lambda calculus by introducing types, giving a more structured way to reason about functions. STLC uses two basic *types*: a base type that could be *int, boolean, etc.*, and a function type that describes the type of its argument and the type of its return value. Since we already have two base types (*int, bool*), we only need to add *function* to our list of types.

$$\begin{array}{lll} \text{types} ::= & \textit{Int, Bool} & \textit{type} \\ & \tau \to \tau & \textit{Function type} \end{array}$$

To check if a type judgment is valid, STLC uses a type system that consists of a set of rules that assign a type to an expression or term. These type systems are useful in programming languages to reduce the possibility of bugs due to typing errors. STLC consists of the following basic typing rules for the expressions defined above.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\text{Var} \tag{1}$$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . t : \tau_1 \to \tau_2}\text{Abs} \tag{2}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\, t_2 : \tau_2}\text{App} \tag{3}$$

The rule for *variable* 1 is simple, as it states that if we can find that x has a type τ in the context Γ, then we can assert that Γ entails that x has type τ. Let's take as an example: a simple variable named x, and an environment Γ that maps x to a type *int*. We can easily look for x in Γ and conclude that it is of type *int*.

The rule for abstraction 2 describes a parameter of a known type τ_1, which we will just add to the context of its body t, and then proceed to get its type τ_2. This will provide the type for the abstraction, which will be of a function type that takes as a parameter the type of x and returns the type of the body t, this is $\tau_1 \to \tau_2$. If we observe the previous example given for *abstraction*, we can observe that it takes an argument of type *int*, which will be added to a constant number, which can only mean that the function takes and returns a type *int*, meaning the function itself would have the type: *int* $\to$ *int*.

Lastly, the rule for *application* 3, that is a function applied to another term, implies that the first term must have a function type $\tau_1 \to \tau_2$, and the second term must have the type of the parameter of the first term, τ_1. If both of these premises are proven correct, then we can assert that the application of t_1 over t_2 is of type τ_2. Let's take the example defined above for *application*, which takes an integer as a parameter and returns another integer, $int \to int$. If we apply this function to the parameter x, which we can assert is of type *int*, then we can conclude that, when evaluated, we will get something of type *int*.

2.2.2 High-Order Functions

High-Order functions can take other functions as parameters and return functions as results. Functional programs treat functions as first-class citizens, which means that they can be used as parameters, returned by functions, and saved as values.

The clearest way of using a function as an argument in another function is first to save the original function, and then pass the the name of the first function to the second function. In order to save a variable as a let-binding, first, we add a Let constructor to our list of terms,

$$\text{terms} ::= \quad \textit{Let } x = t_1 \textit{ in } t_2 \quad \textit{Let-binding}$$

where the first expression is used as an argument that can be used for the second expression. After we get the type of the argument t_1, similarly to lambda in 2, we just add $t_1 : \tau_1$ to the environment of t_2, and then we can proceed to get its type. So, the typing rule looks like this,

$$\frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma, t_1 : \tau_1 \vdash t_2 : \tau_2}{\Gamma \vdash \textit{Let } t_1 : \tau_1 \textit{ in } t_2 : \tau_2} \text{ Let} \tag{4}$$

This now introduces the topic of bound variables and free variables. When we define a function as above, we should note that the expression t_1 defined inside the let-binding, is only visible in the body of the function, this is known as a bound variable. If the function uses a variable that was defined outside the let-binding then it's known as a free variable. For example x here is a free variable,

```
let z = x + 5
```

But in this example, x is a bound variable

```
let x = 5 in x + 1
```

In other words, a variable is bound if it is not free, and a variable is free if it is found in the set of free variables FV defined recursively as [7]:

$$
\begin{array}{lll}
\text{FV(term)} ::= & \{\ x\ \} & \mathit{Var}\ x \\
& \mathit{FV(t1)} \cup \mathit{FV(t2)} & \mathit{App}\ t1\ t2 \\
& \mathit{FV(body)} - x & \lambda x.body
\end{array}
$$

For instance, if we wanted to analyze which are the free variables in the expression that applies a lambda expression to a variable
`((fun z → z) y)`
By using the definition of free variables, we get

```
 FV((λz.z)) ∪ FV(Var y)                Application
(FV(Var z) - FV(Var z)) ∪ FV(Var y)        Lambda
 ({ z } - { z }) ∪ { y }                  Variable
 ∅ ∪ { y }
 { y }
```

So we can observe that the only free variable on this expression is the variable y.

2.3 Rattus

Rattus is a modal FRP language that implements the modal FRP calculi developed by Krishnaswami [4] and Bahr *et al* [5], yet it features a more flexible type system [1]. What distinguishes *Rattus* from others, is its simpler, less restrictive type system, which enhances its suitability for practical applications. *Rattus* employs a modal type system known as Linear Temporal Logic (LTL), which is commonly used in FRP language proposals. This type system introduces two modalities: the *later* modality and the *always* modality, symbolized by $\bigcirc$ (*later*) and $\Box$ (*always*). The $\bigcirc$ modality indicates that the premise is true in the next timestep, while the $\Box$ modality asserts that a premise holds across *all future* timesteps [6]. *Rattus* utilizes these modalities, as defined in *Simply Ratt*, to enhance its type system beyond the STLC calculi, applying $\bigcirc$ for delayed computations and $\Box$ for computations that are always available [5].

Given a stream of type A, denoted *Str* A, the head of that stream has type A and the tail of that stream has type $\bigcirc\ A$ (or *'later' A*). In Rattus such a stream can be represented as:
`Str a = a :::(◯(Str a))`

Observe that the tail of the stream uses the $\bigcirc$ modality, which is introduced by the *term delay*. *Delay* moves the recursive computation one-time step ahead of the current time; each of these steps in time is represented by a ✓. The typing rule for *delay* is as follows:

$$
\frac{\Gamma, \checkmark \vdash t : A}{\Gamma \vdash \mathit{delay}\ t : \bigcirc A} \tag{5}
$$

You read the premise of the typing rule as given the context Γ and a step in time ✓, the term t must have the type A. If this premise holds, we can safely

arrive at the conclusion that in the context of Γ, the delay applied to the term t is of type $\bigcirc\ A$

Notably, the introduction of the ✓ to the typing context ensures, that not only can we represent variables and their respective type, but we can also represent steps in time; the typing context can have multiple ✓s. In a context with one or more ✓s, all variables that are to the left of a ✓ are one timestep older than the ones to the right of that ✓. This also means that for every ✓ introduced, we are delaying the computation one-time step further. Most importantly, a *term* t does not have access to variables in older contexts unless the variable it is accessing is a time-independent variable. These variables are called stable types and refer to base types such as *Int* and *Boolean*; these are always accessible irrespective of whichever timestep a *term* t is in. Other types like the function types and the modal $\bigcirc$ are not considered *stable types*. A variable can be accessed in the context if the following rule holds:

$$\frac{\Gamma' tick - free \quad or \quad A \text{ is stable}}{\Gamma, x : A, \Gamma' \vdash x : A} \tag{6}$$

This simple rule can be read as, given a context Γ, a variable x with type A, and a context Γ', then the variable x is accessible and is of type A, if either of the two above premises holds. The first premise denotes that there is no ✓ to the right of the variable, because that would make the variable too old and therefore should not be accessible. The second premise is that the type A that x has is a *stable type*. To illustrate how the rule for accessing variables and the term *delay* interacts, we look at a function called `constInt` that takes an input and essentially creates an infinite stream of that input.

```
constInt x = x :::  delay(constInt x)
```

In the case of this function, the context is changed for any *term* that is guarded by delay. Namely, for the *terms* inside delay, the context has an added ✓. This means that the first premise of the Rattus variable rule cannot be used, since we cannot access older data, which the x has now become. However, the second premise might still hold; namely that the x has to be of a *stable type*. In this case, the function will type-check, but it will only work for stable types.

Just as we can 'delay' a term and effectively look ahead one timestep into the future, there is also a term that allows us to effectively return to the present: *Adv* (advance). *Adv* is the elimination form of $\bigcirc$. When this term is applied, the variable bindings that were made in the future become out of scope, since *Adv* returns us to the present. Notably, this creates an interesting dynamic, where variables to the right of ✓ go out of scope forever, however, it brings the variables one ✓ to the left back into scope. This means, that variables that go out of scope under a *delay* can be brought into scope again by an *Adv*. Given that advance is the elimination form of the rule for $\bigcirc$, it should be quite clear that it is expected to be used on a term t with a type like $\bigcirc\ A$. *Adv* also is also restricted to contexts with a ✓. The typing rule for*Adv* is:

$$\frac{\Gamma \vdash t : \bigcirc A \quad \Gamma' \text{ tick free}}{\Gamma, \checkmark, \Gamma' \vdash adv\ t : A} \tag{7}$$

This rule has two premises. The first premise states that the term t exists and must have the type $\bigcirc\ A$. The second premise Γ' is tick-free, meaning there are no more steps in time after that. The conclusion means that given a context Γ, a step in time $\checkmark$, and then another context Γ', *advancing* on a term t is of type A.

If we consider a simple example program that transforms a stream. The function *inc* takes a stream of integers and increments each of the integers by one:

```
inc (x:::xs) = (x+1):::Delay(inc Adv(xs))
```

In cases like this where a function takes a stream and transforms it, we must use the elimination type of $\bigcirc$: *Adv*. It is used here to convert the type $\bigcirc$*Str Int* into *Str Int*, given the prerequisites described in the rule. This is because the tail of a stream always has type $\bigcirc$*Str A*, as we see in section 2.3.1.

Next, we discuss how to deal with functions and the *Box* term. Function types are not considered *stable* types. The reasoning behind the classification is that a function can implicitly have temporal values within its closure, and these values could, of course, have arbitrary types. Since functions are not stable types, it should be quite clear that we only have access to a function, if there is no $\checkmark$s; meaning the term trying to access a function must also be in the present. This limits how we can declare and use functions in a big way; if we wanted to apply a function to values situated in future timesteps, this seems impossible. The problem is quite evident in the following code example involving a simple *map* operation. Here, `map` takes a function *f*, a stream *x:::xs* and creates a new stream, where *f* is applied to each of the values.

$(a \to b) \to Str\ A \to Str\ B$

```
map f (x::xs) = f x :::Delay(map f adv(xs))
```

This approach might look correct at first glance, but the problem becomes evident if we remember that f is not a *stable* type and that *delay* introduces a $\checkmark$ into the context. Clearly, inside the delay, a normal function like $a \to b$ is inaccessible. To solve this particular problem, *Rattus* applies the modal type *box* $\square$ that turns a non-stable type of A into a stable type of $\square\ A$. Since we can *"box"* any *term t*, that means that we can also *"box" f*, turning it into $\square(a \to b)$. We can now effectively have 'functions' that are of *stable* type and therefore can be accessed at any point in time. The rule for *box* is:

$$\frac{\Gamma^{\square} \vdash t : A}{\Gamma \vdash\ box\ t :\ \square A} \tag{8}$$

The premise of this uses a new form of context $\Gamma^{\square}$, we call this the "boxed" context. The $\Gamma^{\square}$ context is the present context with all $\checkmark$s removed and with

only *stable* types remaining. It essentially 'flattens' the context and only keeps the stable types. The premise is read as if the *term* t is in a context of only stable types, then it has type A. The conclusion can then be read as, the term *box* applied on t would have the type $\Box A$ if the premise holds.

However, we are not able to apply a 'boxed' function: $\Box(a \to b)$. To apply a 'boxed' function, it is first necessary to eliminate the $\Box$ and turn it back into a regular function like $a \to b$. This is where we introduce the elimination term for $\Box$, namely *unbox*. The *unbox* transforms a $\Box A$ into a A, allowing it to be used in the present. The rule for *unbox* is:

$$\frac{\Gamma \vdash t : \Box A}{\Gamma \vdash \ unbox\ t : \ A} \tag{9}$$

The premise of this states that given a context it has *term* t of type $\Box A$. The conclusion can be read as given the same context, if *unbox* is applied to the same *term* t, it will have type A.

Now, if we try to construct the same *map* function from before with these new terms, we know that the map function should in fact take a 'boxed' function. In the code example below, f needs to be unboxed, whereas it then can be applied to x, then f still being a 'boxed' function, it can still be used inside the context of the delay, since it's still considered stable.

$\Box(a \to b) \to Str\ A \to Str\ B$

```
map f (x::xs) = unbox f x :::  delay(map f adv(xs))
```

We have added the terms described here to our language; we have also added the modal types described, meaning we are now able to evaluate Rattus expressions within the language.

2.3.1 Stream terms and types

Other than the terms introduced by STLC and *Rattus*, we still need a way to represent streams, their head, and their tail, as the whole point is to deal with streams. To represent *streams* of type *Str A*, we included the term *Cons*, which consists of a *head* and a *tail*. The head of the stream represents the value in the present; this means that if we have a stream of type *Str A*, the *head* of that stream must have the type A. The term *Tail* represents the rest of the stream; namely the values that we do not have access to, but that we will have access to in the "future". In this sense, the tail must apply the *later* modality, and therefore the tail of a stream with type *Str A* must be of type $\bigcirc$ *Str A*. The stream, head, and tail have the following typing rules:

$$\frac{\Gamma \vdash t : Str\ a}{\Gamma \vdash Head\ t : A} \qquad \frac{\Gamma \vdash t : Str\ a}{\Gamma \vdash Tail\ t :: \bigcirc Str\ a} \tag{10}$$

$$\frac{\Gamma \vdash s : A \quad \Gamma \vdash t : \bigcirc Str\ A}{\Gamma \vdash s :: t : Str\ A} \tag{11}$$

2.3.2 Recursive functions

Finally, we introduce the term *rec* to recursive functions. The term *Rec* takes a string as an argument, which much like *let* 4, serves as the name of the recursive function. *Rec* works much in the same way as *let*, in fact, but has the advantage of being able to be recursively applied to a stream. *Rec* is given the name of a string denoted as x in the typing rule of type B, a term s representing the body of the same type B, and then another term t that represents another expression (in our case the rest of the program), that results in type A. The first premise assigns the type B to s, but only in the 'boxed' context $\Gamma^{\square}$ where x^{Rec} is recursively bound with type B. This means that x can appear in s, allowing for recursive definitions. As noted, x has the recursive flag ^{Rec}, whereas the x in the second premise does not. This allows us to distinguish between, when a recursive function is called inside itself, and when it is applied outside itself. Variables denoted with the ^{Rec} flag, cannot be type-checked at the time step they were introduced but must be type-checked at least one time step into the future. The next premise has a context, where it has access to x with the assigned type B. The conclusion to the typing rule is that the term s will have the same type as x; namely B. Then, s is accessible in t, which has the type A.

$$\frac{\Gamma^{\square}, x^{Rec} : B \vdash s : B \qquad \Gamma, x : B \vdash t : A}{\Gamma \vdash \textit{let rec (x:B)} = s \textit{ in } t : A} \tag{12}$$

2.4 Productivity, Causality and space leaks

As mentioned earlier, the FRP languages have faced multiple problems, so implementing an FRP language presents special challenges. These problems are productivity, causality, and implicit space leaks. However, the theory behind Rattus guarantees that these problems are accounted for and solved accordingly. In the following sections we discuss what these problems are and how Rattus solves them, and then after the implementation, we discuss whether we solved these problems in section 9.

2.4.1 Causality

How causality is ensured is covered extensively by Bahr [1]. The essential idea is that present values should not be able to depend on future values. Let's say that we have a function that essentially returns the tail of a stream like the *tomorrow* function:

```
let tomorrow (x::xs)= xs;
```

In this case, it seems like the user can define a "present" value that depends on a value given in the future, thus making it a non-causal function. However, the tail of a stream type checks to $\bigcirc$ *Str A*. This means that the value of the tail of the stream has not arrived yet, and will arrive in the next timestep. This

means that the type of the function is:

$$\texttt{Str A} \rightarrow \bigcirc\ \texttt{Str A}$$

The *tomorrow* function returns something that points to a heap location of a value that does not exist yet, as "xs" points to a value that has simply not arrived yet. If we were able to add an *Adv* in an attempt to get a future value like this:

```
let tomorrow (x::xs)=Adv(xs);
```

This would not typecheck, given that *Adv* needs a ✓ in the context. So whereas a user might be able to write a function that returns $\bigcirc$ *Str A*, they cannot access values that have not arrived yet.

2.4.2 Productivity and guarded recursion

For functional reactive programs that deal with coinductive types (infinite recursive types or data structures), managing these recursive calls is critical in ensuring stability and correctness. Guarded recursion is a technique to manage these recursive calls and how they unfold over time, ensuring that coinductive types would not run infinitely when evaluated; this is also called productivity. Typically, guarded recursion refers to recursion executed based on a condition or guarded by a construct ensuring its safe execution. Such a "guard" construct typically ensures that each step progresses towards some base case or otherwise operates within a safe framework that ensures productive behavior; in our case, a recursive step happens at every time step.

Observe the coinductive type Fix $\alpha.A\times\alpha$, which represents an infinite stream of A's. *Fix* is a term commonly used to represent recursive functions. Each element in the stream is paired with another stream of A's; quite obviously this leads to an infinite structure. A naive implementation of operations on such a type might attempt to evaluate or manipulate the entire structure all at once, leading to non-termination, since any operation could never reach the end of the stream. When such a problem occurs, we say the program is non-productive. This is the problem that guarded recursion solves; instead of evaluating the entire structure, only the head of the stream is evaluated, whereas the next recursive call will be evaluated once some condition is met. In the case of Rattus and other FRP languages, the condition that guards the recursion is time passing.

Traditional approaches often introduce a modality similar to Rattus's $\bigcirc$ modality (Delay), which "guards" the recursive part of an infinite structure. This modality is placed before the recursive call of the coinductive type, meaning that we would transform Fix $\alpha.A \times \alpha$ into Fix $\alpha.A \times \bigcirc\alpha$. This delay modality $\bigcirc$ guards and thereby ensures that the recursive part of the coinductive type is not evaluated before time has passed one timestep. It should be noted that in this case occurrences of Fix $\alpha.A \times \alpha$ are explicitly not typable and the type system will detect this as an error. For a recursive type to be typable, it must

be a guarded recursive type, meaning that while Fix $\alpha.A \times \alpha$ is not typable, Fix $\alpha.A \times \bigcirc\alpha$ is typable.

This all means that regular recursion is disallowed and code like the following program will not typecheck:

```
let rec loop x = x:::loop x
```

Here the unguarded recursive call to the loop does not occur under a delay, and the type checker would therefore reject it. For this simple program to type check, it must have a delay. This means that for the following code to type check, it must have a delay included like this:

```
let rec loop x = x:::Delay(loop x)
```

However, this is not the only modality that Rattus needs for guarded recursion to work as intended. The following function that takes a stream, would not type check:

```
let rec replicate (x:::xs) = x:::Delay(replicate xs)
```

As stated earlier, the tail of a stream inherently has the type $\bigcirc$ *Str a*. In this situation, the tail of a stream must have the *Adv* annotation, which serves as the elimination form of $\bigcirc$. Because of this, the *replicate* function must be written as follows:

```
let rec replicate (x:::xs) = x:::Delay(replicate Adv(xs))
```

However, writing code like this is quite tiresome, and in section 6.3.1, we discuss how to implicitly place *Delay* and *Adv*, so the user can write programs without needing detailed knowledge of the underlying complexities these concepts that Rattus introduces. By making the typechecker handle the insertion of these constructs, we aim to simplify the development process and make the language more user-friendly, while still maintaining the necessary safeguards to ensure productive behavior and stability in functional reactive programming.

2.4.3 Space leaks

An inherent problem in building FRP languages is implicit space leaks. An implicit space leak is a space leak that is not intended by the programmer but comes from the implementation of the language itself. The problem is that the language retains old data, that will grow during run time, until the program eventually runs out of memory. The problem seems trivial enough, but higher-order languages can hold references to old data within their closures, which means the data must remain within memory for as long as the closure might be evaluated.

An example of a potentially leaky function is the *constInt*:

```
constInt n = n :::  constInt n;  Type:  A → Str A
```

It's clear that this function is meant to take an integer and produce a stream of integers, yet its polymorphic nature lets the developer pass anything. This includes other streams. These streams would grow with each timestep. If we applied a stream to constInt, we end up with the type *Str (Str A)*. Here is it obvious that at each time step, it would have to store all previously observed input values from time step 0 to the last timestep, thus making its memory usage grow linearly with time. To illustrate this with a concrete example, assume that *leakyConst* is fed the stream of numbers $0, 1, 2, \ldots$ as input. Then, the resulting stream of type `Str (Str Int)` contains, at each time step n, the same stream $0, 1, 2, \ldots$. This makes it quite clear, that it would need to keep previous old data in memory, making it leaky.

Rattus solves this by introducing a delay to this function and by the variable rule 6.

```
constInt n = n :::Delay( constInt n);
```

In this situation, the function expects n to be a stable type, meaning you are not able to pass a stream into *constInt* because anything but a stable type cannot be accessed inside the *Delay* unless it uses *Adv*, which this does not.

Bahr [1] describes another function that has implicit space leaks. Namely, the *leakyMap*.

```
let leakyMap f=
    let rec run (x:::xs) = f x :::Delay( run Addv(xs));
```

This function is not typable. This is because functions can keep values in their closures, and in this case is therefore able to retain old data, that would not be garbage collected in the next time step.

Other than these rules, the interpreter theory also describes an aggressive garbage collection strategy, where old values are discarded. We elaborate more on this in section 7, where we go into detail about it.

3 The language

In this section, we are going to showcase some smaller functions and programs that you can write in this language. The syntax is very reminiscent of the syntax for F#. We show these and discuss them briefly here in the beginning, before we show how the parser, typechecker, or interpreter works because we believe it makes the rest of the paper easier to follow.

3.1 Variables and function declarations

You declare variables and functions in much the same way you do in F#. You can declare simple variables of type *Int* or type *Bool* respectively in the follow-

ing way:

```
let x = 1;  Int
```

```
let b = true;  Bool
```

You can declare functions. Below we see the id function, that takes an argument and returns the same argument unaltered. We also see it's associated type:

```
let id x = x;    'b → 'b
```

You can think of *'b* as a polymorphic type.

The next function takes an integer, increments it by 1, and then returns it:

```
let add x = x + 1;    Int → Int
```

You can also declare an anonymous function:

```
(fun x -> x+1)   Int → Int
```

Notice that none of the functions or variables have type annotation to tell what type its argument is or which type it even returns, yet we still know the type. This is because the type system of the language is quite powerful, and the type checker can infer any typable program. The developer is in fact not even able to add type annotation. We go into explicit detail about how this works in section 6.

It is also able to reject the programs it should reject. If we apply the *add* function from before with a bool like this, we receive an error:

```
add true;    type error:  int and bool
```

3.2 Simple streams and stream manipulation

Streams and stream manipulation are a big part of the language and its main purpose for being built. You are able to create a function that takes an argument and then creates an infinite stream of the argument provided:

```
let rec constInt n = n :::  constInt n;   S'a → Str S'a
```

Notice here, that the type has an associated "S" with it. We use this "S" in the language to denote the stable types Bahr introduced [1]. The typechecker is able to infer that n must be a stable type, given that we also access it on the right side of ":::", where the typechecker has placed an implicit *Delay*; more on

how this works in section 6.3.1. If we apply the *constInt* to an integer *1*, we get the following type and output:

```
constInt 1;

Type:  Str int

Result:  [Int 1; Int 1; Int 1;...]
```

Another function that results in a stream is *from*. This function takes an integer an creates a stream of integers that increment by 1 for each recursive step:

```
let rec from n = n:::from (n + 1);  Int → Str Int
```

The result and type of applying this function to an integer 1 is quite obvious:

```
from 1;

Type:  Str int

Result:  [Int 1; Int 2; Int 3; Int 4;...]
```

In addition to these straightforward stream functions, you can also create more complex ones. For example, the *stutter* function generates a stream where each integer appears twice before incrementing, as follows

```
let rec stutter n = n :::  n :::  stutter (n + 1);  Int → Str Int
```

The result of running this function with a 1 is:

```
stutter 1;

Type:  Str int

Result:  [Int 1; Int 1; Int 2; Int 2; Int 3; Int 3;...]
```

3.3 Sum function and user interaction

The *sum* function is designed to produce a stream where each element is the cumulative sum of the elements provided by a stream. The function is defined recursively as follows:

```
let rec sum acc (x:::xs) = x + acc :::  sum (x + acc) xs;
```

In this definition:
- `acc` is the accumulator that keeps track of the cumulative sum.
- `x` is the current element of the stream.
- `xs` is the rest of the stream.

It typechecks to the following type: `Int → (Str Int → Str Int)`

Each element in the resulting stream is the sum of the current element and the accumulator, followed by the recursive call to `sum` with the updated accumulator.

If we initialize the function in the following way, we produce a function with type `(Str Int → Str Int)`:

```
sum 0;
```

Any function that is type inferred to `Str A → Str B`, requires user input to function correctly. The language now expects user input via the console. Each time the user inputs a value the program reacts to the input.

The *sum* function "reacts" to user input by printing out the head of the stream at every timestep. Below is an example of user input provided one by one, and the corresponding output produced by the *sum* function:

User input:	1	2	3	4	5	6	7	8	9
sum response:	*Int 1*	*Int 3*	*Int 6*	*Int 10*	*Int 15*	*Int 21*	*Int 28*	*Int 36*	*Int 45*

Table 1: User input and *sum* function response

3.4 Box, unbox, functions and time steps

A crucial aspect of Rattus is how functions interact with time steps. Functions are not classified as stable types and therefore are not accessible in a context with a ✓. To manage this, we use the concept of boxing and unboxing functions.

Boxing a function encapsulates the function in a way where it becomes a stable type, allowing it to be safely used in a delayed context. This is achieved using the *Box* constructor. Conversely, unboxing is the process of retrieving the original function from its boxed state using the *Unbox* constructor.

Consider the following example where we define a boxed function, *g*, which negates its input. We also define a function, *from*, that generates an infinite stream of integers starting from a given value. Finally, we define a *map* function that applies a boxed function to each element of a stream:

```
let g = Box(fun x -> x * -1);
let rec from n = n:::from (n + 1);
let rec map f (x:::xs) = Unbox(f) x::: map f xs;
map g (from 1);
```

The inside of *map* can now typecheck correctly, since f is a stable type. We then apply the map function to our boxed function g and the stream of integers from 1. The result of this is exactly what we expect:

```
Type:  Str int
```

```
Result:  [Int -1; Int -2; Int -3;...]
```

If we do not box the function g, but still try to pass it in the map function like this:

```
let g = (fun x -> x * -1);
let rec from n = n:::from (n + 1);
let rec map f (x:::xs) = Unbox(f) x::: map f xs;
map g (from 1);
```

In this case, the function fails, because it cannot unbox a function that is not boxed, and the typechecker gives this error:

```
type error boxed and function
```

However, when we wholeheartedly ignore the Unbox part of map and write the map function like this, it luckily also fails:

```
let g = (fun x -> x * -1);
let rec map f (x:::xs) = f x::: map f xs;
map g (from 1);
```

The error being:

```
Non-stable type found in the stable set.
Found a non-stable type in a place where we expect a stable type.
```

This is, of course, because we are accessing the function f at the right side of *:::*, whose context is treated as having a ✓, because of the implicit *Delay*. Functions are not stable types and can therefore not be accessed in the future. This shows that the type inference and the type rules hold for the implementation

3.5 Showcase of two small games

In this subsection, we show off two small programs, that function as games. This is to show what is possible within the programming language. We have kept the games as simple as possible for the sake of clarity.

3.5.1 Rock, Paper, Scissors

In this first game, we have rock, paper, scissors.

```
let rec RPS n = 1:::0:::-1:::RPS n;

let gameLogic = Box (fun r x -> if r==x then 0 else if
r==1 & x==-1 then -1 else if r==0 & x==1 then -1 else if r==-1 & x==0 then -1 else 1);

let rec game gl (r:::rps) (u:::user) = Unbox(gl) r u ::: game gl rps user;
```

```
let playGame= game gameLogic (RPS 1);"
```

At first, we define a stream called *RPS*. This is the stream of either rock, paper, or scissor represented as 1,0,-1, that the opponent chooses at each timestep. The second function is the *gameLogic*, which carries all the game logic. The third function defines the actual game, which can take a function (gameLogic), an opponent stream of answers (RPS), and the user stream of answers.

The declaration *playGame* just declares the game, which will then run the program and expect user input, when given to the interpreter, since it's a *Str Int* $\rightarrow$ *Str Int*. The user can then play the game by either inputting a -1, 0, or 1 into the console. You then receive an output from the program of either -1, 0, or 1. If you receive a 1, it means you won, a zero means it's a draw and a -1 means you lost.

3.5.2 Hot potato

The second game is hot potato. Essentially in the game, there's a "clock" ticking down. Once it reaches 0, whoever holds it loses. In this game you can either give a 1 or a 0. The 1 means you hold the potato for another round. The 0 means you pass it along to the other player. You can't pass it twice in a row. You win if you don't hold the potato at the end of the countdown.

```
let rec hotPotato n = (if n==0 then 1 else 0)
            ::: if n==0
                then hotPotato 5
                else hotPotato (n - 1);

let gameLogic = Box (fun prev p u -> if prev==0 & u==0 then -1
else if p==1 & u==1 then -1 else if p==1 & u==0 then 1 else 0);

let rec game gl prev (p::: hp) (u:::ui) = Unbox(gl) prev p u  ::: game gl u hp ui;

let runGame= game gameLogic 1 (hotPotato 5)
```

The first function defines the stream for the "clock". Once it reaches 0 it gives a 1, otherwise it gives a zero. The second function holds the game logic. The third function is the actual game that takes a boxed function, an integer, and two streams; one of them being the user input. The last function initializes the game with the game logic, the previous move (set to 1, so you can pass the potato), and the stream from *hotpotato*. The type for *runGame* is *Str Int* $\rightarrow$ *Str Int*, which means it runs and takes user input. The program can react and give you either a 0, meaning nothing happened, 1, meaning the clock reached 0 after you handed it away to your opponent or a -1, which means the clock reached zero while you held it or you tried to pass it twice in a row.

4 Project overview and implementation

The implementation of the programming language is divided into three main components, which are typical for interpreted languages: the parser, the typechecker, and the interpreter. In the following sections, we will explore the theory behind each of these components, their implementation, and key observations.

Before we explore the main components, we will introduce the abstract syntax used to represent programs in the following section. We will then in the next sections examine the parser in very general terms. Next, we will discuss the typechecker, focusing on its theory, how it integrates with modal types, and its implementation. Finally, we will cover the interpreter, its underlying theory, its functionality with Rattus, and its implementation.

4.1 The abstract syntax

What the parser, typechecker, and interpreter all have in common is access to the abstract syntax tree. The parser takes the syntax that the human has written and produces the abstract syntax tree, the type-checker then makes sure that the abstract syntax tree is typeable and follows all the typing rules; it also slightly alters it. After this, the interpreter is given the abstract syntax tree and executes the program that the abstract syntax tree represents. All these parts are written in F#.

We represented the terms of the STLC, Rattus, and our recursive and Cons terms by the type *expr* in F#:

We represented the terms from the STLC as follows:

```
type expr =
...
| Var of string
| App of expr * expr
| Lambda of string * expr
...
```

We represented the Rattus terms like the following, which was quite simple, given that the Rattus terms are only ever applied to one *term*:

```
type expr =
...
| Delay of expr
| Adv of expr
| Box of expr
| Unbox of expr
...
```

We represented the terms for streams, their head and tail as:

```
type expr =
...
| Head of expr
| Tail of expr
| Cons of (expr * expr)
...
```

We represented the *Let* and *Rec*, the variable declarations and recursive definitions as the following:

```
type expr =
...
| Let of string * expr * expr
| Rec of string * expr * expr
...
```

Notice here that the Let and Rec have two *expr*'s. One represents its body, and the last one represents the rest of the program. In this way we effectively chain Lets and Recs together at the last *expr* position to create entire programs.

We represented the integers, booleans, and tuples, as well as the primary operator (+,-, /, *), in the following way:

```
type expr =
...
| CstI of int
| CstB of bool
| Tuple of expr * expr
| Prim of string * expr * expr
...
```

We also represented heap locations as an abstract syntax. The parser and type checker never use it, but the interpreter uses it when dealing with delayed computations:

```
type expr =
...
| Loc of string
...
```

Together the entire expr type looks like this:

```
type expr = | CstI of int | CstB of bool | Var of string | Let
```

```
of string * expr * expr | Prim of string * expr * expr | If of expr
* expr * expr | Letfun of string * string * expr * expr | App of expr
* expr | Lambda of string * expr | Delay of expr | Adv of expr | Box
of expr | Unbox of expr | Head of expr | Tail of expr | Cons of (expr
* expr) | Rec of string * expr * expr | Tuple of expr * expr | Loc
of string
```

5 Parser

Parsers are an important part of any programming language. The role of a parser is to take the human-written syntax and translate it into an abstract syntax tree, which can then be executed by some other process like a type-checker or an interpreter. In the process of parsing syntax, it has to undergo tokenization, where the syntax is translated into a list of tokens so that the parser can construct the abstract syntax. The tokenization part of the program is called a lexer. This is the common and traditional approach for how the tokenization process integrates with the rest of the parsing process. However, we opted for another approach; we implemented an incremental parser that switches between tokenizating the syntax and consuming the tokens to produce the abstract syntax tree. Essentially, once the parser has enough tokens to deduce the only possible interpretation of these tokens, then it proceeds to construct this bit of the abstract syntax. It then goes back to tokenizing again, until a new interpretation is the only possible one, it then parses this into abstract syntax and so on. It keeps going like this until there is no more to tokenize.

The parser uses a simple and powerful parser combinator library: FParsec [10]. FParsec allows any string or set of strings to be parsed into the appropriate token. The library allows the developer to define precedence for each parser combinator; this is incredibly useful because it allows the developer to create context-sensitive tokens. This flexibility enables the same syntax to have a diverse set of interpretations based on the surrounding syntactic context. Consider the examples with "X" below. In the first instance, "X" is evaluated as a variable. In the second instance, "X" is declared as a variable assigned to a constant. In the third instance, the function `myFunction` is declared with "X" as its parameter. In the final instance, "X" is first declared as a function with the parameter a, and then this function is applied to the integer 5. In this case, the "a" that the user has defined after "X" is interpreted as a parameter and not a new variable. This is because the parser is able to deduce from the surrounding context that "a" is in this instance only able to be a parameter. In this sense, the "parameter parser" takes precedence over the "variable/function name" parsers. This ensures that "a" in "X" is not mistaken for a variable or function name, but is in fact a parameter. The key point is that the surrounding context determines how "X", "a" or any name is evaluated.

```
X;
let X=1;,
let myFunction X = ....
let X a = a + 1; X 5;
```

We have simplified how the parser works for the sake of clarity, but the overall structure and idea remain the same. The parser has two main parts: the *declarative* part and the *expression* part. The *declarative* part processes tokens one at a time until it determines what the syntax is attempting to declare. As mentioned earlier, it processes the syntax into a token one at a time, and it then narrows down the set of possible interpretations; if there is only one interpretation left, it then proceeds to consume all tokens and create the abstract syntax, otherwise, it stores the token in short-term memory. This allows it to distinguish between variables, functions, and recursive functions. For instance, when encountering a "let" keyword, the parser can initially consider multiple interpretations, such as a variable, function, or recursive function declaration. However, with each subsequent token, it can eliminate certain interpretations.

The examples below show how the let keyword is first interpreted and what possible interpretations are narrowed down. The parser then tokenizes the next input, further narrowing down possible interpretations, until it only has one interpretation. Given just the first "let" keyword, it has several options; it could be either a variable, a function, or a recursive function declaration. In the code, the options are actually stored as a list of integers representing each interpretation, but that is off-topic. Given the next word "X", it can narrow it down to either being a variable declaration or a function declaration. Finally, once it parses the "a", it deduces that the only interpretation is a function. It then proceeds to construct the abstract syntax.

```
let.../Variable, function, recursive function
let X.../Variable, function
let X a.../function
```

The *expression* part of the parser operates similarly to the *declarative* part but is slightly more advanced. It can temporarily hold previous abstract syntax in short-term memory. For instance, given the input `5+3;`, the parser first recognizes the number 5, eliminates other interpretations, and stores it in memory. It then identifies the "+" sign, narrows down the interpretation to addition, and at this point, the parser hasn't processed the number 3 yet. The parser then constructs the abstract syntax `Prim("+", CstI 5, expression)`. Here, "expression" represents a new run of the parser with reset memory and a reset of possible interpretations. It processes the remaining `3;`, resulting in the final abstract syntax: `Prim("+", CstI 5, CstI 3)`.
Consider another input like `f 5 6...;`. The parser first processes "f" and then "5", recognizing that with "f" in its short-term memory, this should definitely

be a function call. At this stage, the parser hasn't processed "6" or any other variables. It constructs the abstract syntax `App(Var "f", CstI 5)` and stores this in its short-term memory for the next run of the expression parser. When the parser runs again, it encounters "6" and, with the previous abstract syntax in memory, determines that "6" is part of the function call. This results in the final abstract syntax: `App(App(Var "f", CstI 5), CstI 6)`. If the next token had been an operator instead of "6," the parser would have treated the previous abstract syntax as the first argument of the "Prim" operator from earlier. Were we to add another number or variable name like "x " at the end of `f 5 6 x;`, it also treats this as part of the function call and retrieves `App(App(Var "f", CstI 5), CstI 6)` from the short term memory to construct `App(App(App(Var "f", CstI 5), CstI 6), Var "x")`.

5.1 From syntax to abstract syntax

The parser is designed to work with a similar syntax as F#. Primarily this was chosen, as we believe the syntax of F# to be simplistic and easy to pick up. It also makes it easier for people who know F# to switch to this language.

5.1.1 Expressions

For a simple variable assignment declaration, much like in F#, the syntax is written as

```
let X=1;
```

and gets parsed as abstract syntax into:

```
Let ("X", CstI 1, Var "X")
```

It is important to note, that the last part of the *Let* abstract syntax is *Var "X"* because there is nothing else left in the program. In the case, where there are multiple declarations, they are chained together like the following:

```
let X=1; let Y=2; let Z=3;
```

This will result in the following abstract syntax:

```
Let
  ("X", CstI 1,
    Let ("Y", CstI 2,
      Let ("Z", CstI 3, Var "Z")))
```

Notice here, how the *Let*'s are chained together at the last argument. The last argument of a *Let* will always lead to the rest of the program, and in the

case where there is nothing else to parse, it will just refer to itself via a variable: *Var "Z"*.

The parser can encounter and handle a similar syntax, but where one of them is an expression without a *let*:

```
let X=1; 2; let Z=3;
```

The abstract syntax that is generated from this is quite similar to the previous abstract syntax. When the parser encounters an expression that does not have a *let* associated with it, it treats it as a *let* and gives it a generic name "it" similarly to F#:

```
Let
  ("X", CstI 1,
    Let ("it", CstI 2,
      Let ("Z", CstI 3, Var "Z")))
```

5.2 Function and operator declarations

Functions are defined with the same syntax as F#. The id function is a function that takes a parameter x and just returns that same x without manipulation:

```
let id x=x;
```

This function results in the following abstract syntax:

```
Let ("id", Lambda ("x", Var "x"), Var "id")
```

Notice here, that the only difference between a regular variable declaration and a function declaration is the addition of the *Lambda* with the parameter. You are also able to define anonymous functions. An anonymous function that takes a parameter x and just returns an integer 1 looks like this:

```
(fun x -> 1);
```

This will return the following abstract syntax:

```
Lambda ("a", CstI 1)
```

Mathematical operations and calculations are defined by the obvious syntax for the following example expressions:

```
1+2;
1-2;
1*2;
1/2;
```

This translates into the following context respectively:

```
Prim("+", CstI 1, CstI 2)
Prim("-", CstI 1, CstI 2)
Prim("*", CstI 1, CstI 2)
Prim("/", CstI 1, CstI 2)
```

The parser can also parse multiple calculations like this:

```
1+2+3+4+5;
```

This translates into the following abstract syntax

```
Prim("+", CstI 1,
  Prim ("+", CstI 2,
    Prim ("+", CstI 3,
      Prim ("+", CstI 4, CstI 5))))
```

5.3 Box and unbox

You can use the box and unbox terms in the language too. To use the box and unbox you can write the following:

```
Box(x)
Unbox(x)
```

This results in the following abstract syntax:

```
Box(Var "x")
Unbox(Var "x")
```

As noted in 2.3, the box term is primarily introduced to be used on functions. This must be represented by an anonymous function inside the Box term. In the following syntax, we declare a boxed function by boxing an anonymous function; the function takes a parameter and just returns a 1:

```
Box(fun x -> 1)
```

This results in the following abstract syntax:

```
Box (Lambda ("x", CstI 1))
```

5.4 Examples of functions

Here we are going to showcase some examples of functions and how their abstract syntax turns out. We'll not go into detail about how the parser translates this, as this is not the main focus of the paper.
The first example is a recursive function called `from`. It is a pretty simple function that takes an integer and creates a stream of integers which are increased by one at every timestep.

```
let rec from n = n :::  from (n + 1);
```

This gets translated in the abstract syntax:

```
Rec
("from",
 Lambda
   ("n", Cons (Var "n", App (Var "from", Prim ("+", Var "n", CstI 1)))),
 Var "from")
```

The second example is a function called *sum*. This function is pretty simple, it takes an integer (acc) and a stream of integers. The head of the stream is produced by the next value in the stream and the integer it was provided. It then accumulates the value over time:

```
let rec sum acc (x:::xs) = x + acc :::  sum (x + acc) xs;
```

This gets translated into the following abstract syntax:

```
Rec
    ("sum",
     Lambda
       ("acc",
        Lambda
          ("manualStreamYwCVprzzpbZSMce1",
           Let
             ("x", Head (Var "manualStreamYwCVprzzpbZSMce1"),
              Let
                ("xs", Tail (Var "manualStreamYwCVprzzpbZSMce1"),
                 Cons
                   (Prim ("+", Var "x", Var "acc"),
                    App
                      (App (Var "sum", Prim ("+", Var "x", Var "acc")),
                       Var "xs")))))), Var "sum")
```

Notice here, that the stream it takes as a parameter is given a randomized name that increments. This is just to make sure the user can't provide a name that is accidentally the same, it then creates two values inside the function "x" and "xs" that use the terms Head and Tail on the stream respectively. The "x" and "xs" variables are then accessible to the rest of the function when dealing with the type inference and interpreter.

On the last example, we showcase a function representing the classic *map* function. This function takes a boxed function and a stream. It then unboxes the function and applies this function to each of the values of the stream:

```
let rec map f (x:::xs) = Unbox(f) x :::  map f xs;
```

This then creates the following abstract syntax tree:

```
Rec
("map",
 Lambda
   ("f",
    Lambda
      ("manualStreamSrJwVDNZlbPVUau2",
       Let
         ("x", Head (Var "manualStreamSrJwVDNZlbPVUau2"),
          Let
            ("xs", Tail (Var "manualStreamSrJwVDNZlbPVUau2"),
             Cons
               (App (Unbox (Var "f"), Var "x"),
                App (App (Var "map", Var "f"),
                     Var "xs")))))), Var "map")
```

6 Type Inference

The type inference algorithm used for F# and other ML programming languages is the Hindley-Milner algorithm [7]. To start, every term has a type. The basic types are int, bool, string, etc. It also includes other types like lists and pairs that are built using type constructors. In addition, it uses type variables that can be instantiated to any type. The type inference allows a program to infer the type of a term based on its use. For instance, we can easily infer the type of a function that adds a variable to a constant integer

```
let z x = x + 5
int → int
```

As the parameter `x` is being added by a primary operator (+), we can safely infer that `x` must also be an int which implies that `z` takes an integer and returns an integer, *int* → *int*. In this environment, the variables `x` and `z` can only have one fixed type, also known as monomorphic type.

However, it's not always that simple to infer a type. For example, we might also encounter variables that can hold several types in their environment, which we will refer to as polymorphic types. Let's consider a function that takes another function as an argument which is applied to a constant integer,

```
let f g = g 5
(int → α) → α
```

Here, α is what is known as a type variable. This means that α can represent different types depending on the context in which the function `f` is used. For instance, if we pass a function `g` that takes an integer and returns a boolean (e.g., `isEven`), then we can infer that α is a boolean. The type of `f` in this case becomes: `(int → bool) → bool`. On the other hand, if we use a function g that takes an integer and returns another integer (e.g., `factorial`), then α would be an integer and the type of `f` would then be `(int → int) → int`.

In other words, these types (α, β, etc.) can be substituted by a monomorphic type or another type variable, depending on the environment in which they are used. This process is known as *instatiation*. On the contrary, using a type variable instead of a specific type is known as *generalization*, which allows us to have and use these polymorphic types.

In the environment, a polymorphic type is represented by a list of type variables and a type where these variables occur. This is also referred to as a type scheme. For the previous example, the type scheme for g would look like:

```
[ α, int → α ]
```

This is also read as, "for all ways to instantiate the type variable α, the type int $\rightarrow \alpha$ is possible for g", or

```
∀ α.  int → α
```

A monomorphic type t represented by a type scheme would have the form of `([], t)`, or $\forall().t$, where the list of type variables is empty [9].

Let's analyze a more elaborate example of a function using polymorphic types, *fone*, where an anonymous function with an argument h, with another anonymous function in its body that takes an argument x and finally applies h to x,

```
let fone = fun h -> fun x -> h x.
```

which will result in a type and type scheme as follows

```
(α → β) → α → β
∀ αβ.(α → β) → α → β
```

where α and β are type variables, which means that *fone* takes a function that could take an argument of any type and return something of any type, which is not necessarily the same as the argument. This is the most general type for this expression. If we apply a simple alteration to *fone*, lets call it *ftwo*, where the body of the last anonymous function applies h to an application of h to x,

```
let ftwo = fun h -> fun x -> h (h x).
```

this will result in a type and type scheme as follows

$(\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$
$\forall\, \alpha.(\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$

This is because, as we can observe, the application of `h x` must return the same type as the argument that `h` takes. By analyzing these two functions, we can say that the type scheme, which will be referred to from now on as γ, of the first function, is more general than the second one, or $\gamma_1 < \gamma_2$.

To generalize a term to a type scheme, first, it must be possible to assert that the term has the type β in the context, and then we must make sure that the type variable α is not free in the type environment. This translates to,

$$\frac{\Gamma \vdash t : \beta \quad \alpha \notin FV(\Gamma)}{\Gamma \vdash t : \forall \alpha.\beta} \text{ Generalization} \tag{13}$$

On the contrary, to instantiate a type scheme, we substitute types for type variables, also expressed as $[\tau/\alpha]\gamma_1$ where we replace each free occurrence of α in a type scheme γ_1 by τ, resulting in another type scheme γ_2. This can only be done if γ_1 is more general than γ_2. Note that substitution only affects free variables, so instantiation acts on free variables, and generalization acts on bound variables [8].

$$\frac{\Gamma \vdash t : \sigma_1 \quad \sigma_1 > \sigma_2}{\Gamma \vdash t : \sigma_2} \text{ Instantiation} \tag{14}$$

The type inference proposed by Damas and Milner [8] stated rules for inferring types for the STLC. The rules for *variable, lambda* and *application,* remains the same as 1, 2, and 3, respectively. However, for *let*, we must make some adjustments. When we introduce the parameter of the function into the environment, we must first generalize its type, then we can proceed to infer the type of the body of the function. This translates to,

$$\frac{\Gamma \vdash t_1 : \alpha \quad \Gamma \cup \{generalize(\Gamma, \alpha)\} \vdash t_2 : \beta}{\Gamma \vdash Let\ t_1\ in\ t_2 : \beta} \text{ Let} \tag{15}$$

The type inference algorithm applies the unification algorithm of Robinson [8]. The purpose of the unification is that, given two terms, it tries to find a substitution that, when applied to terms, makes them equal. In other words,

the unification of two terms *U(x_1, x_2)* tries to find the most general substitution V such that $x_1V = x_2V$. The algorithm attempts to find the most general substitution by following the cases described in algorithm [7],

Algorithm 1 Unification algorithm

$U(\tau_1, \tau_2)$
if $\tau_1 = x$ and $\tau_2 = y$, and $x = y$ **then**
    [ ]
else if $\tau_1 = \alpha$ and $\tau_2 = \beta$ **then**
    $[\alpha/\beta]$
else if $\tau_1 = \alpha$ and $\tau_2 = t_1$ and α does not occur in t_1 **then**
    [τ_2 / x]
else if $\tau_1 = \alpha \to \beta$ and $\tau_2 = \delta \to \varepsilon$ **then**
    U(α, δ), U(α, ε)
end if

If none of the cases are matched, then the algorithm fails and the type inference is not possible. Note that given a type variable α and a type t_1, we must check that α does not occur in t_1, this is to avoid circular or infinite types [9]. For example, if the type t_1 is α, and the type t_2 has $\alpha \to \alpha$, then this will fail as $\alpha \neq \alpha \to \alpha$.

The Hindley-Milner algorithm uses this unification algorithm to get the most general type for an expression. The algorithm, referred to as algorithm *W*, deals with four cases, one for each term: *Variable*, *Application*, *Lambda*, and *Let* [7]. The algorithm *W*, given a context Γ and a term t, will return a substitution S and a type τ [8]. This substitution, when applied to τ, will instantiate the type variables in order to get the most general type for the term t.

When any of the conditions mentioned in 2 are not met, then W fails [8]. The case where the term t is a variable simply states that given a type for t, the result is the instantiation of its type scheme with fresh (new) type variables. The case for *lambda* describes the creation of a fresh type variable β for x that will be added to the environment that we will use to infer its body. This will give a substitution that will be applied to β, and then used on the resulting type. The case for *application* infers the types for both expressions, τ_1 and τ_2, respectively, and attempts the unification for τ_1 with a new type function consisting τ_2 and a fresh variable β. This unification will create a substitution V that will be applied to the fresh variable, which is returned for this case. The last one corresponds to *let*; we infer the type for the first expression, then we proceed to generalize its type, and then add it to the environment. This new environment will be used to infer the type of the second expression, which will give us the type that the case will return.

Algorithm 2 Algorithm W

$W(\Gamma, t) = (S, \tau)$
if t is $Variable$ and its type is $\forall\alpha_1, \alpha_n\tau'$ in Γ **then**
  $S = [\,]$ and $\tau\ [\beta_i/\alpha_i]\tau'$
  where the β_is are new.
else if t is *Lambda* $\lambda x.e$ **then**
  let β be a new type variable and
  $W(\Gamma \bigcup \{x : \beta\}, e) = (S_1, \tau_1)$
  $S = S_1$ and $\tau = S_1\beta \to \tau_1$
else if t is *Application* t_1t_2 **then**
  $W(t_1, \Gamma) = (S_1, \tau_1)$ and $W(t_2, \Gamma) = (S_2, \tau_2)$
  $U(S_2\tau_1, \tau_2 \to \beta) = V$
  where β is new, then $S = VS_1S_2$ and $\tau = V\beta$
else if t is *Let* in the form *let* $x = t_1$ *in* t_2 **then**
  $W(\Gamma, t_1) = (S_1, \tau_1)$
  $W(S_1\Gamma \bigcup x : S_1\overline{\Gamma(\tau_1)}, t_1) = (S_2, \tau_2)$
  then $S = S_2S_1$ and $\tau = \tau_2$
end if

6.1 Type inference for modal types

6.1.1 Later Modal ◯

Severi [11] proposed a method to infer a temporal modal type • used for delay operations. She introduces a delay type operator • into the terms of the STLC where there is a rule for introducing and eliminating the modality. Then she states a set of constrain typing rules for this version of the STLC that are used in the unification algorithm. This unify returns a set of solutions, instead of one. However, for our type inference algorithm, we opted for a simpler solution. In essence, we kept in mind that we need to infer the types of terms based on their use. Based on this idea, we attempt to infer the modal type ◯ of Rattus based on the type rule for *delay* and *advance*.

For delay, as stated on the typing rule 5, we add a step in time to our context and then infer the type for the delayed term, τ_1. As the delay rule specifies that the type must be of a "later" type, then we just proceed to add the ◯ modality to τ.

Similarly, for advance, based on the typing rule 7, we assume time has passed and we check the type of the advanced term on Γ. If the type of such term is $\bigcirc\tau$, then we return τ. However, there is the case where the type of this term is still a type variable α. As we know from 7, the advanced term must be $\bigcirc\alpha$, so we proceed to create a fresh type variable β with the modality ◯, such that we get $\bigcirc\beta$, and then perform a unification between α and $\bigcirc\beta$. If the unify method succeeds, then we can safely infer that advance is of type β.

6.1.2 Always Modal $\Box$

For stable types, we took a more elaborate approach. As described in rule 6, we can only read types that are in the present, or if we need to take a look back in time, then it must be a stable type. But the problem comes when the type we need is a type variable that is found in another time step since there is no way to know if it is stable or not. At this point, is not possible to know if a type variable could be instantiated by a non-stable type in the future, provoking this to break the rule that states that we can only access stable types when we take steps in time. To solve it, first, we need to keep track of all the types that must be stable in order to maintain the rule established by Rattus in rule 6. Type variables whose type remains unknown, but appear in a context, where only stable types are allowed are added to a set C. The type variables that are added in this set C, must have an indicative, or flag, such that it can only be unified with other stable types. This creates a version of rule 6 to reason about polymorphic types.

$$\frac{\Gamma' tick - free \quad or \quad A \text{ is stable}}{\Gamma, x^{\text{isStable}} : A, \Gamma' \vdash x^{\text{isStable}} : A} \tag{16}$$

A similar problem comes when we need to infer the type of a boxed term t, as we need to create a new context $\Gamma^{\Box}$ with only stable types, according to 8. This environment should only contain all the stable types of the "main" environment at the moment, without any ticks. When we infer the type of t in $\Gamma^{\Box}$, we need another type of indicative, that will only be active on $\Gamma^{\Box}$, such that, when we look for the type of a variable during our type inference process, after we find it, we know that it must be stable, thus should be added to our set C.

Following these rules, we can now safely infer *box* and *unbox*. For *unbox*, we follow a similar approach as in *advance*. We infer the type for the term that we aim to *unbox* and proceed to pattern match. If the type is $\Box\tau$, then we just return the type τ. But, if the type is a type variable α, then we create a new type variable and add the modal always to it, $\Box\beta$, and then proceed to unify α and $\Box\beta$. If the unification process succeeds, then we can safely infer that the unboxed term is of type β.

Similarly to delay, for inferring a *boxed* term, after collecting our stable types in $\Gamma^{\Box}$, as described above, we proceed to infer the type τ for term t and we will just add the modality $\Box$ to it, $\Box\tau$. In addition to this, we will add this $\Box\tau$ to our set C.

6.2 Implementation of type inferer

Our type inference function `inferType` was initially based on the `typ` function created by Peter Sestoft [9] that he designed for a functional language, following

the type inference for ML programs. We adapted some of the functionality and extended the existing constructs with Rattus terms, but the initial basis for the implementation and helper functions was based on Peter Sestoft's work.

The `inferType` function takes an expression built in an abstract syntax tree. Then it calls our extended version of `typ` with an initial empty variable-type environment Γ, a level of binding 0, and an empty set C for collecting stable types. We will explain more in detail about these elements. For the type inference algorithm, we also added cases for the *unify* algorithm to deal with modal types.

6.2.1 Environment, types, and expressions

In terms of code, our variable-type environment Γ is represented like this:

```
type 'v env = ((string * 'v * bool * bool) list) list
```

Notice that the implementation employs a list of lists. This was done to represent the number of ✓'s (ticks) in the context. The innermost lists contain the actual variables in the context, while the separation between these lists signifies a tick. We are thus able to represent an empty context with no-ticks as the following:

$$\emptyset \rightarrow \texttt{[[]]} \tag{17}$$

In this sense, an *env* that only has a single list within a list is said to be tick-free. However, if we had a context with some variables with their corresponding type, and a tick, we represent it as the following:

$$\texttt{x:A, y:B,}\checkmark\texttt{,z:C} \rightarrow \texttt{[[z:C];[x:A,y:B]]} \tag{18}$$

Notice how the inner list in the head of the context, represents the variables that happened later in time. In other words, a list to the left is more recent than the one to its right. This example was, of course, just a high-level view of how we represent ticks in our environment. Our variable in the environment actually takes four arguments, the first value simply represents the name of the variable, the second one corresponds to its type scheme, which consists of `typevar list * typ`, the third argument is a boolean that marks the variable as either recursive or not recursive, only used for inferring recursive functions, and the fourth one is to represent stable types inside a $\Gamma^{\Box}$, as explained in 6.1.2.

For example, if we want to represent a non-recursive, variable `x` of type *int* in an environment where no time has passed, it would look like this,

```
[[("x",TypeScheme ([], TypI), false, false)]]
```

If we introduced a ✓ in this context, and then introduced another non-recursive integer named `y`, the context would look like this:

```
[[("y",TypeScheme ([], TypI), false, false)];
```

```
[("x",TypeScheme ([], TypI), false, false)]]
```

The types inside the type scheme could be either monomorphic or a type variable,

```
type typ =
    | TypI          (* integers *)
    | TypB          (* booleans *)
    | TypF of typ * typ   (* (argumenttype, resulttype) *)
    | Circle of typ
    | Boxed of typ
    | Str of typ
    | Tup of typ * typ
    | TypV of typevar    (* type variable *)
```

Besides the types that we have added from STLC and Rattus, we have the *Tup* which represents a tuple of types, and the *TypV* which represents polymorphic types, or type variables, which take a *typevar*:

```
type tyvarkind =
    | NoLink of string
    | LinkTo of typ
and tyvar = (tyvarkind * int * bool) ref
```

The *typevar* takes a type *tyvarkind* indicating if the variable is connected to another type or not, with `LinkTo` or `NoLink`, respectively. It also takes an integer which represents the level of binding at which the type was introduced. Lastly, it takes a boolean value used to indicate whether the type variable has been set as stable, referred to from now on as *isStable* flag. Note that *typevar* is a *ref*, which allows us to update the values efficiently on all its occurrences. For instance, if the type variable is in the environment, and in our set C, when we change any of its values, it will be updated on both places at once.

Lastly, we built a type *expr* to denote the list of terms that we have been building upon across the paper. To summarize, it is essentially a combination of the expressions given by the STLC, functions, constants, primary operators, stream, and Rattus terms,

```
type typ =
    | CstI of int
    | CstB of bool
    | Var of string
    | Let of string * expr * expr
    | Prim of string * expr * expr
    | App of expr * expr
    | Lambda of string * expr
    | Delay of expr
```

```
| Adv of expr
| Box of expr
| Unbox of expr
| Head of expr
| Tail of expr
| Cons of (expr * expr)
| Rec of string * expr * expr
| Tuple of expr * expr
```

6.2.2 Recursive functions and recursive flag

The third argument for every variable in the environment corresponds to a boolean value to distinguish recursive variables in time, previously stated in 2.3.2. The recursive variables are only introduced in the *Rec* expression, to infer the type of its first expression, as described in the rule for *Rec:* 12. The variables that have this value set to *true* should only be available to find in further steps on time, not in the present. This means that our `lookup` function will read this flag every time it searches for a variable, and only return a type for a recursive variable if it is not in the present. To help us visualize it, a recursive variable should look like this in the environment.

```
[[...];[("map",TypeScheme (...), true, false)]]
```

6.2.3 Stable flag for $\Gamma^{\Box}$

The fourth argument that we added to our variables in the environment is an aid that will help us determine which types should be stable because they were found in $\Gamma^{\Box}$. First of all, this flag will let our `lookup` function add a type to C regardless if it was found in the "present", or in another step in time. It's important to not confuse it with the *isStable* flag that we assigned for the type variable, as their purposes differ. This flag on the environment is exclusive for $\Gamma^{\Box}$, whereas the *isStable* flag, once is set as true, it won't be reversed, in order to keep track of stable type variables. We will discuss more about it below.

6.2.4 Type variables

As mentioned, the type variable consists of a level, a stable flag, and an acyclic graph structure *tyvarkind* consisting of nodes, which are built using a union-find data structure. All nodes that are connected are considered equal and form part of an *equivalence class*. Each of these classes contains a node that represents them all, known as the *canonical representative* [9].
Our data structure works following three main operations. First, when we create a fresh type variable, we create a *new* node `NoLink` with a one-element equivalence class. Secondly, by *unifying* two types, we are joining their equivalence classes into one. Lastly, we *find* the node that is canonical representative of its

class.
Each type variable we have, has either a `NoLink`, which means this node is the canonical representative, or `LinkTo` that is joined to other nodes of the same equivalence class. To *find* the canonical representative of a class, we follow each *LinksTo* until we reach a *NoLink*.

6.2.5 Level of binding

The level *lvl* that the function `typ` takes, is a solution implemented by Sestof [9] where he keeps track of the level of binding of the type environment and the type variables. The outermost level, or our initial state, is zero and increases each time we enter the right-hand side of a *let-binding*, which can occur in *Let*, *Lamda*, and *Rec* in our language. With this level we can easily detect if a variable is bound, for example, if the level of the type is lower than the current level, then it is free in this environment, thus it can't be generalized [9]. It is also used for the unification process, when it is trying to equate two types, we take the lowest binding level and set it for both unified types.

6.2.6 Set of stables C and *isStable* flag

To solve the first problem described in 6.1.2, we included a set to save stable types C and added the *isStable* flag to the type variables. Each time we try to `lookup` for a type variable in a different step of time that is not present, we add the resulting type in C.
In the same way, when we need to infer the type of a term that has been boxed *Box(t)* after we get a type τ for the term t, then we infer that its type should be stable, so we add the boxed version of the type, $\Box\tau$, to C.
When we finish the type inference for the left side of a recursive function, we check that all the types added to C are in fact stable, or fail otherwise. The purpose of doing it at this step is to make sure that if the right side of the function wants to unify a type that is on C, it can only do it with another stable type. When we check the types in C, if we find types are base (int or bool), or boxed, then we can remove them from C as we don't need to keep track of them in our set, while keeping C as short as possible. When we find a type variable there, then we change its *isStable* flag to *true* and keep it in C.

6.2.7 Lookup function

The `lookup` function was initially based on Sestof's proposal for his type inference [9]. It attempts to find the variable's name in the context and returns an instantiated version of the type found. However, our version needed some additional steps for our types.
To begin, we need to distinguish between types that can be found in the present from the types that can be found in other steps in time. First, non-stable types can only be found in the present. On the contrary, recursive variables can't be found in the present. This means that the `lookup` function will start by checking that the *isRec* flag is set to false, returning the value found regardless of the

type, or failing otherwise. If the function cannot find the type in the "present" list, then it will attempt to find it in the next step in time by calling a helper function. This second function will look for the type recursively in every other step in time. It starts by checking the *isRec* flag for the found type, if it is true, now it will be safely returned. If this flag is false, then it must be a stable type so we should add it to our set C.

In addition, it can be the case when we are searching for types in a context $\Gamma^{\Box}$. For this purpose, when the `lookup` function is in the "present", it must also check our fourth flag. If it is true, it can only mean that it was put there by $\Gamma^{\Box}$, so it must be stable, and thus it should be added to C.

6.2.8 Unify algorithm

We used the unify algorithm 1 and the `unify` function proposed by Sestof [9] as a base to solve the type inference for the terms in STLC, however, we extended the cases to be able to equate tuples, modal types, and streams.
First, we added the case where we need to unify tuple types *Tup*. Following the same line for functions, we unify the first elements of both tuples recursively and then proceed to unify the second elements.

Secondly, to deal with stable types, we need to do an additional check before equating type variables. Given that a type variable α has a flag stating if it is stable or not, we need to check it every time we attempt to unify it with anything else. If it is set to false, then α can be safely equated following our algorithm rules. But if the flag is set to true, it means that α can only be unified with another stable type. This means that before proceeding with the unification, we must verify that the second type is stable, and fail the unification otherwise.

Thirdly, for the modality type $\bigcirc$, as stated in 6.1.1, if an advanced term is inferred to be a type variable, we need to equate it with a fresh delayed type. A type variable α can be equated with any non-stable type as long as α does not occur in it, as specified in the unification algorithm, and its *isStable* flag remains `false`. This same argument stands when unifying a type variable with a modal type.

Finally, if we are attempting to unify a delayed-type with a non-type variable, then we can only equate types with the same modality, for instance, a $\bigcirc\alpha$ can only by unified with another $\bigcirc\beta$. The same ideas stand for equating *Str* types.

The table 2 helps us visualize a summary of our final unification algorithm, based on a table created by Sestof showing the first rules [9]

Note that any case not covered here, will cause the unification to fail. How the `unify` method works essentially is that it first tries to find the canonical representation of both types, in case they are type variables, and then proceeds to match the cases accordingly. Then it calls an auxiliary method `linkVarToType` which asserts that we don't have circularity (α does not occur in t_1) and checks the level for both types. Finally, it creates a link to connect the first type with the second, `LinkTo` t_2, and adds it into the type scheme of the first type, joining

type1	type2	Action
int	int	No action needed
$t_1 \to t_2$	$t_3 \to t_4$	Unify t_1 with t_3 and t_2 with t_4
α	α	No action needed
α	β	Make α equal to β
(t_1, t_2)	(t_3, t_4)	Unify t_1 with t_3 and t_2 with t_4
(t_1, t_2)	(t_3, t_4)	Unify t_1 with t_3 and t_2 with t_4
α^{false}	t_1	Make α^{false} equal to t_1, given that α does not occur in t_1
α^{true}	t_1	If t_1 is stable, make α^{true} equal to t_1, given that α does not occur in t_1
$\bigcirc\ t_1$	$\bigcirc\ t_2$	Unify t_1 and t_2
$\square\ t_1$	$\square\ t_2$	Unify t_1 and t_2
$Str\ t_1$	$Str\ t_2$	Unify t_1 and t_2

Table 2: Unification

its equivalence classes.

6.2.9 Type instantiation and generalization

The `generalize` function in our type inference takes two arguments: the level of the environment at the time of the generalization *lvl*, and a type *t*, and returns a type scheme. How it works essentially is that it starts by filtering out the free variables that occur in t that are not free in this environment, which is done by comparing their binding level. This process results in a list of types *tvs* that will be returned in a type-scheme with the type t, as:

```
TypeScheme[(tvs, t)]
```

This generalization process is done in two cases in our type inference. The first is when we try to infer a *Let* function, according to 15. The second case is when we want to infer a recursive *Rec* function. First, we create a fresh variable for the recursive function x and add it as a recursive variable in the environment. After inferring a type for this first expression, we create another environment with x as a non-recursive variable and a generalized type. We shall use this second environment to evaluate the second expression.

For instantiating a type, the `specialize` function takes the level *lvl* of the environment, and a type scheme, and returns an instantiated type *typ*. It creates a substitution list *subst* with each item on the list *tvs* of the type scheme, with creates tuples of a variable and a fresh type variable. Then it uses a helper function `copyType` to apply these substitutions of fresh variables to the type t. This function recursively traverses the type structure, replacing type variables according to the *subst* list and copying the other types. In the case that *tvs* is empty, then it just returns *t*. All this process of using these fresh variables is to

prevent overwriting the types each time we instantiate them. In other words, in one application it can be instantiated to *int*, and in another application, it can be instantiated to *bool*. Recall that this `specialize` function is called each time we look up a type of a variable.

6.3 Automatic Guarded Recursion and Type Inference

This section deals with the problem of guarded recursion, and how we use the type checker to implicitly place the Rattus terms *Delay* and *Adv*. We discuss how Rattus solves the problem of guarded recursion in section 2.4.2. Our language also needs *Delay* and *Adv* to function, since it's based on Rattus. This section explains how we deduced where to place *Delay* and *Adv*, and how the type checker uses this deduction to slightly alter the program and place the terms in their appropriate spots.

6.3.1 Intro to placing Delay and Advance

While *Delay* and *Adv* solve the problem of guarded recursion, it evidently creates a new problem: in these cases, the programmer is burdened with having to control the recursion on their own by writing *Delay* and *Adv* at their appropriate spots in the program they are attempting to write. This has the advantage of making the type inference quite easy, but has the disadvantage of the programmer having to know the rules for how to use the introduction and elimination of the modal operator $\bigcirc$. This is not ideal. Optimally, we want the programmer to be able to write type-safe programs, without having to worry about the rules for guarded recursion. Bahr [1] suggests that the $\bigcirc$ modality should implicitly be inserted in guarded recursive types like Fix $\alpha.A \times \alpha$ should be equivalent to $A \times \bigcirc(\text{Fix } \alpha.A \times \alpha)$.

Traditional approaches have left the handling of recursion control as a burden on the programmer, where the programmer has had to manage and control recursive calls using certain syntactical constructs such as explicit delay and advance markers. However, though this is the easiest to implement, it leaves the programmer with a new level of complexity and makes programs quite error-prone. Severi (2019) [2] addresses this problem and introduces a novel solution that shifts the responsibility of recursion control from the programmer to the compiler. She introduces a silent modal operator, effectively placing the responsibility of recursion outside of syntax. There was certainly some inspiration regarding how Severi (2019) [2] gave the responsibility of handling recursion to the compiler. However, we have different constructs than Severi, and must therefore approach it slightly differently, but admit that Severi solves the problem very well given their constructs. We opted to make our typechecker handle the recursion. So while the typechecker is type infering and checking the program, it also looks out for specific instances and then slightly alters program by adding *Delay* and *Adv* in their appropriate spots.

As stated earlier, regular recursion is not allowed, and when the type system encounters something that looks like regular recursion, then it fails. In fact,

the typechecker can predict this with accuracy, and before we added the implementation to implicitly place *Adv* and *Delay*, it would throw the custom error: *RecursiveFunctionException*.

6.3.2 Delay

In the current implementation, when the typechecker is trying to type infer the tail of a stream with the abstract syntax of *Cons(e1, e2)*, it is attempting to infer the type of e2 by the rules for tail (10) and Cons (11). In such a situation, if *e2* is inferred to have type $\bigcirc StrA$ and *e1* has the type A, then it typechecks. This means that everything is fine. However, such a situation does not involve recursion, and in fact, must look something like this:

`xs:`$\bigcirc StrInt$
```
let rec newStream ys = 1 :::  ys;
newStream xs;
```

In this case, we have a predefined variable, that has the type $\bigcirc StrInt$, it is then attached as the tail of a stream, with an int 1 as its head. Notice here that no delay is inserted.

However, other cases fail. In particular, recursive functions that call themselves fail. In such cases, we simply wrap *e2* from *Cons(e1, e2)* in a *Delay* and then attempt to type infer on this new abstract syntax:

```
Delay(e2)
```

This means that an example like this:

```
let rec from n = n :::  from (n + 1);
```

Will initially fail within the typechecker, this error is caught by the typechecker however. It then takes the code to the right of the *:::*, wraps it in a *Delay* like below and attempts to run the program again:

```
let rec from n = n :::  Delay( from (n + 1));
```

It will attempt to add a delay at the tail of a *Cons* that fails to type infer or fails to follow a typing rule. It then reruns the abstract syntax of the tail, now packed in a *Delay*, and attempts to infer the type and check if it follows the typing rule; if either fails, then the typechecker fails. Note that the typechecker does not actually interact with the syntax of the program, but only interacts and transforms the *abstract* syntax of the program. The above example was to make it easier to digest. The actual abstract syntax looks like this:

```
Rec
```

```
("from",
 Lambda
   ("n", Cons (Var "n", App (Var "from", Prim ("+", Var "n", CstI 1)))),
 Var "from")
```

and is transformed into this:

```
Rec
("from",             delay inserted
 Lambda                   ↓
   ("n", Cons (Var "n", Delay(App (Var "from", Prim ("+", Var "n", CstI 1))))),
 Var "from")
```

6.3.3 Placing Adv

Advance, as stated earlier, is the elimination form of ◯. The typing rule for this also expects the context to have at least one ✓, meaning that when advance will type check we are looking at least one time step into the future; this means that *adv* is always placed inside at least one *delay*. This is the first observation. Next, we know that *adv* should only *advance* something of type $\bigcirc A$. This is the second observation.

The logic given these observations is quite simple, luckily. There is only certain terms that can actually give us a $\bigcirc A$. Obviously, from the typing rule of *delay* we know that once applied to a term t of type A, it gives us $\bigcirc A$. However, *delay* is now controlled by the typechecker, so we can disregard this line of thinking entirely. But we have another *term* that readily gives us a ◯, which is the *term Tail*, that by its typing rule takes a *Str A* and gives us a ◯ *Str A* . The last term that can potentially give us a type $\bigcirc A$, is the term *Var*, which can have variables of type $\bigcirc A$ saved in the environment.

If the typechecker encounters a *Var* term while recursively traveling the abstract syntax tree, it first gets the type that *Var* represents. It then checks whether there is a ✓ in the context. If there is no ✓ in the context, it simply returns the type and does not transform the AST. If it finds a ✓, then it checks whether the type from *Var* is of type $\bigcirc A$ (including ◯ Str A), if this is the case, then it wraps the *Var* in an *Adv* so it looks like:

```
Adv(Var "x" )
```

If this succeeds, the typechecker then gives back the type: *A* (or *Str A*, if it was ◯ *Str A*). It then also transforms the abstract syntax by placing an *Adv* around the *Var*. There is also another case, if the type is evaluating the term *Tail*, and there is a ✓ in the context. In this case, it also wraps the *Tail* in *Adv*. However, this approach is rarely shown due to the less elegant syntax:

```
let rec map f s = Unbox(f) Head(s) ::: map f Tail(s);
```

compared to the more readable form:

```
let rec map f (x::xs) = Unbox(f) x ::: map f xs;
```

To showcase how this works in greater detail, let's say that we have a recursive function that manipulates streams called *Inc*. The function is quite simple. It takes the head of a stream, and increments it by 1. The *Inc* function looks like this:

```
let rec inc (x::xs) = x + 1 :::  inc xs;
```

This then gets translated into this abstract syntax by the parser:

```
Rec
("inc",
 Lambda
   ("manualStreamsTEvnywdrLuNlWO1",
    Let
      ("x", Head (Var "manualStreamsTEvnywdrLuNlWO1"),
       Let
         ("xs", Tail (Var "manualStreamsTEvnywdrLuNlWO1"),
          Cons (Prim ("+", Var "x", CstI 1), App (Var "inc", Var "xs"))))),
 Var "inc")
```

However, we will only focus on the part of the abstract syntax that deals with streams, namely the *Cons*:

```
Cons (Prim ("+", Var "x", CstI 1), App (Var "inc", Var "xs"))
```

The following table is a little verbose but shows the steps the typechecker takes. Nothing happens in step 1, but remark that the typechecker has "x", "inc" and "xs" in its environment. In step 2 it infers that the `Prim ("+", Var "x", CstI 1)` is a TypI (int), and thereby also infers that "x" is `TypI`. In step 3 it starts evaluating the tail of the *Cons*. In step 4 we reach the `Var "inc"`, which ensures that it fails since recursive functions can only be accessed in a context with a ✓. In step 5 we add the *Delay*, thus inserting a ✓. It then tries to infer `App(Var "inc", Var "xs")` again; at first this is fine. In step 6 we reach the *Var "xs"*, which is inferred to have type ◯Str TypI (int). Since it has ◯ and is inside an environment with a ✓, it inserts an `Adv`, which we see in step 7. In step 7, `Adv` has been inserted, and it then infers the `Adv(Var "xs")` to have type `Str(TypI)` (Str int). The actual transformation of the AST is done here, but for the sake of clarity, we have two more steps. In step 8 we try to infer the entire tail, which ends up having type ◯`Str(TypI)`, aka ◯ Str Int. In step 9 we try to infer the entire `Cons`, which gives us the type `Str(TypI)`.

s	AST	ticks	Type	Environment
1	Cons (Prim (”+”, Var ”x”, CstI 1), (App (Var ”inc”, Var ”xs”)))))			`[[("inc", TypV, true); ("x", TypV, false); ("xs", ◯Str(TypV))]]`
2	Cons **(Prim (”+”, Var ”x”, CstI 1),** (App (Var ”inc”, Var ”xs”)))))		TypI	`[[("inc", TypV, true); ("x", TypI, false); ("xs", ◯Str(TypI))]]`
3	Cons (Prim (”+”, Var ”x”, CstI 1), **(App (Var ”inc”,** Var ”xs”)))))			`[[("inc", TypV, true); ("x", TypI, false); ("xs", ◯Str(TypI))]]`
4	Cons (Prim (”+”, Var ”x”, CstI 1), (App (**Var ”inc”**, Var ”xs”)))))		Fail	`[[("inc", TypV, true); ("x", TypI, false); ("xs", ◯Str(TypI))]]`
5	Cons (Prim (”+”, Var ”x”, CstI 1), **Delay(App (Var ”inc”,** **Var ”xs”))**)))	✓		`[[];[("inc", TypV, true); ("x", TypI, false); ("xs", ◯Str(TypI))]]`
6	Cons (Prim (”+”, Var ”x”, CstI 1), Delay(App (Var ”inc”, **Var ”xs”))**)))	✓	◯Str(TypI))	`[[];[("inc", TypV, true); ("x", TypI, false); ("xs", ◯Str(TypI))]]`
7	Cons (Prim (”+”, Var ”x”, CstI 1), Delay(App (Var ”inc”, **Adv(Var ”xs”))**)))	✓	Str(TypI))	`[[];[("inc", TypV, true); ("x", TypI, false); ("xs", Str(TypI))]]`
8	Cons (Prim (”+”, Var ”x”, CstI 1), **Delay(App (Var ”inc”,** **Adv(Var ”xs”))**)))	✓	◯Str(TypI))	`[[];[("inc", TypV, true); ("x", TypI, false); ("xs", Str(TypI))]]`
8	**Cons (Prim (”+”,** **Var ”x”, CstI 1),** **Delay(App (Var ”inc”,** **Adv(Var ”xs”))**)))		Str(TypI))	`[[];[("inc", TypV, true); ("x", TypI, false); ("xs", Str(TypI))]]`

6.4 Examples for type inference

To demonstrate how the complete implementation of the type inference works, we will show it with an example that is commonly used in our language. The function `constInt` is just a simple recursive function that places the same value recursively in a stream. Even though it is very easy to analyze, it demonstrates how the type inference works, including how we deal with stable types. By simple inspection, we can deduce the type of `constInt`. It takes an argument of any

type α and returns a stream of the same type α, so it should look like $\alpha \rightarrow Str\alpha$.

```
Language:  let rec constInt n = n :::  constInt n;
```

This syntax will translate into an initial AST

```
Rec
  ("constInt", Lambda ("n", Cons (Var "n",
      App (Var "constInt", Var "n"))),
      Var "constInt")
```

We will explain every step of the type inference and present a table for a better understanding of how the abstract syntax tree, environment, and set of stables types C change over the process. We will also show the types that the expression returns when its type inference is finished. The environment shown in the table represents its result before moving to the next step. For simplicity, we will only show the type *typ* of each variable, instead of its type scheme.

We start to run the function `inferType` that takes the expression and initializes with an empty environment `[[]]`, level zero, and an empty set of stable types. This starts with the case for *Rec*, which will increase the binding level by one, create a fresh type variable α for "constInt" and add it to the environment with the *isRec* flag set to true. Note that even though we are going into a $\Gamma^{\square}$, the stable flag is false for `constInt`. This is because it is a recursive variable.

AST	Type	Environment	C
Rec("constInt", Lambda ("n", Cons (Var "n", App (Var "constInt", Var "n"))), Var "constInt")		`[[("constInt", (TypV` α`, false), true)]]`	`[]`

With this new environment, we will continue to infer the type for the first expression, which in this case is *Lambda*. This case acts similarly as it also increases the level by one, creates a new type variable β for "n", adds it to the environment, and afterward uses it to infer the type for its body.

AST	Type	Environment	C
Rec("constInt", **Lambda ("n", Cons (Var "n", App (Var "constInt", Var "n"))),** Var "constInt")		`[[("constInt", (TypV` α`, false), true); ("n", (TypV` β`, false), false)]]`	`[]`

The body for the *Lambda* is a *Cons* term. This case starts by inferring the type for its head, which is the variable "n". This means that we will call the

`lookup` function to find the type for "n", which is found in the present as the type variable β.

AST	Type	Environment	C
Rec("constInt", Lambda ("n", **Cons (Var "n",** App (Var "constInt", Var "n"))), Var "constInt")	`TypV` β	`[[("constInt", (TypV` α`, false), true); ("n", (TypV` β`, false), false)]]`	`[]`

Now we can infer the type for the tail of *Cons*, which is a term *App*. This case will attempt to infer the type for the variable "constInt" which will fail, as we thoroughly described in 6.3.1, and will change the AST to include a *Delay*, and then proceed to infer it. *Delay* will add a step in time, and try to infer again the *App* term.

AST	Type	Environment	C
Rec("constInt", Lambda ("n", Cons (Var "n", **Delay(App (Var "constInt", Var "n"))))**, Var "constInt")		`[[];[("constInt", (TypV` α`, false), true); ("n", (TypV` β`, false), false)]]`	`[]`

The *App* expression will start by looking up its first term, the variable "constInt". This time, this will succeed as this variable is in another time step and is recursive, which results in the type variable α.

AST	Type	Environment	C
Rec("constInt", Lambda ("n", Cons (Var "n", Delay(App(**Var "constInt",** Var "n")))), Var "constInt")	`TypV` α	`[[];[("constInt", (TypV` α`, false), true); ("n", (TypV` β`, false), false)]]`	`[]`

Now it will continue with the second term of the application, which is the *variable* "n". The `lookup` function will find this variable in another time step, which means that it must be a stable type, adding it to C.

AST	Type	Environment	C
Rec("constInt", Lambda ("n", Cons (Var "n", Delay(App(Var "constInt", **Var "n")**))), Var "constInt")	`TypV` β	`[[];[("constInt", (TypV` α`, false), true); ("n", (TypV` β`, false), false)]]`	`[TypV` β`]`

The expression *App* can continue and will attempt to unify the type from its first term with a function type that takes as its first argument the type of the second term and a type of a fresh type variable γ for the second argument, $TypV\ \beta \rightarrow TypV\ \gamma$. If the unification is possible, this function type will be linked to α, and then *App* will return the type variable γ.

AST	Type	Environment	C
Rec("constInt", Lambda ("n", Cons (Var "n", Delay(**App(Var "constInt", Var "n")**)), Var "constInt")	`TypV` γ	`[[];[("constInt", (TypF(`$\beta \rightarrow \gamma$`), false), true); ("n", (TypV` β`, false), false)]]`	`[TypV` β`]`

Now we can continue with the term *Delay*, where we will take the type we got from inferring *App* and add a ◯ modal to it.

AST	Type	Environment	C
Rec("constInt", Lambda ("n", Cons (Var "n", **Delay(App(Var "constInt", Var "n"))**), Var "constInt")	◯`TypV` γ	`[[];[("constInt", (TypF(`$\beta \rightarrow \gamma$`), false), true); ("n", (TypV` β`, false), false)]]`	`[TypV` β`]`

After finishing with *delay*, we can go back to *Cons*, where now we have the type for its tail, ◯γ. As described on 11, if the head is of type A, and the tail is of type ◯$Str\ A$, then *Cons* is of type $Str\ A$. We can check this by unifying the type we got from the tail, with a ◯*Str* added to the type we got from the head. This is `unify(`◯`TypV`γ`,` ◯`Str TypV` β`)`. This will add the link to ◯ $Str\ TypV\ \beta$ for γ. If this unification is possible, then we can infer type $Str\ TypV\ \beta$.

AST	Type	Environment	C
Rec("constInt", Lambda ("n", **Cons (Var "n", Delay(App(Var "constInt", Var "n"))))**, Var "constInt")	`Str TypV` β	`[[("constInt", (TypF(`$\beta\rightarrow$ $\bigcirc$`Str`β`), false), true); ("n", (TypV` β`, false), false)]]`	`[TypV` β`]`

This result is the type that is inferred for the body of our *Lambda* expression. To finish inferring the type for this term, we will return a function type built with the first argument corresponding to the first expression, and the second argument as the type we just got from the body.

AST	Type	Environment	C
Rec("constInt", **Lambda ("n", Cons (Var "n", Delay(App(Var "constInt", Var "n"))))**, Var "constInt")	`TypF(TypV`$\beta \rightarrow$ `Str TypV`β`)`	`[[("constInt", (TypF(`$\beta\rightarrow$ $\bigcirc$`Str`β`), false), true); ("n", (TypV` β`, false), false)]]`	`[TypV` β`]`

Now we can finally continue with our first term *Rec*. As described in 12, the type that we get from the first term, should be the same type as the first argument. This means that the type variable we created for "constInt" should be the same as the type we got from the first expression, which means that we can unify them. As we can observe, the type of "constInt" is the same as the type resulting from the first term. At this point, we will also `reduce` the set of stable types. It contains the type variable β, which is only a `NoLink` at this point, which means it hasn't been linked to any non-stable type, so we just proceed to change its *isStable* flag to true. Now we should generalize the type variable of "constInt" and then add it to the environment with its *isRec* flag set to false.

AST	Type	Environment	C
Rec("constInt",Lambda ("n", Cons (Var "n", Delay(App(Var "constInt", Var "n")))), Var "constInt")		`[[("constInt", (TypF(`$\beta\rightarrow$ $\bigcirc$`Str`β`), false), false); ("n", (TypV` β`, true), false)]]`	`[TypV` β`]`

Lastly, we infer the type of the second argument, which in this case is just the variable "constInt", which is $\beta \rightarrow Str \bigcirc\beta$.

AST	Type	Environment	C
Rec("constInt",Lambda ("n", Cons (Var "n", Delay(App(Var "constInt", Var "n")))), **Var "constInt")**	`TypF(`$\beta\rightarrow$ $\bigcirc$`Str`β`)`	`[[("constInt", (TypF(`$\beta\rightarrow$ $\bigcirc$`Str`β`), false), false); ("n", (TypV` β`, true), false)]]`	`[TypV` β`]`

Note that according to our type inference, the type of the argument is the same type that is in C. This means that when we want to apply this function, the argument must be a stable type. For instance, a function `constInt 1` will type check, as the type of 1 is int. But if we attempt to type check `constInt (fun x -> 1)`, when it tries to unify β with $\delta \rightarrow int$, it will fail with the message "Stable type error: Tried to match Stable Type: S'b with non-stable type: ('c $\rightarrow$ int)".

7 Interpreter

An interpreter is essentially a program that takes an input and returns a result. It's important to denote the difference between the interpreter language L (the language that we are using for our version of Rattus) and the implementation language I (the language that we used to write the interpreter, F#). The program in L is a sequence of operations that resembles machine code, which is why the interpreter is known as an abstract machine [9].

Similar to type inference, the evaluation of expressions can be described by rules. For this language, we are using the structural operational semantics, which read similarly to the type system rules. Similarly, it contains zero or more premises above a horizontal line, and if all premises hold, then the conclusion must be true [9]. The premises and conclusions consist of *judgments*, which are written as $\langle t, \rho\rangle \Downarrow \langle t_2, \rho\rangle$ where the symbol $\Downarrow$ reads as *evaluates*, and the symbol ρ is the environment of variables with its respective value. In the same way as in type inference, `[t/x]s` refers to substituting every occurrence of `t` with `x` in `s`. We are using the big-step semantic as it directly shows the final values for the evaluations. The rules for our language are defined as:

$$\frac{\rho(x) = v}{\langle x, \rho\rangle \Downarrow \langle v, \rho\rangle}\ \text{Variable} \tag{19}$$

$$\frac{\langle t_1, \rho\rangle \Downarrow \langle v_1, \rho'\rangle \qquad \langle t_2, \rho'\rangle \Downarrow \langle v_2, \rho''\rangle}{\langle t_1 + t_2, \rho\rangle \Downarrow \langle v_1 + v_2, \rho''\rangle}\ \text{Primary operators} \tag{20}$$

$$\frac{\langle t, \rho\rangle \Downarrow \langle \lambda x.s, \rho'\rangle \qquad \langle t', \rho'\rangle \Downarrow \langle v, \rho''\rangle \qquad \langle s[v/x], \rho''\rangle \Downarrow \langle v', \rho'''\rangle}{\langle t't, \rho\rangle \Downarrow \langle v', \rho'''\rangle}\ \text{Application} \tag{21}$$

$$\frac{}{\langle \lambda x.t, \rho \rangle \Downarrow \langle \lambda x.t, \rho \rangle} \text{ Lambda} \tag{22}$$

$$\frac{\langle t_1, \rho \rangle \Downarrow \langle v_1, \rho' \rangle \qquad \langle t_2[v_1/x], \rho' \rangle \Downarrow \langle v_2, \rho'' \rangle}{\langle \text{let } x = t_1 \text{ in } t_2, \rho \rangle \Downarrow \langle v_2, \rho'' \rangle} \text{ Let} \tag{23}$$

7.1 Values

The parser will take the language given by the user, and return the abstract syntax, which the typechecker then slightly alters, where finally the interpreter will evaluate recursively to return a value. Since we are working with high-order functions, an evaluation may return either an integer or another function. A value of a function is known as a *Closure* which consists of a name, the function body, and the environment that contains the types of the free variables at the point of the evaluation [9].

values ::=	*Int of int*	*Integer*
	*Closure of string * expr * env*	*Closure*

As previously stated in the rules 21 and 22, after the evaluation of the expression, we get a function value. For instance, for a simple evaluation of an anonymous function, the parser will return the abstract syntax of a Lambda term. The interpreter will evaluate the *Lambda* term and determine that it should add the expression to the environment and return the non-evaluated value of the expression,

```
Language:  (fun x -> x + 1)
Abstract syntax:  Lambda("x", Prim("+", Var"x", CSTI 1))
Evaluation:  Closure("x", Lambda("x", Prim("+", Var"x", CSTI 1)), ["x",
Closure("x", Lambda("x", Prim("+", Var"x", CSTI 1)), []])
```

In the same way, when the interpreter wants to evaluate an *Application* as shown in the rule 21, it evaluates the first term, and as it should be a function, it matches the return value with a *Closure* value and fails otherwise.

7.2 Rattus Abstract Machine

This abstract machine must be able to move forward in time, but should also be absent of space leaks. In order to accomplish this, we must erase all data from the previous time step each time we move forward. Bahr [1] proposes the idea of using a store σ that contains up to two separate heaps η, one *now* heap where we save delayed operations that we can look for in order to execute in the present, and a *later* heap where we save operations that we can use in the next time step. This means that we can only read from *now* heap and write on *later*

heap. Each time we advance in time, this *now* heap gets erased and replaced by the *later* heap, leaving a new empty *later* heap. This prevents the storage of old data which could lead to implicit space leaks.

Additionally, Bahr [1] proposes to split the abstract machine in two components: the *evaluation* semantics which describes how the terms are evaluated in a single step of time, and a *step* semantics that describe how to evaluate the program over time by taking and constructing streams.

7.2.1 Evaluation Semantics

The *evaluation* semantics for the terms in our language are similar to 19, 21, 22, and 23, but besides the use of an environment, he proposes the use of the store σ. This store can consist of a single heap η_L or two heaps η_N and η_L, for now and later heap respectively, represented as $\eta_N \checkmark \eta_L$.

The heaps are essentially finite maps from heap location l to terms. Each time we need to create a new heap location, we should call the method *alloc* that will return a fresh l for η_L. Recall that we can only write in η_L. This method is called when we need to *delay* a term, by placing the delayed term in the fresh l on η_L, and returning l. In the same line, when we need to advance a term, we get the term that was delayed and now must be found in the present η_N and evaluated. This translates to:

$$\frac{\langle t; \eta_N \rangle \Downarrow \langle l; \eta'_N \rangle \qquad \langle \eta'_N(l); \eta'_N \checkmark \eta_L \rangle \Downarrow \langle v; \sigma' \rangle}{\langle \text{adv } t; \eta_N \cup \eta_L \rangle \Downarrow \langle v; \sigma' \rangle} \text{ Advance} \tag{24}$$

$$\frac{l = \text{alloc}(\sigma)}{\langle \text{delay } t; \sigma \rangle \Downarrow \langle l; \sigma, l \mapsto t \rangle} \text{ Delay} \tag{25}$$

As observed in both of these terms, we should be able to return and get a value of l, adding it to our list of values. However, the user cannot use or see this value directly and it doesn't have a typing rule, it is just used for simplicity in the abstract machine [1].

Similarly to *lambda* in the rule 22, to evaluate the term *Box*, we will not evaluate the term inside, but return it as a `box` value, which is constructed similar to a closure of a function. When we need to *unbox* it, we will evaluate the term inside and check that the value is precisely of a `box`, and then we will proceed to evaluate it.

$$\frac{}{\langle \text{Box t}, \sigma \rangle \Downarrow \langle \text{box t}, \sigma \rangle} \text{ Box} \tag{26}$$

$$\frac{\langle t, \sigma \rangle \Downarrow \langle \text{box } t', \sigma' \rangle \qquad \langle t', \sigma' \rangle \Downarrow \langle v, \sigma'' \rangle}{\langle \text{unbox } t, \sigma \rangle \Downarrow \langle v, \sigma'' \rangle} \text{ Unbox} \tag{27}$$

For evaluating the *head* and *tail* of Stream, we need to decompose it. We can do this by using projection functions *head* and *tail* that extract the head

and tail, respectively, of a stream [1].
`head` = λ`x.`π_1
`tail` = λ`x.`π_2

$$\frac{\langle t;\sigma\rangle \Downarrow \langle\langle v_1, v_2\rangle;\sigma'\rangle \qquad i \in \{1,2\}}{\langle \pi_i(t);\sigma\rangle \Downarrow \langle v_i;\sigma'\rangle}\ \text{Head and Tail} \tag{28}$$

This means, if we want to evaluate a head *Head(t)* or a tail *Tail(t)*, we will evaluate the expression t which should be a stream. This means we can pattern match and extract the head or the tail value of the stream and return it.

7.2.2 Step Semantic

The step semantics, represented with $\overset{v}{\Longrightarrow}$, where v is the output of the expression in a step in time, describe how to operate the streams of type *Str A* [1].

$$\frac{\langle t;\eta\checkmark\rangle \Downarrow \langle v :: l, \eta_N \checkmark \eta_L\rangle}{\langle t;\eta\rangle \overset{v}{\Longrightarrow} \langle \text{adv } l;\eta_L\rangle} \tag{29}$$

At each step of time, when we evaluate an expression t with a store that contains a heap *now* η and an empty heap *later* η_L, we expect to get a value of a stream with a value of type A in its head and a location for a heap in the tail. Additionally, it contains a store with a heap η_N that has the same values in η and possibly some additional computations from the evaluation, and η_L that contains delayed computations for the next step. To get to the next step in time, we need to advance the location we got from the tail with the new *now* heap η_L. Notice how we don't use the η_N for the next step in time to not save old data.

These operations should return an infinite sequence like [1],

$$\langle t;\varnothing\rangle \overset{v_0}{\Longrightarrow} \langle t_1;\eta_1\rangle \overset{v_1}{\Longrightarrow} \langle t_2;\eta_2\rangle ...$$

For instance, if we operate a `from n` function where we increment n by 1 at each step of time, starting with n as 1,

```
Int → Str Int
let rec from n= n:::Delay(from(n+1));
from 1;
```

Then we should expect to get something like this,

$$\langle from\ 1;\varnothing\rangle \overset{1}{\Longrightarrow} \langle adv\ l_1;\eta_1 = l_1 \mapsto from(1+1)\rangle$$
$$\langle adv\ l_1\ 1;\eta_1\rangle \overset{2}{\Longrightarrow} \langle adv\ l_2;\eta_1 = l_2 \mapsto from(2+1)\rangle ...$$

As we can observe, at each step in time, we get a value of Int, given that we are operating a stream of integers. Additionally, we can also observe that the store, after the first iteration, contains a heap location containing the closure of

the recursive function *from*, put in there by *delay*, with its environment where n is incremented by 1, and other values that resulted from evaluating the terms following the evaluation semantics.

The second case for the step semantics, represented with $\xrightarrow{v_0/v_0'}$, where v_0 is an input that the stream will receive and v_0' is the output of the expression in a step in time, describes how to operate streams of type $Str\ A \to Str\ B$ [1].

$$\frac{\langle t; \eta, l^* \mapsto v :: l^* \mapsto \langle\rangle\rangle \Downarrow \langle v' :: l; \eta_N \checkmark \eta_L, l^* \mapsto \langle\rangle\rangle}{\langle t; \eta\rangle \xrightarrow{v/v'} \langle \text{adv } l; \eta_L\rangle} \tag{30}$$

We require a dedicated location in the heap l^* for the input, so it will only contain the present input and it's also ready to use for future inputs. It waits for an input to evaluate the term with a heap η that initially contains the heap location l^* that is ready to hold an input. Similarly to 29, this evaluates to a stream that has a head with a value, that in this case would be of $Str\ B$, and a tail with the location that will be evaluated next. Similarly, it returns a store with two heaps, η_N and η_L, and we will erase η_N and replace it with η_L, with the difference that this heap will contain an empty location l^* that will wait for the next input.

To illustrate this process, let's use a `sum` example, where we have a function that takes an input and adds it to an accumulator,

```
Str Int → Str Int
let rec sum acc (x::xs) = x + acc ::  Delay((sum (x + acc)) Adv(xs));
sum 0;
```

This expression would evaluate to,

$$\langle sum(adv\ l^*); \varnothing\rangle \xrightarrow{2/2} \langle adv\ l_1; \eta_1 = l_1 \mapsto sum, l^* \mapsto \langle\rangle\rangle$$
$$\langle sum(adv\ l^*); \varnothing\rangle \xrightarrow{10/22} \langle adv\ l_1; \eta_1 = l_1 \mapsto sum, l^* \mapsto \langle\rangle\rangle ...$$

It starts by waiting for an input which will be added to l^* and then it will evaluate the recursive function sum, which takes and returns a stream of integers. As the variable *acc* was initialized with zero, the function will add the number of the input to it, which in this case is 2. This will return 2 as an output and then will proceed to advance the tail of the evaluation l_1. Similarly, as the previous example, the store will consist of one heap that will contain the recursive function, which was put in there by delay, but in this case, it will also have the reserved heap location l^* with an empty value, waiting for the next input. Note that as we did in the previous example, the heap is replaced by η_L, in this way avoiding implicit space leaks.

7.3 Interpreter Implementation

The interpreter also applies the *expr* abstract syntax tree, with the additional expression for *Loc*. It also includes *values* that consist of integers, functions, tuples, and values gotten from evaluating Rattus terms.

```
    type value =
...
| Int of int
| Bool of bool
| VTuple of (value * value)
| Closure of (string * expr * intenv<value>)
...
```

Additionally, the value type holds a few ones that are important for the internal workings of the interpreter. The stream value, although, not explicitly returned to the user, comes into play during terms like *head* and *tail*, that rely on *stream* to produce necessary interpretations. *Location* takes a string and points to a location on the heap. We cover the heap in section 7.2, but the heap is essentially a map of heap locations and their values. *Location* is intrinsically tied to delayed computations, as the term *Delay* in fact returns a *Location*, without executing the computation, as described in 7.2.1. Lastly, it holds *Empty*, which is there as the placeholder value for l^* explained by the interpreter rule 30 for programs of type *Str A*$\rightarrow$ *Str B*.

```
    type value =
...
| Stream of (value * value)
| Location of string
| Empty
...
```

The interpreter applies another structure for its context. It applies an environment that we have called *intenv*. This environment is to keep track of variables during run-time. This is a simple Map structure that maps the variable names, saved as strings in the Map to values. The *Closure* term from before holds an entire *intenv*.

```
    type intenv<'x> = Map<string, 'x>
```

The last structure that the interpreter uses is the *store* and *heap*. The heap is a Map of strings to values. The store employs two heaps and names them the *now_heap* and the *later_heap*. The later heap is used to store delayed computations.

```
type heap = Map<string, value>

type store = { nowHeap: heap; laterHeap: heap }
```

7.3.1 General overview of the interpreter

The interpreter has three main functions, as well as multiple helper functions, that together allow us to run functional reactive programs created by the parser and type checker. The first function is the *eval* function, which runs the abstract syntax tree, produces output, and produces delayed computations. Essentially, it evaluates the *evaluation semantics* described in 7.2.1. The two other functions handle how delayed computations are interpreted depending on their type, as described on 7.2.2; *RunR* runs programs of type $Str\ A \rightarrow Str\ A$ and expects user input. Essentially *RunR* runs any program that expects one or more streams as input. The function *runXTimes* handles all other forms of programs that deal with delayed computation and takes a parameter X that is the number of times it should run delayed computations, before stopping. This is set to 10.000 in our current implementation for demonstration purposes but can run infinitely. Both of these functions use *eval* to first run the initial program produced by the parser. Then they are given the delayed computations by the evaluation of the program by *eval*. The functions then use the *eval* function again to run the newly created delayed computations. These then produce their own delayed computations, which are then run again by *eval* and so on until an exit condition is met.

7.4 Interpreter and environment

The interpreter uses an environment-based evaluation approach, which differs from direct substitution methods. Any branch of the abstract syntax tree is evaluated into a *value*, which is then returned, and possibly stored in the environment for the rest of the program to access.

7.4.1 Evaluating the program

As stated earlier, the entire evaluation of the abstract syntax tree is done by the *eval* function. In this section, we're going to go through how it works in detail and how it aligns with the theory provided by section 7.

The *eval* function applies *evalEnv*. The evalEnv takes the abstract syntax tree: *expr*, an environment: *intenv* and a heap store *store*. Eval itself only takes the *expr* and the *store*, but evaluates evalEnv with these and an empty environment. The *evalEnv* is a recursive function that continuously calls itself during evaluation and consistently updates its environment and the store. Finally, each call of *evalEnv* will eventually return a value and a new store.

7.4.2 Ints, booleans, and tuples

Integers, Booleans and tuples are quite simple for the interpreter to evaluate. The abstract syntax for these looks like this:

```
CstI of int
```

```
CstB of bool
```

```
Tuple of expr * expr
```

The evaluation of CstI and the CstB are almost self-explanatory and just return an int and boolean respectively, as well as the store they were passed. The *Tuple* takes two expressions. It just evaluates both of these expressions into their respective values. Then it returns the *VTuple* value with the values from the two expressions at the place they were represented in the expr `Tuple`.

```
VTuple of (value * value)
```

7.4.3 Variable introduction and accessing

The variable introduction in the interpreter deals with the abstract syntax:

```
Let of string * expr * expr
```

The first argument is the *string* and represents the name of the variable. The second argument is an *expr* the body of the variable, which can be any other expression. The third argument is also en *expr* and represents the rest of the abstract syntax tree; meaning the rest of the program. Firstly, the body of the variable is evaluated using *evalEnv*. This then provides a new *value* and store. The value is then added to a new environment with the name provided by the string. The last *expr*, which represents the rest of the program, is then evaluated by *evalEnv* using the new environment and store. Thus, we have successfully introduced a variable and passed it to the next step in the abstract syntax tree. The rule for let is described by the interpreter rule 23.

When accessing variables in the interpreter, it deals with the abstract syntax:

```
Var of string
```

When this is evaluated by *evalEnv*, it will look up the given string in the environment, that was passed to it. If it finds it, then it simply returns the value associated with the string, as well as the current store. If it does not find it, then the program fails, and it simply stops running. See rule 19.

7.4.4 Functions and recursive structures

Functions and recursive functions are not evaluated, but instead defer the evaluation until it is applied. There is not much difference between how functions and recursive functions defer the evaluation. A function is represented as:

```
Lambda of string * expr
```

This structure takes a string, which is a name for the parameter of the function. Next, it takes a *expr*, whereas a pattern with *expr* in *evalEnv* is usually

evaluated, in this case, the *expr* is saved inside a *Closure*, with it's current environment. The Closure is then returned.

In recursive functions, it functions similarly. A recursive function is represented as this:

```
Rec of string * expr * expr
```

This structure takes a string, which is the name of the recursive function. It then takes the first expr, which is the body of the recursive function. Finally, it takes another expr, which represents the rest of the program, much like the *Let*. When *Rec* is evaluated, the first *expr*, meaning the body, is not evaluated, but instead saved in a *Closure* with the current environment. It then adds this *Closure* to a new environment, with the name given by the string. Then, it evaluates the last *expr* with this new environment.

7.4.5 Applying functions

When applying functions, the interpreter has to deal with the abstract syntax:

```
App of expr * expr
```

Firstly, it evaluates both of the *expr*'s, where they each respectively return a value and a store. Let's denote the value returned from the first expression as *val1* and the one from the second as *val2*. *Val1* represents the function that is being applied to something. In our case, if everything goes correctly, *val1* contains a *Closure* with a function in the form of a *Lambda* in it. Both the *val1* and *val2* are then added to a new environment with their respective names. Finally, the body of the lambda is then evaluated with the new environment. This allows for both functions and recursive functions to be applied to a value. See rule the interpreter rule 21.

7.4.6 Streams

When dealing with streams, we, as noted, need something to identify and access the head and the tail of a stream, as well as representing the stream itself. The abstract syntax that deals with these are:

```
Head of expr

Tail of expr

Cons of (expr * expr)
```

As you can see, both the *expr Head* and *Tail* take an *expr*. The logic behind this is quite similar and quite simple. Firstly, they evaluate the expression with the given *environment* and *store*. We are then given a value; this value is then matched with the value *Stream*: *Stream of (value * value)*. If a stream is not returned, this results in an error. However, if a stream is returned the *Head* will return the **value** of the stream, whereas the *Tail* will return the last value of the stream, which is always the value *Location*, as the tail is not available

yet, before the next time step. For more information on *Head* and *Tail* see the interpreter rule 28. The *Cons* term is quite simple, as you can see, it takes two *expr*. Both of these *expr* are then evaluated as values. Let's denote these as *val1* and *val2*. These values are then added to the *value Stream* that looks like *Stream of (value * value)*, and it ends up being represented as:

```
Stream(Val1, Val2)
```

7.4.7 Box and Unbox

Box and unbox work similarly to how the lambda works and how it's applied. The Box and Unbox has the following abstract syntax:

```
Box of expr
```

```
Unbox of expr
```

The box takes an *expr*, and similarly to the *Lambda*, an *expr* that is *boxed*, is not evaluated. It merely puts the *expr* into the value *Vbox* and the current environment necessary for the *expr*. It then returns the *Vbox* value. Remember, that Vbox is a value that can hold an expression inside of it: *VBox of (expr * intenv⟨value⟩)*. The unbox also takes an expression. First, it evaluates this function and checks whether the value returned is a Vbox or not. If not, then the interpreter throws an error; however, this is unlikely, since the typechecker should've made sure this couldn't happen. If it finds a Vbox, it then evaluates what was previously not evaluated, however, this might result in a function, which is then also not evaluated until it is applied. This follows the interpreter rule 26 and 27.

7.4.8 Delay and Advance

The part of the abstract that deals with delaying computations is, of course, the Delay.

```
Delay of expr
```

Observe, that Delay takes an expr. However, it does not evaluate this expression. Instead, it saves the expr in a closure. It then stores the closure with a new heap location name in the *laterHeap*. The name of the location is then stored in the value *Location*, which is finally returned by Delay. The idea behind this is that the tail of the stream, meaning the part that the delay interacts with, is only available later, and therefore must be in the later heap, where it is not accessible. We will access it later, once we have taken one timestep. This follows the interpreter rule 24 for *Delay*.

Advance is the elimination form of delay, and as seen earlier it only ever interacts with the tail of a stream. The abstract syntax is:

```
Adv of expr
```

Adv takes an expression and expects this expression to be either the expr *Loc of expr* or to evaluate to the value *Location*. Both *Loc* and *Location* contain a string that points to a location on the *nowHeap*. It then looks up the location on the *nowHeap*, given the string either provided. In our implementation, this either results in a stream or a closure, which then is evaluated to a stream; meaning that *adv* always results in the value *Stream*. See rule 25 for more details.

7.5 Timesteps and reactivity

It should be clear now that the *Delay* is the one that delays computations and puts them into the later heap. Now, as stated earlier in the section, there are two ways to actually run the interpreter. Firstly, we can run programs that result in *Str A*. This is done via the *runXTimes* function. This function takes an integer and an expr as parameters; the amount of times it should run the delayed computations and the abstract syntax tree. It first runs the initial *eval* with the initial abstract syntax tree created by the parser and possibly altered by the typechecker. Once it's been run, it produces a value and a store containing a nowHeap and a laterHeap. The later heap should at this moment hold the delayed computations from the previous run of *eval*. The value produced by *eval* should be a *Stream* with a head and a tail. The head is added to a list that accumulates all heads of the produced *stream* values generated by the program. The tail is a *Location*.

We then advance time, where the nowHeap is set as the laterHeap, and the laterHeap is set to an empty heap. The *Location* from the tail is then packed in the *expr*, so it ends up going from *Location("loc1")* to *Adv(Loc("loc1"))*. Remember that *adv* looks up the location in the nowHeap, and that time has advanced, so that the nowHeap is now the previous *laterHeap*. This means that *Adv* can now look up the delayed computation given by the previous tail and actually evaluate the expression that it finds there. This produces its own delayed computation, which is then again run in the same manner.

Let's say we have the following function. The function *from* produces an infinite stream of the same argument it was given. However, in each recursive step, it increments n by 1. Let's say in this case we call it with an integer 1:

let rec from n = n ::: from (n + 1);

from 1;

In this instance, the *from* function is applied to *integer 1*. The interpreter will first run this simple function with the *eval* function. After that, it receives a value in the form of a stream. It should look like this:

Stream(Int 1, Location("Loc1"))

The head is the present value *Int 1* that can be accessed and used. The tail of the stream has the value *Location* with some string in it denoting the location

of the delayed computation in the laterHeap. It also receives a store. This store has the nowHeap and the laterHeap. Remember that this is the initial run of the computation, and therefore at this stage, the nowHeap should be empty and the laterHeap should hold the delayed computation we receive from the implicit delay in *from*; namely the right side of *:::*, the *from (n+1)* :

```
now_heap:  map []
```

```
later_heap:  map [("loc1", Closure ("loc1", App (Var "constInt"  ,
Prim("+", Var "n", CstI 1), intEnv)]
```

The closure in the laterHeap also carries an environment with it; it requires this to run the delayed computation. We have denoted it as intEnv in the section above, but it looks like this:

```
map
  [("from",
    Closure
      ("from",
        Lambda
          ("n", Cons (Var "n", Delay (App (Var "constInt",
            Prim("+", Var "n", CstI 1))))), map []));
  ("n", Int 1)]
```

The environment inside the closure carries the *from* function and the variable *n*. This environment was saved to the *Closure* as part of the *Delay*.

In our current implementation, we add the head of the stream to a list to merely observe the complete stream at the end of the run time. Time is then advanced by shifting the laterHeap onto the nowHeap, so they end up looking like this:

```
now_heap:  map [("loc1", Closure ("loc1", App (Var "from", Prim("+",
Var "n", CstI 1)), intEnv)]
```

```
later_heap:  map []
```

The value from tail is *Location("Loc1")* and is then constructed into the *expr Loc("Loc1")* and is then packed inside an *Adv*:

```
Adv(Loc("Loc1"))
```

This newly created abstract syntax is then run together with the advanced store (meaning time passed) on the *eval* function again:

```
e = Adv(Loc("Loc1"))
```

```
   s = now_heap:  map [("loc1", Closure ("loc1", App (Var "from", Prim("+",
Var "n", CstI 1)), intEnv)]
later_heap:  map []

                                eval e s
```

Remember that *Adv* uses this *"loc1"* to look into the nowHeap and run that computation. The evaluation of eval will then again produce a *stream* value and a new delayed computation, which will be run again, and so on, eventually producing a long list of the values created by the program:

```
[Int 1; Int 2; Int 3; Int 4; Int 5; Int 6; Int 7; Int 8; Int 9; Int 10; Int 11; ...  ]
```

7.6 Running Str A → Str B

The language can run functions of type *Str A → Str B*. This is done via the *RunR* functions. It just takes an *expr*, which is the abstract syntax tree produced by the parser and type checker. However, this abstract syntax tree has been manipulated slightly. For each *Str A* it expects as input, the entire abstract syntax tree is put inside an *App*. This means, that the AST is treated as a function when put inside the *App*, which it is. The other *expr* for *App* is then *Adv(Loc("-1"))*. The *Loc("-1")* is a location in the heap that will never be hit during string generation for delayed computation and thereby remains "free". The *Loc("-1")* signifies the place on the heap, where we can put user input, l^* as described in rule 30. Here is an overview of how the type affects how the AST is affected:

$$Str\ A = AST$$

$$(Str\ A \rightarrow Str\ A) = App(AST,\ Adv(Loc(\text{“-1”})))$$

$$(Str\ A \rightarrow Str\ A \rightarrow Str\ A) = App(App(AST,\ Adv(Loc(\text{“-1”}))),\ Adv(Loc(\text{“-1”})))$$

Before the function evaluates the new AST, it first asks the user for input through the console. This input is then stored on the nowHeap on the location "-1" in the form of a *Stream* value. The input is the head of the stream, with the tail pointing to the same "-1" via *Location("-1")* like: *Stream(input, Location("-1"))*. It also adds the value *Empty* on the "-1" location, though only on the laterHeap. This decision directly reflects the interpreter rule 30 regarding programs that end in the type *Str A → Str B*.

Similarly to the other function, it runs the initial, now manipulated, abstract syntax tree. This then produces delayed computations, some of which point to *"-1"* on the heap. It then advances time, so the nowHeap becomes the laterHeap and the laterHeap becomes empty; then it runs the eval again with the delayed computations and so on. The major difference here for user experience

is that the program will react and print out, how the user's input has affected the program. In the following example, we see the *sum* function that we have shown before:

```
let rec sum acc (x::xs) = x + acc ::  Delay((sum (x + acc)) Adv(xs));
```

This function initially has the type: $Int \rightarrow (StrInt \rightarrow StrInt)$. However, when initialized with 0 like:

```
sum 0;
```

it suddenly has the type $StrInt \rightarrow StrInt$. Firstly the abstract syntax is changed from AST to this:

App(AST, Adv(Loc("-1")))

This is then run by the *runR* function that handles user input. The user can provide input via the console, and this will change the accumulated value. If we give an input with the int 2, it at first reacts and adds the 2 to the initial 0 value. The store has also been initialized and changed:

```
s = now_heap:  map [("-1", Stream (Int 2, Location "-1"))]
  later_heap:  map [
    ("-1", Empty);
    ("loc1",
      Closure (
        "loc1",
        App (App (Var "sum", Prim ("+", Var "x", Var "acc")), Adv (Var "xs")),
        map [("acc", Int 0);
          ("manualStreamNRBuleVuAIiWlhq1", Stream (Int 2, Location "-1"));
          ("sum", Closure ("sum", ..., map []));
          ("x", Int 2); ("xs", Location "-1")]
      )
    )
  ]
```

The now_heap has the stream, that the user initialized with their input of 2. The later_heap also has the location "-1", as well as the closure for delayed computation of the *sum* function and its current environment, including "acc", the stream we provided, the actual "sum" function and the int "x" that we provided. The "-1" location is then retrieved from the tail of the stream on the

now_heap. It then similarly advances time, so that the nowHeap becomes the later_heap, and leaves the later_heap empty, except for the fact, that it still has a place for the location "-1", but the value is just the value *Empty*.

The *eval* function is then run again with the new advanced store (s) and *Adv(Loc(-1))* as input:

```
eval Adv(Loc(-1)) s
```

This is how it keeps going, consistently reacting to the user's input. Below we show the user input one by one, and what the *sum* function responds with:

User input:	1	2	3	4	5	6	7	8	9
sum response:	*Int 1*	*Int 3*	*Int 6*	*Int 10*	*Int 15*	*Int 21*	*Int 28*	*Int 36*	*Int 45*

8 How it all comes together

As stated at the beginning of section 4, the entire project can be broken into 3 parts: The parser, the type checker, and the interpreter. These parts are packed inside a small function called `rattus`. It is essentially one function to bring them all together: The parser parses the syntax into the abstract syntax. The type-checker then infers the type of the abstract syntax tree and slightly alters it by inserting *Delay* and *Adv* at their appropriate spots. Then depending on the type produced by the typechecker, it runs the interpreter in one of its three modes.

To showcase this, let's follow the `from` example. The user will write this syntax

```
let rec from n= n :::  from (n + 1);

from 1;
```

Our function `rattus` will start running this syntax with the parser, which will translate to the following abstract syntax tree

```
Rec
("from",
 Lambda
   ("n", Cons (Var "n", App (Var "from", Prim ("+", Var "n", CstI 1)))),
 Let ("it", App (Var "from", CstI 1), Var "it")))
```

Then, `rattus` gives this abstract syntax to the type checker, which will add the missing *delay* term and will infer that it is of type *Str Int*. Then, as the type is *Str A* and not *Str A* $\rightarrow$ *Str B*, it runs the interpreter in a mode where it does not require user input, but just runs the program and the following delayed computations 10.000 times. This will result in a list of integers:

```
   [Int 1; Int 2; Int 3; Int 4; Int 5; Int 6; Int 7; Int 8; Int 9; Int 10;
Int 11; Int 12; Int 13; Int 14; Int 15; Int 16; Int 17; Int 18; Int 19;
Int 20; Int 21; Int 22; Int 23; Int 24; Int 25; Int 26; Int 27; Int 28;
Int 29; Int 30; Int 31; Int 32; Int 33; Int 34; Int 35; Int 36; Int 37;
Int 38; Int 39; Int 40; Int 41; Int 42; Int 43; Int 44; Int 45; Int 46;
Int 47; Int 48; Int 49; Int 50; Int 51; Int 52; Int 53; Int 54; Int 55;
Int 56; Int 57; Int 58; Int 59; Int 60; Int 61; Int 62; Int 63; Int 64;
Int 65; Int 66; Int 67; Int 68; Int 69; Int 70; Int 71; Int 72; Int 73;
Int 74; Int 75; Int 76; Int 77; Int 78; Int 79; Int 80; Int 81; Int 82;
Int 83; Int 84; Int 85; Int 86; Int 87; Int 88; Int 89; Int 90; Int 91;
Int 92; Int 93; Int 94; Int 95; Int 96; Int 97; Int 98; Int 99; Int 100;
...]
```

9 Testing and discussion

As stated earlier, some of the main problems when building FRP languages have been: causality, productivity, and implicit space leaks. In this paper, we have implemented a faithful version of Rattus, that reports to have conquered these problems.

9.1 Causality

If we take a look at the *tomorrow* function from section 2.4.1, it at first seems that this breaks causality or that this function would not typecheck.

```
let tomorrow (x::xs)= xs;
```

However, a user is actually able to use this `tomorrow` function while programming. If we look at the following example, we see an example of the *tomorrow* function in use:

```
let rec from n = n ::: from (n + 1); from 1;
let tomorrow (x::xs) = xs;
let fromButStartsWithN n = n ::: tomorrow (from 1);
fromButStartsWithN 13;
```

In this example, we declare the *from* function we have seen many times now. We then declare the *tomorrow* function and then declare a function *fromButStartsWithN*, that takes an integer as the head of the stream and otherwise produces the same Stream as *from 1*.

```
Type:  Str Int

Result:  [Int 13; Int 2; Int 3; Int 4; Int 5; Int 6; Int 7; Int 8; Int 9;...
```

So while you are not able to define present values that depend on future values, you are still able to manipulate and transform streams in meaningful ways. Thus, causality is ensured.

9.2 Productivity

As described earlier, productivity is ensured via the modal type *Delay*. This is implicitly inserted into recursive functions that call themselves at the moment such a function calls itself, effectively transforming regular recursion into guarded recursion. This ensures that any stream is evaluated in finite time, thereby maintaining productivity and preventing non-termination issues associated with coinductive types.

Through the implicit insertion of *Delay* and *Adv*, we have simplified the development process for programmers, by allowing the users to write recursive functions without the need to manually manage these modal types. By automating this aspect within the typechecker, the implementation of guarded recursion has been simplified, ensuring that programs are productive and stable, as well as easy to write.

We have tested our implementation with various different programs seen throughout the entire paper. In all the tested cases, the implicit placement of *Delay* and *Adv* has performed just as intended. The typechecker successfully prevents non-productve recursion patterns, ensuring that only guarded recursive functions are allowed.

This approach not only simplifies the syntax that the user has to write, but it also guarantees the safety and correctness required for functional reactive programming. The transformation of recursive calls to guarded recursion by use of the *Delay* modality ensures that each time step progresses correctly.

9.3 Space leaks

Bahr [1] ensures that Rattus does not have any space leaks if implemented correctly. It does this via an aggressive garbage collection strategy, where old data is discarded and thereby not retained for the next time step. This holds for our implementation as well, and we thereby have no space leaks. Given a common function like *from* that we have seen a couple of times by now:

```
let rec from n = n :::  from (n + 1); from 1;
```

if we run this for 100 million time steps, it uses no more space than had it run once. The size of the heap store stays the same during runtime. In the case of *from*, both the nowHeap and the laterHeap stay at the size of 1 during the entire runtime. This is because the values in the nowHeap are discarded at every timestep and the laterHeap is transferred to the nowHeap just afterward, leaving the laterHeap empty until the end of the next delayed computation.

However, if we were to try to create a few functions that look like they have implicit spaceleaks, for instance, these don't typecheck:

```
let leakyMap f = let rec run (x:::xs) = f x :::  run xs;;

let rec leakyMap2 f (x:::xs) = f x :::  leakyMap2 f xs;
```

The f is typechecked to be a function type, and given the type system, such a type cannot be accessed inside a delay and cannot be referenced in recursive functions. Notably, the following function typechecks:

```
let rec constInt n = n :::  constInt

Type:  (S'c -> Str S'c)
```

But it does so with the "S" marker that marks the type as Stable; meaning the function can only take a stable type and return a stream of stable types. This ensures that you are not able to pass something like a *Stream (str)* into the *constInt* function that would cause a spaceleak.

In conclusion, there are no implicit space leaks in our implementation.

10 Conclusion

In this paper, we covered the theoretical background behind functional reactive programming languages, specifically focusing on modal types and the theory behind Rattus. We then implemented a parser, typechecker, and interpreter, which together form an FRP language based on Rattus. We argue that the language showcases significant advancements in handling data streams and behaviors without falling into common pitfalls like non-causal programs, non-productive recursions, and implicit space leaks. By integrating modal types and the special rules for Rattus with the Hindley-Milner algorithm, we were able to create a powerful typechecker capable of inferring the type of any writable program. Thus, ensuring that the programmer will not accidentally write a program that is either non-causal, non-productive, or has implicit space leaks, as the typecheker rejects these programs. Additionally, by following the strategy for avoiding implicit space leaks on the interpreter, proposed by Bahr, and implementing it in our language, we also prevent the user from keeping old and unnecessary data.

Throughout the paper, we have showcased various programs that typecheck and run, and various that does not typecheck. Notably, in section 3.5 we show off two smaller games that showcase the complexity of the language while maintaining type safety. The typechecker can typecheck and reject various programs; it is notably able to infer which types are *stable types* and describe them via the stable type annotation *"S'c"* as seen with *constInt*, where it can correctly infer the type $S'c \rightarrow Str\ S'c$. It can identify and reject quite complex attempts at creating space leaks, as seen with *leakyMap* in section 2.4.3. Other than this, the language hides a lot of complexity from the programmer with the implicit placing of *Delay* and *Adv*, making the language robust and easy to write.

Another achievement is how the typechecker is able to take programs that would not typecheck, and then insert the introduction and elimination terms

for the ◯ modality, thus removing the burden of handling guarded recursion from the programmer, and reducing the complexity needed to actually write programs in an FRP language. Moreover, our approach exemplifies how you might develop a functional reactive programming language that is more user-friendly, and therefore is a milestone in making FRP languages more accessible to developers. We have effectively abstracted guarded recursion management away from the user, but hope that future work will be able to abstract even more "difficult" concepts away and make FRP languages even more accessible since our implementation still needs some esoteric understanding to write programs effectively.

The true advancement in this field lies in merging the theoretical robustness of Rattus with the practical implementation of an FRP language, demonstrating that Bahr's theoretical framework can be effectively translated into a real-world language, thereby validating Bahr's concepts.

Other than the technical achievements described in the paper, our work contributes to the broader understanding and development of FRP languages. By implementing a concrete example of Rattus, we have offered a pillar of knowledge for further research and development in functional reactive programming. Our implementation serves as a proof of concept that might lead to future enhancements and optimizations in the field of functional reactive programming based on Rattus or alterations of Rattus.

References

[1] Bahr, P. (2022). Modal FRP for all: Functional reactive programming without space leaks in Haskell. *Journal of Functional Programming, 32*, E15. doi:10.1017/S0956796822000132

[2] Severi, P. (2019). A Light Modality for Recursion. Logical Methods in Computer Science Volume 15, Issue 1, 15(1). `https://dblp.uni-trier.de/db/journals/lmcs/lmcs15.html#Severi1`

[3] Elliott, C., & Hudak, P. (1997). Functional reactive animation. Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming - ICFP '97. `https://doi.org/10.1145/258948.258973`

[4] Krishnaswami, N. R. (2013). Higher-order functional reactive programming without spacetime leaks. *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming.* `https://doi.org/10.1145/2500365.2500588`

[5] Bahr, P., Christian Uldal Graulund, & Rasmus Ejlers Møgelberg. (2019). Simply RaTT: a fitch-style modal calculus for reactive programming without space leaks. *Proceedings of the ACM on Programming Languages*, *3*(ICFP), 1–27. `https://doi.org/10.1145/3341713`

[6] Graulund, C. U. (2021). *Type Theories for Reactive Programming*. IT-Universitetet i København. ITU-DS No. 175

[7] Grant, I. (2011). The Hindley-Milner Type Inference Algorithm. https://steshaw.org/hm/hindley-milner.pdf

[8] Damas, L. (1982). Principal type-schemes for functional programs. `https://courses.cs.washington.edu/courses/cse503/10wi/readings/p207-damas.pdf`

[9] Sestoft, P., & Hallenberg, N. (2017). Programming language concepts. Springer.

[10] Tolksdorf, S. (n.d.). FParsec Documentation. FPARSEC documentation. `https://www.quanttec.com/fparsec/2023`

[11] Severi, P. (2019). A LIGHT MODALITY FOR RECURSION. Logical Methods in Computer Science, 15(1), 32. `https://doi.org/10.23638/LMCS-15(1:8)2019`