Li, Kwing Hei

Type Systems for Functional Reactive Programming

Part II Project in Computer Science

Churchill College

2022

Declaration

I, Li Kwing Hei of Churchill College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

I am content for my dissertation to be made available to the students and staff of the University.

Signed 

Date April 11, 2022

Proforma

Candidate Number: 2378F

Project Title: Type Systems for Functional Reactive Programming

Examination: Computer Science Tripos — Part II, 2022

Word Count: 12000[1]

Line Count: 8264[2]

Project Originators: The dissertation author and Alan Mycroft

Project Supervisor: Alan Mycroft

Original Aims of the Project

The original aims of the project were to design, implement, and evaluate a functional reactive programming language *Eva* based on the Lively RaTT calculus from the paper *Diamonds are not forever – Liveness in reactive programming with guarded recursion*. The project should include a parser, a type-checker, and an interpreter. For the evaluation criteria, the dissertation author should consider the soundness, expressiveness, and runtime efficiency of the language.

Work Completed

Exceeded core and implemented various extensions. Eva enriches Lively RaTT's type system in various ways, e.g. it supports parametric polymorphism and generalizes various typing rules while preserving RaTT's correctness guarantees. Eva supports type synonyms and modular programming. The industrial synchronous dataflow language Lustre is implemented as a domain-specific language within Eva via the *Lust4Eva* transformation. The soundness, theoretical power, and usability of Eva are evaluated.

Special Difficulties

None.

[1] Calculated using texcount (from `https://app.uio.no/ifi/texcount`). All code listings included.
[2] Calculated using cloc (from `https://github.com/AlDanial/cloc`).

Without safety, liveness is illusory;
Without liveness, safety is ephemeral.
- Neil D. Jones

Contents

Chapter 1

Introduction

A **reactive program** is one that continuously interacts with the environment, e.g. software for servers, graphical user interfaces, and control software in vehicles. Today, many critical software programs are reactive in nature, so there is significant value in designing expressive programming languages which enable us to implement bug-free and memory-efficient reactive programs that are also easy to reason about. Unfortunately, designing and implementing such a programming language is widely acknowledged to be challenging.

Traditionally, **imperative languages** are used to implement asynchronous reactive programs. They provide a range of complex features like shared states and callbacks. Each of these features on its own is challenging to reason about, and the combination of all these features only makes programs even more error-prone and difficult to reason about.

Synchronous dataflow languages, like Lucid [36] and Lustre [19], implement reactive programs through a network of stream-processing nodes that communicate with each other. At each clock tick, each node produces and consumes a statically-known amount of data, thus providing strong guarantees on time and space usages. These programs are more predictable than their asynchronous counterparts, but their expressiveness is more limited, since the dataflow network is fixed and cannot be dynamically modified.

Functional reactive programming (FRP) languages, first introduced by Elliot and Hudak [17], implement synchronous reactive programs via a high level of abstraction from the functional paradigm. By modelling time-varying values as an infinite list, programs dynamically alter the dataflow network through features like high-order functions and signal-valued signals. It is also easy to conduct equational reasoning on programs. However, many FRP models accept programs that violate expected reactive properties like causality. It is also easy to write programs that incur significant resource leaks as the memory management abstraction is not exposed to the programmer.

This dissertation project concerns the design, implementation, and evaluation of a practical FRP language called ***Eva***, whose type system is an enhanced version of a recent theoretical calculus called **Lively RaTT** [7]. Eva's type system addresses the above limitations by:

1. Certifying that all programs possess certain statically-known guarantees (subsections 2.3.1 and 2.3.2).

2. Ensuring no data resides in the heap for more than one time step during a program's execution (subsection 2.3.3). Programs need to be explicit in their retention of data and, hence, do not exhibit any implicit space leaks.

3. Supporting polymorphism and treating all data as first-class citizens, making Eva expressive enough to implement a wide spectrum of programs (subsection 3.2.3).

4. Providing information on **both** safety and liveness properties of certain reactive programs via their types, a feature not provided by any other programming language (section 2.5).

As a significant proof of concept, I embedded Lustre, an established dataflow programming language for implementing critical control software, as a domain-specific language within Eva via a program transformation process called ***Lust4Eva*** (subsections 3.3.3 and 3.7.4). This allows us to write reactive programs with Lustre's higher abstraction model, while leveraging Eva's type-checker and interpreter.

Multiple aspects of Eva were evaluated, including:

1. Showing Eva type-rejects **all** programs that are non-causal, non-generative, or space-leaky (section 4.1).

2. Demonstrating the computational power of Eva is higher-order primitive recursive in a single time step and Turing-complete over time (subsection 4.2.1), and that the actual time efficiency of Eva's operations is consistent with the theoretical complexities (subsection 4.2.2).

3. Demonstrating how Eva implements various single or multi-step programs, while highlighting how the type of Eva programs provides information regarding its safety, liveness, or even fairness guarantees (section 4.3).

Chapter 2

Preparation

In order to implement a programming language based on RaTT, I needed to understand the type system and calculus in detail. This chapter details the various concepts RaTT is built on, as well as the software engineering aspect of my project.

2.1 Type Systems

A type system is a set of rules defining when a computer program is valid. It assigns a piece of information, called a type, to each inductively-defined expression, and verifies the composition of expressions follows expected rules.

Consider the expression $e_1 + e_2$, that adds two integers. A typing rule might state that this expression is valid if both e_1 and e_2 have the type `Integer`. Moreover, if the expression is valid, the return type of the expression is `Integer`. In a simple type system, this rule might appear in the form:

$$\text{addSimple}\frac{\Gamma \vdash e_1 : \texttt{Integer} \qquad \Gamma \vdash e_2 : \texttt{Integer}}{\Gamma \vdash e_1 + e_2 : \texttt{Integer}}$$

This is known as an inference rule. This can be read as: if all the judgements above are true, then the judgement at the bottom will follow. The Γ symbol is called the context, which stores the type of free variables of the expression concerned.

Sometimes, variables in the context need to be treated differently. Consider the following Java [3] program with one function nested within another:

```
void f(int n){
  Function<int, int> g = (m)->{
    C;
    return n+m;
  }
}
```

The statement `C` can modify `m`, but not `n` in general. One can encode these rules within the context by using delimiters called **tokens**. The context Γ remains a list, but it may contain tokens to describe how variables are treated differently. In RaTT and Eva, such tokens are used to introduce a notion of time for the type-checking algorithm.

A type system is important in a programming language because it enables us to statically capture validity aspects or invariants of programs before running them. Under a sound type system, type-

checked programs incur no errors during run-time. In more sophisticated type systems, types can provide richer information about the semantic properties and guarantees of a program.

2.2 Functional Reactive Programming

Functional Reactive Programming (FRP) is a common programming paradigm for implementing reactive programs via a high-level abstraction provided by functional programming. FRP encodes discrete-time-varying values, called signals, as infinite lazy lists called streams. Elements of the stream denote how the value changes through time, with the n-th element in the list denoting the value n time steps from now. One can then define the relationship between input and output signals via familiar functional programming building blocks, e.g. `map`, `filter`. For example, a program that repeatedly takes a number as input and returns its successor can be implemented as:

```
mapSuc (x::xs) = (x+1)::(mapSuc xs)
```

or simply

```
mapSuc = map (fun n => n+1)
```

Under the functional paradigm, it is easier to reason about pure functions having no hidden side effects in a modular way. Most functional languages also come with a type system, which helps detect errors during compile time.

However, if we directly borrow the same type system from a typical functional language, one can write valid functional programs that do not make sense when we interpret streams as signals.

Consider the following function:

```
badTl (x::xs) = xs
```

A traditional ML-like type system would type-check and accept `badTl`, a function that returns the tail of a stream. However, when we interpret the stream as a signal, `badTl` is invalid as it violates **causality** and **generativity**. Here, causal means that current behavior does not depend on future inputs, and generative means that a value is eventually produced at each time step provided the program has not halted. At time n, `badTl` is unable to produce an output value without knowing the input value for time $n + 1$, thus representing a non-realizable reactive function.

Another problem of FRP languages is that programs might incur memory leaks which are caused by the language's implementation and are not exposed to the programmer via the source code. We call these **implicit space leaks**. (We allow programs to incur explicit space leaks, i.e. memory leaks that are visible from the code, like a finite list explicitly defined to increase in size through time.) In this dissertation, all space leaks are assumed to be implicit, unless otherwise specified.

Consider the following function:

```
badCons (x::xs) = (x::xs) :: (badCons (x::xs))
```

This is a higher-order function that receives a stream and returns a constant stream of streams. Observe that the interpretation of `badCons` incurs a space leak: to produce an element of the output stream at time n, the interpreter needs to retain all values produced by the input stream from time 1 to $n - 1$. We call programs that incur space leaks, like `badCons`, **space-leaky**.

In summary, the naïve ML-like type system is not sophisticated enough to support FRP, since it allows non-causal, non-generative or space-leaky reactive programs. One wants a richer type system that forbids such invalid reactive programs.

2.3 Language Primitives for FRP Properties

Traditional type systems are not rich enough to forbid non-causal, non-generative, or space-leaky FRP programs. Consequently, various theoretical FRP type systems have been proposed to address these limitations. In subsections 2.3.1, 2.3.2, and 2.3.3, I discuss common language primitives which are used to enrich type systems to achieve desired FRP properties.

Recently, Bahr proposed a theoretical calculus with a type system called Lively ReActive Type Theory, or simply **Lively RaTT** [7], which possesses all the desirable language properties above. For this dissertation, I use RaTT to denote Lively RaTT, unless otherwise specified. All typing rules presented for the rest of the chapter are taken from the RaTT calculus.

2.3.1 Causality and Nakano's Fixed-Point Operator

Recall that if we represent a signal as a simple stream, we can implement the non-causal function `badTl` from section 2.2 representing the semantically-impossible program that returns the value it is to be given in the next time step. In a simple type system, the type of a stream satisfies the type isomorphism $\texttt{Str(A)} \cong \texttt{A} \times \texttt{Str(A)}$, where the tail of a stream is isomorphic to the entire stream. Consequently, `badTl` has the type $\texttt{Str(A)} \to \texttt{Str(A)}$, suggesting that the tail of the stream could be used freely, including during the current step.

Various FRP calculi, like Atkey's and McBride's calculus [4], prevent this with a type-based solution proposed by Nakano [28]; A modal operator $\triangleright$ (for both types and expressions) is used to describe data coming in the next time step, which thus cannot be used in earlier steps.

Once we introduce a notion of time on data with $\triangleright$, we can represent a signal as an enhanced stream with a guarded-recursive type constructed by Nakano's fixed-point operator primitive: $\texttt{NFix}\ \alpha.\texttt{A} \times \alpha$.[1] When unfolded, the `NFix type` introduces a $\triangleright$ before the recursion variable, forming the type $\texttt{A} \times \triangleright(\texttt{NFix}\ \alpha.\texttt{A} \times \alpha)$. The type of a stream then instead satisfies the stronger type isomorphism $\texttt{Str(A)} \cong \texttt{A} \times \triangleright\texttt{Str(A)}$. Intuitively, the $\triangleright$ modality labels the tail of a stream to be data that can only be accessed in the future. A simple `tl` function on a stream now has the type $\texttt{Str(A)} \to \triangleright\texttt{Str(A)}$, which although type-checks, prevents us from using the returned tail freely for other operations in this current time step.

Once we introduce the $\triangleright$ modality, we also need `delay` and `adv` constructs to relate computations at different time steps. The former allows us to describe future operations one step into the future, while the latter turns the clock one step back to describe computations in the current step. To encode this change of time through the context, we add the tick-token $\checkmark_\triangleright$ (see section 2.1) to the right of a context when type-checking a `delay` expression, and remove it when type-checking an `adv` expression. Whether we have a tick-token in the context or not determines whether we are type-checking an expression in the next time step or the current one. For RaTT in particular, we can only define operations one time step in the future but not further, i.e. there can be at most one tick-token in the context.

$$\text{delayRule}\frac{\Gamma, \checkmark_\triangleright \vdash t : A}{\Gamma \vdash \texttt{delay}\ t : \triangleright A} \qquad \text{advRule}\frac{\Gamma \vdash t : \triangleright A}{\Gamma, \checkmark_\triangleright, \Gamma' \vdash \texttt{adv}\ t : A}$$

Some calculi, including RaTT, provide Nakano's fixed-point combinator for recursive expressions that span over time, denoted as `nfix`. Unlike a normal fixed-point for defining recursive programs which

[1] For RaTT, the author re-uses the construct `Fix` to represent Nakano's fixed-point operator primitive. I use `NFix` (and `nfix`) for clarity.

has type $(\texttt{A}\rightarrow\texttt{A})\rightarrow\texttt{A}$, this fixed-point construct for expressions has type $(\Box\triangleright\texttt{A}\rightarrow\texttt{A})\rightarrow\Box\texttt{A}$ to ensure that causality is enforced over time. (It uses the modality $\Box$, which is explained below.)

$$\text{nfixRule}\frac{\Gamma, x : \Box \triangleright A, \sharp \vdash t : A}{\Gamma \vdash \texttt{nfix}\ x.t : \Box A}$$

2.3.2 Generativity and Higher-Order Primitive Recursion

In section 2.2, we defined generativity as a value being eventually produced at each time step provided the program has not halted. In other words, the program cannot get stuck in an infinite loop in one time step. Consequently, generative programs cannot allow arbitrary recursive functions, such as the following `badLoop` function that never produces the head of the return stream given a stream of data, as it calls itself repeatedly:

```
badLoop (x::xs) = badLoop (x::xs)
```

One trivial way of achieving generativity is by removing recursion completely. Consider the Simply Typed Lambda Calculus [8], which is strongly-normalizing. If we restrict expressions to be encoded within the Simply Typed Lambda Calculus, we never write programs that are non-generative! Unfortunately, by doing so, we sacrifice the potential to write various useful recursive programs, e.g. factorial.

Many calculi achieve a nice balance between generativity and expressiveness by introducing a total recursion construct for the Peano integers $\texttt{rec}_{\texttt{Nat}}$ that spans over space (meaning that the recursion occurs within one time step). Because $\texttt{rec}_{\texttt{Nat}}$ below treats functions as first-class citizens, it is not only primitive recursive, but **higher-order primitive recursive**. This means that one can implement non-primitive recursive functions like the Ackermann function, or the quicksort function in one time step. In fact, it is known that all recursive functions that are provably total in first-order arithmetic can be implemented with $\texttt{rec}_{\texttt{Nat}}$ [34].

$$\text{rec}_{\text{Nat}}\text{Rule}\frac{\Gamma \vdash s : A \qquad \Gamma, x : \texttt{Nat}, y : A \vdash t : A \qquad \Gamma \vdash n : \texttt{Nat}}{\Gamma \vdash \texttt{rec}_{\texttt{Nat}}(s, x\ y.t, n) : A}$$

In addition to representing `Nat` via the Peano numbers, we can represent the type of integers `Int` as `Nat+Nat`, and the type of rational numbers `Real` as `Int*Nat`.

2.3.3 Eliminating Space Leaks and `Stable` Types

Recall that `badCons` from section 2.2 exhibits space leaks since its execution requires storing every element seen from the input stream. We ideally want this space leak to be explicitly visible in the source code, so programs which incur space leaks not observable from the source code level are type-rejected.

One solution to eliminate space leaks is to use an interpreter with a two-heap store model [14]. At each step, the program accesses values of elements stored in the previous time step from the first heap, while storing values to be used in the next step with the second heap. During a clock tick, all elements in the first heap are garbage-collected and the two heaps are swapped for the next step. To retain the value of an element for more than one step, the program has to explicitly propagate the element from one heap to the other for every step. (This is allowed as this is considered an explicit space leak, where the action of retaining a value is obvious from the program.) This model ensures no element of the

two-heap store is more than one time step-old, thus eliminating all space leaks. The semantics of the `delay` and `adv` constructs below describes how an expression interacts with the two-heap store[2]:

$$\text{delayEval}\frac{l = \texttt{alloc}(\sigma) \qquad \sigma \neq \bullet}{\langle \texttt{delay}\ \ t; \sigma\rangle \Downarrow \langle l; (\sigma, l \mapsto t)\rangle} \qquad \text{advEval}\frac{\langle t; \eta_N\rangle \Downarrow \langle l; \eta_N'\rangle \qquad \langle \eta_N'(l); \eta_N' \checkmark \eta_L\rangle \Downarrow \langle v; \sigma'\rangle}{\langle \texttt{adv}\ \ t; \eta_N \checkmark \eta L\rangle \Downarrow \langle v; \sigma'\rangle}$$

Sometimes, we want to access time-independent data which should theoretically be accessible at any point of a program's execution even if it is not stored in the heap, e.g. the integer `57`, or any data of type $\Box$`A` (where `A` can be any type). These data have the type qualifier `Stable`, whose type are given by:

$$\texttt{Stable}\ \ S, S' ::= 1 \mid \texttt{Nat} \mid S + S' \mid S \times S' \mid \Box A$$

Intuitively, a $\Box$ type can only be constructed without depending on time-dependent data and can be safely accessed at any time step. The constructor and destructor for the $\Box$ type are the `unbox` and `box` constructs, respectively. To type-check an expression of $\Box$ type, we introduce a second token $\sharp$ for the context, which separates time-independent variables on the left from the time-dependent variables on the right. Additionally, $\checkmark_\triangleright$ can only appear on the right of $\sharp$; anything to the right of a $\checkmark_\triangleright$ are data defined in the next time step, and thus must be time-dependent. This means a context with both tokens must be of the form $\Gamma, \sharp, \Gamma', \checkmark_\triangleright, \Gamma''$, where $\Gamma, \Gamma', \Gamma''$ contain no tokens.

$$\text{unboxRule}\frac{\Gamma \vdash t : \Box A}{\Gamma, \sharp, \Gamma' \vdash \texttt{unbox}\ \ t : A} \qquad \text{boxRule}\frac{\Gamma, \sharp \vdash t : A}{\Gamma \vdash \texttt{box}\ \ t : \Box A}$$

I finally present the `var` rule, which describes whether an expression can access a variable in a context. It states that a variable in a context can be accessed if and only if the variable has a `Stable` type, or there is no token to the right of it. This is intuitive, since we expect some data to be accessible if the data itself is time-independent regardless of when it is created, or if it is generated for the use of this time step, i.e. no tokens to its right.

$$\text{varRule}\frac{\texttt{token-free}(\Gamma') \vee A\ \ \texttt{Stable}}{\Gamma, x : A, \Gamma' \vdash x : A}$$

2.4 Safety and Liveness

For a reactive system, we are often interested in two classes of properties: **safety** and **liveness** [2]. Informally, a safety property requires that nothing bad will ever happen, e.g. a system will never crash. On the other hand, liveness states that something good will eventually happen, e.g. the system will eventually terminate. In fact, safety and liveness properties are so fundamental and expressive that all linear-time properties can be expressed as the intersection of safety and liveness properties.

For this dissertation, I partition safety and liveness properties into two finer classes: **static** and **dynamic**. A static property only depends on the program's execution in one time step; a dynamic property is defined over multiple steps.

Recall the properties causality, generativity, and non-space-leaking from section 2.2. Causality is a static safety property: at each time step, current computation steps do not depend on future events. Generativity is a static liveness property: at each time step, we will eventually see a value as the output

[2] The $\checkmark$ token in the advEval rule splits the store into two heaps.

(unless the program halted previously). Non-space-leaking is a static safety property: at each time step, no space leaks not observable from the source code are incurred.

In addition, we are interested in two dynamic properties. **Productivity** is a dynamic safety property stating that the program never terminates, and repeatedly produces a value at every time step. **Termination** is a dynamic liveness property stating that the program will eventually halt at some time step in the future. Table 2.1 summarizes various properties based on their characteristics.

	Safety	**Liveness**
Static	Causality, Non-space-leaking	Generativity
Dynamic	Productivity	Termination

Table 2.1: Program properties

In FRP, a productive program is represented by a stream, while a terminating one is represented by a finite list that spans over time[3].

There are various FRP calculi that support either productive or terminating programs via their type systems, but not both. This is because infinite and finite lists spanning over time are built differently: the former is a co-inductive type with no base case, and the latter is an inductive one. Naïvely equating the two types breaks the soundness of the type system, see section 2.5.

2.5 Combining Safety and Liveness in RaTT

Above, we identified various limitations of existing FRP languages. We saw language constructs and typing rules that enable us to type-reject non-generative, non-causal, or space-leaky programs at compile-time, while maintaining a high level of expressiveness. However, these features are not unique to RaTT. Nakano's fixed-point operator led to a flurry of proposals on guarded recursion, like Clouston et al.'s Guarded λ-calculus [15]. The total recursion scheme for `Nat` can first be found in Gödel's System T calculus [18]. RaTT's method of eliminating space leaks is highly inspired by Krishnaswami's AdjS calculus [23].

We now turn to RaTT's unique feature: the ability to support both productive and terminating programs in a single system. Its typing rules enable us to encode both streams and finite lists that span over time in one language, allowing us to assert dynamic safety and liveness properties via the type of the programs.

As emphasized in section 2.4, from a reactive-system perspective, implementing programs with dynamic safety properties like productivity is insufficient. We also want to implement ones with dynamic liveness properties, e.g. terminating programs. By realizing Linear Temporal Logic (LTL) as a type system [33], there were attempts to encode reactive programs with dynamic liveness properties, like Jeffrey's calculus [21] and Cave et al.'s calculus [12]. Similar to $\triangleright$, LTL has its own modality, $\bigcirc$, for representing data coming one time step from now. Like the tick-token $\checkmark_{\triangleright}$ for $\triangleright$, we also introduce another tick-token $\checkmark_{\bigcirc}$ within contexts Γ for the $\bigcirc$ modality. The constructs `delay` and `adv` in RaTT operate on $\bigcirc$ in the same way as on $\triangleright$:

$$\text{delayRule}\frac{\Gamma, \checkmark_{\mathtt{m}} \vdash t : A}{\Gamma \vdash \mathtt{delay}\ t : \mathtt{m}\ A} \qquad \text{advRule}\frac{\Gamma \vdash t : \mathtt{m}\ A \qquad \mathtt{m} \leq \mathtt{m'} \vee A\ \mathtt{Limit}}{\Gamma, \checkmark_{\mathtt{m}'}, \Gamma' \vdash \mathtt{adv}\ t : A}$$

RaTT also introduces the until operator $\mathcal{U}$ from LTL, to represent finite lists that span over time. This

[3] We distinguish two kinds of lists: those that span over space (existing for one time step) and those that span over time. Under this distinction, a stream is equivalent to an infinite list that spans over time.

inductive type comes together with two constructors, `now` and `wait`.

$$\text{nowRule}\frac{\Gamma \vdash t : B}{\Gamma \vdash \texttt{now}\ t : A\mathcal{U}B} \qquad \text{waitRule}\frac{\Gamma \vdash s : A \qquad \Gamma \vdash t : \bigcirc(A\mathcal{U}B)}{\Gamma \vdash \texttt{wait}\ s\ t : A\mathcal{U}B}$$

A $\mathcal{U}$ list is similar to a normal finite list from functional programming except for two aspects: Firstly, the $\mathcal{U}$ list spans over time instead of space, and secondly, the `now` construct (similar to `nil`) takes in an element of type B as an argument. Similar to $\texttt{rec}_{\texttt{Nat}}$, $\mathcal{U}$ types come with their own recursive construct, though it is one that unfolds over time, rather than within a time step.

$$\text{rec}_{\text{Until}}\text{Rule}\frac{\Gamma, \sharp, x : B \vdash s : C \qquad \Gamma, \sharp, x : A, y : \bigcirc(A\mathcal{U}B), z : \bigcirc C \vdash t : C \qquad \Gamma, \sharp, \Gamma' \vdash u : A\mathcal{U}B}{\Gamma, \sharp, \Gamma' \vdash \texttt{rec}_{\mathcal{U}}(x.s, x\ y\ z.t, u) : C}$$

We now have two similar modalities in RaTT for describing data coming in the next time step: $\triangleright$ and $\bigcirc$. Ideally, we want to be able to write programs that interact with both modalities safely, e.g. a program that takes a stream as input and returns a $\mathcal{U}$ list containing the first 57 elements of the stream.

One might naïvely believe we can directly equate $\triangleright$ and $\bigcirc$ in one single type system to support both productive and terminating programs. Unfortunately, this is too good to be true. Consider the following RaTT function which has type $1\mathcal{U}\texttt{A}$:

```
wrongTerminate =
  nfix finiteList. wait () (unbox finiteList)
```

We expect this to return a finite `Until` list, with the last element being an element of type `A`. Unfortunately, the co-inductive property of Nakano's fixed-point breaks the inductive property of $\mathcal{U}$, causing `wrongTerminate` to return an infinite stream, and violating the termination guarantee we seek to achieve.

RaTT establishes a marriage between productive and terminating programs in one type system by considering the $\bigcirc$ to be a submodality of $\triangleright$, i.e. $\bigcirc \leq \triangleright$, instead of directly equating them (see the `m ≤ m'` precondition in advRule). Intuitively, one can coerce any element of type $\bigcirc$`A` to type $\triangleright$`A` freely, but not the other round.[4]

Under this submodality rule, to type-check `wrongTerminate`, one would have to coerce `(unbox wrongTerminate)` from type $\bigcirc$`(1`$\mathcal{U}$`A)` to type $\triangleright$`(1`$\mathcal{U}$`A)`, which is invalid in RaTT, thus `wrongTerminate` is type-rejected by RaTT as intended.

Though built with different constructs, a stream and an $\mathcal{U}$ list are both valid types of reactive programs in RaTT. Because of this, the operational semantics of RaTT is divided into two parts: the **evaluation semantics** which dictates the computational behavior of a RaTT program at each time step, and the **step semantics** which captures its dynamic behavior between one time step and the next. The evaluation semantics for a stream and an $\mathcal{U}$ list are the same, as how we evaluate an element in a stream is identical to how we evaluate one in an $\mathcal{U}$ list. Their step semantics however differ, since getting the second element of a stream is not the same as getting the second element of an $\mathcal{U}$ list.

Because of this complexity, RaTT needs multiple interpreters. It provides six reactive interpreters, one for each type that represents a valid reactive program. These interpreters have the same evaluation semantics, but their step semantics differ from one another. After type-checking a program, RaTT attempts to unify its type with one of the interpreter's corresponding types. If one matches, the program will be interpreted by the corresponding interpreter. Otherwise, the program is treated as

[4] We can coerce an element of type $\triangleright$`A` to $\bigcirc$`A` in RaTT if `A` is a `Limit` type.

single-step and is only evaluated into a value via the evaluation semantics, and terminates eventually in one time step.

The six interpreters are equally split into three classes. The first two classes are for interpreting streams and $\mathcal{U}$ lists, respectively. The last class is for interpreting `Fair` streams, which have types defined as:

$$\texttt{Fair(A,B)} = \texttt{NFix}\ \alpha.\texttt{A}\mathcal{U}(\texttt{B}\times \triangleright(\texttt{B}\mathcal{U}(\texttt{A}\times\alpha)))$$

Interpreting a program of type `Fair(A,B)` produces a `Fair` stream where data of types `A` and `B` each appears infinitely often. This neatly highlights the strength of RaTT, since fairness is a combination of dynamic safety and dynamic liveness properties. (Note that the term "fairness" has various definitions in different contexts [24].)

For each class, the first interpreter evaluates pure programs and returns the corresponding productive stream, terminating $\mathcal{U}$-list, or `Fair` stream. The second interpreter is an interactive and impure variant of the first, i.e. it repeatedly interacts with the environment/user's input and returns the corresponding reactive datatype. (This is similar to that in Haskell: if the program is an `Integer`, the result is just printed. If the program is an `IO` monad type, it might accept user input which affects future computation.)

2.6 Starting Point

I had no prior experience with implementing parsers, type-checkers, interpreters or functional programming languages (except for a bit of OCaml in the Foundations of Computer Science course). I studied Semantics of Programming Languages, Compiler Construction, and Computation Theory, which were relevant to this project. I read relevant papers on RaTT-like calculi during the summer though no project code was written before the start of Michaelmas term.

2.7 Requirements Analysis and Code Licensing

This project should consist of three components: a parser for converting the source code of Eva into an abstract syntax tree (AST), a type-checker that type-checks the AST according to the rules of the (possibly enhanced) RaTT's type system, and an interpreter that executes the type-checked program using a two-heap store model. I opted to use the spiral development model.

My choice of programming language for implementing my project is Haskell [27]. I always wanted to master Haskell, and this project provided the perfect opportunity to explore it. In addition, Haskell is a statically typed, purely functional programming language, and it guided the design of Eva's syntax, which is also statically typed and functional.

Throughout the project implementation, I used Git [13] for version control, and frequently backed up my repository on GitHub.

This project has two dependencies, the Parsec library for implementing the parser, and the Clock library for benchmarking. The libraries are released under a BSD-2 and BSD-3 license, respectively, so I am not required to include their licenses when distributing the source code of my project since none of their source or binary is included. However, if I were to distribute my project as a binary program, I would need to include their licenses in addition to my BSD-3 license.

Chapter 3

Implementation

Despite its wonderful properties, RaTT was previously only a theoretical calculus, and its potential for implementing sound and expressive reactive programs was not fully realized. Using RaTT as the underlying calculus, I designed and implemented **Eva**, a practical FRP language. In this chapter, I describe Eva's language design choices, and the implementation of various components in Eva's toolchain, including but not limited to the parser, program analyzer, and interpreters.

3.1 Designing the Syntax

RaTT's syntax is inappropriate for a programming language for various reasons. Firstly, RaTT's syntax involves mathematical symbols that are not typable using a standard keyboard. Moreover, the language constructs of RaTT do not lend itself easily to a syntax-directed type-checking algorithm. Therefore, I redesigned the syntax for Eva, taking inspiration from Haskell, OCaml, and Python. Table 3.1 highlights some syntax differences between Eva and RaTT. Like in Haskell, some symbols are used for both expressions and types.

Eva	**RaTT**
`fun`	λ
`>`	`delay` or $\triangleright$
`@`	`delay` or $\bigcirc$
`<`	`adv`
`#`	`box` or $\Box$
`?`	`unbox`
`Until`	$\mathcal{U}$
`primrec`	$\texttt{rec}_{\texttt{Nat}}$
`urec`	$\texttt{rec}_{\mathcal{U}}$
`:::`	Syntactic sugar for prepend on streams

Table 3.1: Syntax differences

3.2 Designing the Type System

In addition to the syntax, I improved RaTT's type system to provide a better programming experience with Eva. Here, I present various design choices to highlight how Eva enhances RaTT's type system.

3.2.1 Representing Nat and Arithmetic Operations

In RaTT, `Nat` types are inductively defined in a Peano-number fashion, i.e. via the `0` and `suc` constructors. This simplifies its proofs of correctness, but is inconvenient for programming in the real world, since simple operations on `Nat` like multiplication must be built from the ground up via `primrec`. This also presents huge time efficiency issues, as the time complexity of various arithmetic operations are polynomial in the size of the operands in Peano representation, instead of logarithmic in binary representation.

Eva overcomes these problems by optionally representing `Nat` types with Haskell's `Integer` types in the abstract syntax tree (AST). It also supports common operators for `Nat` found in most programming languages, e.g. addition +, exponentiation ^. The user then need not define common arithmetic operations from scratch, and since these operators are executed with Haskell's default operators on `Integer`, they are of complexity $O(1)$, allowing almost all programs to be executed without significant slow-down. A user can still force the interpreter to use the Peano representation for integers through options, see section 3.9.

However, since RaTT does not support negative numbers, operators like minus - have to be interpreted differently to reflect how they are expected to be implemented with `primrec`. In Eva, `(x-y)` is equivalent to taking the maximum of `0`, and the difference of `x` and `y`. Similarly, I rewrote the semantics of operations like mod `%` and divide `/` to account for edge cases like taking mod of `0`. Below are the typing rule and evaluation semantics for `%` as an example:

$$\text{modRule}\ \frac{\Gamma \vdash m : \texttt{Nat} \qquad \Gamma \vdash n : \texttt{Nat}}{\Gamma \vdash m \texttt{\%} n : \texttt{Nat}}$$

$$\text{modEval}\ \frac{\langle m;\sigma\rangle \Downarrow \langle m';\sigma'\rangle \qquad \langle n;\sigma'\rangle \Downarrow \langle n';\sigma''\rangle}{\langle m \texttt{\%} n;\sigma\rangle \Downarrow \langle v;\sigma''\rangle \qquad \texttt{where}\ v = m'\ \texttt{mod}\ (\texttt{max}(1,n'))}$$

3.2.2 `Bool` and `List`

In addition to the two basic datatype primitives of RaTT: `Unit`[1] and `Nat`, Eva also supports `Bool` for booleans, and `List` for constructing finite lists that span over space.

The constructors for `Bool` type are `true` and `false`, and its destructor is the `if...then...else` construct. I also introduced the non-short-circuiting boolean operators: `and`, `or`, and `not`. As an example, the rules for the construct `and` are as follows:

$$\text{andRule}\ \frac{\Gamma \vdash b : \texttt{Bool} \qquad \Gamma \vdash c : \texttt{Bool}}{\Gamma \vdash b\ \texttt{and}\ c : \texttt{Bool}}$$

$$\text{andEval}\ \frac{\langle b;\sigma\rangle \Downarrow \langle b';\sigma'\rangle \qquad \langle c;\sigma'\rangle \Downarrow \langle c';\sigma''\rangle}{\langle b\ \texttt{and}\ c;\sigma\rangle \Downarrow \langle v;\sigma''\rangle \qquad \texttt{where}\ v = (b'\ \texttt{\&\&}\ c')}$$

The `List` type has base constructors `[]` representing the empty list, and cons `::` for prepending an element in front.[2] One can destruct a `List` with the `primrec` construct. Just like `Nat` is represented by Haskell's `Integer`, `List` is represented directly by Haskell's list types. As an example, I present `primrec`'s rules for `List` types:

$$\text{primrecListRule}\ \frac{\Gamma \vdash xs : \texttt{List}(A) \qquad \Gamma \vdash s : B \qquad \Gamma, x : A, xs' : \texttt{List}(A), r : B \vdash t : B}{\Gamma \vdash \texttt{primrec}\ xs\ \texttt{with}\ |\ [] \Rightarrow s\ |\ x :: xs', r \Rightarrow t : B}$$

[1] The type `Unit` is written as `1` in RaTT, which is not adopted by Eva for obvious reasons.

[2] Eva also supports append ++.

$$\text{primrecListEval1}\ \frac{\langle xs;\sigma\rangle \Downarrow \langle [];\sigma'\rangle \qquad \langle x;\sigma'\rangle \Downarrow \langle v;\sigma''\rangle}{\langle \texttt{primrec}\ xs\ \texttt{with}\ \mid [] \Rightarrow s \mid x :: xs', r \Rightarrow t;\sigma\rangle \Downarrow \langle v;\sigma''\rangle}$$

$$\text{primrecListEval2}\ \frac{\begin{array}{c}\langle xs;\sigma\rangle \Downarrow \langle v :: vs;\sigma'\rangle \\ \langle \texttt{primrec}\ vs\ \texttt{with}\ \mid [] \Rightarrow s \mid x :: xs', r \Rightarrow t;\sigma'\rangle \Downarrow \langle r';\sigma''\rangle \\ \langle t[v/x, vs/xs', r'/r]\rangle \Downarrow \langle w;\sigma'''\rangle\end{array}}{\langle \texttt{primrec}\ xs\ \texttt{with}\ \mid [] \Rightarrow s \mid x :: xs', r \Rightarrow t;\sigma\rangle \Downarrow \langle w;\sigma'''\rangle}$$

Moreover, I introduced the equals == and not-equals != operators for certain types that admit comparison. Inspired by Haskell's type classes, I defined the type qualifier `Comparable` specifying the types that are comparable. This qualifier, together with qualifiers `Stable` and `Limit` from RaTT, are used specifically for defining type parameters of polymorphic functions, see subsection 3.2.3.

$$\texttt{Comparable}\ C, C' :== \texttt{Unit} \mid \texttt{Nat} \mid \texttt{Bool} \mid \texttt{List}(C) \mid C\texttt{+}C' \mid C\texttt{*}C'$$

3.2.3 Parametric Polymorphism

RaTT's calculus is monomorphic. This means functions cannot be written generically, and functions that intuitively should work on multiple data-types must be redefined for each specific type, making it inconvenient for coding. As an example, to implement an identity function `id` for both `Unit` and `Nat`, one must explicitly define two separate functions:

```
def idUnit x:Unit = x

def idNat x:Nat = x
```

Eva enhances RaTT to allow explicit parametric polymorphism in a System F manner [11]. The universal quantification of the type must be bound to the outermost level of a function definition, i.e. when using the `def` construct. For example, to define the generic `id` function, the user specifies the type variables to be universally quantified in curly brackets:

```
def id{a} x:a = x
```

To use `id` for a certain type, say for `Unit`, the user just provides the type in the curly brackets similarly, e.g. `id{Unit}`.

This addition of parametric polymorphism is not neccessary, for one can produce a specialized version of a polymorphic function for each type the function is used on, similar to the monomorphisation translation in the optimizing compiler MLton [37]. Thus, type-checking the polymorphic Eva calculus is reduced to type-checking the monomorphic RaTT one, which is proven to be sound.

Since there are some constructs that can only be performed on expressions with certain type qualifiers, the user has to provide the required type qualifiers of certain type parameters, e.g. the following polymorphic `compareTrue` function only accepts arguments which are comparable (the function does not type-check if the type qualifier `Comparable` is removed):

```
def compareTrue{Comparable a} x:a = (x==x)
```

To type-check polymorphic functions, I introduce a new context Θ, which maintains a list of polymorphic type variables and their type qualifiers, together with Γ for Eva's type-checking judgements.

3.2.4 Generalizing RaTT's Typing Rules

Two generalisations were made to the type system of RaTT and incorporated into Eva (these adjustments were confirmed to be semantically valid by Bahr, the author of RaTT).

The first concerns the addition of the `let...in...` construct, that enables users to bind local variables. This construct increases readability of long Eva programs and makes the programming experience easier. However, adding it is not trivial. A typical computer scientist might propose desugaring the construct as such:

$$\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \qquad \Longrightarrow \qquad (\texttt{fun}\ x : A \texttt{=>}\ e_2)\ e_1 \qquad \text{(with } A \text{ being the type of } e_1\text{)}$$

Though this proposal is valid, it is too restrictive. This is because RaTT's typing rule for creating anonymous functions can only be done under a tick-free context, preventing us from using the `let` construct for computations in the next time step, as illustrated below:

$$\frac{\Theta; \Gamma, x : A \vdash e : B \qquad \text{tick-free}(\Gamma) \qquad \Theta \vdash A\ \text{type}}{\Theta; \Gamma \vdash \texttt{fun}\ x : A \texttt{=>}\ e : B}$$

RaTT showed that this restriction is necessary for `fun` to prevent the function body retrieving delayed-computation that has been garbage collected.[3]

However, this problem is not present for the `let...in...` construct as the lambda abstraction of `let...in...` is immediately applied in the same time step. It turns out that `let...in...` can use a more relaxed typing rule:

$$\frac{\Theta; \Gamma \vdash e : A \qquad \Theta; \Gamma, x : A \vdash e' : B}{\Theta; \Gamma \vdash \texttt{let}\ x = e\ \texttt{in}\ e' : B}$$

Another interesting change to RaTT's type system is observed in the typing rules of `?` and `urec`. The unmodified version of the two rules share a common characteristic: the preconditions require a stronger context than the conclusion:

$$\frac{\Theta; \Gamma \vdash e : \#A}{\Theta; \Gamma, \#, \Gamma' \vdash \texttt{?}e : A}$$

$$\frac{\begin{array}{c}\Theta; \Gamma, \#, \Gamma' \vdash e : A\ \texttt{Until}\ B \\ \Theta; \Gamma, \#, x : B \vdash e_1 : C \\ \Theta; \Gamma, \#, x' : A, y : \texttt{@}(A\ \texttt{Until}\ B), z : \texttt{@}C \vdash e_2 : C\end{array}}{\Theta; \Gamma, \#, \Gamma' \vdash \texttt{urec}\ e\ \texttt{with|now}\ x \texttt{=>}\ e_1 \texttt{|wait}\ x'\ y, z \texttt{=>}\ e_2 : C}$$

After implementing the basic type-checker and experimenting with Eva programs, I realized these two rules were too restrictive, because `Stable` elements in Γ' are removed when type-checking the preconditions. Since `Stable` elements are time-independent and free from interference of the two-heap store, expressions in the preconditions should still be able to access those elements safely. After discussing with Bahr, he confirmed that his proofs allow the generalization of the above two rules, allowing us to retain `Stable` type elements in the context when type-checking the preconditions of both rules:

$$\frac{\Theta; \Gamma, \text{token-less-stable}(\Gamma') \vdash e : \#A}{\Theta; \Gamma, \#, \Gamma' \vdash \texttt{?}e : A}$$

[3] See the `leaky` function from the Simply RaTT paper [6] for more details.

$$\frac{\begin{array}{c}\Theta;\Gamma,\#,\Gamma' \vdash e : A\ \texttt{Until}\ B \\ \Theta;\Gamma,\#,\text{token-less-stable}(\Gamma'),x : B \vdash e_1 : C \\ \Theta;\Gamma,\#,\text{token-less-stable}(\Gamma'),x' : A, y : @(A\ \texttt{Until}\ B), z : @C \vdash e_2 : C\end{array}}{\Theta;\Gamma,\#,\Gamma' \vdash \texttt{urec}\ e\ \texttt{with|now}\ x \texttt{=>}\ e_1\ \texttt{|wait}\ x'\ y, z \texttt{=>}\ e_2 : C}$$

3.3 Eva's Features

In addition to having a rich type system, Eva provides various useful programming features: support for type synonyms, support for modular programming, and hosting Lustre as an embedded domain-specific language. I discuss these features below.

3.3.1 Type Synonyms

To make the type-checking process syntax-guided, type ascriptions are needed to annotate possibly type-ambiguous expressions. For example, one has to specify the intended sum type of an `inl` expression to remove ambiguity:

```
inl 57:Nat+(Unit*Bool)
```

However, as the types used become increasingly complex in more sophisticated programs, writing the types in plain form becomes cumbersome. Eva allows users to define transparent polymorphic type synonyms, so they can use specialized type synonyms when providing type ascriptions. Type parameters for type synonyms are provided in round brackets. As an example, the template type of a `Fair` stream can first be defined as a type synonym:

```
type Fair(a,b) =
  NFix x --> a Until (b * >(b Until (a*x)))
```

The type ascription of a `Fair` stream that produces `Bool` and `List(Unit)` can then be written as:

```
Fair(Bool,List(Unit))
```

3.3.2 Modular Programming

Eva supports modular programming, allowing the user to separate large programs into multiple modules, each containing functions and type synonyms for one aspect of the software project. Modules can thus be reused which improves manageability and readability of programs. This is shown to be very useful in my evaluation when various common functions for `Nat`, `List`, and streams are implemented in a few modules, and re-used frequently when building complex programs.

Eva's module importer algorithm automatically imports modules forming a directed acyclic graph in a topological sort manner and detects circular dependencies, a phenomenon that might break the generativity of Eva programs, see subsection 4.1.2. To prevent name-clashing of imported functions and type synonyms from different modules, a user can provide an optional alias for each module, similar to how it is done in Haskell and Python [35]:

```
import Prelude.List as PL
```

3.3.3 Lustre as a Domain-Specific Language

I implemented Lustre, an industrial synchronous dataflow language, as an embedded domain-specific language within Eva, meaning one can define Eva programs written in Lustre syntax.

After parsing Lustre code, Eva desugars the Lustre AST into an Eva one. Because Lustre and Eva fall under fundamentally different language paradigms (dataflow and FRP, respectively), this transformation involves much complex analysis on the Lustre AST. I call this transformation **Lust4Eva**, which is elaborated in subsection 3.7.4.

Implementing Lustre as a domain-specific language allows the user to write productive programs more easily with the higher level of abstraction provided by Lustre, while achieving correctness guarantees provided by Eva's type checker. As an example, the function `runningTotal` which outputs the running total of an input stream can be implemented in Lustre syntax as follows:

```
{-
node runningTotal (input:int) returns (output:int);
let
  output = input -> input+pre(output);
tel
-}
```

3.4 Eva's Abstract Syntax for Expressions

I now present a subset of Eva's abstract syntax (specifically for representing expressions). The full abstract syntax rules can be found in Appendix section A.1.

$$
\begin{aligned}
e ::= {} & x \mid n \mid () \mid \texttt{fun}\ x : t \texttt{ => } e \mid e\ e \mid \texttt{nfix}\ x : t \texttt{ => } e \mid \texttt{let}\ x = e\ \texttt{in}\ e \\
& \mid \texttt{true} \mid \texttt{false} \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
& \mid o_1\ e \mid e\ o_2\ e \mid (e,e) \mid \texttt{let}\ (x,x) = e\ \texttt{in}\ e \\
& \mid \texttt{inl}\ e : t \mid \texttt{inr}\ e : t \mid \texttt{match}\ e\ \texttt{with}\ \texttt{|inl}\ x \texttt{ => } e\ \texttt{|inr}\ x \texttt{ => } e \\
& \mid [] : t \mid [e, ..., e] \\
& \mid \texttt{let}\ x\texttt{::}x = e\ \texttt{in}\ e \\
& \mid \texttt{primrec}\ e\ \texttt{with}\ \texttt{|0 => }e\ \texttt{|suc}\ x,\ x \texttt{ => } e \\
& \mid \texttt{primrec}\ e\ \texttt{with}\ \texttt{|[] => }e\ \texttt{|}x\texttt{::}x,\ x \texttt{ => } e \\
& \mid \texttt{now}\ e : t \mid \texttt{wait}\ e\ e \\
& \mid \texttt{urec}\ e\ \texttt{with}\ \texttt{|now}\ x \texttt{ => } e\ \texttt{|wait}\ x\ x,\ x \texttt{ => } e \\
& \mid \texttt{into}\ e : t \\
o_1 ::= {} & \texttt{not} \mid \texttt{fst} \mid \texttt{snd} \mid \texttt{>} \mid \texttt{@} \mid \texttt{<} \mid \texttt{\#} \mid \texttt{?} \mid \texttt{out} \mid \texttt{suc} \\
o_2 ::= {} & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{\^{}} \mid \texttt{and} \mid \texttt{or} \mid \texttt{==} \mid \texttt{!=} \mid \texttt{::} \mid \texttt{++} \mid \texttt{:::}
\end{aligned}
$$

3.5 Repository Overview

Eva's toolchain is written in Haskell, and built by Stack, a cross-platform program for developing Haskell projects. The Eva repository is split into three main directories. The `app/` directory contains `Main.hs`, the main entry point of the entire Eva toolchain. The `src/` directory contains various

modules, grouped into libraries corresponding to each stage of the toolchain. The `example/` directory contains various Eva code I wrote for evaluation, see chapter 4.

Table 3.2 presents an overview of the various directories and files of the Eva repository. Figure 3.1 presents an overview of Eva's entire toolchain.

Folder/File	**Description**	**Number of Files**	**Lines of Code**
`app/Main.hs`	Main entry point of the Eva toolchain	n/a	39
`src/Datatype.hs`	Datatypes used by the Eva toolchain	n/a	245
`src/MainFunctions/`	Functions to detect potential execution options	1	69
`src/StringFunctions/`	Functions to remove comments from source code before parsing	1	19
`src/Parser/`	Parsing code	5	1759
`src/ProgramAnalyzer/`	Analyzes parsed AST and converts it to a simpler AST for the interpreter	5	459
`src/ExpTypeConverters/`	Code to resolve type synonyms in ascriptions	5	230
`src/TypeChecker/`	Eva's type-checking algorithm	2	703
`src/TypeFunctions/`	Functions for analyzing types, e.g. determining whether a type is `Stable`	7	140
`src/Interpreter/`	Code for the various interpreters	10	711
`src/ExpFunctions/`	Common functions used by interpreters, e.g. substitution	1	50
`src/PrintFunctions/`	Code to display the type-checker and interpreter's output onto the terminal	3	600
`example/`	Eva code for evaluation	42	3240
Total		82	8264

Table 3.2: Repository overview

3.6 Parser

Eva's parser is implemented using the Haskell library Parsec [26]. Parsec is a monadic parser combinator library, enabling the programmer to build complex parsers from small ones with monadic operations and Haskell's monad syntactic sugar.

One tricky issue with Parsec is that general left-recursive grammars cause parsers to loop forever. This might seem like a problem since Eva's grammar contains various operations which are left recursive, e.g. +. One might propose rewriting the grammar to remove left-recursion, and add semantic actions in the parser to translate the parsed AST back into the left-recursive one, but this is quite cumbersome. Luckily, one can use the `chainl` or `chainl1` combinators from the Parsec library to automatically parse left-recursive grammars.

The `src/Parser/` folder contains five modules: `ExpParser.hs` for parsing expressions, `TypeParser.hs` for parsing type ascriptions, `VarParser.hs` for parsing variable names, `LustreParser.hs` for parsing Lustre code, and `MainParser.hs` which utilizes all the above parsers to form the function

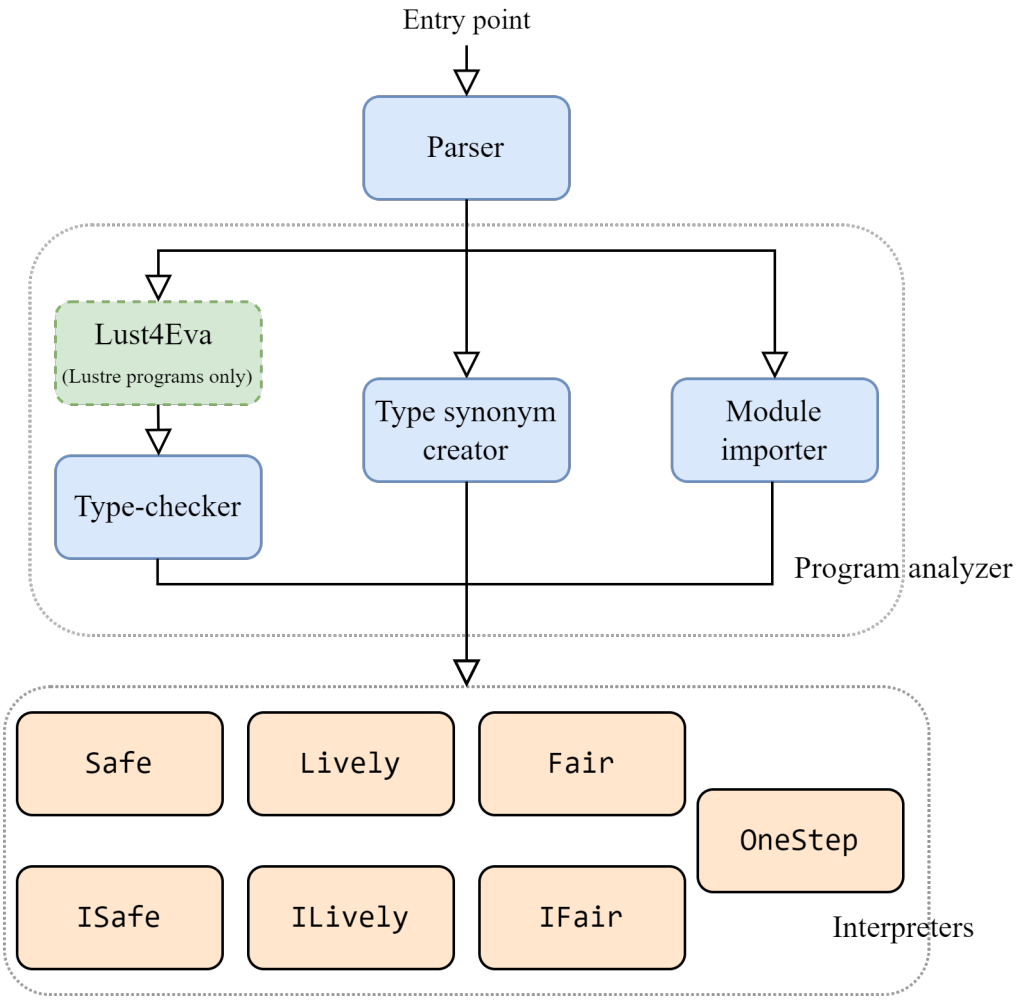

Figure 3.1: Eva's toolchain

`mainParser`. This function takes in two strings, the file name and the source code, and returns a list of ASTs of type `AExp` representing the entire program.

3.7 Program Analyzer

After the parser produces a list of ASTs (each representing a top-level phrase), the program analyzer goes through each tree and performs the corresponding analysis actions. There are three basic types of ASTs, each produced by a different kind of top-level phrase: `def`, `type`, and `import`, respectively. Each of these ASTs is processed by a different component of the program analyzer, see subsections 3.7.1, 3.7.2, and 3.7.3. Subsection 3.7.4 describes Lust4Eva, the desugarer for the Lustre domain-specific language.

3.7.1 Type-Checker

The type-checker is the heart of the Eva language. Responsible for type-checking functions defined by the `def` keyword, it passes the AST of datatype `AExp` through two stages: inlining type synonyms in ascriptions, and type-checking the overall expression.

In the first stage, the type-checker rewrites all type synonyms found in ascriptions anywhere in the expression into the full definition with the function `abExpConverter`, producing an AST of type

`BExp`. The removal of type synonyms simplifies the type-checking algorithm, as we only need to process ascriptions written in plain RaTT syntax. The type-checker also verifies that the name of the function does not clash with previously-defined functions' names, including those imported from other modules.

In the second stage, the type-checker passes the simplified `BExp` AST through the `mainTypeChecker` function, which then type-checks that the expression is valid according to Eva's typing rules.[4] In addition to the AST, the filename and function name is also passed to the function, so if the type-check process fails, a helpful error message, including the filename, function name, the source of the error, and the reason for the error, is displayed back to the user on the terminal via the `typeCheckerErrorMsg` function.

During the type-checking process, an expression might reference another function defined previously in the same file, or in an imported module. In this case, the type-checker verifies that the referenced function's type follows typing rules (and that the passed type arguments for specializing polymorphic functions match the intended type qualifiers), and if successful, inlines the AST of the referenced function into the calling one.

The `mainTypeChecker` function returns two values if both stages are successful: a new AST of the program of type `CExp`, specifically designed for interpreters, and the type of the program when type-checked under the empty context. All `CExp` ASTs contain De Bruijn indices instead of bound variables, and all referenced functions are inlined. Additionally, they do not contain type ascriptions since type ascriptions do not affect semantics rules.

3.7.2 Type Synonym Creator

When a `type` AST is encountered as a top-level phrase, the type synonym creator defines the type synonym for the rest of the program after passing the type synonym definition through two stages.

Firstly, it verifies that the name of the synonym does not clash with those of previously defined synonyms or synonyms imported from other modules. Then, like how the type-checker unfolds type synonyms in ascriptions, the type synonym creator inlines the definition of referenced type synonyms into the definition of the current one.

The type synonym creator also checks that the definition is valid, and has no free type variables. For example, the following type synonym `invalidSynonym` is invalid as the type variable `d` is free in the definition:

```
type invalidSynonym(a,b) = NFix c --> d
```

3.7.3 Module Importer

When an `import` AST is encountered, the program analyzer runs the module importer to import all the functions and type synonyms defined in the module located in the filepath provided. These definitions can be used for the rest of the current source file. However, definitions imported into the imported module from other modules are not transitively imported to the current file. As an example, the `cannotFind` function in `Test3.eva` cannot be type-checked as `x` is only being imported from `Test1.eva` to `Test2.eva`, but not to `Test3.eva`.

[4] Some authors call this type-checking algorithm type-inference instead. I consider this process as type-checking as type ascriptions are provided to guide the type-checker in a syntax-directed manner.

`def x=()`	`import Test1`	`import Test2` `def cannotFind = x`
From `Test1.eva`	From `Test2.eva`	From `Test3.eva`

Before importing a module, a number of checks are performed. Firstly, circular imports are forbidden. The module importer achieves this by storing a stack of filepaths representing the 'path of traversal' as it imports files in a depth-first-search fashion. If the file to be imported is present in the stack, a circular dependency is detected and thus the overall program analyzer aborts and returns an error message. For example, the following two programs form a circular dependency and are not accepted by Eva:

`import Test5`	`import Test4`
From `Test4.eva`	From `Test5.eva`

The module importer also checks whether the file to be imported has ever been imported beforehand into the current source file, and if yes, the AST is ignored, e.g. the second statement is ignored by the program analyzer:

```
import SameFile as File1
//Following line is ignored by the module importer
import SameFile as File2
```

Afterwards, if the module to be imported has already been parsed and type-checked previously, then the function and type synonyms definitions of the module are imported directly without parsing the same module again. Otherwise, the module is parsed and type-checked in a similar recursive fashion. Before importing function and type synonym definitions into the calling module, the module importer verifies there is no name clash with previously defined or imported function and type synonyms.

3.7.4 Lust4Eva

Code written in the Lustre domain-specific language is encapsulated with the `{- -}` constructs. The parser first generates a `LustreExp` AST for each Lustre `node` definition. Lust4Eva then analyzes the statement and returns an `AExp` AST to be processed by the type-checker, just like a normal Eva `def` statement. Type-checking Lustre programs with Eva's type-checker allows us to leverage the guarantees of Eva's type system, e.g. type-checked Lustre programs are all generative.

One interesting subtlety of the Lust4Eva transformation concerns the Lustre `pre` construct which accesses the previous element of a stream. Consider the following Lustre program:

```
{-
node invalid (input:int) returns (output:int);
let
  output = pre(input);
tel
-}
```

This is a valid Lustre function, where given an input stream $a_0, a_1, a_2, ...$, it returns `null`, $a_0, a_1, ...$. However, this function is not semantically valid in Eva as Eva programs must be generative, meaning `null` cannot be returned for any time step.

Although the `pre` construct is not necessarily generative, it is too restrictive to forbid the use of `pre` as Lustre programs like `valid` below can be interpreted by Eva without trouble, which takes in a stream $a_0, a_1, a_2, ...$, and returns $0, a_0, a_1, ...$:

```
{-
node valid (input:int) returns (output:int);
let
  output = 0 -> pre(input);
tel
-}
```

Unfortunately, determining whether a Lustre program is semantically generative, i.e. every possible execution of the program is generative, is undecidable[5]. To allow the use of `pre` without breaking generativity guarantees, Lust4Eva checks that `pre` is never used in the first time step of a function. In this case, Lust4Eva identifies all syntactically generative Lustre programs, i.e. every path on the flowgraph is generative, a decidable subset of all semantically generative programs.

At the moment, we do not allow arbitrary nested `pre` constructs in Lustre terms, e.g. `true -> if input then pre(input) else pre(pre(input))`. This is because `Lust4Eva` transforms Lustre programs into a recursive Eva program where only one past element of the input streams is stored at each step. There are two ways one can improve the transformation to support nested `pre` constructs:

1. The transformed Eva program stores a list of all past elements of all input streams. This incurs a space leak for all Lustre programs, even those that can be interpreted with constant memory, which violates the non-space-leaky guarantee of Eva.

2. Apply a similar co-effect analysis proposed by Petricek et al. [32] to infer the minimum number of past elements the Eva program needs to store so that it can be interpreted correctly. I leave this idea for future work.

3.8 Interpreters

After the main entry file type-checks completely, Eva interprets the `main` function defined in the file, if it exists. If the `main` function is not found, Eva simply type-checks all functions and informs the user that the program is valid, but no `main` function is found. This `main` function cannot be one that is imported to the origin file, e.g. the `main` function defined in `Test6.eva` is not executed if one runs `eva Test7.eva` on the command prompt:

`def main = 0`	`import Test6`
From `Test6.eva`	From `Test7.eva`

The main function cannot be polymorphic because its type is used to select the appropriate interpreter. For example, the following `main` function is not interpreted even though it type-checks.

```
def main{a}=()
```

Building on RaTT's theoretical interpreters (see section 2.5), Eva supports six types of reactive interpreters (together with a non-reactive `OneStep` interpreter), see table 3.3. These tree-interpreters

[5] This is established through a reduction to the undecidability of equality in first-order logic.

all use the function `evaluationInterpreter` of type `CExp→Store→(CExp, Store)`, which implements Eva's evaluation semantics. With the exception of the `OneStep` interpreter, each interpreter also has its own function for its corresponding step semantics, which transforms the `CExp` and `Store` at the end of each time step.

Interpreter	**Type of `main` function**
`Safe`	`#Str`(A)
`ISafe`	`#(Str`$(A) \to$ `Str`$(B))$
`Lively`	`#(A Until B)`
`ILively`	`#(Str`$(A) \to (B$ `Until` $C))$
`Fair`	`#Fair`(A, B)
`IFair`	`#(Str`$(A) \to$ `Fair`$(B, C))$
`OneStep`	All other types

Table 3.3: Eva's Interpreters

The interpreter to be run for the `main` program depends on the type of `main`, and each interpreter is proven to provide certain safety, liveness, or fairness guarantees. These guarantees are summarized by the Fundamental Theorems of Eva, which are provided in Appendix A.8. As an example, I provide the Fundamental Theorem of Eva's `ISafe` interpreter:

Fundamental Theorem of Eva's ISafe Interpreter *If* $\cdot;\cdot \vdash e$: `#(Str`$(A) \to$ `Str`$(B))$*, then there is an infinite sequence of reduction steps:*

$$\langle (?e)\ (<l_0>); \emptyset; l_0 \rangle \overset{v_1/v_1'}{\Longrightarrow}_{\texttt{ISafe}} \langle e_1; \eta_1; l_1 \rangle \overset{v_2/v_2'}{\Longrightarrow}_{\texttt{ISafe}} \langle e_2; \eta_2; l_2 \rangle \overset{v_3/v_3'}{\Longrightarrow}_{\texttt{ISafe}} \cdots$$

Moreover, if B is a value type, then $\cdot;\cdot \vdash v_i' : B$ *for all* $i \geq 1$.

3.9 Execution Options

After compiling the Eva toolchain into an executable (and including it in the `PATH` environment variable), one can run any Eva project by providing the file name of the main entry file (ending with `.eva` extension):

```
eva <file> [options]
```

The following options are available:

`--peano`: Represents numbers in the Peano notation, i.e. `0` and `suc` constructs, instead of the default Haskell `Integer` representation.

`--stepNum=<n>`: Pauses the program after every `<n>` time steps when running the `Safe`, `Lively`, or `Fair` interpreters. Pressing any key resumes the program, which pauses again after another `<n>` time steps. The default `stepNum` is `10`.

`--src=<dir>`: Changes the source directory for importing modules to `<dir>`. The default source directory is the current directory of the terminal.

`--time`: Outputs the time taken to interpret the program at each time step.

3.10 Implementation Summary

RaTT is presented in the context of a theoretical calculus, which is not rich enough to be translated directly into a programming language. I discussed how I redesigned RaTT's syntax and typing rules, and implemented various features to improve the programming experience of Eva. I then discussed implementation details of various components in Eva's toolchain, specifically the parser, program analyzer, and interpreters. Lastly, I explained how one runs an Eva project with the toolchain and presented various command-line options to control Eva's behavior.

Chapter 4

Evaluation

My project concerns the design of a programming language, Eva, and its type system, which does not lend itself well to a traditional evaluation. Instead, I evaluate my project based on three aspects:

1. I investigate the soundness of Eva's type system, which should type-reject programs that are non-causal, non-generative, or space-leaky.
2. I evaluate Eva's theoretical power, both on its computational power and time complexity.
3. I demonstrate that Eva is expressive enough to implement various one-step and reactive programs, and that the type of type-checked reactive programs provides safety, liveness, or fairness guarantees of their execution.

All Eva programs presented below form a small subset of the **540 programs** in the `example/` directory, which contains up to **3240 lines** of Eva code, see section 3.5. These programs are obtained from various sources, including relevant academic papers and industrial stream-processing libraries.

In this chapter, free variables in types represent universally-quantified polymorphic variables. We define the following type synonyms:

```
type Str(a) = NFix x --> a*x
type Maybe(a) = Unit + a
type BStr = Str(Bool)
type Ev(a) = NFix x --> a+x
type Fair(a,b) = NFix x -->
  a Until (b * >(b Until (a*x)))
```

All programs are run on a Xiaomi Notebook Pro with the following specifications:

Processor: Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz

Memory: 16GB RAM

OS: Windows 10 Pro

GHC: -O2 optimization flag

4.1 Soundness

RaTT's type system ensures type-checked programs are causal, generative, and non-space-leaky. As Eva is an enriched version of the RaTT calculus, Eva should provide similar guarantees. In

subsections 4.1.1, 4.1.2, 4.1.3, I demonstrate that Eva type-rejects non-causal, non-generative, and space-leaky programs, respectively.

4.1.1 Causality

Recall the `badTl` function in section 2.2 which is non-causal in a naïve type system as the function returns the $n+1$-st element of the input stream at time step n. An Eva implementation is as follows:

```
def badTl{a} # xs:Str(a) =
  let _:::xs' = xs in
  < xs'
```

Eva throws a type-error stating that < cannot be applied to `xs'`, informing us that `xs'` cannot be advanced and used in the current time step. However, the function is valid if we leave the tail of the stream as it is:

```
def tl{a} # xs:Str(a) =
  let _:::xs' = xs in
  xs'
```

This is type-checked by Eva with the type `#(Str(a) -> >Str(a))`. This is consistent with our expectations as the tail of the stream has a > modality, meaning it can be used safely at the next time step. For example, the function `first0` which replaces the first element of a `Nat` stream with `0` can be implemented as follows:

```
def first0 # xs:Str(Nat) =
  let _:::xs' = xs in
  0:::xs'
```

Eva gives this the type `#(Str(Nat) -> Str(Nat))`.

For completeness, we show the `@` modality obeys similar causality rules, as the following `badTl'` function for `Until` lists does not type-check:

```
def badTl'{a} # xs:a Until a =
  urec xs with
    | now x=>now x:a Until a
    | wait _ y, _ => <y
```

Eva correctly points out that the variable `y` cannot be advanced and used in the current time step. On the other hand, a similar `first0'` function for `Until` lists can be implemented as such:

```
def first0' # xs:Nat Until Nat =
  urec xs with
    | now _ => now 0:Nat Until Nat
    | wait _ y, _ => wait 0 y
```

This is type-checked by Eva with the type `#(Nat Until Nat -> Nat Until Nat)`.

4.1.2 Generativity

The functions `badTl` and `badTl'` from subsection 4.1.1 are non-generative as well, as no values can be returned at each time step without waiting for the next element of the input stream. Eva type-rejecting the two functions gives us additional confidence that its calculus is indeed generative.

Another design choice of Eva which helps to enforce generativity is that we disallow arbitrary mutual recursion to prevent circular dependencies, e.g. functions cannot refer to other functions defined later in the same module. This allows Eva to inline functions within each other in a topological sort manner, reducing the `main` function into one that does not reference other definitions, including itself. Since all basic constructs of Eva are generative, this design choice allows us to flexibly re-use previous function definitions without breaking generativity. As an example, this non-generative function `circular` cannot be type-checked, as it calls itself repeatedly:

```
def circular = circular
```

Eva type-rejects `circular` with the error stating it cannot reference itself in the body.

Note that Eva does not support any re-ordering analysis of function definitions so this program is invalid:

```
def zero1 = zero2
def zero2 = 0
```

I argue this is not too restrictive as proof assistants such as Coq [9] also impose the same restrictions. After all, computer scientists generally reason about large programs in a bottom-up manner, and one should still be able to write Eva programs easily without being hindered significantly by this restriction.

4.1.3 Non-Space-Leaking

Recall the `badCons` function in section 2.2 which incurs a space leak. The Eva implementation of the function is as follows:

```
def badCons{a} =
  nfix loop:#(Str(a) -> Str(Str(a))) =>
    fun xs:Str(a) =>
      xs ::: >(<?loop xs)
```

Eva type-rejects this, stating that the final `xs` is not `Stable` and thus cannot be accessed in the next time step without causing space leaks. Contrast this to `simpleCons`, where given a `Stable` type in the first time step, a stream with the same repeated `Stable` element is returned:

```
def simpleCons{Stable a} =
  nfix loop:#(a -> Str(a)) =>
    fun x:a =>
      x ::: >(<?loop x)
```

Eva type-accepts this with the type `#(a -> Str(a))`, given `a` is `Stable`. Unlike `badCons`, `SimpleCons` is valid because the type `a` is guaranteed to be `Stable`, meaning it can be accessed in the next time step freely without incurring implicit space leaks, e.g. `Nat`. Storing the `Stable` value for future time steps only requires constant space in the store, unlike storing a stream in our `badCons` example.

For completeness, I present the function `badConsCorrected`, which has the intended semantics of `badCons`, but is type-accepted by Eva as the space leak is explicitly coded. We do this by explicitly storing every element of the input stream in a list, and passing the list across each time step.[1] Note that the elements of the input stream must be `Stable`, otherwise the list cannot be passed across each time step without space leaks.

```
def badConsCorrected{Stable a} =
  let helper =
    nfix loop:#(List(a) -> Str(a) -> Str(Str(a))) =>
      fun l:List(a) xs:Str(a) =>
        let hd = ?streamCreate{a} l xs in
        let x:::xs' = xs in
        let l' = l++[x] in
        hd ::: >(<?loop l' <xs')
  in
  #(?helper []:List(a))
```

This is type-checked by Eva with the type `#(Str(a) -> Str(Str(a)))`, given `a` is `Stable`.

4.2 Theoretical Power

We now examine Eva's theoretical computational power and asymptotic time complexity. In subsection 4.2.1, I show that Eva's evaluation semantics is higher-order primitive recursive, and its full semantics is Turing-complete. In subsection 4.2.2, I show that Eva's constructs match the theoretical complexities, e.g. the basic arithmetic operations for binary representation are $O(1)$, and do not depend on the size of the operands.

4.2.1 Computability

Recall from subsection 2.3.2 that RaTT's evaluation semantics is higher-order primitive recursion. Though Eva has a richer type system, its evaluation semantics remains higher-order primitive recursive. It is not Turing-complete as we need to forbid non-termination in each time step, so that Eva's programs are guaranteed to be generative.

To provide empirical evidence that its evaluation semantics is higher-order primitive recursive, I implemented various one-step higher-order primitive recursive functions. For example, Eva can implement both the Ackermann function [1] of type `#(Nat -> Nat -> Nat)`, and the quicksort algorithm of type `#(List(Nat) -> List(Nat))`, both of which are higher-order primitive recursive functions that are not primitive recursive, see appendix B.1 and B.2, respectively.

When we consider the full semantics of Eva, the step semantics allows us to use streams to represent the potentially non-terminating nature of computations. I demonstrate that Eva's full semantics is Turing-complete by implementing various Turing-complete models of computation:

1. Partial-recursive functions [22]
2. Universal register machine [25]

[1] Here the pre-defined function `streamCreate` takes in a list and a stream, and returns the stream with the list appended in front. Note the explicit use of ++ append to create a larger list `l'`.

3. Brainfuck interpreter [16]

Here, I show that all partial-recursive functions can be implemented in Eva. By definition, the class of partial-recursive functions is the smallest class of functions containing the basic functions (constant, projection, successor) and is closed under composition, primitive recursion, and minimization. Eva's evaluation semantics is higher-order primitive recursive, and it is trivial to see how the basic functions, composition, and primitive recursion of functions can be implemented in one time step. To show all partial-recursive functions are implementable, it suffices to demonstrate that we can represent minimization via a stream.

To encode minimization, we use the type `Str(Maybe(Nat))` to represent a stream. An `inl ()` means we are still looking for a `Nat` to cause the argument function to return 0, and `inr n` means `n` is the smallest `Nat` for the function to return 0. In particular, I present the minimization function of type `(#(Nat -> Nat -> Nat -> Nat) -> #(Nat -> Nat -> Nat))`, that finds the smallest first argument that causes the ternary function `f` to return 0 below:

```
def minimization2
  f:#(Nat -> Nat -> Nat -> Nat) =
    fun x1:Nat x2:Nat =>
      let repeat =
        (nfix loop:#(Nat -> Str(Maybe(Nat))) =>
          fun i:Nat =>
            let b = (?f i x1 x2) == 0 in
            let success =
              ?simpleCons{Maybe(Nat)}
              (inr i:Maybe(Nat)) in
            let failureHd = inl ():Maybe(Nat) in
            let failureTl = >(<?loop (suc i)) in
            let failure = failureHd ::: failureTl in
            if b then success else failure
        ) in
      #(?repeat 0)
```

If no argument causes `f` to return 0, `minimization2` returns a constant stream of `inl ()`. Otherwise, if `n` is the smallest `Nat` to make `f` return 0, the output stream returns `inr n` repeatedly after the `n`-th time step.

4.2.2 Time Complexity

In RaTT, `Nat` numbers are inductively built from the Peano axioms. This means that basic operations on `Nat` like addition or multiplication are not built-in constructs, but have to be defined via primitive recursion. As an example, if we just limit ourselves to RaTT's `suc` and `primrec` constructs, the program for adding two numbers of type `#(Nat -> Nat -> Nat)` is written as:

```
def add # n:Nat m:Nat =
 primrec n with
   | 0 => m
   | suc _, rest => suc rest
```

The function `add` repeats the `suc` operation `n` number of times, meaning the addition of `n` and `m` is $O(n)$. If we use similar methods to build more complex functions, like exponential, these functions

are very slow, even when provided with small arguments. For example, figure 4.1 shows the time taken[2] to compute the n-th Fibonnacci number in Peano representation iteratively with the following code (the function `fib` has type `#Str(Nat)`):

```
def fibHelper =
  nfix repeat:#(Nat -> Nat -> Str(Nat)) =>
    fun x:Nat y:Nat =>
      let sum = x `?add` y in
      let rest = >(<?repeat y sum) in
      sum ::: rest

def fib # = ?fibHelper 0 1
```

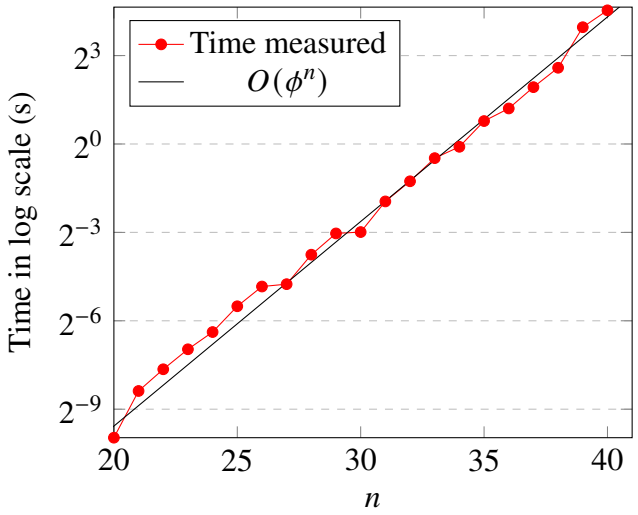

Figure 4.1: Time taken to compute the n-th Fibonacci number in Peano representation (averaged over 5 runs)

Note that the time complexity grows roughly as fast as the Fibonacci sequence, i.e. $O(\phi^n)$. (The golden ratio ϕ is roughly 1.618.) This is consistent with the two cost analysis equations below:

$$\text{fibHelperCost}(0, a, b) = 1$$
$$\text{fibHelperCost}(n+1, a, b) = 1 + \text{addCost}(a, b) + \text{fibHelper}(n, b, a+b)$$

In addition, Eva runs out of memory computing beyond the 40-th Fibonacci number (102334155) since the space required to store a number in Peano representation is linear to the magnitude of the number.

By contrast, Eva implements `Nat` with Haskell's `Integer` numbers by default. For operations like + or ^, it directly applies Haskell's built-in $O(1)$ operations on Haskell's `Integer`, though as mentioned in subsection 3.2.1, we rewrite certain operations to account for edge cases to preserve generativity, e.g. dividing by 0. When we re-compute the Fibonacci numbers with Eva's default settings, the recorded time taken to compute each of the first 10000 Fibonacci numbers is negligible, i.e. 0.000s, which matches the expected constant time complexity.

To show that other Eva constructs match the theoretical time complexities, I measured the time taken to insertion-sort a random list of length n, with n ranging from 50 to 1000, see figure 4.2, which matches the expected time complexity of $O(n^2)$.

[2] The time taken to compute the first 19 Fibonacci numbers is too insignificant to be plotted on a logarithmic graph.

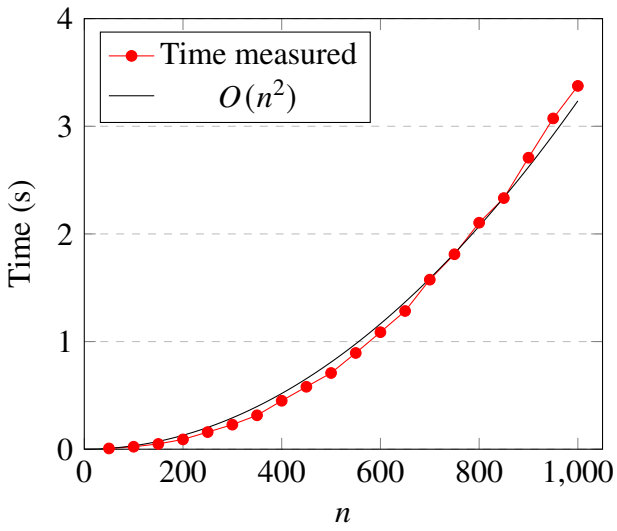

Figure 4.2: Time taken to insertion sort a random list of integers in Haskell `Integer` representation of length n (averaged over 5 runs)

4.3 Usability

Though RaTT has clear syntax, typing and semantics rules, it is difficult to program with pure RaTT constructs. Eva enriches RaTT's syntax and semantics so that they lend themselves more easily to programming. The `example/` directory contains a corpus of Eva programs obtained from various sources that illustrates Eva's practicality and usability. These programs can be classed into two categories:

1. One-step non-reactive programs for the `OneStep` interpreter
2. Multi-step reactive programs for the six reactive interpreters

In particular, the reactive programs highlight the heart of Eva's strength – using types to verify the dynamic safety or liveness properties of a program's execution in a single language, by implementing reactive programs which are productive, terminating, or fair.

4.3.1 One-Step Programs

We first consider one-step non-reactive programs, meaning the program only executes the evaluation semantics via the `OneStep` interpreter, and terminates after the first time step. These programs are classed into four modules:

1. `Prelude/Nat.eva`, which implements various interesting functions on `Nat`, e.g. logarithm and square root.
2. `Prelude/Bool.eva`, which implements basic functions for `Bool`.
3. `Prelude/List.eva`, which implements a significant subset of the Haskell functions on finite lists from the `Data.List` library.
4. `Prelude/Maybe.eva`, which implements all the Haskell functions on the `Maybe` monad from the `Data.Maybe` library.

As an interesting example from `Prelude/List.eva`, assuming `foldr` is pre-defined, we can write the `subsequences` function of type `#(List(a) -> List(List(a)))`, which returns a list of all possible subsequences of the argument list:

```
def subsequences{a} xs:List(a) =
 let nonEmptySubsequences ys:List(a) =
   primrec ys with
     | [] => []:List(List(a))
     | x::_, rest =>
       let f zs:List(a) r:List(List(a)) =
         zs::(x::zs)::r in
       [x]::foldr{List(a), List(List(a))}
         f []:List(List(a)) rest
 in
 []:List(a) :: (nonEmptySubsequences xs)
```

4.3.2 Productive Programs

Most people consider reactive programs to be productive by definition, meaning that they continue to interact with the environment forever. As a result, there is a large corpus of productive programs from various sources I can implement and evaluate with Eva. Most of these functions in this subsection can be interpreted by the `Safe` or `ISafe` interpreters, or they operate on types involving the > modality.

The module `Modality/Stream.eva` contains basic functions operating on streams which are heavily used in many of the following modules. For example, the function `natStr` in the module returns the stream of `Nat` numbers, i.e. `0`, `1`, `2`,....

Modules in the `Papers/` directory contain the implementation of various example programs listed in academic papers introducing similar reactive calculi, namely:

1. AdjS [23]
2. Simply Typed Lambda Calculus with clocks and guards [4]
3. Simply RaTT [6]
4. Lively RaTT [7]
5. Rattus [5]

It is worth mentioning that by implementing these example programs from academic papers, I identified type-checking errors for three example programs in the Simply RaTT paper, namely `edgeAux`, `current`, and `counter`. All three errors were reported to the main author Bahr, who subsequently corrected them in the local copy of the paper on his website.

In addition, I implemented functions from three stream-processing programming tools, namely:

1. Reactive Banana, a Haskell higher-order FRP library, in `ReactiveBanana/`
2. Yampa, a Haskell arrowized FRP libary, in `Yampa/`
3. Milan [10], a data-oriented progamming language, in `Milan/`

Moreover, hardware circuits with a global clock can also be considered as productive programs. Programs in the `Hardware/` directory illustrate how various hardware circuits can be written by Eva. For example, the `ff` function of type `(Bool -> #BStr -> #BStr)` in `Hardware/Definitions.eva` simulates the action of a D flip-flop, which delays a stream of `Bool` by one time step:

```
def ff initial:Bool xs:#BStr=
  let helper = nfix loop:#(Bool -> BStr -> BStr) =>
    fun current:Bool xs:BStr=>
      let x:::xs' = xs in
      current ::: >(<?loop x <xs')
  in #(?helper initial ?xs)
```

All Lustre programs are productive (see subsection 3.3.3), and through the Lust4Eva transformation, `InterestingPrograms/LustreExamples.eva` demonstrates how one can leverage Lustre's higher abstraction model to implement reactive programs, while achieving correctness guarantees from Eva's type system. Here I present the famous `posClockEdge` function, found in many Lustre tutorials, which outputs `true` at every positive edge transition:

```
{-
node posClockEdge (clock:bool) returns (output:bool);
let
  output = false -> clock and not pre(clock);
tel
-}
```

Eva type-checks this with the type `#(BStr -> BStr)`, and runs it with the interactive `ISafe` interpreter.

4.3.3 Terminating Programs

In addition to supporting productive programs, one of RaTT's most exciting strengths is that it can support terminating programs with the `@` modality in the same language, meaning they should eventually halt at a certain time step in the future. Here, I show that Eva possesses this strength as well, by presenting various programs that are executed by the `Lively` or `ILively` interpreters, or they operate on types involving the `@` modality.

To start with, I present the simple `countdown` function, where given the argument `n`, it returns the `Until` list containing $n, n-1, ..., 0$, which has type `(Nat -> #(Nat Until Nat))`:

```
def countdown n:Nat =
  primrec n with
    | 0 => # now 0:Nat Until Nat
    | suc x, rest => # wait (suc x) @(?rest)
```

In general, terminating programs are more difficult to program in Eva than productive ones. Under the Curry-Howard correspondence [20], writing a program for a type is similar to finding a proof for a theorem. With a rich type system like Eva's, writing a program to match the intended type seems challenging. Moreover, it is well-known that proving liveness properties, like termination, is harder than proving safety properties, like productivity [31]. For the function `countdown` above, the `primrec` construct implicitly "proves" that the `Until` list constructed is finite under the Curry-Howard programs-as-proofs model since `primrec` is total.

However, I argue this is not a problem because many interesting programs are built by composing various smaller constructs. After doing much of the hard work implementing a library containing functions for terminating programs, stringing them together to form more sophisticated ones is not too difficult a task.

To evaluate whether Eva succeeds in combining both productive and terminating programs in a single language, it is important that one investigates how the ○ and ▷ modalities interact in Eva. Recall the submodality inequality from section 2.5, i.e. ○ ≤ ▷, meaning we can coerce any element of type ○A to type ▷A freely, but not the other round. We can see this is the case for Eva through the following two functions in `Modality/Primitives.eva`:

```
def atToAngle{a} # x:@a = ><x
def angleToAtWrong{a} # x:>a = @<x
```

Eva type-checks `atToAngle` with the type `#(@a -> >a)` as intended, which shows that one can indeed coerce a type ○A to type ▷A arbitrarily. On the other hand, Eva type-rejects `angleToAtWrong` with an error message, which suggests that `x` cannot be coerced from a type ▷A to a type ○A freely. These two type-checking results match the expected effects of the submodality rule. For completeness, I also show the correct implementation of `angleToAt` of type `#(>a -> @a)`, which is allowed given that the polymorphic type variable `a` is `Limit`:

```
def angleToAt{Limit a} # x:>a = @<x
```

The RaTT paper presents a large corpus of programs using both ○ and ▷ in interesting ways. Eva is able to type-accept every one of them, suggesting that Eva indeed inherits RaTT's strength in handling both modalities in one system correctly. As an example, the `timeout` function of type `(Nat -> #(Ev(a) -> (Unit Until Maybe(a))))` in `Papers/LivelyRaTT.eva` has the effect similar to taking the first `n` elements of an infinite input stream to form an `Until` list of length `n`.

4.3.4 Fair Programs

In addition to productive and terminating programs, Eva can also interpret fair programs with the reactive `Fair` and `IFair` interpreters. Fair programs are interesting for two reasons. Firstly, fairness is an intersection of safety and liveness properties, which highlights the elegant marriage between both properties within Eva. Secondly, proving fairness of a program is even more challenging than proving termination, so under the Curry-Howard programs-as-proofs model, writing a fair program that type-checks successfully is an enthralling task. However, just like terminating programs, as long as most of the common functions for `Fair` streams are first implemented in a standard library, one can compose them together to produce more complicated ones while preserving static and dynamic properties.

The RaTT paper and the paper by Cave et al. [12] provide a corpus of fair programs, where each of them is type-checked and interpreted by Eva correctly, the most interesting being the fair scheduler function `sch` of type `#(Nat -> Str(a) -> Str(b) -> Fair(a, b))`[3]. This function interleaves two streams into an output `Fair` stream, where it selects a progressively increasing number of elements from the first stream for each time it selects an element from the second. In particular, `?sch{a, b} 0` takes in two streams of type `Str(a)` and `Str(b)`, to produce a `Fair` stream of the form:

```
b a a b a a a b a a a a b a a a a a b . . .
```

Though the `b` stream gets an infinite number of turns, its proportion in the return stream reduces steadily to zero.

[3] The type variables `a` and `b` are required to be `Limit`.

4.4 Evaluation Summary

The design and implementation of Eva achieve all the core success criteria and extensions listed in chapter 1:

1. Matching RaTT's theoretical correctness guarantees, Eva correctly type-rejects all invalid programs, i.e. non-causal, non-generative, or space-leaky programs.
2. I showed that Eva is highly expressive in that it is higher-order primitive recursive in one time step, and Turing-complete over a stream, by implementing various models of computation.
3. By benchmarking Eva's execution time on various programs, I demonstrated that its language constructs match expected theoretical time complexities.
4. I showed that Eva can be realized as a practical programming language by implementing a large corpus of both non-reactive and reactive programs taken from various sources.

Chapter 5

Conclusion

5.1 Summary

This dissertation project was a success.

I successfully enriched the **theoretical RaTT calculus** to form the **practical programming language Eva**, by enhancing RaTT's syntax and type system. Eva supports type synonyms and modular programming. The industrial dataflow language Lustre was also implemented as an embedded DSL within Eva via the Lust4Eva transformation.

I demonstrated Eva's correctness guarantees by showing that Eva type-rejects all non-causal, non-generative, or space-leaky programs. I also demonstrated Eva is higher-order primitive recursive in a single time step and Turing-complete over time by implementing various models of computation, and that Eva's performance is consistent with the theoretical complexities via benchmarking. Most importantly, I showed that Eva is highly expressive and implemented a plethora of programs taken from various sources, and demonstrated how one could implement programs with their types guaranteeing dynamic safety or liveness properties, including productive, terminating, and fair programs, and interpret them with one of the six reactive interpreters, which is selected by their program type.

5.2 Lessons Learned

RaTT is the result of several man-years of work by various authors. Therefore, working on a project built on RaTT required significant understanding of the literature. Although understanding the literature enough to implement Eva was relatively straightforward, understanding it well enough to be able to write a tutorial-style explanation of the calculus took a lot more effort. In addition, the design and implementation of Eva cover various topics beyond RaTT, e.g. parametric polymorphism, the Lustre dataflow paradigm. It is near impossible to explain all the concepts of Eva well enough to be understood by a Part IB student, while fulfilling the word limit. As such, tough decisions on choosing which topics should be dropped from the dissertation and which should be elaborated in greater detail had to be made frequently.

5.3 Future Work

There are various directions for continuing this project.

Eva currently does not support type inference, and many expressions such as `nfix` and `now` require the

use of type ascriptions to guide the type-checker. One future direction is to extend Eva's type checker to enable type inference, so programs can be type checked without every required type ascription.

In order to use implement higher-order primitive recursive functions for a single time step, the programmer must use the basic `primrec` construct to build the targeted function. For a function with multiple arguments to be primitive-recursed on, one has to use `primrec` on every single argument individually, making the code hard to write, and equally difficult to read. In addition, mutually-recursive functions must be defined together using a "coupling" method. This is highly inconvenient, especially if the number of mutually-recursive functions increases. A possible extension is to redesign Eva's syntax to allow users to define mutually-recursive functions in a more traditional fashion, and allow the analysis and translation to the plain Eva's calculus be performed by the type-checker, similarly to Coq, Agda [30], and Isabelle [29].

Lastly, the Eva language is currently supported by seven high-level interpreters. This means that we have less control over memory on the machine code level, and we are unable to confidently guarantee the elimination of space leaks on a lower-level even though it is the case on the abstract data level. In addition, we do not perform any optimization on the machine code to increase its time and space efficiency. It would be great to investigate whether Eva programs can be directly compiled into machine code, or even to hardware, thus allowing us to reason about memory usage better and introduce interesting compiler optimizations to the compiled machine-code.

Bjarne Stroustrup, the inventor of C++, once said, "There are only two kinds of languages: the ones people complain about and the ones nobody uses." I look forward to a future where Eva is disliked by more computer scientists.

Bibliography

[1] W. Ackermann. Zum Hilbertschen Aufbau der reellen Zahlen. *Mathematische Annalen*, 99:118–133, 1928.

[2] B. Alpern and F. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, Sep 1987.

[3] K. Arnold, J. Gosling, and D. Holmes. *The Java programming language*. Addison Wesley Professional, 2005.

[4] R. Atkey and C. McBride. Productive coprogramming with guarded recursion. *SIGPLAN Not.*, 48(9):197–208, Sep 2013.

[5] P. Bahr. Modal FRP for all: Functional reactive programming without space leaks in Haskell. Submitted to JFP, July 2021.

[6] P. Bahr, C. Graulund, and R. E. Møgelberg. Simply RaTT: A Fitch-style modal calculus for reactive programming. *CoRR*, abs/1903.05879, 2019.

[7] P. Bahr, C. Graulund, and R. E. Møgelberg. Diamonds are not forever: Liveness in reactive programming with guarded recursion. *Proc. ACM Program. Lang.*, 5(POPL), Jan. 2021.

[8] H. Barendregt, W. Dekkers, and R. Statman. *Lambda Calculus with Types*. Perspectives in Logic. Cambridge University Press, 2013.

[9] Y. Bertot and P. Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[10] T. Borchert. Milan: An evolution of data-oriented programming. *Tom Borchert's Programming Blog*, 2020. `https://tborchertblog.wordpress.com/2020/02/13/28/`.

[11] K. B. Bruce, A. R. Meyer, and J. C. Mitchell. The semantics of second-order lambda calculus. *Information and Computation*, 85(1):76–134, 1990.

[12] A. Cave, F. Ferreira, P. Panangaden, and B. Pientka. Fair reactive programming. *SIGPLAN Not.*, 49(1):361–372, Jan 2014.

[13] S. Chacon and B. Straub. *Pro git*. Apress, 2014.

[14] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, Nov 1970.

[15] R. Clouston, A. Bizjak, H. B. Grathwohl, and L. Birkedal. The guarded lambda-calculus: Programming and reasoning with guarded recursion for coinductive types. *Log. Methods Comput. Sci.*, 12(3), 2016.

[16] B. Easter. Fully human, fully machine: Rhetorics of digital disembodiment in programming. *Rhetoric Review*, 39(2):202–215, 2020.

[17] C. Elliott and P. Hudak. Functional reactive animation. *SIGPLAN Not.*, 32(8):263–273, Aug 1997.

[18] K. Gödel. On an extension of finitary mathematics which has not yet been used. In S. Feferman, J. Dawson, and S. Kleene, editors, *Kurt Gödel: Collected Works Vol. Ii*, pages 271–284. Oxford University Press, 1972.

[19] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79:1305 – 1320, Oct 1991.

[20] W. A. Howard. The formulae-as-types notion of construction. In H. Curry, H. B., S. J. Roger, and P. Jonathan, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, 1980.

[21] A. Jeffrey. LTL types FRP: Linear-time temporal logic propositions as types, proofs as functional reactive programs. In *Proceedings of the Sixth Workshop on Programming Languages Meets Program Verification*, PLPV '12, page 49–60, New York, NY, USA, 2012. Association for Computing Machinery.

[22] S. C. Kleene. λ-definability and recursiveness. *Duke Mathematical Journal*, 2(2):340 – 353, 1936.

[23] N. R. Krishnaswami. Higher-order functional reactive programming without spacetime leaks. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, page 221–232, New York, NY, USA, 2013. Association for Computing Machinery.

[24] M. Kwiatkowska. Survey of fairness notions. *Information & Software Technology*, 31:371–386, 1989.

[25] J. Lambek. How to program an infinite abacus. *Canadian Mathematical Bulletin*, 4(3):295–302, 1961.

[26] D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Departement of Computer Science, Universiteit Utrecht, July 2001. User Modeling 2007, 11th International Conference, UM 2007, Corfu, Greece, June 25-29, 2007.

[27] S. Marlow. Haskell 2010 language report. *Available online http://www. haskell. org/(May 2011)*, 2010.

[28] H. Nakano. A modality for recursion. In *Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.99CB36332)*, pages 255–266, 2000.

[29] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[30] U. Norell. Towards a practical programming language based on dependent type theory. In *PhD Thesis - Chalmers University of Technology*, 2007.

[31] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst.*, 4(3):455–495, Jul 1982.

[32] T. Petricek, D. Orchard, and A. Mycroft. Coeffects: A calculus of context-dependent computation. In *Proceedings of International Conference on Functional Programming*, ICFP 2014, 2014.

[33] A. Pnueli. The temporal logic of programs. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 46–57, Los Alamitos, CA, USA, Oct 1977. IEEE Computer Society.

[34] J. Smith. The identification of propositions and types in Martin-L öf's type theory: A programming example. *International Conference on Fundamentals of Computation Theory*, pages 445–456, 1983.

[35] G. Van Rossum and F. L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[36] W. W. Wadge and E. A. Ashcroft. *LUCID, the Dataflow Programming Language*. Academic Press Professional, Inc., USA, 1985.

[37] S. Weeks. Whole-program compilation in MLton. In *Proceedings of the 2006 Workshop on ML*, ML '06, page 1, New York, NY, USA, 2006. Association for Computing Machinery.

Appendix A

Eva's Specifications

A.1 Abstract Syntax

⟨*program*⟩ ::= ⟨*stmt*⟩ | ⟨*program*⟩ ⟨*stmt*⟩ | ⟨*program*⟩ '`{-`' ⟨*lustre-stmts*⟩ '`-}`'

⟨*stmt*⟩ ::= ⟨*def-stmt*⟩ | ⟨*type-stmt*⟩ | ⟨*import-stmt*⟩

⟨*def-stmt*⟩ ::= '`def`' ⟨*lower-str*⟩ ⟨*opt-curly-parameters*⟩ ⟨*opt-arguments*⟩ ⟨*opt-box*⟩ ⟨*opt-arguments*⟩ '=' ⟨*exp*⟩

⟨*opt-curly-parameters*⟩ ::= '' | '`{`' ⟨*parameters*⟩ '`}`'

⟨*parameters*⟩ ::= ⟨*parameter*⟩ | ⟨*parameters*⟩ '`,`' ⟨*parameter*⟩

⟨*parameter*⟩ ::= ⟨*opt-property*⟩ ⟨*lower-str*⟩

⟨*opt-property*⟩ ::= '' |'`Stable`' | '`Limit`' | '`Stable Limit`' | '`Comparable`'

⟨*opt-arguments*⟩ ::= '' | ⟨*arguments*⟩

⟨*arguments*⟩ ::= ⟨*argument*⟩ | ⟨*argument*⟩ ⟨*arguments*⟩

⟨*argument*⟩ ::= ⟨*lower-str*⟩ ⟨*ascription*⟩

⟨*opt-box*⟩ ::= '' | '`#`'

⟨*exp*⟩ ::= ⟨*opt-dotted-lower-str*⟩ ⟨*opt-curly-type-list*⟩ | ⟨*number*⟩ | '`()`'
| '`fun`' ⟨*arguments*⟩ '`=>`' ⟨*exp*⟩
| ⟨*exp*⟩ ⟨*exp*⟩
| '`nfix`' ⟨*lower-str*⟩ ⟨*ascription*⟩ '`=>`' ⟨*exp*⟩
| '`let`' ⟨*lower-str*⟩ ⟨*opt-arguments*⟩ ⟨*opt-box*⟩ ⟨*opt-arguments*⟩ '=' ⟨*exp*⟩ '`in`' ⟨*exp*⟩
| ⟨*exp*⟩ '`` ` ``' ⟨*exp*⟩ '`` ` ``' ⟨*exp*⟩ | '`true`' | '`false`'
| '`if`' ⟨*exp*⟩ '`then`' ⟨*exp*⟩ '`else`' ⟨*exp*⟩
| ⟨*unary-exp-op*⟩ ⟨*exp*⟩ | ⟨*exp*⟩ ⟨*binary-exp-op*⟩ ⟨*exp*⟩
| '`(`' ⟨*exp*⟩ '`,`' ⟨*exp*⟩ '`)`'
| '`let`' '`(`' ⟨*wc-var*⟩ '`,`' ⟨*wc-var*⟩ '`)`' '=' ⟨*exp*⟩ '`in`' ⟨*exp*⟩
| '`inl`' ⟨*exp*⟩ ⟨*ascription*⟩ | '`inr`' ⟨*exp*⟩ ⟨*ascription*⟩
| '`match`' ⟨*exp*⟩ '`with`' '`|`' '`inl`' ⟨*wc-var*⟩ '`=>`' ⟨*exp*⟩ '`|`' '`inr`' ⟨*wc-var*⟩ '`=>`' ⟨*exp*⟩
| '`[]`' ⟨*ascription*⟩ | '`[`' ⟨*list-elems*⟩ '`]`'
| '`let`' ⟨*wc-var*⟩ '`:::`' ⟨*wc-var*⟩ '=' ⟨*exp*⟩ '`in`' ⟨*exp*⟩

| ‘primrec’ ⟨*exp*⟩ ‘with’ ‘|’ ‘0’ ‘=>’ ⟨*exp*⟩ ‘|’ ‘suc’ ⟨*wc-var*⟩ ‘,’ ⟨*wc-var*⟩ ‘=>’ ⟨*exp*⟩
| ‘primrec’ ⟨*exp*⟩ ‘with’ ‘|’ ‘[]’ ‘=>’ ⟨*exp*⟩ ‘|’ ⟨*wc-var*⟩ ‘::’ ⟨*wc-var*⟩ ‘,’ ⟨*wc-var*⟩ ‘=>’ ⟨*exp*⟩
| ‘now’ ⟨*exp*⟩ ⟨*ascription*⟩ | ‘wait’ ⟨*exp*⟩ ⟨*exp*⟩
| ‘urec’ ⟨*exp*⟩ ‘with’ ‘|’ ‘now’ ⟨*wc-var*⟩ ‘=>’ ⟨*exp*⟩ ‘|’ ‘wait’ ⟨*wc-var*⟩ ⟨*wc-var*⟩ ‘,’ ⟨*wc-var*⟩ ‘=>’ ⟨*exp*⟩
| ‘into’ ⟨*exp*⟩ ⟨*ascription*⟩

⟨*opt-dotted-lower-str*⟩ ::= ⟨*lower-str*⟩ | ⟨*upper-str*⟩ ‘.’ ⟨*lower-str*⟩

⟨*opt-curly-type-list*⟩ ::= ‘’ | ‘{’ ⟨*type-list*⟩ ‘}’

⟨*type-list*⟩ ::= ⟨*type*⟩ | ⟨*type*⟩ ‘,’ ⟨*type-list*⟩

⟨*unary-exp-op*⟩ ::= ‘not’ | ‘fst’ | ‘snd’ | ‘>’ | ‘@’ | ‘<’ | ‘#’ | ‘?’ | ‘out’ | ‘suc’

⟨*binary-exp-op*⟩ ::= ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘%’ | ‘^’ | ‘and’ | ‘or’ | ‘==’ | ‘!=’ | ‘::’ | ‘++’ | ‘:::’

⟨*list-elems*⟩ ::= ⟨*exp*⟩ | ⟨*exp*⟩ ‘,’ ⟨*list-elems*⟩

⟨*wc-var*⟩ ::= ‘_’ | ⟨*lower-str*⟩

⟨*ascription*⟩ ::= ‘:’ ⟨*type*⟩

⟨*type-stmt*⟩ ::= ‘type’ ⟨*upper-str*⟩ ⟨*opt-round-arguments*⟩ ‘=’ ⟨*type*⟩

⟨*opt-round-arguments*⟩ ::= ‘’ | ‘(’ ⟨*lower-str-list*⟩ ‘)’

⟨*lower-str-list*⟩ ::= ⟨*lower-str*⟩ | ⟨*lower-str*⟩ ‘,’ ⟨*lower-str-list*⟩

⟨*type*⟩ ::= ⟨*opt-dotted-upper-str*⟩ ⟨*opt-round-type-list*⟩ | ⟨*lower-str*⟩ | ‘(’ ⟨*type*⟩ ‘)’
| ‘Unit’ | ‘Nat’ | ‘Bool’ | ‘List’ ‘(’ ⟨*type*⟩ ‘)’
| ‘NFix’ ⟨*lower-str*⟩ ‘-->’ ⟨*type*⟩
| ⟨*unary-type-op*⟩ ⟨*type* | <*type*⟩ ⟨*binary-type-op*⟩ ⟨*type*⟩

⟨*opt-round-type-list*⟩ ::= ‘’ | ‘(’ ⟨*type-list*⟩ ‘)’

⟨*opt-dotted-upper-str*⟩ ::= ⟨*upper-str*⟩ | ⟨*upper-str*⟩ ‘.’ ⟨*upper-str*⟩

⟨*unary-type-op*⟩ ::= ‘>’ | ‘@’ | ‘#’

⟨*binary-type-op*⟩ ::= ‘->’ | ‘Until’ | ‘+’ | ‘*’

⟨*import-stmt*⟩ ::= ‘import’ ⟨*file-path*⟩ ⟨*opt-name*⟩

⟨*opt-name*⟩ ::= ‘’ | ‘as’ ⟨*upper-str*⟩

⟨*lustre-stmts*⟩ ::= ‘’ | ⟨*lustre-stmts*⟩ ⟨*lustre-stmt*⟩

⟨*lustre-stmt*⟩ ::= ‘node’ ⟨*lower-str*⟩ ‘(’ ⟨*lustre-args*⟩ ‘)’ ‘returns’ ⟨*lustre-var-ascrip*⟩ ‘;’ ‘let’ ⟨*lustre-body*⟩ ‘tel’

⟨*lustre-args*⟩ ::= ‘’ | ⟨*lustre-args-multiple*⟩

⟨*lustre-args-multiple*⟩ ::= ⟨*lustre-var-ascrip*⟩ | ⟨*lustre-args-multiple*⟩ ‘,’ ⟨*lustre-var-ascrip*⟩

⟨*lustre-var-ascrip*⟩ ::= ⟨*lower-str*⟩ ‘`:`’ ⟨*lustre-type*⟩

⟨*lustre-type*⟩ ::= ‘`int`’ | ‘`bool`’

⟨*lustre-body*⟩ ::= ⟨*lower-str*⟩ ‘=’ ⟨*lustre-exp*⟩ ‘`;`’ | ⟨*lower-str*⟩ ‘=’ ⟨*lustre-exp*⟩ ‘`->`’ ⟨*lustre-exp*⟩ ‘`;`’

⟨*lustre-exp*⟩ ::= ⟨*number*⟩ | ⟨*lower-str*⟩ | ‘`pre(`’ ⟨*lower-str*⟩ ‘`)`’ | ⟨*lustre-exp*⟩ ⟨*lustre-bin-op*⟩ ⟨*lustre-exp*⟩
| ‘`not`’ ⟨*lustre-exp*⟩ | ‘`true`’ |‘`false`’ | ‘`if`’ ⟨*lustre-exp*⟩ ‘`then`’ ⟨*lustre-exp*⟩ ‘`else`’
⟨*lustre-exp*⟩

⟨*lustre-bin-op*⟩ ::= ‘=’ | ‘<’ | ‘>’ | ‘`or`’ | ‘`and`’ | ‘+’ | ‘-’ | ‘*’ | ‘/’

A.2 Definitions

$$
\begin{aligned}
\text{Types } A\ B\ C &::= x \mid \texttt{Unit} \mid \texttt{Nat} \mid \texttt{Bool} \mid \texttt{List}(A) \mid \texttt{NFix}\ x \longrightarrow A \\
&\mid {>}A \mid \texttt{@}A \mid \#A \mid A \to A \mid A\ \texttt{Until}\ A \mid A + A \mid A * A \\
\text{Expressions } e &::= x \mid n \mid () \mid \texttt{fun}\ x : A \texttt{=>}\ e \mid e\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \\
&\mid e + e \mid e - e \mid e * e \mid e/e \mid e\%e \mid e\texttt{\^{}}e \mid \texttt{suc}\ e \\
&\mid \texttt{primrec}\ e\ \texttt{with} \mid \texttt{0} \texttt{=>}\ e \mid \texttt{suc}\ x, x \texttt{=>}\ e \\
&\mid \texttt{true} \mid \texttt{false} \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid e\ \texttt{and}\ e \mid e\ \texttt{or}\ e \mid \texttt{not}\ e \\
&\mid e == e \mid e != e \\
&\mid (e, e) \mid \texttt{fst}\ e \mid \texttt{snd}\ e \\
&\mid \texttt{inl}\ e : A \mid \texttt{inr}\ e : A \mid \texttt{match}\ e\ \texttt{with} \mid \texttt{inl}\ x \texttt{=>}\ e \mid \texttt{inr}\ x \texttt{=>}\ e \\
&\mid [] : A \mid [e, \ldots, e] \mid e :: e \mid e +\!+ e \\
&\mid \texttt{primrec}\ e\ \texttt{with} \mid [] \texttt{=>}\ e \mid x :: x, x \texttt{=>}\ e \\
&\mid {>}e \mid \texttt{@}e \mid {<}e \mid \#e \mid \texttt{?}e \mid \texttt{nfix}\ x : A \texttt{=>}\ e \mid e\texttt{:::}e \mid \texttt{let}\ x\texttt{:::}x = e\ \texttt{in}\ e \\
&\mid \texttt{now}\ e : A \mid \texttt{wait}\ e\ e \mid \texttt{urec}\ e\ \texttt{with} \mid \texttt{now}\ \texttt{=>}\ e \mid \texttt{wait}\ x\ x, x \texttt{=>}\ e \\
&\mid \texttt{into}\ e : A \mid \texttt{out}\ e \\
\text{Type Qualifiers } \omega &::= \texttt{None} \mid \texttt{Stable} \mid \texttt{Limit} \mid \texttt{Comparable} \mid \texttt{LimitStable} \\
\text{Type Contexts } \Theta &::= \cdot \mid \Theta, (x, \omega) \\
\text{Term Variable Contexts } \Gamma &::= \cdot \mid \Gamma, x : A \mid \Gamma, \# \mid \Gamma, \checkmark_{>} \mid \Gamma, \checkmark_{@} \\
\text{Values } v\ w &::= () \mid n \mid \texttt{fun}\ x : A \texttt{=>}\ e \mid (v, v) \mid \texttt{inl}\ v \mid \texttt{inr}\ v \\
&\mid \texttt{true} \mid \texttt{false} \mid [] : A \mid [v, \ldots, v] \\
&\mid \#e \mid \texttt{@}e \mid {>}e \mid \texttt{nfix}\ x : A \texttt{=>}\ e \mid l \mid \texttt{into}\ v : A \mid \texttt{now}\ v : A \mid \texttt{wait}\ v\ v \\
\text{Heaps } \eta &::= \{l \mapsto v, \ldots, l \mapsto v\} \\
\text{Stores } \sigma &::= \cdot \mid \eta \mid \eta \checkmark \eta \\
\text{Value types } U\ V &::= \texttt{Unit} \mid \texttt{Nat} \mid \texttt{Bool} \mid \texttt{List}(U) \mid U * U \mid U + U \\
\texttt{Str}(A) &::= \texttt{NFix}\ x \longrightarrow (A * x) \\
\texttt{Fair}(A, B) &::= \texttt{NFix}\ x \longrightarrow A\ \texttt{Until}\ (B *{>} (B\ \texttt{Until}\ (A * x)))
\end{aligned}
$$

A.3 Judgement for Types

$$\frac{(x, \omega) \in \Theta}{\Theta \vdash x \text{ type}}$$

$$\frac{}{\Theta \vdash \texttt{Unit/Nat/Bool type}} \qquad \frac{\Theta \vdash A \text{ type}}{\Theta \vdash \texttt{List}(A)/{>}A/\texttt{@}A/\#A \text{ type}} \qquad \frac{\Theta, (x, \texttt{Limit}) \vdash A \text{ type}}{\Theta \vdash \texttt{NFix } x \longrightarrow A \text{ type}}$$

$$\frac{\Theta \vdash A \text{ type} \qquad \Theta \vdash A' \text{ type}}{\Theta \vdash A \ \texttt{Until} \ A'/A + A'/A * A' \text{ type}} \qquad \frac{(x, \omega) \in \Theta \qquad \omega \in \{\texttt{Stable}, \texttt{LimitStable}, \texttt{Comparable}\}}{\Theta \vdash x \text{ stable}}$$

$$\frac{}{\Theta \vdash \texttt{Unit/Nat/Bool}/\#A \text{ Stable}} \qquad \frac{\Theta \vdash A \text{ stable}}{\Theta \vdash \texttt{List}(A) \text{ stable}} \qquad \frac{\Theta \vdash A \text{ stable} \qquad \Theta \vdash A' \text{ stable}}{\Theta \vdash A + A'/A * A' \text{ stable}}$$

$$\frac{(x, \omega) \in \Theta \qquad \omega \in \{\texttt{Limit}, \texttt{LimitStable}, \texttt{Comparable}\}}{\Theta \vdash x \text{ limit}}$$

$$\frac{}{\Theta \vdash \texttt{Unit/Nat/Bool limit}} \qquad \frac{\Theta \vdash A \text{ limit}}{\Theta \vdash \texttt{List}(A)/\texttt{@}A/\#A \text{ limit}} \qquad \frac{\Theta, (x, \texttt{Limit}) \vdash A \text{ limit}}{\Theta \vdash \texttt{NFix } x \longrightarrow A \text{ limit}}$$

$$\frac{\Theta \vdash A \text{ type}^1}{\Theta \vdash {>}A \text{ limit}} \qquad \frac{\Theta \vdash A \text{ type}^1 \qquad \Theta \vdash A' \text{ limit}}{\Theta \vdash A \to A' \text{ limit}} \qquad \frac{\Theta \vdash A \text{ limit} \qquad \Theta \vdash A' \text{ limit}}{\Theta \vdash A + A'/A * A' \text{ limit}}$$

$$\frac{(x, \omega) \in \Theta \qquad \omega \in \{\texttt{Comparable}\}}{\Theta \vdash x \text{ comparable}}$$

$$\frac{}{\Theta \vdash \texttt{Unit/Nat/Bool comparable}} \qquad \frac{\Theta \vdash A \text{ comparable}}{\Theta \vdash \texttt{List}(A) \text{ comparable}}$$

$$\frac{\Theta \vdash A \text{ comparable} \qquad \Theta \vdash A' \text{ comparable}}{\Theta \vdash A + A'/A * A' \text{ comparable}}$$

A.4 Judgement for Context

$$\frac{}{\Theta \vdash \cdot \text{ ctx}} \qquad \frac{\Theta \vdash \Gamma \text{ ctx} \qquad \Theta \vdash A \text{ type}}{\Theta \vdash \Gamma, x : A \text{ ctx}} \qquad \frac{\Theta \vdash \Gamma \text{ ctx} \qquad \# \notin \Gamma}{\Theta \vdash \Gamma, \# \text{ ctx}}$$

$$\frac{\Theta \vdash \Gamma \text{ ctx} \qquad \# \in \Gamma \qquad \text{tick-free}(\Gamma)}{\Theta \vdash \Gamma, \checkmark_{>} \text{ ctx}} \qquad \frac{\Theta \vdash \Gamma \text{ ctx} \qquad \# \in \Gamma \qquad \text{tick-free}(\Gamma)}{\Theta \vdash \Gamma, \checkmark_{@} \text{ ctx}}$$

[1] Not a typo

A.5 Typing Rules

$$\frac{\text{token-free}(\Gamma') \quad \vee \quad \Theta \vdash A \text{ stable}}{\Theta; \Gamma, x : A, \Gamma' \vdash x : A} \qquad \frac{}{\Theta; \Gamma \vdash n : \texttt{Nat}} \qquad \frac{}{\Theta; \Gamma \vdash \texttt{()} : \texttt{Nat}}$$

$$\frac{\begin{array}{c}\Theta; \Gamma, x : A \vdash e : B \\ \text{tick-free}(\Gamma) \\ \Theta \vdash A \text{ type}\end{array}}{\Theta; \Gamma \vdash \texttt{fun}\ x : A \texttt{=>}\ e : B} \qquad \frac{\Theta; \Gamma \vdash e : A \to B \qquad \Theta; \Gamma \vdash e' : A}{\Theta; \Gamma \vdash e\ e' : B}$$

$$\frac{\begin{array}{c}\Theta; \Gamma \vdash e : A \\ \Theta; \Gamma, x : A \vdash e' : B\end{array}}{\Theta; \Gamma \vdash \texttt{let}\ x = e\ \texttt{in}\ e' : B} \qquad \frac{\begin{array}{c}\Theta; \Gamma \vdash e : \texttt{Nat} \\ \Theta; \Gamma \vdash e' : \texttt{Nat} \\ op \in \{\texttt{+}, \texttt{-}, *, /, \%, \texttt{\^{}}\}\end{array}}{\Theta; \Gamma \vdash e\ op\ e' : \texttt{Nat}} \qquad \frac{\Theta; \Gamma \vdash e : \texttt{Nat}}{\Theta; \Gamma \vdash \texttt{suc}\ e : \texttt{Nat}}$$

$$\frac{\begin{array}{c}\Theta; \Gamma \vdash e : \texttt{Nat} \\ \Theta; \Gamma \vdash e_1 : A \\ \Theta; \Gamma, x : \texttt{Nat}, y : A \vdash e_2 : A\end{array}}{\Theta; \Gamma \vdash \texttt{primrec}\ e\ \texttt{with}\ \texttt{|0 =>}\ e_1 \texttt{|suc}\ x, y \texttt{=>}\ e_2 : A}$$

$$\frac{}{\Theta; \Gamma \vdash \texttt{true} : \texttt{Bool}} \qquad \frac{}{\Theta; \Gamma \vdash \texttt{false} : \texttt{Bool}} \qquad \frac{\begin{array}{c}\Theta; \Gamma \vdash e : \texttt{Bool} \\ \Theta; \Gamma \vdash e_1 : A \\ \Theta; \Gamma \vdash e_2 : A\end{array}}{\Theta; \Gamma \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : A}$$

$$\frac{\begin{array}{c}\Theta; \Gamma \vdash e_1 : \texttt{Bool} \\ \Theta; \Gamma \vdash e_2 : \texttt{Bool}\end{array}}{\Theta; \Gamma \vdash e_1\ \texttt{and}\ e_2 : \texttt{Bool}} \qquad \frac{\begin{array}{c}\Theta; \Gamma \vdash e_1 : \texttt{Bool} \\ \Theta; \Gamma \vdash e_2 : \texttt{Bool}\end{array}}{\Theta; \Gamma \vdash e_1\ \texttt{or}\ e_2 : \texttt{Bool}} \qquad \frac{\Theta; \Gamma \vdash e : \texttt{Bool}}{\Theta; \Gamma \vdash \texttt{not}\ e : \texttt{Bool}}$$

$$\frac{\begin{array}{c}\Theta; \Gamma \vdash e_1 : A \\ \Theta; \Gamma \vdash e_2 : A \\ \Theta \vdash A \text{ comparable}\end{array}}{\Theta; \Gamma \vdash e_1 == e_2 : \texttt{Bool}} \qquad \frac{\begin{array}{c}\Theta; \Gamma \vdash e_1 : A \\ \Theta; \Gamma \vdash e_2 : A \\ \Theta \vdash A \text{ comparable}\end{array}}{\Theta; \Gamma \vdash e_1 \texttt{!} = e_2 : \texttt{Bool}}$$

$$\frac{\begin{array}{c}\Theta; \Gamma \vdash e_1 : A \\ \Theta; \Gamma \vdash e_2 : B\end{array}}{\Theta; \Gamma \vdash (e_1, e_2) : A * B} \qquad \frac{\Theta; \Gamma \vdash e : A * B}{\Theta; \Gamma \vdash \texttt{fst}\ e : A} \qquad \frac{\Theta; \Gamma \vdash e : A * B}{\Theta; \Gamma \vdash \texttt{snd}\ e : B}$$

$$\frac{\Theta;\Gamma \vdash e : A \qquad \Theta \vdash A + B \text{ type}}{\Theta;\Gamma \vdash \texttt{inl}\ \ e : A + B : A + B} \qquad \frac{\Theta;\Gamma \vdash e : B \qquad \Theta \vdash A + B \text{ type}}{\Theta;\Gamma \vdash \texttt{inr}\ \ e : A + B : A + B}$$

$$\frac{\begin{array}{c}\Theta;\Gamma \vdash e : A + B \\ \Theta;\Gamma, x : A \vdash e_1 : C \\ \Theta;\Gamma, y : B \vdash e_2 : C\end{array}}{\Theta;\Gamma \vdash \texttt{match}\ \ e\ \ \texttt{with|inl}\ \ x \texttt{=>}\ e_1 \texttt{|inr}\ \ y \texttt{=>}\ e_2 : C}$$

$$\frac{\Theta \vdash \texttt{List}(A) \text{ type}}{\Theta;\Gamma \vdash \texttt{[]} : \texttt{List}(A) : \texttt{List}(A)} \qquad \frac{\Theta;\Gamma \vdash e_i : A \qquad (\forall i = 1, \ldots, n)}{\Theta;\Gamma \vdash [e_1, \ldots, e_n] : \texttt{List}(A)}$$

$$\frac{\begin{array}{c}\Theta;\Gamma \vdash e_1 : A \\ \Theta;\Gamma \vdash e_2 : \texttt{List}(A)\end{array}}{\Theta;\Gamma \vdash e_1 \texttt{::} e_2 : \texttt{List}(A)} \qquad \frac{\begin{array}{c}\Theta;\Gamma \vdash e_1 : \texttt{List}(A) \\ \Theta;\Gamma \vdash e_2 : \texttt{List}(A)\end{array}}{\Theta;\Gamma \vdash e_1 \texttt{++} e_2 : \texttt{List}(A)}$$

$$\frac{\begin{array}{c}\Theta;\Gamma \vdash e : \texttt{List}(A) \\ \Theta;\Gamma \vdash e_1 : B \\ \Theta;\Gamma, x : A, y : \texttt{List}(A), z : B \vdash e_2 : B\end{array}}{\Theta;\Gamma \vdash \texttt{primrec}\ \ e\ \ \texttt{with|[]} \texttt{=>}\ e_1 \texttt{|}x\texttt{::}y, z \texttt{=>}\ e_2 : B}$$

$$\frac{\Theta;\Gamma, \checkmark_{>} \vdash e : A}{\Theta;\Gamma \vdash \texttt{>}e : \texttt{>}A} \qquad \frac{\Theta;\Gamma, \checkmark_{@} \vdash e : A}{\Theta;\Gamma \vdash \texttt{@}e : \texttt{@}A} \qquad \frac{\begin{array}{c}\Theta;\Gamma \vdash e : m\ A \\ m \leq m' \quad \vee \quad \Theta \vdash A \text{ limit}\end{array}}{\Theta;\Gamma, \checkmark_{m'}, \Gamma' \vdash \texttt{<}e : A}\ (\text{where } \texttt{@} \leq\ \texttt{>})$$

$$\frac{\Theta;\Gamma, \text{token-less-stable}(\Gamma') \vdash e : \#A}{\Theta;\Gamma, \#, \Gamma' \vdash \texttt{?}e : A} \qquad \frac{\Theta;\Gamma, \# \vdash e : A}{\Theta;\Gamma \vdash \#e : \#A} \qquad \frac{\begin{array}{c}\Theta;\Gamma, x : \#\texttt{>}A, \# \vdash e : A \\ \Theta \vdash \#A \text{ type}\end{array}}{\Theta;\Gamma \vdash \texttt{nfix}\ \ x : \#A \texttt{=>}\ e : \#A}$$

$$\frac{\begin{array}{c}\Theta;\Gamma \vdash e_1 : A \\ \Theta;\Gamma \vdash e_2 : \texttt{>Str}(A)\end{array}}{\Theta;\Gamma \vdash e_1 \texttt{:::} e_2 : \texttt{Str}(A)} \qquad \frac{\begin{array}{c}\Theta;\Gamma \vdash e_1 : \texttt{Str}(A) \\ \Theta;\Gamma, x : A, y : \texttt{>}(\texttt{Str}(A)) \vdash e_2 : B\end{array}}{\Theta;\Gamma \vdash \texttt{let}\ \ x\texttt{:::}y = e_1\ \ \texttt{in}\ \ e_2 : B}$$

$$\frac{\Theta;\Gamma \vdash e : B}{\Theta;\Gamma \vdash \texttt{now}\ \ e : A\ \ \texttt{Until}\ \ B : A\ \ \texttt{Until}\ \ B} \qquad \frac{\begin{array}{c}\Theta;\Gamma \vdash e_1 : A \\ \Theta;\Gamma \vdash e_2 : \texttt{@}(A\ \ \texttt{Until}\ \ B)\end{array}}{\Theta;\Gamma \vdash \texttt{wait}\ \ e_1\ e_2 : A\ \ \texttt{Until}\ \ B}$$

$$\frac{\begin{array}{c}\Theta;\Gamma, \#, \Gamma' \vdash e : A\ \ \texttt{Until}\ \ B \\ \Theta;\Gamma, \#, \text{token-less-stable}(\Gamma'), x : B \vdash e_1 : C \\ \Theta;\Gamma, \#, \text{token-less-stable}(\Gamma'), x' : A, y : \texttt{@}(A\ \ \texttt{Until}\ \ B), z : \texttt{@}C \vdash e_2 : C\end{array}}{\Theta;\Gamma, \#, \Gamma' \vdash \texttt{urec}\ \ e\ \ \texttt{with|now}\ \ x \texttt{=>}\ e_1 \texttt{|wait}\ \ x'\ y, z \texttt{=>}\ e_2 : C}$$

$$\frac{\begin{array}{c}\Theta;\Gamma \vdash e : A[\texttt{>}(\texttt{NFix}\ \ x \longrightarrow A)/x] \\ \Theta \vdash \texttt{NFix}\ \ x \longrightarrow A \text{ type}\end{array}}{\Theta;\Gamma \vdash \texttt{into}\ \ e : \texttt{NFix}\ \ x \longrightarrow A : \texttt{NFix}\ \ x \longrightarrow A} \qquad \frac{\Theta;\Gamma \vdash e : \texttt{NFix}\ \ x \longrightarrow A}{\Theta;\Gamma \vdash \texttt{out}\ \ e : A[\texttt{>}(\texttt{NFix}\ \ x \longrightarrow A)/x]}$$

A.6 Evaluation Semantics

$$\frac{}{\langle v;\sigma\rangle \Downarrow \langle v;\sigma\rangle} \qquad \frac{\langle e_1;\sigma\rangle \Downarrow \langle \texttt{fun}\ x : A \Rightarrow e_1';\sigma'\rangle \quad \langle e_2;\sigma'\rangle \Downarrow \langle v;\sigma''\rangle \quad \langle e_1'[v/x];\sigma''\rangle \Downarrow \langle v';\sigma'''\rangle}{\langle e_1\ e_2;\sigma\rangle \Downarrow \langle v';\sigma'''\rangle}$$

$$\frac{\langle e_1;\sigma\rangle \Downarrow \langle v;\sigma'\rangle \quad \langle e_2;\sigma'\rangle \Downarrow \langle \texttt{fun}\ x : A \Rightarrow e_2';\sigma''\rangle \quad \langle e_2'[v/x];\sigma''\rangle \Downarrow \langle v';\sigma'''\rangle}{\langle \texttt{let}\ x = e_1\ \texttt{in}\ e_2;\sigma\rangle \Downarrow \langle v';\sigma'''\rangle}$$

$$\frac{\langle e_1;\sigma\rangle \Downarrow \langle n;\sigma'\rangle \quad \langle e_2;\sigma'\rangle \Downarrow \langle m;\sigma''\rangle \quad v = n + m}{\langle e_1 + e_2;\sigma\rangle \Downarrow \langle v;\sigma''\rangle} \qquad \frac{\langle e_1;\sigma\rangle \Downarrow \langle n;\sigma'\rangle \quad \langle e_2;\sigma'\rangle \Downarrow \langle m;\sigma''\rangle \quad v = \max(0, n - m)}{\langle e_1 - e_2;\sigma\rangle \Downarrow \langle v;\sigma''\rangle}$$

$$\frac{\langle e_1;\sigma\rangle \Downarrow \langle n;\sigma'\rangle \quad \langle e_2;\sigma'\rangle \Downarrow \langle m;\sigma''\rangle \quad v = n \times m}{\langle e_1 * e_2;\sigma\rangle \Downarrow \langle v;\sigma''\rangle}$$

$$\frac{\langle e_1;\sigma\rangle \Downarrow \langle n;\sigma'\rangle \quad \langle e_2;\sigma'\rangle \Downarrow \langle m;\sigma''\rangle \quad v = \lfloor n/(\max(1, m))\rfloor}{\langle e_1 * e_2;\sigma\rangle \Downarrow \langle v;\sigma''\rangle} \qquad \frac{\langle e_1;\sigma\rangle \Downarrow \langle n;\sigma'\rangle \quad \langle e_2;\sigma'\rangle \Downarrow \langle m;\sigma''\rangle \quad v = n \bmod (\max(1, m))}{\langle e_1 \% e_2;\sigma\rangle \Downarrow \langle v;\sigma''\rangle}$$

$$\frac{\langle e_1;\sigma\rangle \Downarrow \langle n;\sigma'\rangle \quad \langle e_2;\sigma'\rangle \Downarrow \langle m;\sigma''\rangle \quad v = n\texttt{\^{}}m}{\langle e_1\texttt{\^{}}e_2;\sigma\rangle \Downarrow \langle v;\sigma''\rangle}\ \text{where } 0\texttt{\^{}}0 = 1$$

$$\frac{\langle e;\sigma\rangle \Downarrow \langle n;\sigma'\rangle \quad v = n + 1}{\langle \texttt{suc}\ e;\sigma\rangle \Downarrow \langle v;\sigma'\rangle} \qquad \frac{\langle e;\sigma\rangle \Downarrow \langle 0;\sigma'\rangle \quad \langle e_1;\sigma'\rangle \Downarrow \langle v;\sigma''\rangle}{\langle \texttt{primrec}\ e\ \texttt{with}\ \texttt{|0} \Rightarrow e_1 \texttt{|suc}\ x, y \Rightarrow e_2;\sigma\rangle \Downarrow \langle v;\sigma''\rangle}$$

$$\frac{\langle e;\sigma\rangle \Downarrow \langle n;\sigma'\rangle \quad (\text{with n} \neq 0) \quad \langle \texttt{primrec}\ n - 1\ \texttt{with}\ \texttt{|0} \Rightarrow e_1 \texttt{|suc}\ x, y \Rightarrow e_2;\sigma'\rangle \Downarrow \langle v;\sigma''\rangle \quad \langle e_2[n - 1/x, v/y];\sigma''\rangle \Downarrow \langle v';\sigma'''\rangle}{\langle \texttt{primrec}\ e\ \texttt{with}\ \texttt{|0} \Rightarrow e_1 \texttt{|suc}\ x, y \Rightarrow e_2;\sigma\rangle \Downarrow \langle v';\sigma'''\rangle}$$

$$\frac{\langle e;\sigma\rangle \Downarrow \langle \texttt{true};\sigma'\rangle \quad \langle e_1;\sigma'\rangle \Downarrow \langle v;\sigma''\rangle}{\langle \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2;\sigma\rangle \Downarrow \langle v;\sigma''\rangle} \qquad \frac{\langle e;\sigma\rangle \Downarrow \langle \texttt{false};\sigma'\rangle \quad \langle e_2;\sigma'\rangle \Downarrow \langle v;\sigma''\rangle}{\langle \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2;\sigma\rangle \Downarrow \langle v;\sigma''\rangle}$$

$$\frac{\langle e_1;\sigma\rangle \Downarrow \langle v_1;\sigma'\rangle \quad \langle e_2;\sigma'\rangle \Downarrow \langle v_2;\sigma''\rangle \quad v = v_1 \&\& v_2}{\langle e_1\ \texttt{and}\ e_2;\sigma\rangle \Downarrow \langle v;\sigma''\rangle} \qquad \frac{\langle e_1;\sigma\rangle \Downarrow \langle v_1;\sigma'\rangle \quad \langle e_2;\sigma'\rangle \Downarrow \langle v_2;\sigma''\rangle \quad v = v_1 || v_2}{\langle e_1\ \texttt{or}\ e_2;\sigma\rangle \Downarrow \langle v;\sigma''\rangle}$$

$$\frac{\langle e;\sigma\rangle \Downarrow \langle v;\sigma'\rangle \quad v' = !v}{\langle \texttt{not}\ e;\sigma\rangle \Downarrow \langle v';\sigma'\rangle}$$

$$\frac{\langle e_1;\sigma\rangle \Downarrow \langle v_1;\sigma'\rangle \quad \langle e_2;\sigma'\rangle \Downarrow \langle v_2;\sigma''\rangle \quad v_1 = v_2}{\langle e_1 == e_2;\sigma\rangle \Downarrow \langle \texttt{true};\sigma''\rangle} \qquad \frac{\langle e_1;\sigma\rangle \Downarrow \langle v_1;\sigma'\rangle \quad \langle e_2;\sigma'\rangle \Downarrow \langle v_2;\sigma''\rangle \quad v_1 \neq v_2}{\langle e_1 == e_2;\sigma\rangle \Downarrow \langle \texttt{false};\sigma''\rangle}$$

$$\frac{\langle \texttt{not}\ (e_1 == e_2);\sigma\rangle \Downarrow \langle v;\sigma'\rangle}{\langle e_1 != e_2;\sigma\rangle \Downarrow \langle v;\sigma'\rangle}$$

$$\frac{\langle e_1;\sigma\rangle \Downarrow \langle v_1;\sigma'\rangle \quad \langle e_2;\sigma'\rangle \Downarrow \langle v_2;\sigma''\rangle}{\langle (e_1,e_2);\sigma\rangle \Downarrow \langle (v_1,v_2);\sigma''\rangle} \qquad \frac{\langle e;\sigma\rangle \Downarrow \langle (v_1,v_2);\sigma'\rangle}{\langle \texttt{fst}\ e;\sigma\rangle \Downarrow \langle v_1;\sigma'\rangle} \qquad \frac{\langle e;\sigma\rangle \Downarrow \langle (v_1,v_2);\sigma'\rangle}{\langle \texttt{snd}\ e;\sigma\rangle \Downarrow \langle v_2;\sigma'\rangle}$$

$$\frac{\langle e;\sigma\rangle \Downarrow \langle v;\sigma'\rangle}{\langle \texttt{inl}\ e : A;\sigma\rangle \Downarrow \langle \texttt{inl}\ v : A;\sigma'\rangle} \qquad \frac{\langle e;\sigma\rangle \Downarrow \langle v;\sigma'\rangle}{\langle \texttt{inr}\ e : A;\sigma\rangle \Downarrow \langle \texttt{inr}\ v : A;\sigma'\rangle}$$

$$\frac{\langle e;\sigma\rangle \Downarrow \langle \texttt{inl}\ v : A;\sigma'\rangle \quad \langle e_1[v/x];\sigma'\rangle \Downarrow \langle v';\sigma''\rangle}{\langle \texttt{match}\ e\ \texttt{with|inl}\ x \texttt{=>}\ e_1\texttt{|inr}\ y \texttt{=>}\ e_2;\sigma\rangle \Downarrow \langle v';\sigma''\rangle}$$

$$\frac{\langle e;\sigma\rangle \Downarrow \langle \texttt{inr}\ v : A;\sigma'\rangle \quad \langle e_2[v/y];\sigma'\rangle \Downarrow \langle v';\sigma''\rangle}{\langle \texttt{match}\ e\ \texttt{with|inl}\ x \texttt{=>}\ e_1\texttt{|inr}\ y \texttt{=>}\ e_2;\sigma\rangle \Downarrow \langle v';\sigma''\rangle}$$

$$\frac{\langle e_i;\sigma_{i-1}\rangle \Downarrow \langle v_i;\sigma_i\rangle \qquad (\forall i = 1,\ldots,n)}{\langle [e_1,\ldots,e_n];\sigma_0\rangle \Downarrow \langle [v_1,\ldots,v_n];\sigma_n\rangle} \qquad \frac{\langle e_1;\sigma\rangle \Downarrow \langle v;\sigma'\rangle \quad \langle e_2;\sigma'\rangle \Downarrow \langle [v_1,\ldots,v_n];\sigma''\rangle}{\langle e_1\texttt{::}e_2;\sigma\rangle \Downarrow \langle [v,v_1,\ldots,v_n];\sigma''\rangle}$$

$$\frac{\langle e_1;\sigma\rangle \Downarrow \langle [v_1,\ldots,v_n];\sigma'\rangle \quad \langle e_2;\sigma'\rangle \Downarrow \langle [w_1,\ldots,w_m];\sigma''\rangle}{\langle e_1\texttt{++}e_2;\sigma\rangle \Downarrow \langle [v_1,\ldots,v_n,w_1,\ldots,w_m];\sigma''\rangle}$$

$$\frac{\langle e;\sigma\rangle \Downarrow \langle [] : A;\sigma'\rangle \quad \langle e_1;\sigma'\rangle \Downarrow \langle v;\sigma''\rangle}{\langle \texttt{primrec}\ e\ \texttt{with}\ \texttt{|[]} \texttt{=>}\ e_1\texttt{|}x\texttt{::}y, z \texttt{=>}\ e_2;\sigma\rangle \Downarrow \langle v;\sigma''\rangle}$$

$$\frac{\begin{array}{c}\langle e;\sigma\rangle \Downarrow \langle [v_1,\ldots,v_n];\sigma'\rangle \qquad (\text{with n} \neq 0)\\ \langle \texttt{primrec}\ [v_2,\ldots,v_n]\ \texttt{with}\ \texttt{|[]} \texttt{=>}\ e_1\texttt{|}x\texttt{::}y, z \texttt{=>}\ e_2;\sigma'\rangle \Downarrow \langle v;\sigma''\rangle\\ \langle e_2[v_1/x,[v_2,\ldots,v_n]/y,v/z];\sigma''\rangle \Downarrow \langle v';\sigma'''\rangle\end{array}}{\langle \texttt{primrec}\ e\ \texttt{with}\ \texttt{|[]} \texttt{=>}\ e_1\texttt{|}x\texttt{::}y, z \texttt{=>}\ e_2;\sigma\rangle \Downarrow \langle v;\sigma'''\rangle}$$

$$\frac{l = \text{alloc}(\sigma) \quad \sigma \neq \cdot}{\langle >e; \sigma\rangle \Downarrow \langle l; \sigma + \{l \mapsto e\}\rangle} \qquad \frac{l = \text{alloc}(\sigma) \quad \sigma \neq \cdot}{\langle @e; \sigma\rangle \Downarrow \langle l; \sigma + l \mapsto e\rangle} \qquad \frac{\langle e; \eta_N\rangle \Downarrow \langle l; \eta'_N\rangle \quad \langle \eta'_N(l); (\eta'_N \checkmark \eta_L)\rangle \Downarrow \langle v; \sigma'\rangle}{\langle <e; (\eta_N \checkmark \eta_L)\rangle \Downarrow \langle v; \sigma'\rangle}$$

$$\frac{\langle e_1; \sigma\rangle \Downarrow \langle v_1; \sigma'\rangle \quad \langle e_2; \sigma'\rangle \Downarrow \langle v_2; \sigma''\rangle}{\langle e_1 \texttt{:::} e_2; \sigma\rangle \Downarrow \langle \texttt{into}\ (v_1, v_2) : A; \sigma''\rangle}\ [1]$$

$$\frac{\langle e_1; \sigma\rangle \Downarrow \langle v_1; \sigma'\rangle \quad \langle e_2[\texttt{fst(out}\ v_1)/x, \texttt{snd(out}\ v_1)/y]; \sigma'\rangle \Downarrow \langle v_2; \sigma''\rangle}{\langle \texttt{let}\ x\texttt{:::}y = e_1\ \texttt{in}\ e_2; \sigma\rangle \Downarrow \langle v_2; \sigma''\rangle}$$

$$\frac{\langle e; \cdot\rangle \Downarrow \langle \#e'; \cdot\rangle \quad \langle e'; \sigma\rangle \Downarrow \langle v; \sigma'\rangle \quad \sigma \neq \cdot}{\langle ?e; \sigma\rangle \Downarrow \langle v; \sigma'\rangle} \qquad \frac{\langle e; \cdot\rangle \Downarrow \langle \texttt{nfix}\ x : A \texttt{=>}\ e'; \cdot\rangle \quad \langle e'[\#(>(?\texttt{nfix}\ x : A \texttt{=>}\ e'))/x]; \sigma\rangle \Downarrow \langle v; \sigma'\rangle \quad \sigma \neq \cdot}{\langle ?e; \sigma\rangle \Downarrow \langle v; \sigma'\rangle}$$

$$\frac{\langle e; \sigma\rangle \Downarrow \langle v; \sigma'\rangle}{\langle \texttt{now}\ e; \sigma\rangle \Downarrow \langle \texttt{now}\ v; \sigma'\rangle} \qquad \frac{\langle e_1; \sigma\rangle \Downarrow \langle v_1; \sigma'\rangle \quad \langle e_2; \sigma'\rangle \Downarrow \langle v_2; \sigma''\rangle}{\langle \texttt{wait}\ e_1\ e_2; \sigma\rangle \Downarrow \langle \texttt{wait}\ v_1\ v_2; \sigma''\rangle}$$

$$\frac{\langle e; \sigma\rangle \Downarrow \langle \texttt{now}\ v; \sigma'\rangle \quad \langle e_1[v/x]; \sigma'\rangle \Downarrow \langle v'; \sigma''\rangle}{\langle \texttt{urec}\ e\ \texttt{with|now}\ x \texttt{=>}\ e_1\texttt{|wait}\ x'\ y, z \texttt{=>}\ e_2; \sigma\rangle \Downarrow \langle v'; \sigma''\rangle}$$

$$\frac{\langle e; \sigma\rangle \Downarrow \langle \texttt{wait}\ v_1\ v_2; \sigma'\rangle \quad l = \text{alloc}(\sigma') \quad \langle e_2[v_1/x', v_2/y, l/z]; \sigma' + \{l \mapsto (\texttt{urec}\ <v_2\ \texttt{with|now}\ x \texttt{=>}\ e_1\texttt{|wait}\ x'\ y, z \texttt{=>}\ e_2)\}\rangle \Downarrow \langle v'; \sigma''\rangle}{\langle \texttt{urec}\ e\ \texttt{with|now}\ x \texttt{=>}\ e_1\texttt{|wait}\ x'\ y, z \texttt{=>}\ e_2; \sigma\rangle \Downarrow \langle v'; \sigma''\rangle}$$

$$\frac{\langle e; \sigma\rangle \Downarrow \langle v; \sigma'\rangle}{\langle \texttt{into}\ e; \sigma\rangle \Downarrow \langle \texttt{into}\ v; \sigma'\rangle} \qquad \frac{\langle e; \sigma\rangle \Downarrow \langle \texttt{into}\ v; \sigma'\rangle}{\langle \texttt{out}\ e; \sigma\rangle \Downarrow \langle v; \sigma'\rangle}$$

[1] Type ascription A not important

A.7 Step Semantics

$$\frac{\langle e;\eta\checkmark\rangle \Downarrow \langle v :: w;\eta_N\checkmark\eta_L\rangle}{\langle e;\eta\rangle \overset{v}{\Longrightarrow}_{\texttt{Safe}} \langle <w;\eta_L\rangle}$$

$$\frac{\langle e;\eta\checkmark\rangle \Downarrow \langle \texttt{wait}\ \ v\ w;\eta_N\checkmark\eta_L\rangle}{\langle e;\eta\rangle \overset{v}{\Longrightarrow}_{\texttt{Lively}} \langle <w;\eta_L\rangle} \qquad \frac{\langle e;\eta\checkmark\rangle \Downarrow \langle \texttt{now}\ \ v;\eta_N\checkmark\eta_L\rangle}{\langle e;\eta\rangle \overset{v}{\Longrightarrow}_{\texttt{Lively}} \langle \texttt{HALT};\eta_L\rangle}$$

$$\frac{\langle e;\eta\rangle \overset{v}{\Longrightarrow}_{\texttt{Lively}} \langle e';\eta'\rangle}{\langle e;\eta;1\rangle \overset{\texttt{inl}\ v}{\Longrightarrow}_{\texttt{Fair}} \langle e';\eta';1\rangle} \qquad \frac{\langle e;\eta\rangle \overset{v}{\Longrightarrow}_{\texttt{Lively}} \langle e';\eta'\rangle}{\langle e;\eta;2\rangle \overset{\texttt{inr}\ v}{\Longrightarrow}_{\texttt{Fair}} \langle e';\eta';2\rangle}$$

$$\frac{\langle e;\eta\rangle \overset{(v,w)}{\Longrightarrow}_{\texttt{Lively}} \langle \texttt{HALT};\eta'\rangle}{\langle e;\eta;1\rangle \overset{\texttt{inr}\ v}{\Longrightarrow}_{\texttt{Fair}} \langle <w;\eta';2\rangle} \qquad \frac{\langle e;\eta\rangle \overset{(v,w)}{\Longrightarrow}_{\texttt{Lively}} \langle \texttt{HALT};\eta'\rangle}{\langle e;\eta;2\rangle \overset{\texttt{inl}\ v}{\Longrightarrow}_{\texttt{Fair}} \langle \texttt{out}(<w);\eta';1\rangle}$$

$$\frac{\langle e;(\eta+\{l\mapsto v :: l'\})\checkmark\{l'\mapsto ()\}\rangle \Downarrow \langle v' :: w;\eta_N\checkmark(\eta_L+\{l'\mapsto ()\})\rangle \quad l' = \mathrm{alloc}(\eta\checkmark)}{\langle e;\eta;l\rangle \overset{v/v'}{\Longrightarrow}_{\texttt{ISafe}} \langle <w;\eta_L;l'\rangle}$$

$$\frac{\langle e;(\eta+\{l\mapsto v :: l'\})\checkmark\{l'\mapsto ()\}\rangle \Downarrow \langle \texttt{wait}\ \ v'\ w;\eta_N\checkmark(\eta_L+\{l'\mapsto ()\})\rangle \quad l' = \mathrm{alloc}(\eta\checkmark)}{\langle e;\eta;l\rangle \overset{v/v'}{\Longrightarrow}_{\texttt{ILively}} \langle <w;\eta_L;l'\rangle}$$

$$\frac{\langle e;(\eta+\{l\mapsto v :: l'\})\checkmark\{l'\mapsto ()\}\rangle \Downarrow \langle \texttt{now}\ \ v';\eta_N\checkmark(\eta_L+\{l'\mapsto ()\})\rangle \quad l' = \mathrm{alloc}(\eta\checkmark)}{\langle e;\eta;l\rangle \overset{v/v'}{\Longrightarrow}_{\texttt{ILively}} \langle \texttt{HALT};\eta_L;l'\rangle}$$

$$\frac{\langle e;\eta;l\rangle \overset{v/v'}{\Longrightarrow}_{\texttt{ILively}} \langle e';\eta';l'\rangle}{\langle e;\eta;l;1\rangle \overset{v/\texttt{inl}\ v'}{\Longrightarrow}_{\texttt{IFair}} \langle e';\eta';l';1\rangle} \qquad \frac{\langle e;\eta;l\rangle \overset{v/v'}{\Longrightarrow}_{\texttt{ILively}} \langle e';\eta';l'\rangle}{\langle e;\eta;l;2\rangle \overset{v/\texttt{inr}\ v'}{\Longrightarrow}_{\texttt{IFair}} \langle e';\eta';l';2\rangle}$$

$$\frac{\langle e;\eta;l\rangle \overset{v/(v',w)}{\Longrightarrow}_{\texttt{ILively}} \langle \texttt{HALT};\eta';l'\rangle}{\langle e;\eta;l;1\rangle \overset{v/\texttt{inr}\ v'}{\Longrightarrow}_{\texttt{IFair}} \langle <w;\eta';l';2\rangle} \qquad \frac{\langle e;\eta;l\rangle \overset{v/(v',w)}{\Longrightarrow}_{\texttt{ILively}} \langle \texttt{HALT};\eta';l'\rangle}{\langle e;\eta;l;2\rangle \overset{v/\texttt{inr}\ v'}{\Longrightarrow}_{\texttt{IFair}} \langle \texttt{out}(<w);\eta';l';1\rangle}$$

A.8 Fundamental Theorems of Eva

A.8.1 Safe Interpreter

If $\cdot;\cdot \vdash e : \#\texttt{Str}(A)$, then there is an infinite sequence of reduction steps:

$$\langle ?e;\emptyset\rangle \overset{v_1}{\Longrightarrow}_{\texttt{Safe}} \langle e_1;\eta_1\rangle \overset{v_2}{\Longrightarrow}_{\texttt{Safe}} \langle e_2;\eta_2\rangle \overset{v_3}{\Longrightarrow}_{\texttt{Safe}} \cdots$$

Moreover, if A is a value type, then $\cdot;\cdot \vdash v_i : A$ for all $i \geq 1$.

A.8.2 Lively Interpreter

If $\cdot;\cdot \vdash e : \#(A\ \texttt{Until}\ B)$, then there is a finite sequence of reduction steps:

$$\langle ?e;\emptyset\rangle \overset{v_1}{\Longrightarrow}_{\texttt{Lively}} \langle e_1;\eta_1\rangle \overset{v_2}{\Longrightarrow}_{\texttt{Lively}} \langle e_2;\eta_2\rangle \overset{v_3}{\Longrightarrow}_{\texttt{Lively}} \cdots \overset{v_n}{\Longrightarrow}_{\texttt{Lively}} \langle \texttt{HALT};\eta_n\rangle$$

Moreover, if A and B are value types, then $\cdot;\cdot \vdash v_i : A$ for all $0 < i < n$, and $\cdot;\cdot \vdash v_n : B$.

A.8.3 Fair Interpreter

If $\cdot;\cdot \vdash e : \#\texttt{Fair}(A, B)$, then there is an infinite sequence of reduction steps:

$$\langle \texttt{out}(?e);\emptyset;1\rangle \overset{v_1}{\Longrightarrow}_{\texttt{Fair}} \langle e_1;\eta_1;p_1\rangle \overset{v_2}{\Longrightarrow}_{\texttt{Fair}} \langle e_2;\eta_2;p_2\rangle \overset{v_3}{\Longrightarrow}_{\texttt{Fair}} \cdots$$

such that for each $p \in \{1,2\}$, we have $p_i = p$ for infinitely many $i \geq 1$. Moreover, if A and B are value types, then $\cdot;\cdot \vdash v_i : A + B$ for all $i \geq 1$.

A.8.4 ISafe Interpreter

If $\cdot;\cdot \vdash e : \#(\texttt{Str}(A) \to \texttt{Str}(B))$, then there is an infinite sequence of reduction steps:

$$\langle (?e)\ (<l_0);\emptyset;l_0\rangle \overset{v_1/v_1'}{\Longrightarrow}_{\texttt{ISafe}} \langle e_1;\eta_1;l_1\rangle \overset{v_2/v_2'}{\Longrightarrow}_{\texttt{ISafe}} \langle e_2;\eta_2;l_2\rangle \overset{v_3/v_3'}{\Longrightarrow}_{\texttt{ISafe}} \cdots$$

Moreover, if B is a value type, then $\cdot;\cdot \vdash v_i' : B$ for all $i \geq 1$.

A.8.5 ILively Interpreter

If $\cdot;\cdot \vdash e : \#(\texttt{Str}(A) \to (B\ \texttt{Until}\ C))$, then there is a finite sequence of reduction steps:

$$\langle (?e)\ (<l_0);\emptyset;l_0\rangle \overset{v_1/v_1'}{\Longrightarrow}_{\texttt{ILively}} \langle e_1;\eta_1;l_1\rangle \overset{v_2/v_2'}{\Longrightarrow}_{\texttt{ILively}} \langle e_2;\eta_2;l_2\rangle \overset{v_3/v_3'}{\Longrightarrow}_{\texttt{ILively}} \cdots \overset{v_n/v_n'}{\Longrightarrow}_{\texttt{ILively}} \langle \texttt{HALT};\eta_n;l_n\rangle$$

Moreover, if B and C are value types, then $\cdot;\cdot \vdash v_i' : B$ for all $0 < i < n$, and $\cdot;\cdot \vdash v_n' : C$.

A.8.6 IFair Interpreter

If $\cdot;\cdot \vdash e : \#(\texttt{Str}(A) \to \texttt{Fair}(B, C))$, then there is an infinite sequence of reduction steps:

$$\langle \texttt{out}((?e)\ (<l_0));\emptyset;l_0;1\rangle \overset{v_1/v_1'}{\Longrightarrow}_{\texttt{IFair}} \langle e_1;\eta_1;l_1;p_1\rangle \overset{v_2/v_2'}{\Longrightarrow}_{\texttt{IFair}} \langle e_2;\eta_2;l_2;p_2\rangle \overset{v_3/v_3'}{\Longrightarrow}_{\texttt{IFair}} \cdots$$

such that for each $p \in \{1,2\}$, we have $p_i = p$ for infinitely many $i \geq 1$. Moreover, if B and C are value types, then $\cdot;\cdot \vdash v_i' : B + C$ for all $i \geq 1$.

Appendix B

Eva Code Samples

B.1 Ackermann Function

```
def ackermann # m:Nat n:Nat =
  let iter f:Nat -> Nat n':Nat =
    (primrec n' with
      | 0 => f 1
      | suc _, rest => f rest
    ) in
  (primrec m with
    | 0 => (fun x:Nat => suc x)
    | suc _, rest => iter rest) n
```

B.2 Quicksort Function

```
def length{a} # xs:List(a) =
  primrec xs with
    | [] => 0
    | _::_, rest => 1+rest


def leq # x:Nat y:Nat =
  (x - y) == 0


def filter{a} # f:(a->Bool) xs:List(a) =
  primrec xs with
    | [] => []:List(a)
    | x::_, rest =>
      if f x
        then x::rest
        else rest


def partition{a} # f:(a->Bool) xs:List(a) =
  (?filter{a} f xs,
  ?filter{a} (fun k:a => not (f k)) xs)


def quickSortHelper # x:Nat l:List(Nat) =
  let f n:Nat = n `?leq` x in
  ?partition{Nat} f l


def quickSort # l:List(Nat) =
  let len = ?length{Nat} l in
  let q n:Nat= (
    primrec n with
      | 0 => (fun l:List(Nat) => []:List(Nat))
      | suc _, y =>
        fun l:List(Nat) =>
          primrec l with
            | [] => []:List(Nat)
            | u::v, _ =>
              let (small,large) =
                ?quickSortHelper u v in
              let one = y small in
              let two = u::y large in
              one++two
  ) in
  (q len) l
```

Appendix C

Project Proposal

Type Systems for Functional Reactive Programming

2378F

Project Originators: 2378F and Alan Mycroft

Project Supervisors: Alan Mycroft

Director of Studies: John Fawcett

Overseers: Robert Mullins and Marcelo Fiore

Introduction

Functional Reactive Programming (FRP) is a common programming paradigm for implementing asynchronous dataflow models, such as graphical user interfaces and control software in vehicles. FRP languages aim to provide a high level of abstraction while allowing the program to be run efficiently at the hardware level after compilation. They also bring known benefits of functional programming to reactive programming, such as being less error-prone and easier to reason about.

In most FRP languages, signals (time-varying values) are represented in the form of infinite lazy-lists called streams, where elements of the list denotes how the value changes in different time steps, with the head of the list specifically denoting value in the current time step. However, valid programs that operate on streams do not make sense when we interpret the streams as signals. Consider the following three functions in pseudo code, all three taking a signal in the form of a stream as input and returning one as output:

```
f x = 0::x
g x = x
h (x::xs) = xs
```

A traditional ML-like type system would type check and accept all three functions. `f` is a function that delays a stream by one time step by padding the beginning with a 0, `g` is the identity function, and `h` advances the stream by one time step.

However, when we interpret the stream as a signal, `h` is be considered invalid as it is non-causal and non-generative. Here, causal means that current behavior cannot depend on future inputs and generative means that a value is eventually produced at each time step as long as the program has not halted. At each time step, `h` is unable to produce an output value at each time step without knowing the future input value in the next time step, thus representing a non-implementable reactive function.

Ideally, we want a type system for FRP languages that not only identifies a subset of valid programs (so we can only implement classically well-typed programs that are causal and progressive), but also allows us to reason about the associated semantics properties of programs based on its type. The latter condition is useful as reasoning about safety and liveness properties of a reactive system is often critical to verifying its correctness.

Krishnaswami [Kri13] proposed a FRP calculus that ensures productivity (a type of safety property) by introducing modal operators for guarded recursion to the type system. Cave et al. [CFPP14] then proposed a similar FRP calculus that satisfies liveness properties (like fairness) by introducing operators from linear temporal logic. However, the operators introduced in both calculus are not compatible with each other, and designing a FRP calculus with both safety and liveness guarantees is non-trivial.

Recently, Bahr [BGM21] proposed a theoretical calculus, Lively RaTT, that succeeded in combining the operators from the two above logic systems, by considering one to be a sub-modality of the other. Its semantics rules out non-causal programs by construction and asserts safety and liveness properties of a type-checked program with respect to its type. Depending on the type of a program, RaTT can either directly evaluates it into a value in a single time step (evaluation semantics) or performs a series of computation over a (possibly infinite) sequence of time steps (step semantics). Bahr also showed that RaTT could be implemented with a two heap-model (where time-based values are allocated in one of the two heaps interchangeably in each time step, and garbage-collected immediately after two time-steps), thus eliminating implicit space leaks by construction.

In this project, I will design and implement a FRP language, provisionally dubbed Eva. Eva will be a programming variant of RaTT, but her syntax may be refined to simplify the type checking algorithm and provide a more convenient programming experience. The core deliverables include implementing the parser, type-checker and abstract syntax tree (AST) interpreter. All these tools will be implemented in Haskell, but Eva's syntax will use features from both Haskell's and OCaml's syntax. For evaluation, I will consider Eva's soundness, expressiveness and runtime efficiency.

Starting Point

I have no prior experience with implementing parsers, type-checkers, interpreters or functional programming languages. I have studied Semantics of Programming Languages, Compiler Construction and Computation Theory, which will be relevant in this project. I have read relevant papers on RaTT-like calculi during the summer though no project code was written before the start of Michaelmas term.

Success Criteria

1. Design the syntax and semantics of Eva, taking inspiration from Lively RaTT
2. Implement the parser, type-checker and AST interpreter with the two-heap model
3. Evaluate the soundness, expressiveness and runtime efficiency of Eva

Evaluation

The nature of this project does not lend itself to quantitative evaluation easily. Instead, a collection of qualitative evaluation will be used to evaluate multiple aspects of Eva.

1. Guarantees Evaluation: I will test Eva against a corpus of programs found in relevant academic papers, Haskell stream libraries or handcrafted myself. I will present a collection of type-valid programs based on safety and liveness properties together with a suite of erroneous programs that violates safety and liveness properties and evaluate whether they are accepted by the type system.
2. Expressiveness Evaluation: I will evaluate the expressiveness of Eva's computability power by implementing programs of various computability power, e.g. Turing-complete or primitive recursive. I will also investigate Eva's limitations by considering whether there are natural programs outside its scope.
3. Leaks Evaluation: I will present a suite of programs with space and time leaks obtained from relevant academic papers or handcrafted myself, testing whether or not they are accepted by the type checker and, if accepted, investigate their run-time behaviour.

Possible Extensions

1. Add syntactic sugar
2. Provide support for modular programming
3. Provide support for streams representing interactive IO
4. Enhance the type system e.g. enriching the rules of the type system, implementing type inference
5. Implement a compiler that compiles a subset of the language to a netlist representation

Resource Declaration

I will be mainly using my personal laptop (Xiaomi Notebook Pro – 1.80 GHz, 16 GB RAM) for software development. I will be using the Computing Service's MCS as backup in case my laptop breaks down. In addition, I will also perform daily backups with Git version control and Google Drive.

Timeline and Milestone

Week 1–2 (21st Oct – 3rd Nov):

Design the syntax and semantics for Eva based on Lively RaTT

Practice using Haskell by writing small programs

Week 3–4 (4th Nov – 17th Nov):

Implement the parser

Milestone: Given a text file of a valid Eva program, generate the AST

Week 5–6 (18th Nov – 1st Dec):

Implement the type checker

Milestone: ASTs can be type-checked in a standard recursive-directed manner

Week 7 – 8 (2nd Dec – 15th Dec):

Implement the interpreter with a two-heap model

Milestone: Eva programs can be run using the interpreter

Week 9–10 (16th Dec – 29th Dec):

Prepare test cases for the project evaluation

Milestone: Implement programs for all three subsections of the evaluation

Week 11 – 12 (30th Dec – 12th Jan):

Slack time.

Finish core deliverables

Milestone: Pass the success criteria

Week 13–14 (13th Jan – 26th Jan):

Work on extensions

Week 15–16 (27th Jan – 9th Feb):

Continue working on extensions

Write up section 1 and 2 of the dissertation

Write up progress report and prepare presentation

Milestone(3rd Feb): Submit progress report

Milestone: Finish Introduction and Preparation chapter

Week 17–18 (10th Feb – 23rd Feb):

Continue working on extensions

Write up section 3 of the dissertation

Milestone: Finish Implementation chapter

Week 19–20 (24th Feb – 9th Mar):

Continue working on extensions

Write up section 4 of the dissertation

Milestone: Finish evaluation chapter

Week 21–22 (10th Mar – 23rd Mar):

Slack time

Week 23–24 (24th Mar – 6th Apr):

Finish Dissertation and send to supervisor for feedback

Milestone (1st April): Submit first draft of the dissertation to supervisor and DoS

Week 25–26 (7th Apr – 20th Apr):

Wait for supervisor and DoS' comments and make corrections

Milestone: Submit second draft of the dissertation to supervisor and DoS

Week 27–29 (21st Apr – 11th May):

Make final corrections and submit dissertation

Milestone: Submit final dissertation

References

[BGM21] Patrick Bahr, Christian Uldal Graulund, and Rasmus Ejlers Møgelberg. Diamonds are not forever: Liveness in reactive programming with guarded recursion. *Proc. ACM Program. Lang.*, 5(POPL), January 2021.

[CFPP14] Andrew Cave, Francisco Ferreira, Prakash Panangaden, and Brigitte Pientka. Fair reactive programming. In *Proceedings of the 41st ACM*

SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, page 361–372, New York, NY, USA, 2014. Association for Computing Machinery.

[Kri13] Neelakantan R. Krishnaswami. Higher-order reactive programming without spacetime leaks. In *International Conference on Functional Programming (ICFP)*, September 2013.