Implementing a Standalone Language for Asynchronous Modal FRP

Alexander Berg alebe@itu.dk

Emil Jäpelt emja@itu.dk

Date: 03-06-2024
Course code: KISPECI1SE

Abstract

This thesis presents an implementation of a programming language based on the Async RaTT calculus the theory of which is defined by Møgelberg and Bahr [2]. The language has previously been implemented as an embedded language in Haskell but has never been implemented as a standalone language. The paper describes such a language implementation with type inference, transpilation to OCaml, and gives several examples of programs written in Async RaTT. Thorough testing and measurements of memory usage indicates that the guarantees of no implicit space-leak, causality, and productivity are present.

1 Introduction

Functional Reactive Programming (FRP) is a programming paradigm first described in 1997 [4] which combines functional programming and reactive programming resulting in declarative programming of reactive systems. These reactive systems in question are high-level abstractions of computations which use a number of input channels. Such a system could be a robot with sensors or a graphical user interfaces with input fields.

Among a handful of suggested languages within this paradigm Async RaTT [2] stands out by having an operational semantics that allows output channels to be dependent on multiple asynchronous input channels. These channels may change at runtime. The operational semantics have additionally been formally proven. However, the language has prior to this project only been implemented as an embedded language within Haskell [1].

This paper introduces an implementation of Async RaTT as a standalone programming language. In section 2 the basic concepts behind, and features of, the language are explained. This is followed by a more in-depth showcase of all of the available terms, types, and rules of the implementation.

Section 3 goes through the implemented language infrastructure, including an introduction of how one can utilize the implementations for their own use cases.

Section 4 explains how the different types of inference have been implemented. This includes type-, clock-, and stable type inference.

Section 5 explains the interpreter. This includes explanations of how both the operational and reactive semantics are implemented, in regards to both the functional implementation and the data structures.

Finally, section 6 explains how Async RaTT is transpiled to OCaml.

The source code of the implementation can be found at `https://github.com/bueskyd/AsyncRaTT`.

2 Language Introduction

Programs written in Async RaTT work by reacting to updates in an environment which, for example, could be a computer with a clock, keyboard, and console, or a robot with sensors and actuators. Essentially, a program will wait for these updates, and then produce some effect once they occur. Both of these actions work on the concept of channels.

Input channels are all the channels from which the program can receive information. These channels can be either `push-only` channels on which a program must wait for an update to occur, `buffered-only` channels from which the program can always read the newest value, or `push-buffered` that combines `push-only` and `buffered-only` channels.
As an example, the keyboard of a computer could be modelled as a push channel which produces a value each time a key is pressed. It could also be buffered if said value is saved such that the latest value is always accessible. A battery sensor on a robot might be a buffered channel from which the charge percentage could be read.

Output channels are all the channels on which the program can produce an effect by writing a value to it. In this implementation these channels can be typed, requiring a certain type of value, or untyped, in which case the output can handle any type of value.
The console of a computer could be an untyped output channel, and a DC motor of a robot might be typed strictly require a number describing how fast and in which direction to rotate.

The implementation includes type inference, explained further in section 4. This means that in most places where a type is needed it can optionally be left out resulting in less verbose source code without compromising on type safety.

2.1 Program Structure

```
let fst = \(x,_) -> x;

type 'a option =
    | Some of 'a
    | None;

type int_option = int option;

type int_tuple = int * int;

let rec key_handle = delay(Some(adv(wait keyboard)) :: key_handle);

console <- None :: key_handle;
```

Listing 1: Example program

In the concrete syntax of the language implementation you can do three things. Define types which includes type aliases and algebraic data types (ADTs), define bindings via let-expressions, and bind signals to output channels. All of these top-level language elements are terminated with a semicolon.

As an example the program in listing 1 does the following:

1. Defines a function returning the first element of a pair.
2. Defines the polymorphic algebraic data type 'option'.
3. Defines a type alias for options of integers.
4. Defines a type alias for pairs of integers.
5. Binds the name 'key_handle' to a delayed signal over the keyboard.
6. Binds a signal to the output channel called `console`.

For a program to actually do anything there must be at least one binding of a signal to an output channel.

Generally, a program is written by defining some delayed signals by using `delay`, the signal construction operator, ::, with the left argument being a value of type `'a`, and the right argument being a delayed signal over values of type `'a`.

Such a delayed signal can then be bound to an output channel. The output channel must be given an initial value. In listing 1 this is done by signal construction. In this case the initial value is `None` representing that initially, no key has been pressed.

2.2 Types

The language implementation comes with a handful of built-in types and the mentioned capability of type definitions. An important note about types in Async RaTT is that some types are considered to be stable. If a type is stable then values of this type will be allowed to be moved into the future. The importance and rules of this is explained in further detail in section 4.1.1. Table 1 gives an overview of what types are available and how they are written in the concrete syntax.

Name	Syntax
Unit	`unit`
Integer	`int`
Boolean	`bool`
String	`string`
Tuple	t_0 `*` t_1
Function	t_0 `->` t_1
Delayed	t `later`
Stable	t `boxed`
Signal	t `signal`
Named	*name*
Named	(`(`t_0`, ..., `t_n`)` *name*

Table 1: Type Overview

The implementation provides a handful of common types namely `unit`, `int`, `bool`, and `string`. All of these are considered to be stable types. Naturally, types from the theory of the language

are also included. Although, they have been given a concrete syntax for the sake of readability and usability. `'a later` is the syntax for the modal type $\textcircled{\exists}\,\text{`}a$, and `'a boxed` is the syntax for the modal type $\Box\text{`}a$. The type `'a signal` is essentially a recursive type, which could be written as `'a * 'a signal later`.

2.2.1 Built-in ADTs

Two built-in ADTs are provided, namely `selection` and `option`.

selection relates to the `select` term described further in section 2.3.2. It encodes the possibility that out of two delayed computations one may complete before the other or both may complete at the same time.

```
type 'a 'b selection =
    | Left of 'a * 'b later
    | Rigth of 'a later * 'b
    | Both of 'a * 'b;
```

Listing 2: Definition of `selection`

option is a common type, representing the possible lack of a value. Similar data types are found in many languages of differing paradigms including OCaml, F#, C++, Scala, and Haskell. It is included here, only such that we won't have to manually implement the definition given in listing 3 in nearly every file.

```
type 'a option =
    | Some of 'a
    | None;
```

Listing 3: Definition of `option`

2.2.2 Custom ADTs

The implementation does not limit users to only use the built-in ADTs. New ones can be defined given a name, a list of constructors, and optionally a number of type variables. The constructor names must start with a capital letter.

```
type direction =
    | North
    | East
    | South
    | West;
type 'a 'b tuple =
    | Tuple of 'a * 'b;
```

2.2.3 Type Aliasing

Sometimes types become very long and tedious to write explicitly. Type definitions can be used to introduce an alias for any type.

```
type int_tuple = int * int;
type 'a si = 'a signal;
type 'a to_string = 'a -> string;
type str = string;
```

2.3 Language Terms

2.3.1 Common Functional Language Terms

Async RaTT is a FRP language and incorporates many well known functional language terms. Some term have in this implementation, been replaced to create a more generally usable language. For example, natural numbers in Peano representation are replaced by OCaml integers which uses Two's component representations. Other terms which are not formally part of Async RaTT have been added to broaden what can be expressed. For example, the addition of booleans, strings, and ADTs. The semantics of these added terms are similar to those of other functional languages.

Both a polymorphic equality- and inequality operator are provided available via the syntax `=` and `!=`. Both of these operators result in values of the `bool` type.

unit : `()`
This is just the unit value i.e. a value which can only take one shape. It has the type `unit`.

boolean : `true | false`
The common boolean value which can be either true or false and has the type `bool`. There are two binary operations on booleans: Conjunction represented by `&&`, and disjunction represented by `||`.

integer : `0 | 12 | -4`
The whole number representation which can be both positive and negative. These have the type `int`. A handful of arithmetic binary operators are defined for `int`:

$$+, -, /, *, <, >, <=, >=$$

string : `"Hello world!"`
The representation of text. These have the type `string`. A single binary operation is defined for this type, namely concatenation using the `^` operator.

ADT construction : `Some 42 | None`
Each constructor defined in an ADT definition can be instantiated, given a value of the type which defines it. For example, the `Some` constructor of `'a option`, defined in listing 3, is defined with the polymorphic type. So given an `int`, as in the example above, the type `int option` is produced. If the constructor is not defined with any type, no value should be given. This is the case for the `None` constructor of `'a option`.

binding : `let x : t = e in b`
This defines `x` to be of type `t` and immutably take on the value produced by evaluating `e`. `x` will then be available during the evaluation of `b`. The evaluation of `b` is the result of the entire term.

pattern matching : `match e with | a ... -> b ...`
This allows evaluation of terms depending on the shape of some term `e`. If there is only one alternative the beginning `|` is optional. An alternative can have multiple patterns separated with `|` and ending with an arrow pointing to a term.

It must be possible for every pattern to match `e`. For example, the type checker will complain if `e` is of type `int` while one of the patterns expects it to be a `bool` or a tuple. Additionally, every alternative must produce values of the same type, `'a`. The type of the entire expression will then also be `'a`.

The version of the implementation described in this paper does not guarantee that pattern matching is exhaustive. It does however guarantee that all patterns of an alternative contain the same set of bindings.

```
let is_zero = \x -> ( match x with
    | 0 -> true
    | _ -> false
);
let single_digit = \x -> ( match x with
    | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 -> true
    | _ -> false
);
let get = \(opt : 'a option) def -> ( match opt with
    | Some x -> x
    | None -> def
);
```

pattern	matching with e
_	always matches
x	always matches and binds x to e
i	matches if e is an *int* and $i = e$
b	matches if e is a *bool* and $i = e$
(p_0, p_1)	matches if e is a tuple and p_0 matches the first element and p_1 the second.
$p_0 :: p_1$	matches if e is a signal
$Name\ p$	matches if e is an instance of $Name$ and p matches e's argument

Table 2: Available patterns

if-then-else : `if c then a else b`
This term is syntactic sugar for pattern matching on a value of type `bool`.
It is equivalent to the term `match c with | true -> a | false -> b`.

lambda expression : `\(x : t) ... -> b`
This is the syntax for defining a lambda expression. It can take any number of arguments each of which can either be explicitly typed by providing `t` or inferred by leaving it out. Additionally, `x` can be a pattern allowing for argument deconstructing.

An important restriction comes from the parser which does not allow `b` to directly be a pattern matching term. If this is needed it has to be surrounded by parenthesis.

```
let abs : int -> int = \x -> if x < 0 then -x else x;
let fst = \(x,_) -> x;
let add = \a b -> a + b;
```

tuple : `(a, b)`
In this implementation tuples are structures of strictly two elements. If `a` is of type `'a` and `b` is of type `'b` then the tuple is of type `'a * 'b`.

```
let user : string * int = ("John", 24);
let primes = (17, 13);
```

application : `f x`
Function applications work as in most other languages. If `f` is of type `'a -> 'b` and `x` is of type `'a` then the application of `f` to `x` gives a value of type `'b`. Functions having multiple parameters are implemented as functions having one parameter and returning new functions. This allows for partial function applications.

2.3.2 Async RaTT specific terms

The following language constructs are those which make Async RaTT stand out. These relate to handling delayed computations as well as the unusual scoping rules involving time. The scoping rules are explained in more detail in section 2.4.

`wait c`, $\dfrac{k :_c \text{`}a \in \Delta \qquad c \in \{p, pb\}}{\Gamma \vdash_\Delta \texttt{wait}\ k : \text{`}a\ \texttt{later}}$

Some input channels are push channels, meaning that they trigger updates of the program. This can, for example, be after some amount of time has passed or when a button is pressed. Waiting is used to acquire a delayed computation which can be evaluated once the channel being waited on has updated. If the channel produces values of type `'a` then waiting for this gives a values of type `'a later`.
This can, for example, be used to write a function that returns a signal and appends some string to whichever key is pressed on the keyboard:

```
let rec s = \str -> delay(adv(wait keyboard) ^ str :: sig);
```

`read c`, $\dfrac{k :_c \text{`}a \in \Delta \qquad c \in \{b, pb\}}{\Gamma \vdash_\Delta \texttt{read}\ k : \text{`}a}$

Some channels are buffered meaning that there is always a value available. This value is accessed by reading the channel which will immediately fetch the buffered value.

For example, this could be used to write a signal giving the latest keystroke each minute:

```
let rec s = delay(let w = wait minute in read keyboard :: sig);
```

`a :: b`, $\dfrac{\Gamma \vdash_\Delta t_1 : \text{`}a \qquad \Gamma \vdash_\Delta t_2 : \text{`}a \texttt{ signal later}}{\Gamma \vdash_\Delta t_1 \texttt{ :: } t_2 : \text{`}a \texttt{ signal}}$

This is the operator used for constructing signals. The left hand argument is of type `'a`, the right hand argument is of type `'a signal later`, and the term itself has the type `'a signal`.

`delay t`, $\dfrac{\Gamma, \surd_{cl(t)}, \Gamma' \vdash_\Delta t : \text{`}a \qquad \Gamma \vdash_\Delta cl(t) : Clock}{\Gamma \vdash_\Delta \texttt{delay } t : \text{`}a \texttt{ later}}$

This delays the evaluation of `t`. This requires that `t` advances on another delayed value. If `t` has the type `'a` then `delay t` has the type `'a later`. The term `t` can then be evaluated once the required data is available. `delay` introduces a tick in the context, meaning that all binding that existed before the delay are now considered as such. In practice this results in unstable values becoming unavailable and values of the type `'a later` becoming available via advancing.

`adv t`, $\dfrac{\Gamma \vdash_\Delta t : \text{`}a \texttt{ later}}{\Gamma, \surd_{cl(v)}, \Gamma' \vdash_\Delta \texttt{adv } t : \text{`}a}$

This is how a delayed computation is evaluated. This is only allowed inside of a `delay` as a tick in the context is required, and `t` must be of type `'a later`. The result of advancing a value of this type is a value of type `'a`.

`let rec x : t = e in b`, $\dfrac{\Gamma \vdash_\Delta a : \text{`}a \quad \Gamma, x : \circledcirc \text{`}a \vdash_\Delta b : \text{`}b}{\Gamma \vdash_\Delta \texttt{let rec } x = a \texttt{ in } b : \text{`}b}$

This works the same as other bindings except that `x` is implicitly given the type $\circledcirc t$ inside `e`. Advancing on `x` inside `e` results in a value of type `t`. This effectively means that recursion is only available inside a `delay` which gives the language its productivity guarantee. Advancing `x` is done automatically, and there does not exist a concrete syntax for the type $\circledcirc t$.

`select a b`, $\dfrac{\Gamma \vdash_\Delta t_1 : \text{`}a \texttt{ later} \qquad \Gamma \vdash_\Delta t_2 : \text{`}b \texttt{ later}}{\Gamma, \surd_{\theta_1 \sqcup \theta_2}, \Gamma' \vdash_\Delta \texttt{select } t_1\ t_2 : (\text{`}a, \text{`}b) \texttt{ selection}}$

If a computation relies on multiple delayed computations synchronization must be used. This is done by using the `select` keyword with two delayed computations, `a` and `b`. `a` and `b` must be values of the types `'a later` and `'b later` which will make the entire `select` term have the type `('a * 'b) selection`. This captures the possibility that one computation is done before the other or that both are done at the same time. Pattern matching can then be used to express how to handle each of those cases. It is important to note that `select` is required to be inside a `delay` as this is the only way for a tick to have been introduced into the context.

`select` can, for example, be used to write a signal that marks the passage of time in a stream of strings:

```
let rec keyboard_ticks = delay(match select (wait keyboard) (wait minute) with
    | Left(s::ss,_) -> s::keyboard_ticks
    | Right(_,_)
    | Both(_,_) -> "tick"::keyboard_ticks
);
```

`box t`, $\dfrac{\Gamma^\square \vdash_\Delta t : \text{`}a}{\Gamma \vdash_\Delta \texttt{box } t : \text{`}a \texttt{ boxed}}$

Boxing is the mechanic with which terms that are not inherently stable can be made stable. Before evaluating the term `t` the context is stabilized, meaning that all bindings of unstable

types are no longer in scope. As an example, in the following code snippet the term `a` is invalid as `f` is a function type which is inherently unstable. It is therefore removed from the context when evaluating `box f`. In contrast, `b` is valid as it does not refer to any unstable bindings.

```
let a =
    let f = \i -> i + 1 in
    box f;

let b = box (\i -> i + 1);
```

If the term `t` has the type `'a` then `box t` has the type `'a boxed` indicating that the term can safely be moved into the future.

`unbox t`, $\dfrac{\Gamma \vdash_{\Delta} t : \text{`}a \texttt{ boxed}}{\Gamma \vdash_{\Delta} \texttt{unbox } t : \text{`}a}$

This is the counterpart of boxing a term. The term `t` is evaluated and is expected to have the type `'a boxed`. Unboxing `t` will then give a value of type `'a`.

`never`, $\dfrac{}{\Gamma \vdash_{\Delta} \texttt{never} : \text{`}a}$

This is the delayed computation that is never evaluated. This could be used to set an output channel to an initial value which would then never change. For example, this could be used to implement the "Hello World!" program like so:

```
console <- "Hello world!" :: never;
```

2.4 Scope rules

The set of variables which are available at any time is called the context. The content of the context changes in different scenarios. For example, most people who are familiar with functional programming knows that a let-expression, `let x = e in b`, will add the binding `x` to the context in which `b` is evaluated. This is also the case in Async RaTT. However, there are a few more rules to keep in mind which exist to ensure the guarantees of the language. This includes that a program does not contain implicit space-leaks, that it is productive in finite time, and that computations are causal, i.e. that computations do not require future data to complete.

The most central concept here is that any term of the language exists at some point in time. As such, a let-expression defines a binding in the context that exists "now". When a term is delayed, `let x = e in delay t`, the context moves to the next time step and the binding `x` which existed "now" now exists "before". If this `x` is of a stable type, i.e. a type that can safely be moved into the future, it can still be referenced in `t` otherwise it will be unavailable. Additionally, because time has now passed for `t` any term `a` of type `'a later` can now be made to produce a value by writing `adv a`.

To ensure productivity any recursively defined binding only becomes available within its own definition inside of a delay. Thus, a function such as:

```
let rec foo =
    \i -> foo (i + 1);
```

does not type check while `key_handle` in listing 1 does.

3 Language infrastructure

All the language infrastructure for this project in implemented in OCaml5.1 and is build using Dune. Additionally, the lexer generator tool `ocamllex` is used for implementing the lexical analysis, and the parser generator `menhir` is used to implement parsing to the abstract syntax tree. Figure 1 gives an overview of how the different modules of the infrastructure interact.

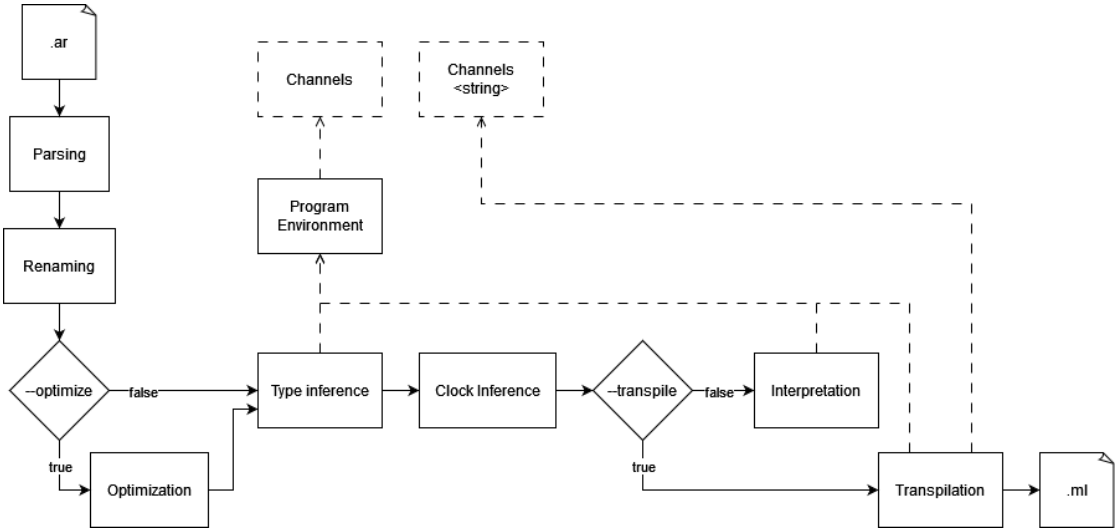

Figure 1: Language infrastructure

3.1 Channels

As previously mentioned Async RaTT describes programs in some environment, Δ, of input and output channels. This means that the theory can work for a multitude of environments. Bahr and Møgelberg [2] gives the examples of a GUI. To encode this in the implementation channels are described in terms of their types which decouples their concrete implementation from the language. The remaining modules can then be constructed given a list of concrete channel implementations resulting in a usable language infrastructure.

A concrete implementation of channels is a list of pairs containing a string and one of four channel constructors. The string is the name of the channel while the channel constructor carries the information of how the channel behaves. The four constructors are `PushChannel`, `BufferedChannel`, `PushBufferedChannel`, and `OutputChannel`. The output channel is defined by a function for outputting values and optionally the type which it can consume. The other three are defined by the type of values that they produce, optionally a delay between updates, the function that actually produces values, and an initial buffer value if the channel is buffered.

3.2 Program environment

Given a list of channels, as described in section 3.1, the implementation can construct a program environment module with the following signature:

```
module type ProgramEnvironment = sig
    val is_buffered_channel : string -> bool
    val is_push_channel : string -> bool
    val input_channel_type : string -> typ
    val output_channel_type : string -> typ option
    val channel_values : unit -> runtime_value val StringMap.t
    val buffer_values : unit -> runtime_value val StringMap.t
    val output_channels : (typ option * val (runtime_value -> unit)) StringMap.t
    val handle_events : unit -> runtime_value val StringMap.t
    val start_channels : unit -> Thread.t list
    val buffers : input_channel StringMap.t
end
```

The first four values are required by the type system implementation such that it can query for typing information on the available input- and output-channels.

The next five values are required by the interpreter to interact with the environment. `buffer_values` gives access to the current values of the buffered input channels. `output_channels` gives access to the output channels. This importantly includes functions with which to actually produce an output. For example, printing to the console. `handle_events` gets all the updates on the set of input channels. Finally, `start_channels` starts up a thread to handle each input channel.

The final value, `buffers`, provides information about buffers and is required by the transpiler for transpiling Async RaTT buffers to OCaml functions.

The rest of the language infrastructure can then use this module to query and interact with the implemented environment.

3.2.1 Provided program environment

For the purpose of testing the language implementation a simple environment is designed. This allows us to write programs that react to, and interact with, the computer. It contains 4 input channels and 2 output channels. This allows for the environment used for the examples in this paper.

second is a push input channel which produces a `unit` value once every second. This enables writing programs that update every second.

minute is a push input channel which produces a `unit` value once every minute. This enables writing programs that update every minute.

keyboard is a push buffered input channel which produces a string each time a key on the keyboard is pressed. The string simply contains the character value of the key pressed. This character values is also saved to a buffer.

system_time is a buffered input channel that contains an integer representing how many seconds have passed since the program was started.

console is an untyped output channel that allows for printing arbitrary values to the console from which the program was started.

console_int is like **console** except that it requires the given values to be of type `int`.

Each input channel is handled on a separate thread which makes the system asynchronous. To avoid race conditions mutual exclusion is used by each thread when writing values to the data structure keeping track of what values exist in any given time step.

An example of how this environment is implemented, and used to construct language infrastructure, can be seen in listing 18.

3.3 Interpreter

The interpreter is the system responsible for evaluating a given abstract syntax tree in the current environment. Additionally, the interpreter communicates with the program environment requesting the input channels to be initialized, querying these for values, and writing to output channels.

3.4 Transpiler

Having to execute programs via an interpreter is not necessarily the best solution for all cases, as it sacrifices performance for a simple execution pipeline i.e. having no compilation step. To get around this, the language infrastructure includes the ability to transpile programs written in Async RaTT into OCaml source code which can then be compiled with a native-code compiler, e.g `ocamlopt`. To enable this feature a module containing the program environment implementation as a string is required. This allows the transpiled program to be bundled with the environment that it relies upon. In order to transpile an Async RaTT program the `-t` or `--transpile` flags have to be set. How the transpilation is done is explained in detail in section 6.

3.5 Optimization

We have not extensively implemented optimizations. Rather we have attempted two simple optimizations. The purpose of this is mostly to have the infrastructure in place. Optimization can be enabled via the `-o` or `--optimize` flags. With this enabled the optimizations are applied on the abstract syntax tree.

3.5.1 Arithmetic Preprocessing

Arithmetic operations consisting exclusively of value literals will be reduced to their result. This means that a term such as `let x = 2 + 2 * 2 in e`, will be transformed into `let x = 6 in e`, while `let x = 2 + y in e` would not be transformed as `y` is not a value literal.

3.5.2 Pattern-matching Reduction

In some cases, it is obvious that an alternative in a match-term will never be matched due to the shape of the pattern compared to the term being matched on. In these cases we discard that

alternative. As an example consider the term:

```
match (1,x) with | (2,x) -> a | (1,2) -> b
```

Here the first alternative is not viable as `1 = 2` is clearly false. Therefore, it can be discarded reducing the amount of alternatives that would be checked at runtime.

Additionally, when a term being matched on does not vary, i.e. its value can be determined at compile time, determining which alternative will be matched also becomes possible. If the alternative would create bindings and any of these are used in the following term every other alternative is discarded. Otherwise the entire match-term is replaced by the term of the alternative. Using this, these terms can be reduced such that a term:

```
match 2 with | 1 -> a | x -> b
```

can be reduced to just `b`. While a term:

```
match 2 with | 1 -> a | x -> x + 1
```

can be reduced to:

```
match 2 with | x -> x + 1.
```

4 Inference

4.1 Type inference

Async RaTT uses a Hindley-Milner type system which is capable of automatically inferring the most general type of any program without the need for type-annotations. This is implemented using a map, `type_vars`, from type variable names to types as the primary data structure:

```
type_vars : typ StringMap.t
```

where `StringMap.t` is defined as:

```
module StringMap = Map.Make(String)
```

We have chosen to implement type inference purely functionally using a state monad. This allows us to "mutate" `type_vars` while remaining purely functional.
The type `typ` represents a type in Async RaTT and is defined as the recursive ADT:

```
type typ =
    | T_Poly of string
    | T_Unit
    | T_Int
    | T_String
    | T_Multiplicative of typ * typ
    | T_Additive of (string * typ) list
    | T_Function of typ * typ
    | T_Existential of typ
    | T_Universal of typ
    | T_Fix of string * typ
    | T_Boxed of typ
    | T_Named of typ list * string
    | T_Bool
    | T_Signal of typ
```

When a new type variable is created it is inserted in `type_vars` with the key being its name. In other words, new type variables map to themselves via their names.
The implementation needs to be able to lookup a type. However, since `typ` is a recursive ADT it may contain other instances of `typ`. Some of these `typ`s may be polymorphic and may therefore have been unified with other types. This means that simply looking up a type in `type_vars` may not result in the correct type. To solve this issue when a type, `t`, is looked up in `type_vars` all type variables, `t_0, ..., t_n`, in `t` are replaced by the result of looking up `t_0, ..., t_n` in `type_vars`. For example if looking up type `'a` results in `'b * 'c` then `'b` and `'c` are also looked up in the map. If looking up `'b` and `'c` results in `int` and `string` respectively then the final result of looking up `'a` is `int * string`.

The implementation needs to ensure that only bindings in the current scope can be accessed. In most other programming languages scopes can be represented by a `binding StringMap.t list` where each `StringMap.t` represents a scope. In Async RaTT this is complicated by the fact that bindings declared outside a `delay` generally cannot be accessed inside a `delay`. To model this the implementation of Async RaTT using the following binding:

```
context : var_env * (var_env option)
```

where `var_env` is defined as:

```
type var_env = binding StringMap.t list
```

Each map represents a scope allowing the implementation to lookup information about a binding. When entering a new scope a new map is added to the front of the list. When a scope is exited the head of the list is removed. This allows for keeping track of available bindings while performing type inference.
The first `var_env` in `context` represents the bindings declared outside a `delay`. The case when the `var_env option` is `None` represents there being no tick in the context (currently inferring types outside a `delay`), while the case when it is `Some` represents the, i.e. currently inferring types inside a `delay`. The `var_env` inside the `option` represents all bindings declared inside the current `delay`.
The `binding` type is defined as follows:

```
type binding = {
    typ : typ;
    top_level : bool;
    declared_in_normal_context : bool;
    declared_outside_local_recursive : bool
}
```

A binding is a record containing a type and three booleans to help determine whether the binding is available when accessed or if an error should be raised. Here the term "normal_context" refers to a context that is not stable and does not have a tick.

4.1.1 Inference of Stable Types

$$\frac{\texttt{'a} : \mathit{stable} \in \phi}{\phi \vdash \texttt{'a} : \mathit{stable}} \qquad \frac{}{\phi \vdash \texttt{unit} : \mathit{stable}} \qquad \frac{}{\phi \vdash \texttt{int} : \mathit{stable}} \qquad \frac{}{\phi \vdash \texttt{bool} : \mathit{stable}}$$

$$\frac{}{\phi \vdash \texttt{string} : \mathit{stable}} \qquad \frac{\phi \vdash t_1 : \mathit{stable} \quad \phi \vdash t_2 : \mathit{stable}}{\phi \vdash t_2 * t_2 : \mathit{stable}} \qquad \frac{}{\phi \vdash t\ \texttt{boxed} : \mathit{stable}}$$

$$\frac{\forall d \in \texttt{def}((t_0, ..., t_n)\ \mathit{name}),\ \phi \vdash d : \mathit{stable}}{\phi \vdash (t_0, ..., t_n)\ \mathit{name} : \mathit{stable}}$$

$\texttt{def}((t_0, ..., t_n)\ \mathit{name})$ is the set of types used to define *name*, with its type variables replaced with $t_0, ..., t_n$

Figure 2: Type stability rules

In Async RaTT a binding defined outside a delay can only be accessed inside a delay if the type of the binding is stable or if it is being advanced on. This means that the types of bindings defined outside delays and accessed inside delays, but not being advanced on, must be constrained to be stable. As an example, consider the function, `f`, shown below.

```
let f = \x -> delay (
    let y = x in
    adv (wait keyboard))
```

The parameter `x` is accessed inside the delay and must therefore be stable. Otherwise, `x` may no longer exist when the delayed computation is advanced. This also means that `f` cannot be applied to unstable type.

Inference of stable types is implemented using a set, ϕ, which in the implementation is called `required_stable`, of strings representing type variable names. When a binding, `x` of type `t`, defined outside a `delay` is accessed inside a `delay`, then `t` will be marked as being required to be stable. This also happens if `x` is defined outside a `box` and is accessed inside a `box`, or if `x` is defined in a top-level function and is accessed inside a local recursive function. This is done by first checking if the type could possibly be stable. For example, the type `'a -> 'b` cannot possibly be stable while the type `'a * 'b` could potentially be stable. If the type cannot possibly be stable then type inference fails. Otherwise, all type variables in `t` are added to `required_stable`. After all types have been inferred `required_stable` contains type variables that may have been unified with other types. These types may or may not be stable. Therefore, the type system iterates over ϕ. For each type variable, the final type is looked up in `type_vars` and checked to see if it can be stable. If it cannot be stable then type inference will fail. The reason that the validity of stable types is, for the most part, only checked once type inference has completed is that the type of a binding may not always be known when it is accessed. Inference of stable types constraints is, like type inference, implemented completely functionally using a state monad.

This implementation ensures that bindings defined outside delays and accessed inside delays are constrained to be stable, and that functions whose parameters are constrained to be stable can only be applied to stable values.

4.2 Clock Inference

$$\frac{}{adv\ v : cl(v)} \qquad \frac{}{v : \emptyset} \qquad \frac{}{adv(wait\ k) : k} \qquad \frac{}{\texttt{read}\ k : \emptyset} \qquad \frac{}{\texttt{never} : \theta} \qquad \frac{}{\texttt{unbox} : \theta}$$

$$\frac{(\theta_y \cap \theta_x = \emptyset) \vee (\theta_x = \theta_y)}{\xi(\theta_x, \theta_y)} \qquad \frac{x : \theta_x \quad y : \theta_y \quad \xi(\theta_x, \theta_y)}{(x, y) : \theta_x \cup \theta_y} \qquad \frac{f : \theta_f \quad a : \theta_a \quad \xi(\theta_f, \theta_a)}{f\ a : \theta_f \cup \theta_a}$$

$$\frac{a : \theta_a \quad b : \theta_b \quad \xi(\theta_a, \theta_b)}{\texttt{let}\ x = a\ \texttt{in}\ b : \theta_a \cup \theta_b} \qquad \frac{e_0 : \theta_0\ ...\ e_n : \theta_n \quad \xi(\theta_0, \theta_1)\ ...\ \xi(\theta_{n-1}, \theta_n)}{\texttt{match}\ e_0\ \texttt{with}\ p_1 \rightarrow e_1,\ ...,\ p_n \rightarrow e_n : \theta_0 \cup ... \cup \theta_n}$$

$$\frac{}{\texttt{select}\ a\ b : cl(a) \cup cl(b)}$$

Figure 3: Clock inference rules

In order for Async RaTT to ensure productivity, no implicit space leaks, and causality it has to know what clock each delayed computation depends on. This can be inferred at compile time by finding all delays in the AST, and finding the clocks that these advance on. The implementation defines a function, `infer_clock`, of type `'a term -> clock_result` that infers the clock of a delay. `clock_result` is an ADT used to determine if all clocks advanced on in a delay are valid. Its definition is as follows:

```
type clock_result =
    | Single of clock_set
    | NoClock
    | MultipleClocks
```

The `Single` constructor represents the case when a single clock has been inferred. `NoClock` represents the case before any `adv` or `select` expression is found. `MultipleClocks` represents the case when a `delay` advances on different clocks. If `NoClock` or `MultipleClocks` is the result of inferring the clock of a delay, then clock inference fails, because a delay must advance exactly one clock.
A `clock_set` is the representation of a clock at compile time. This data type is defined as:

```
and clock_set = CS of {
    channels : StringSet.t;
    variables : StringSet.t
}
```

Where `channels` is the set of all channels advanced on that can be inferred at compile time, and `variables` is the set of variables whose clock is advanced on. The `variables` binding is needed because the clocks of bindings may depend on runtime information. The exact channels in the

clock can therefore not be determined at compile time.

Because we can only advance on clocks of values of type `'a later` the values must be either `wait_k` or bindings of type `'a later`. In the case of `adv wait_k` the clock is simply the singleton clock consisting of `k`. In the case of advancing a binding, `x`, as in `adv x` we add the name of `x` to `variables`. In the following we write `cl(a)` to represent the clock of the binding `a`. Note that in certain situations, it is possible to infer the clock of bindings. For example, the clock of the expression `let x = wait keyboard in adv x` is clearly just `{keyboard}`. However, the implementation does not infer this for two reasons. Firstly, it should be as clear as possible why the compiler either fails or succeeds to infer a clock. Partially inferring the clocks of bindings might cause this to be less clear and therefore make Async RaTT harder to use. Secondly, it would make the implementation much more complicated. We do not deem the additional complexity acceptable as this feature would only work in simple cases, and it would increase the risk of bugs.

For clock inference to succeed the clocks that are advanced on, must be equal. We say that two clocks are equal if they consist of the same set of channels, i.e., for two clocks, θ_1 and θ_2, $\theta_1 = \theta_2$ must hold.

When reaching an `adv` during clock inference we check if the clock advanced on is compatible with the clock inferred so far. For example, consider the expression `delay(adv a + adv b)` where `a` and `b` are bindings. This delay advances on two different clocks: `cl(a)` and `cl(b)`. When iterating through the AST during clock inference we first encounter `adv a`. Since this is the first clock advanced on inside the delay we return `Single (cl(a))`. Then later when reaching `adv b` we check equality of the clocks `cl(a)` and `cl(b)`. In this case the clocks are not equal and we return `MultipleClocks`. This happens because the exact channels of `cl(a)` and `cl(b)` may not be possible to infer at compile time. Therefore, we must take the safe approach, in terms of ensuring the guarantees of Async RaTT, and simply assume that they are different. On the other hand the expression `delay(adv a + adv a)` will succeed since the delay only advances the clock `cl(a)`.

The only case in which two advanced clocks are allowed to be different is through the use of the `select` term. The `select` term takes two values of type `'a later` as parameters, and advances either one or both of their clocks depending on which clock has ticked. The clocks are therefore allowed to be different. The clocks of the given values are combined by set union and wrapped in `Single`.

Consider the expression `delay(match select a b with ...)`. This expression advances on both `cl(a)` and `cl(b)` which normally would not type check. However, since `select` is used the clocks `cl(a)` and `cl(b)` are unioned so the clock of the expression is `cl(a)` $\cup$ `cl(b)`.

In short the algorithm for inferring clocks is as follows:

1. Find all `delay` expressions.
2. Apply `infer_clock` to each `delay` expression.
3. Attach the resulting clock to the `delay`.

The `infer_clock` function works as follows:

1. Iterate through the given expression.

2. When a `adv t` is found compare $cl(t)$ with the result so far:
   - If the previous result was `NoClock` return `Single (cl(t))`.
   - If the previous result was `Single c` then do one of the following:
     - If `cl(t) = c` then return `Single c`.
     - Otherwise return `MultipleClocks`.
   - If the previous result was `MultipleClocks` then return `MultipleClocks`.
3. If the final result is `Single c` then `c` is returned. Otherwise, the result is `MultipleClocks` or `NoClock` and clock inference fails.

5 Interpretation

5.1 Description of Runtime Values

Async RaTT uses the ADT shown in listing 4 for representing values at runtime.

```
type runtime_value =
    | Binding_rv of string
    | Unit_rv
    | Int_rv of int
    | Bool_rv of bool
    | String_rv of string
    | Tuple_rv of runtime_value * runtime_value
    | Closure of string * value_env * typed_term
    | Construct_rv of string * runtime_value
    | Built_in_1 of (runtime_value -> runtime_value)
    | Built_in_2 of (runtime_value -> runtime_value -> runtime_value)
    | Location of int * clock
    | Box of typed_term * value_env
    | Signal_rv of runtime_value * runtime_value
    | Wait_rv
```

Listing 4: Representation of runtime values

A binding is represented by its name as a string. This allows for easily looking up the value of a binding. The representation of a unit value, `Unit_rv`, does not carry any information as it is not needed. Integers (`Int_rv`), booleans (`Bool_rv`), and strings (`String_rv`) simply carry the information which would be expected. Tuples are composed of two arbitrary values and are therefore represented simply as a pair of two `runtime_value`s. A closure consists of the name of a parameter (a string), an environment containing free bindings, and a term which is what is executed when the closure is called. This corresponds to how closures are implemented in other functional languages. A `value_env` is a an ADT shown in listing 5 which is mutually recursive with `runtime_value`.

```
and value_env = VE of { vars : (runtime_binding StringMap.t }
```

Listing 5: Representation of closure environments

A `value_env` is simply a wrapper around a map from binding names to the associated data. When a closure is evaluated (such as when a function is called) a new scope is pushed and all the bindings in the `value_env` are declared. After evaluation of the term the scope is then popped again.

The `Construct_rv` constructor in `runtime_value` represents an ADT. The `string` is the name of the ADT and the `runtime_value` is the value given as argument during runtime.

Built in operators like integer addition and boolean conjunction are implemented using the constructors `Built_in_1` and `Built_in_2`. These are lambda expressions having one and two `runtime_values` as parameters respectively. This allows for easy implementation of operators in Async RaTT that are otherwise built in to OCaml. For example, integer addition is implemented as the function shown in listing 6.

```
let add_integers a b =
    match a, b with
    | Int_rv a, Int_rv b -> Int_rv (a + b)
    | _ -> failwith "Not integers"
```

Listing 6: Representation of closure environments

This function is of type `runtime_value`→ `runtime_value`→ `runtime_value` and can therefore be wrapped in a `Built_in_2`.

A `Location` consists of an integer representing the heap location of a delayed computation and the clock of the delayed computation. `Location` values are used to manage delayed computations. This is a major part of the implementation of Async RaTT and is thus described in both section 5.2 and 5.3.

The `Box` constructor wraps a term to be evaluated later and a `value_env` containing free bindings in the term.

The `Signal_rv` constructor consists of the head and tail of a signal. Both of these are `runtime_values`.

The `Wait_rv` constructor is what is created when the interpreter reaches a `Wait` value. `Wait_rv` does not carry any information as the updated channel is always known when a `Wait_rv` is advanced.

5.2 Implementation of Reactive Semantics

5.2.1 Implementation of the Heap

Async RaTT requires a heap for delaying computations into the future. The heap can be in one of two states: η_L or $\eta_N\langle\kappa \mapsto v\rangle\eta_L$ where η_N is called the now heap, η_L is called the later heap, κ is a channel, and v is a value. The now heap contains delayed computations that are safe to evaluate because their clocks have received an update. The later heap contains delayed computations that can only be evaluated in the future once their clocks have ticked. The expressions $\kappa \mapsto v$ means that the channel κ has received a value v. We model the heap using an algebraic data type:

```
type heap =
    | OneHeap of delayed IntMap.t
    | TwoHeap of {
        now : delayed IntMap.t;
        update : string * runtime_value;
        later : delayed IntMap.t
    }
```

Listing 7: Implementation of the heap

This algebraic data type allows for describing the two states the heap can be in and allows for easily performing garbage collection and ensuring correctness. In accordance with the theory, when the heap is a `OneHeap` it consists only of a map from heap locations to delayed computations. When the heap is a `TwoHeap` it consists of a now heap, a later heap, and an update on a channel. Each integer key in the maps corresponds to a `Location` created during runtime. The `update` binding reflects the channel update $\kappa \mapsto v$ where the string is the name of the channel and the `runtime_value` is the value on the channel.

5.2.2 Implementation of Transitions

$$\text{INIT}\ \frac{\langle\langle t\rangle;\emptyset\rangle \Downarrow^{\iota} \langle\langle v_1 :: l_1, ..., v_m :: l_m\rangle;\eta\rangle}{\langle t;\iota\rangle \xrightarrow{x_1\mapsto v_1,...,x_m\mapsto v_m} \langle x_1 \mapsto l_1, ..., x_m \mapsto l_m;\eta;\iota\rangle}$$

$$\text{INPUT}\ \frac{\iota' = \iota[\kappa \mapsto v] \text{ if } \kappa \in \text{dom}(\iota) \text{ otherwise } \iota' = \iota}{\langle N;\eta;\iota\rangle \xrightarrow{\kappa\mapsto v} \langle N;[\eta]_{\kappa\in}\langle\kappa \mapsto v\rangle[\eta]_{\kappa\notin};\iota'\rangle}$$

$$\text{OUTPUT-END}\ \frac{}{\langle\cdot;\eta_N\langle\kappa \mapsto v\rangle\eta_L;\iota\rangle \Rightarrow \langle\cdot;\eta_L;\iota\rangle}$$

$$\text{OUTPUT-SKIP}\ \frac{\kappa \notin cl(l) \qquad \langle N;\eta_N\langle\kappa \mapsto v\rangle\eta_L;l\rangle \overset{O}{\Longrightarrow} \langle N';\eta;\iota\rangle}{\langle x \mapsto l, N;\eta_N\langle\kappa \mapsto v\rangle\eta_L;\iota\rangle \overset{O}{\Longrightarrow} \langle x \mapsto l, N';\eta;\iota\rangle}$$

$$\text{OUTPUT-COMPUTE}\ \frac{\kappa \in cl(l) \qquad \langle \text{adv } l;\eta_N\langle\kappa \mapsto v\rangle\eta_L\rangle \Downarrow^{\iota} \langle v' :: l';\sigma\rangle \qquad \langle N;\sigma;\iota\rangle \overset{O}{\Longrightarrow} \langle N';\eta;\iota\rangle}{\langle x \mapsto l, N;\eta_N\langle\kappa \mapsto v\rangle\eta_L;\iota\rangle \xrightarrow{x\mapsto v',O} \langle x \mapsto l', N';\eta;\iota\rangle}$$

Figure 4: Reactive semantics from [2]

As described by Bahr and Møgelberg[2] the reactive machine can take three different transitions: Initialization transitions, input transitions, and output transitions. The reactive semantics of

these transitions can be seen in figure 4. Input transitions defined by INIT are denoted using the shorthand:

$$\langle t; \iota \rangle \overset{O}{\Longrightarrow} \langle N; \eta; \iota \rangle$$

Where t is the reactive program, ι is the initial input buffer, O is a sequence that maps output channels to values, N is a map from output channels to heap locations, and η is the heap.
In our implementation t is a list of top-level declarations, and N is implemented as a map:

```
outputs : int StringMap.t
```

Where the keys are output channel names, and the values are heap locations.
We represent ι as a binding, `buffers : runtime_value StringMap.t`, that maps buffer names to their newest values.
We implement INIT by iterating over the list of top-level definitions. Both top-level let-bindings and output channels have an associated term. Top-level let-bindings are implicitly boxed and the term is therefore boxed before the binding is bound to it. This also means that top-level bindings need to be implicitly unboxed when accessed. Output channels need to be initialized and the associated term is therefore evaluated. The term is always of type `'a signal` so the head of the signal, being immediately available, is written to the output channel and the key-value pair consisting of the channel name and the location of the tail is added to `outputs`.
We initialize `buffers` by querying the initial values of all available buffers from the `ProgramEnvironment`.
After the initialization transition, the heap is a one heap where the delayed computations are the delayed tails of the signals initializing the output channels.

Once an updated value v is received on a channel κ an input transition defined by INPUT is performed:

$$\langle N; \eta; \iota \rangle \overset{\kappa \mapsto v}{\Longrightarrow} \langle N; \eta_N \langle \kappa \mapsto v \rangle \eta_L; \iota' \rangle$$

An input transition require that the heap starts out as `OneHeap` and then creates a `TwoHeap`. This is done by taking all the key-value pairs in the `OneHeap` that do not depend on κ and moving them to the `later` map in the newly created `TwoHeap`. The `now` map in the `TwoHeap` is initialised to the map of the old `OneHeap`. The `update` binding is initialised to the pair consisting of the pair (κ, v).
Each input transition also pulls the latest value on all buffered channels and inserts these values into `buffers`.

After an input transition one or more output transitions defined by OUTPUT-SKIP and OUTPUT-COMPUTE are performed.

$$\langle N; \eta_N \langle \kappa \mapsto v \rangle \eta_L; \iota \rangle \overset{O}{\Longrightarrow} \langle N'; \eta_L; \iota \rangle$$

When performing output transitions, the heap will always be a `TwoHeap` since they are always performed after an input transition. To perform an output transition a location depending on κ and its associated delayed computation is found in the now heap. Since a delayed computation is a closure all the bindings stored within are declared. Then the computation itself is evaluated. The bindings from the closures environment are then released to prevent memory leaks. Now the channel in the `outputs` map is remapped to the new location of the tail of the resulting signal.

Finally, the head of the signal is written to the output channel.

After all output transitions for a specific channel update have been performed a garbage collection is done. In correspondence with OUTPUT-END a garbage collection can be done simply by creating a new `OneHeap` and initialising the map within using the `later` binding in the old `TwoHeap`. The old `TwoHeap` is then released. This is always possible because output transitions never change a `TwoHeap` to a `OneHeap`. Thus garbage collections are performed by the following simple function:

```
let collect_garbage (S state) =
    match state.heap with
    | OneHeap _ -> failwith "Cannot perform a garbage collection on a one heap"
    | TwoHeap heap ->
        let heap' = OneHeap heap.later in
        S { state with heap = heap' }
```

Where `state` contains information about bindings, the heap, `outputs` and buffers.

5.3 Interpretation of Terms

Async RaTT is a functional language and has all the usual terms found in many other functional languages. These are: let-bindings, functions, lambda expressions, pattern matching, if-then-else expressions, algebraic data types, and tuples. These constructs work exactly as would be expected of a functional programming language. The details of how these expressions are evaluated will therefore not be discussed. Instead, evaluation of the terms that make Async RaTT interesting will now be described.

First off is the `box` keyword which constructs a delayed computation that can be evaluated at any time in the future. This is implemented by wrapping the term given to `box` in a `Box` value. In addition, boxed values may depend on other bindings which may have been released. Therefore, `Box` not only consists of a computation but also an environment containing the bindings and their values, which were accessible at construction. Put simply, `Box` values are closures.

5.3.1 unbox

$$\frac{\langle t;\sigma\rangle \Downarrow^{\iota} \langle \mathrm{box}\, t';\sigma'\rangle \qquad \langle t';\sigma'\rangle \Downarrow^{\iota} \langle v;\sigma''\rangle}{\langle \mathrm{unbox}\, t;\sigma\rangle \Downarrow^{\iota} \langle v;\sigma''\rangle}$$

The `unbox` keyword takes a boxed term and evaluates it. This can happen both inside and outside a delay. This functionality is implemented by declaring the bindings stored in the environment of the boxed value and then evaluating the delayed computation.

The `wait` keyword constructs a delayed value whose clock contains only the given channel. For example, `wait keyboard` is a delayed value that has the clock `keyboard`. When encountering a `wait` keyword the interpreter simply initializes a `Wait` value and returns it together with the given clock.

5.3.2 delay

$$\frac{l = \text{alloc}^{|\theta|}(\sigma)}{\langle \text{delay}_\theta\ t; \sigma\rangle \Downarrow^\iota \langle l; (\sigma, l \mapsto t)\rangle}$$

The `delay` keyword constructs a delayed computation that can be evaluated using `adv` or `select` once a tick on its clock has occurred. The clock is automatically inferred as described in section 4.2. When evaluated it creates a `Location` value where the stored integer is uniquely generated. The integer and the computation is inserted into the later heap as a key-value pair.

5.3.3 adv

$$\frac{}{\langle \text{adv wait}_\kappa; \eta_N\langle\kappa \mapsto v\rangle\eta_L\rangle \Downarrow^\iota \langle v; \eta_L\langle\kappa \mapsto v\rangle\eta_L\rangle}$$

$$\frac{\langle \eta_N(l); \eta_N\langle\kappa \mapsto v\rangle\eta_L\rangle \Downarrow^\iota \langle w; \sigma\rangle}{\langle \text{adv } l; \eta_N\langle\kappa \mapsto v\rangle\eta_L\rangle \Downarrow^\iota \langle w; \sigma\rangle}$$

The `adv` keyword advances the clock of a value of type `'a later`. When `adv` is applied to a value the location of that value is looked up in the heap giving a delayed computation. This delayed computation is then evaluated and the result of type `'a` is returned.

The expression `adv wait_k` has a special meaning. Instead of returning a `Wait` value, like `wait_k` normally does, the updated value of the channel `k` is simply looked up and returned. This is valid because the clock inference ensures that the delayed computation that the `adv` resides in can only be advanced once `k` has a new value.

5.3.4 never

$$\frac{l = \text{alloc}^{\emptyset}(\sigma)}{\langle \text{never}; \sigma\rangle \Downarrow^\iota \langle l; \sigma\rangle}$$

The `never` keyword, like `delay`, allocates on the heap. However, this location does not have an associated clock and is thus never advanced.

5.3.5 read

$$\frac{\kappa \in \text{dom}(\iota)}{\langle read_\kappa; \sigma\rangle \Downarrow^\iota \langle \iota(\kappa); \sigma)\rangle}$$

The `read` keyword returns the latest value in the given buffer. This is done by simply looking up the `runtime_value` of the buffer in `buffers`.

5.3.6 select

$$\frac{\kappa \in |\mathrm{cl}(v_1)| \qquad \langle \mathrm{adv}\ v_1; \eta_N\langle \kappa \mapsto w\rangle \eta_L\rangle \Downarrow^{\iota} \langle u_1; \sigma\rangle}{\langle \mathrm{select}\ v_1\ v_2; \eta_N\langle \kappa \mapsto w\rangle \eta_L\rangle \Downarrow^{\iota} \langle \mathrm{Left}(u_1, v_2); \sigma'\rangle}$$

$$\frac{\kappa \in |\mathrm{cl}(v_2)| \qquad \langle \mathrm{adv}\ v_2; \eta_N\langle \kappa \mapsto w\rangle \eta_L\rangle \Downarrow^{\iota} \langle u_2; \sigma\rangle}{\langle \mathrm{select}\ v_2\ v_2; \eta_N\langle \kappa \mapsto w\rangle \eta_L\rangle \Downarrow^{\iota} \langle \mathrm{Right}(v_1, u_2); \sigma'\rangle}$$

$$\frac{\kappa \in |\mathrm{cl}(v_1)| \cap |\mathrm{cl}(v_2)| \qquad \langle \mathrm{adv}\ v_1; \eta_N\langle \kappa \mapsto w\rangle \eta_L\rangle \Downarrow^{\iota} \langle u_1; \sigma\rangle \qquad \langle \mathrm{adv}\ v_2; \sigma\rangle \Downarrow \iota\langle u_2; \sigma'\rangle}{\langle \mathrm{select}\ v_1\ v_2; \eta_N\langle \kappa \mapsto w\rangle \eta_L\rangle \Downarrow^{\iota} \langle \mathrm{Both}(u_1, u_2); \sigma'\rangle}$$

The final keyword `select` takes two values and advances one or both of them depending on what channel has been advanced the current tick. Consider the expression `select a b` and let the updated channel be `k`. If `k` is in `cl(a)` but not in `cl(b)` then `a` is advanced and `Left(adv a, b)` is returned. If `k` is not in `cl(a)` but it is in `cl(b)` then `b` is advanced and `Right(a, adv b)` is returned. If `k` is in `cl(a)` and `cl(b)` then both `a` and `b` are advanced and `Both(adv a, adv b)` is returned. This ensures that only clocks which have ticked are advanced.

6 Transpilation

The interpreter described in section 5 implements Async RaTT based on the theory in[2]. However, interpretation comes with the set of problems described in section 3.4. In order to resolve these issues, transpilation from Async RaTT to OCaml has been implemented. The basic operation of the transpiler is traversing the AST and translating its components to OCaml compliant syntax.

Since much of the syntax and semantics of our concrete syntax and OCaml is similar, a large part of transpilation is trivial. For example, let-bindings, pattern matching, and anonymous functions can be transpiled to OCaml, by outputting the syntax almost as is. The only significant difference being our usage of a backslash instead of the `fun` keyword, for anonymous functions.

The parts that require more complex translation, and further explanation, are, once again, the terms that make Async RaTT special: `delay`, `adv`, `select`, `box`, `unbox`, `wait`, and `::`. Note that the `::` operator exists in both OCaml and Async RaTT, but differ semantically between the languages.

In Async RaTT clocks are sets of channels. We transpile channels to strings, i.e. their names, and clocks can therefore be transpiled to sets of strings, utilizing the Set module type of OCaml constructed over the String module. This allows for quick checking of whether a clock contains an updated channel.

In the following unless stated otherwise an underlined piece of text refers to the result of transpiling an Async RaTT term to OCaml. For example, $\underline{\texttt{t}}$ refers to the result of transpiling the Async RaTT term `t` to OCaml, and $\underline{\mathrm{cl(v)}}$ refers to the result of transpiling the clock of the Async RaTT value `v` to OCaml.

Delayed computations consist of a closure and a clock. We transpile the closure to values of type `unit -> 'a` and the clock to a `StringSet.t` resulting in pairs of `(unit -> 'a) * StringSet.t`.

This allows for executing the computation once the required data is available. Therefore, an expression `delay t` is transpiled to:

```
((fun () -> t), cl(t))
```

We need to transpile `cl(t)`, a `clock_set`, from the Async RaTT AST to a OCaml `StringSet.t`. The `clock_set` type contains two string sets, namely `channels` and `variables`. The `channels` are transpiled to the expression:

```
StringSet.of_list [c_1;...;c_n]
```

Where c_i is a unique string from the `channels` set, with n elements.

We also need to transpile the `variables` in the `clock_set` to OCaml. Since delayed computations are pairs consisting of a closure and a clock we can get the clock of a delayed computation, `b`, at runtime using:

```
snd b
```

Since `variables` is a set we can get all its contained channels at runtime by transpiling it to:

```
(List.fold_left StringSet.union StringSet.empty [snd b_1; ...; snd b_n])
```

Now that we know how to transpile both the `channels` and the `variables` of a `clock_set` we can transpile the entire `clock_set` by combining the two results of transpilation:

```
StringSet.union
    (StringSet.of_list [c_1;...;c_n])
    (List.fold_left StringSet.union StringSet.empty [snd b_1; ...; snd b_n]))
```

We transpile input channels, `k`, to a top-level let-binding:

```
let k_update = ref default_value_k
```

Where default_value_k is a default value of the type of channel `k`. This allows for both reading and writing an updated value on the channel `k`.

`adv t` advances a delayed computation. Since a delayed computation is a pair `(unit -> 'a) * StringSet.t`, an `Adv t` in the AST can be transpiled to `fst (t') ()` as this fetches and invokes the computation in question.

The `select` keyword advances one of its arguments depending on which clock has ticked. This is transpiled to lookups in the two sets of strings and a match expression, with three cases. The match expression computes one or both thunks depending on the matched pattern. For example the Async RaTT expression `select a b` is transpiled to:

```
match
    StringSet.mem !updated_channel_name cl(a),
    StringSet.mem !updated_channel_name cl(b) with
| true, true -> Both (fst a (), fst b ())
| true, false -> Left (fst a (), b)
| false, true -> Right (a, fst b ())
```

Where `updated_channel_name` is the name of the updated channel. The transpiled code performs a lookup of `!updated_channel_name` in each clock, and advances either `a`, `b`, or both

depending on the result of the lookups.

Because `box` creates a delayed computation it is simply transpiled to a thunk that evaluates the given term. That is, a value `box t` is transpiled to:

```
(fun () -> t)
```

Since `unbox t` evaluates the delayed computation `t` the term is transpiled to:

```
(t ())
```

which evaluates the thunk t.

`wait k` creates a delayed computation and can therefore be transpiled to values of type `(unit -> 'a) * StringSet.t`. In this case the set just contains a single channel, k. For example the expression `wait k` is transpiled to:

```
(fun () -> !k_update, StringSet.singleton "k")
```

Where `k_update` gets the most recent value on the channel `k`. Similarly, the expression `wait second` is transpiled to:

```
(fun () -> !second_update, StringSet.singleton "second")
```

In order to transpile the `::` operator we first need to define the signal type in the transpilation result. We define it as:

```
type 'a signal =
    | Cons of ('a * ((unit -> 'a signal) * StringSet.t))
```

That is, an `'a signal` consists of a value of type `'a` "now", and a delayed `'a signal`. With this, an expression `a :: b`, is transpiled to `Cons (a, b)`.

With all of these translations in place, it is possible to transpile terms of Async RaTT to OCaml expressions. An example of such a transpilation can be seen in listing 8.

```
let rec map_0 = fun f_1 -> fun s_2 -> match s_2 with
| Cons (x_3, xs_4) -> Cons (
  ( (f_1) () ) ( x_3 ),
  (
    (fun () -> (( ( map_0 ) ( f_1 ) ) ( fst (xs_4) () ))),
    (StringSet.union
      (StringSet.of_list [])
      (List.fold_left StringSet.union StringSet.empty [snd xs_4])
    )
  )
)
```

Listing 8: definition of 'map' from listing 13 transpiled to OCaml

7 Language guarantees

Figures 5 and 6 show how much memory was allocated to native complied programs written in Async RaTT throughout a 10-hour execution. Admittedly, we are not intimately familiar with the details of the OCaml garbage collection systems, but the figures do still suggest that after a while garbage collection is able to entirely satisfy the memory needs. Therefore, no longer requiring the kernel to allocate further memory to the process. In short, this suggests the absence of space-leaks.

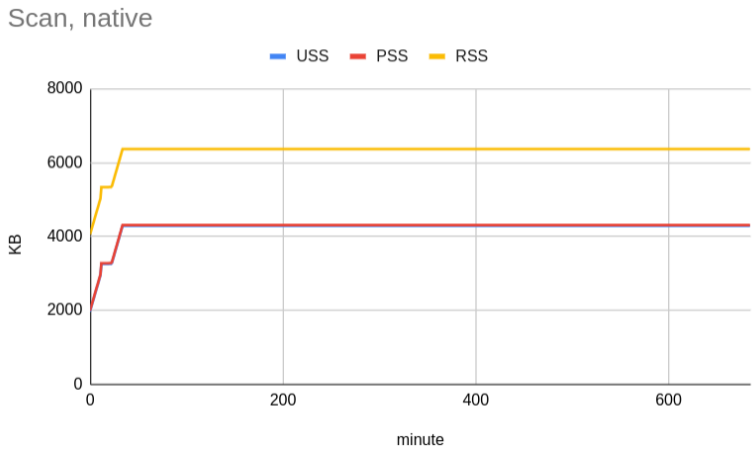

Figure 5: Memory allocated, in KB, to running the 'scan' program from listing 14, as a native executable, according to `smem` [2]

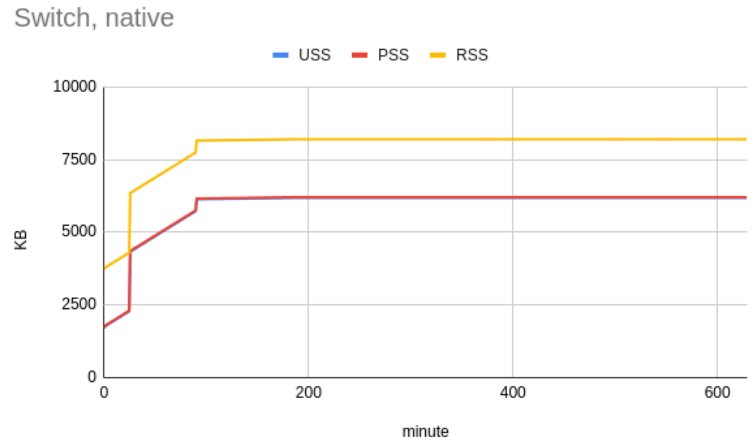

Figure 6: Memory allocated, in KB, to running the 'switch' program from listing 16, as a native executable, according to `smem` [4]

[2] `https://linux.die.net/man/8/smem`

The guarantees of causality and productivity are suggested to be present by our testing suite. Specifically, tests that attempt to type check invalid programs fail as they should. Listing 9 shows example functions that fail due to being unproductive, while 10 show examples of function that are not causal, i.e. relies on values from an earlier time step.

```
let rec func_0 : 'a -> 'a = \x -> func_0 x;

let rec func_1 : int -> int = \i -> if i = 0 then 1 else (func_1 i);
```

Listing 9: Non-productive recursive functions

```
let rec func : 'a -> 'a signal later = \x -> delay(x :: func x);

let rec foo = \f ->
    let x = box (f 7) in
    delay (x :: foo f);

let map = \f ->
    let rec run = \(x :: xs) ->
        f x :: delay(run (adv xs)) in
    run;
```

Listing 10: Non-causal functions

8 Future work

8.1 Input channels as values

Input channels being just a name, instead of a typed value, in some cases reduces code reusability. For example, writing a signal that counts how many times a channel has ticked must be done for each channel individually. Similarly, writing a signal that reads from a buffered channel when some push channel updates must be written for each combination needed. This quickly becomes quite tedious work.
Listing 11 gives an example of a few signals written as they would be, without channels as values, while listing 12 gives the same signals with channels as values.

[4] https://linux.die.net/man/8/smem

```
let second_count =
    let rec aux = \n ->
        delay (let x = wait second in n :: aux (n + 1)) in
    aux 0;

let minute_count =
    let rec aux = \n ->
        delay (let x = wait minute in n :: aux (n + 1)) in
    aux 0;

let rec key_time =
    delay (let x = wait keyboard in (read system_time) :: key_time);
```

Listing 11: Examples without input channels as values

```
let count = \channel -> (
    let rec aux = \n ->
        delay (let x = wait channel in n :: aux (n + 1)) in
    aux 0);

let rec bind = \push_c buf_c -> (
    delay (let x = wait push_c in (read buf_c) :: bind push_c buf_c)
);

let second_count = count second;
let minute_count = count minute;

let key_time = bind keyboard system_time;
```

Listing 12: Example with input channels as values

We have done some exploratory work on this feature but did not complete it. It does seem possible to implement it in a safe fashion, although it is not entirely trivial as the channel types have to work slightly differently to all other types currently implemented. For example, when passing a channel to a function expecting a push channel a push buffered channel should be allowed. This is because a push buffered channel can be used as a push channel.

Furthermore, with channels as values `wait` would essentially have the type `'a p channel -> 'a later`, while `read` would have the type `'a b channel -> 'a`, where `p` and `b` denoted push channels and buffered channels respectively. The abstract syntax should have a `Channel` constructor and the lexical analysis would need to be able to query the program environment for whether a certain string referees to an input channel. This essentially adds channel names as keywords.

Finally, verifying that this change to the language does not ruin any of the guarantees that it should provide would be needed.

8.2 Simultaneous channel updates

The reactive semantics of Async RaTT describe a system in which only one channel can have an update at a time. As such, when synchronizing two signals of strictly different clocks the *Both* case can never be executed. This may come as a surprise to users of the language as two simultaneous events won't be registered as occurring at the same time, unless they happen on the same clock. This can be solved by replacing the `update` binding in the `TwoHeap` constructor with a binding of type `string * runtime_value list`. This allows for storing updates on multiple channels simultaneously.

It has not been formally verified that having multiple simultaneous channel updates would keep the guarantees of the language. However, Async Rattus [1] does support this.

8.3 Nested delays

The typing rules of Async RaTT makes it such that one cannot explicitly write a `delay` inside the term of another `delay`. Defining a binding bound to the result of a `delay` and then referencing said binding inside a delay is however allowed. This is reflected in our implementation, but it might seem weird to anyone not familiar with the formal description of the language, who might be expecting referential transparency properties from the language. This could be alleviated by a program transformation step that moves any nested `delay` terms to bindings outside the parent `delay` term.

8.4 Pattern totality checking

The presented version of the language implementation contain pattern matching. However, it does not perform any checking of whether or not the alternatives cover every possible pattern of the term being matched on. This is problematic as it allows the user to write partial functions which will crash at run time.

Adding this feature should be possible considering its inclusion in other functional languages. One way would be to study and implement the algorithm described by Maranget [6].

9 Related work

In this paper a standalone implementation of Async RaTT is presented. This approach gives a lot of flexibility, allowing us to build and design the language infrastructure to best support the theory. This of cause comes with the downside that everything must built manually.

Another approach would be to implement Async RaTT as an embedded language, providing the implementation with access to the host language's infrastructure and ecosystem. This was explored by Houlborg et al.[5] who embedded Async RaTT into Haskell as a compiler plugin.

Using the functional language paradigm as a platform is not the only choice when designing a reactive language. Oeyen et al.[8] presents the reactive language Haai. A pure reactive programming language using 'reactors' as its unit of computation.

10 Conclusion

The goal of this project was to implement the Async RaTT language described by Møgelberg and Bahr [2]. The project has resulted in an adaptable standalone language implementation. The language is expanded with features and inference systems broadening what can be expressed. This includes pattern matching, ADTs, type inference inspired by Algorithm W which is described and proven by Milner and Damas [7][3], inference of type stability constraints, and inference of term clocks.
The inference rules of stable types and clocks have been formally defined and implemented.
Our empirical evidence, presented in section 7 in the form of testing and memory measurements, suggests that the guarantees of Async RaTT are kept by the implementation.

References

[1] Patrick Bahr, Emil Houlborg, and Gregers Thomas Skat Rørdam. "Asynchronous Reactive Programming with Modal Types in Haskell". In: *Practical Aspects of Declarative Languages*. Ed. by Martin Gebser and Ilya Sergey. Cham: Springer Nature Switzerland, 2023, pp. 18–36. ISBN: 978-3-031-52038-9.

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

[3] Luis Damas and Robin Milner. "Principal type-schemes for functional programs". In: *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1982, pp. 207–212.

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

[5] Emil Houlborg, Gregers Rørdam, Patrick Bahr. *AsyncRattus: An asynchronous modal FRP language*. URL: https://hackage.haskell.org/package/AsyncRattus. (accessed: 26-01-2024).

[6] Luc Maranget. "Warnings for pattern matching". In: *Journal of Functional Programming* 17.3 (2007), pp. 387–421. DOI: 10.1017/S0956796807006223.

[7] Robin Milner. "A theory of type polymorphism in programming". In: *Journal of computer and system sciences* 17.3 (1978), pp. 348–375.

[8] Bjarno Oeyen, Joeri De Koster, and Wolfgang De Meuter. "Reactive Programming without Functions". In: *The Art, Science, and Engineering of Programming* 8.3 (Feb. 2024). ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2024/8/11. URL: http://dx.doi.org/10.22152/programming-journal.org/2024/8/11.