Rizzo memory management

Master of Science in Computer Science
Master's Thesis
KISPECI1SE

IT University of Copenhagen

Patrick Bohn Matthiesen `pmat@itu.dk`
Tobias Skovbæk Brund `tbru@itu.dk`
Supervisor: Patrick Bahr `paba@itu.dk`
GitHub: `https://github.com/itu-msc/RizzoMemory`

May 30, 2026

Abstract

This thesis presents an implementation of Rizzo by Bahr [2]. We extend Rizzo with reference counting and memory reuse as presented by Ullrich and Moura [9]. We develop a compiler that transforms Rizzo programs into a form that adheres to the structural requirements of Ullrich and Moura [9] and outputs C code.

The main contribution of this work is showing that reference counting enables the immediate erasure of signals, which is necessary to avoid duplicate work. We also show that memory reuse for signals is generally limited, since recursive signals combinators depend on their input signals to compute future values. In contrast, reuse of general values works well, and signal head values can sometimes even be reused repeatedly.

We conclude that reference counting is a viable memory management strategy for Rizzo, and that our implementation supports the core language features needed to evaluate it.

Contents

1 Introduction **4**
- 1.1 Functional Reactive Programming . . . 5
- 1.2 Rizzo . . . 6
  - 1.2.1 Signals and the heap . . . 9

2 Reference Counting **10**
- 2.1 What is reference counting . . . 10
  - 2.1.1 Naïve reference counting . . . 11
- 2.2 Counting immutable beans . . . 12
  - 2.2.1 Reference Counting with Ownership . . . 13
- 2.3 AST structure requirements . . . 15

3 RizzoLang Syntax and Semantics **16**
- 3.1 Syntax . . . 17
  - 3.1.1 Top-level Declarations . . . 17
  - 3.1.2 Expressions . . . 17
  - 3.1.3 Patterns . . . 18
- 3.2 Types . . . 19
  - 3.2.1 Primitive Types and compound types . . . 19
  - 3.2.2 Modal Types: Later and Delayed . . . 19
- 3.3 Semantics . . . 20
  - 3.3.1 Semantics: Rizzo . . . 21
  - 3.3.2 Piecing it all together . . . 23

4 Automatic reference counting of Rizzo **25**
- 4.1 Transformations . . . 27
- 4.2 Counting Rizzo Beans . . . 29
  - 4.2.1 Reusing memory . . . 29
  - 4.2.2 Can I borrow that? . . . 31
  - 4.2.3 Learning to count . . . 32

5 Evaluation **36**
- 5.1 Signal heap behaviour . . . 36
- 5.2 No bit is forgotten . . . 40
- 5.3 Reusing memory at runtime . . . 42

6 Limitations and Future Work **45**
6.1 Reference counting overhead in signal combinators 45
6.2 Specialised implementations of combinators 46
6.3 The later applications forgotten to time 46
6.4 To avoid allocating intermediate signals 48
6.5 The gain of indirections 49
6.6 Improving the structure of the signal heap 51

7 Conclusion **52**

References **55**

A Heap Behaviour Evaluation

B Valgrind evaluation
B.1 Program . . . . . .
B.2 Program output . . . . . .
B.3 Advance function . . . . . .
B.4 Advance function lowered to IR . . . . . .

1 Introduction

Functional reactive programming (FRP) is a programming paradigm for building *reactive* programs in a declarative and functional style. FRP languages model time-varying values as first-class values called *signals*. However, without proper restrictions, this style of programming can easily lead to programs that are non-causal, non-productive, or contain space leaks.

The FRP language Rizzo [2] solves these issues using modal types and operational semantics for updating signals in place. Rizzo maintains a global heap of signals from which all signals are updated when an input is received from the environment. This process may create intermediate signals which serve no purpose after an update and should therefore be erased early to avoid superfluous work. This makes memory management particularly important for Rizzo. Reference counting is a strategy for managing dynamic memory that allows immediate erasure of unused heap objects, and this property is ideal for Rizzo.

In this thesis, we evaluate reference counting as a memory management strategy for Rizzo. To do so, we adapt the work of Ullrich et al. [9], who present an algorithm for reference counting functional programming languages with memory reuse for *functional-but-in-place* programming. To transform Rizzo programs into the intermediate representation required by their algorithm, we develop a lexer, parser, and transformation pipeline. We then evaluate this memory management approach by observing the memory behaviour of transpiled Rizzo programs.

The results of our thesis show that reference counting efficiently deallocates signals, avoiding the unnecessary work that would otherwise occur. We also show that reuse works well for most objects like lists, but does not work for signals created from recursive signal combinators.

1.1 Functional Reactive Programming

Reactive programs are interactive programs that communicate with the environment through inputs and outputs. Examples of such programs include servers, graphical user interfaces, and games.

Functional Reactive Programming (FRP) is a programming paradigm that combines functional programming with reactive programming. It allows developers to create programs that can react to changes in data over time, making it easier to handle asynchronous events and manage state in a declarative way.

The main abstraction of FRP is the *signal*, which models time-varying values. Conceptually, a signal is a sequence of values produced by interacting with the environment. Signals can be transformed, combined, and manipulated using familiar functional techniques.

However, representing signals as infinite streams makes it easy to write declarative programs that cannot be run effectively. Such programs may violate one of the three main properties: *causality* (computations are not dependent on the results of future computations), *productivity* (computations eventually produce an output), and the *absence of space leaks*.

To overcome these issues, there exists a family of FRP languages [7, 4, 2] that use *modal types* to guarantee these properties. In the literature [4, 2], these modal types are the `Later` (written in literature as $\bigcirc a$) and the `Box` (written $\Box a$) type constructors. The type `Later` a expresses a promise that a value of type a will become available at some later time step, whereas `Box` a describes a value of type a that is *time-independent* and therefore safe to keep across time steps.

Given these types, it is possible to define a signal as a tuple of a current value (head) and a delayed computation (tail) for computing future values: type `Signal` $a = a \times$ `Later` (`Signal` a).

1.2 Rizzo

Rizzo [2] is an asynchronous FRP language designed to provide the same properties, but with a simpler type system. Rizzo changes the semantics of signals by defining them as mutable references:

$$\text{type } \texttt{Signal } a = \texttt{Ref } (a \times \texttt{Later } (\texttt{Signal } a))$$

Signals are updated in place whenever the environment produces a value that is relevant for the signal. These updates are performed not by the programmer, but by the reactive semantics of Rizzo; see Section 1.2.1. For convenience, we define the signal constructor as an infix operator:

$$\_ :: \_ : a \to \texttt{Later } (\texttt{Signal } a) \to \texttt{Signal } a$$

Rizzo, like other languages [4], differentiates between two variants of the 'later' type. Rizzo has `Later` a, which describes a value of type a becoming available at some future time step. The "some" here is existential; we do not know exactly when the value will arrive, only that it may eventually arrive. It also has `Delayed` a, which describes a value of type a that is available at *all* (universally quantified) future time steps. This universally quantified nature makes `Delayed` values safe to use across time boundaries.

Rizzo has a number of constructors for working with delayed computations [2]. We shall present some of them here, but for a full overview, see Section 3.2. Starting with the `Delayed` type, its main constructor is $\mathsf{delay} : a \to \texttt{Delayed } a$ which simply pushes any value into the future. Then for the `Later` type, we may use the constructor $\mathsf{wait} : \texttt{Chan } a \to \texttt{Later } a$, which is used to wait for the environment to produce input on a particular channel. One can think of a channel (`Chan` type) as an opening through which the environment can pass values to Rizzo programs. Another later constructor is $\mathsf{tail} : \texttt{Signal } a \to \texttt{Later } (\texttt{Signal } a)$, which lets us access the delayed computation for the future value of a signal. However, probably the

most useful of all is the applicative action on `Later` computations:

$$\mathsf{laterapp}\ f\ x : \texttt{Delayed}\ \ (a \to b) \to \texttt{Later}\ a \to \texttt{Later}\ b$$

Intuitively, laterapp delays the application of a function until the argument x is available. This operator allows us to apply delayed functions to later values. However, when writing Rizzo programs it is often more convenient to use the functorial action or pipe operator:

$$\begin{aligned} &\_ \triangleright \_ : (a \to b) \to \texttt{Later}\ a \to \texttt{Later}\ b \\ &f \triangleright x = \mathsf{laterapp}\ (\mathsf{delay}\ f)\ x \end{aligned}$$

With these constructors, we already have the necessary tools to define some signal combinators! We first define *mk_sig*, which lets us create a signal from any delayed computation:

$$\begin{aligned} &mk\_sig : \texttt{Later}\ \ a \to \texttt{Later}\ \ (\texttt{Signal}\ \ a) \\ &mk\_sig\ d = (\lambda x.x :: mk\_sig\ d) \triangleright d \end{aligned}$$

Assuming there exists a channel *console* : `Chan String` that reads from the console, we can define signals that react to console input. An example of such a signal is ("" :: *mk_sig* (wait *console*)), which echoes strings read from the console.

Let us take it one step further and define the familiar *map* function.

$$\begin{aligned} &map : (a \to b) \to \texttt{Signal}\ \ a \to \texttt{Signal}\ \ b \\ &map\ f\ (x :: xs) = f\ x :: (map\ f) \triangleright xs \end{aligned}$$

In our definition of *map* we make use of pattern matching on signals. We could alternatively have used explicit head and tail operations to extract x and xs respectively.

The previous combinators transform a single signal without changing where future values come from. In reactive programs we often need a signal to follow one behaviour until some later computation produces a new signal. We call this operation *switch*.

To define it, we use Rizzo's sync primitive. Given two delayed computations which may produce values asynchronously, sync waits until either one produces a value. Its result records whether the left, right, or both computations produced a value:

$$\mathsf{sync}\ x\ y : \mathtt{Later}\ \ a \to \mathtt{Later}\ \ b \to \mathtt{Later}\ \ (\mathtt{Sync}\ \ (a,\ b))$$

where $\mathtt{Sync}\ \ (a,\ b)$ has constructors $\mathsf{Left}\ a$, $\mathsf{Right}\ b$, and $\mathsf{Both}\ (a,\ b)$.

The *switch* combinator initially behaves like its first signal. When the signal ticks, if the default Left case of sync is triggered, *switch* recursively continues to behave like the first signal. If the Right case is triggered, *switch* immediately switches to the new signal. If both cases are triggered in the same time step, *switch* also immediately switches to the new signal.

$$\begin{aligned}
&\mathit{switch} : \mathtt{Signal}\ \ a \to \mathtt{Later}\ \ (\mathtt{Signal}\ \ a) \to \mathtt{Signal}\ \ a \\
&\mathit{switch}\ (x :: \mathit{xs})\ d = x :: (\mathit{cont} \triangleright \mathsf{sync}\ \mathit{xs}\ d) \\
&\quad \textbf{where}\ \mathtt{cont} : \mathtt{Sync}\ \ (\mathtt{Signal}\ \ a,\ \mathtt{Signal}\ \ a) \to \mathtt{Signal}\ \ a \\
&\qquad\qquad \mathtt{cont}\ \ (\mathsf{Left}\ \mathit{xs}') = \mathtt{switch}\ \mathit{xs}'\ d \\
&\qquad\qquad \mathtt{cont}\ \ (\mathsf{Right}\ d') = d' \\
&\qquad\qquad \mathtt{cont}\ \ (\mathsf{Both}\ _\ d') = d'
\end{aligned}$$

Let us return to what makes Rizzo different, its semantics of signals. Because signals are updated in place as mutable references rather than stored as immutable sequences, it becomes possible to define combinators such as

sample:

$$sample : \texttt{Signal}\ a \to \texttt{Signal}\ b \to \texttt{Signal}\ (a \times b)$$
$$sample\ xs\ ys = map\ (\lambda x.(x,\ \mathsf{head}\ ys))\ xs$$

The *sample* combinator samples the current value of ys each time a new value is produced on the signal xs. If signals were immutable, the combinator would instead reuse the first value of ys for every subsequent value of xs, which would force the runtime to retain the entire history of ys and would therefore introduce a space leak.

1.2.1 Signals and the heap

In Rizzo, time steps forward when a value is produced by any channel. This could happen because of a network message, a sensor reading, or user input from the console. Time steps are discrete and sequential, meaning they happen one after another and never at the same time.

When time steps forward, the data stored in every signal becomes stale since it now belongs to the past. The runtime must therefore bring all signals into the new time step. It does so by iterating over the heap, evaluating each signal's tail to obtain a potentially fresh intermediate signal, and writing the resulting data back into the original signal. Without the proper clean-up, this process leaves both the intermediate signal and the original signal alive, both of which must then be brought into future time steps, leading to duplicate work.

Reference counting provides the precise clean-up needed to avoid this problem. It allows the runtime to detect when an intermediate signal is no longer referenced and can be immediately discarded, thus avoiding unnecessary work during future heap updates.

2 Reference Counting

This section introduces reference counting as a memory management strategy and explains how the approach of Ullrich et al. [9] uses **reset/reuse** instructions to reduce memory allocations and reference ownership information to reduce the overhead of reference counting. We then present the constraints required by the algorithm of Ullrich et al. [9], since these become important when we later adapt the algorithm to Rizzo in Section 4.

2.1 What is reference counting

Early work on reference counting like that of Collins [5] introduced the idea of reference counting as an alternative to other memory management methods like tracing garbage collection. The basic idea of reference counting is to attach a counter to each object in memory that keeps track of how many references point to the object. When a new reference is created, the counter is incremented, and when a reference is destroyed, the counter is decremented. When the counter reaches zero, it means that there are no more references to the object, and it can be safely deallocated.

Early implementations were implemented in the runtime system of the programming language. This was a common approach, but it is hard to optimize. Newer techniques, like the one proposed by Ullrich et al. [9], do static analysis during compilation in order to achieve better performance and more efficient memory management.

By using new **reset** and **reuse** instructions and ownership annotations, the compiler can determine when memory of immutable data can be safely overwritten, and when increment/decrement instructions can or cannot be safely omitted.

Later in this section, we will be using the term *RC token* (Reference Counting token). An RC token is one counted reference to an object. A token is created when the reference count is incremented, and it is consumed

when the reference count is decremented, either directly or by being passed to a context that eventually decrements it, such as a function or a memory allocated value.

2.1.1 Naïve reference counting

In a naïve reference counting implementation, every time a new reference to an object is created, the reference count of that object is incremented. This means the creator of the object has to increment the reference count, and every time the object is taken as input, the reference count has to be incremented. We have to do this because we cannot be sure if the object is shared or not, and we have to be safe. This can lead to a lot of unnecessary increments and decrements, especially in cases where many short-lived objects are created and destroyed, or an object is passed multiple times to a function.

Using a notation close to the reference counted intermediate language generated by the compiler, we can write an example of a naïve reference counting implementation. Let us assume we have a function f that takes an object as input, and then we use it from the main function.

```
fun f x =
    inc x;              -- ref: 2
    ...                 -- use x
    dec x;              -- ref: 1

let main =
    let z = Obj() in    -- start count 0
    inc z;              -- ref: 1
    f z;
    dec z;              -- end count 0
```

As we can see, the reference to the object is being incremented twice and decremented twice. We increment it before we pass it to f, and then have to increment it again inside f because we don't know how the object is used. We then have to decrement it twice, once inside f before we return, and once

after we call f in the main function. This is quite inefficient, especially if we passed z further to another function inside f.

2.2 Counting immutable beans

Ullrich et al. [9] gain their performance improvements by using a combination of ownership annotations, and new **reset** and **reuse** instructions. The ownership annotations in "Counting immutable beans" [9] work by distinguishing between $\mathbb{O}wned$ and $\mathbb{B}orrowed$ references, and let the compiler track which parts of the program are responsible for managing the memory of each object. The compiler can then use this knowledge to determine when an object needs to be incremented or decremented, which reduces the number of reference counting operations that need to be performed. For owned references we need to ensure that the object is alive as long as we need it, which is similar to the naïve reference counting implementation. If we instead borrow an object, then we assume that the caller has the responsibility to ensure that the object is alive as long as we need it. This means we can avoid incrementing the reference count of the object, which is a great performance improvement in nested scopes where all the functions can borrow their parameters. In this case, we can increment the reference count once at the top of the scope, instead of incrementing it for each function call as we would need to if it was owned. We can typically borrow an object when we only need to read from it, and then own the object if we intend to reuse it.

We can then extend this with the new **reset** and **reuse** instructions. These instructions allow the compiler to optimize memory allocations of new objects by reusing memory from old objects that are no longer needed, instead of always allocating new memory for each object. This leads to performance improvements, especially in cases where many short-lived objects or nested objects are created and destroyed.

The paper defines three main steps for implementing their reference counting technique, where each step depends on the previous one:

- Inserting **reset**/**reuse** pairs
- Inferring borrowed parameters
- Inserting **inc**/**dec** instructions

2.2.1 Reference Counting with Ownership

To show how ownership annotations and **reset**/**reuse** instructions can optimize reference counting, we will go through some examples. These examples will use similar notation as the one we used in the naïve example. The notation is based on our intermediate representation, where **inc** and **dec** are used to increment and decrement the reference count of a variable, while **reset** and **reuse** are used to reclaim and recycle storage.

As a first example, consider the expression below where the same variable is passed twice to a function:

```
let z = f y y in
ret z
```

Assume that the function f takes two parameters and consumes an RC token for each of them. Consuming an RC token for a parameter will mark the parameter as $\mathbb{O}wned$, which means that the caller is responsible for the reference count of the objects. In this case, since y is passed twice to f, both occurrences of y are treated as owned, and both will consume an RC token. The compiler must then insert an **inc** before the call to make the second owned use valid:

```
let y = Obj() in
inc y;
let z = f y y in
ret z
```

No matching **dec** is needed afterwards, since both RC tokens are consumed by the call itself. This is the safe but cautious behaviour we get when both arguments are treated as owned.

Borrowing may remove some of this overhead. If we instead assume both parameters of f are borrowed, then there will only be a single decrement instruction to clean up the Obj stored in y:

```
let y = Obj() in
let z = f y y in
dec y;
ret z
```

However, sometimes it can lead to additional or at least the same overhead. If the first parameter of f is borrowed and the second is owned, then only one of the two uses of y consumes an RC token. The borrowed use still requires the object to stay alive for the duration of the call, so the compiler inserts a temporary increment before the call and a matching decrement afterwards:

```
let y = Obj() in
inc y;
let z = f y y in
dec y;
ret z
```

Compared to the previous example, this makes the ownership discipline more precise: one occurrence of y is only observed, while the other is actually consumed. The compiler can therefore avoid treating both arguments as fully owned.

The more interesting optimisation is enabled by the combination of **reset** and **reuse**. In Rizzo this matters when a combinator stops using an input signal and immediately constructs a replacement signal. Consider the function below:

```
fun stop f (x :: xs) : ('a -> Bool) -> Signal 'a ->
   Signal 'a =
    if f x
    then x :: never
    else x :: (stop f |> xs)
```

In the 'then' branch, the input signal $(x :: xs)$ is not used. This makes it possible to reuse the memory cell to store the new signal $x ::$ never, avoiding an allocation.

Not every signal combinator admits this optimisation. For a function such as *map*, the input signal must remain alive because its tail is needed to produce future values:

```
fun map f (x :: xs) =
    f x :: (map f |> xs)
```

Here the delayed computation still depends on xs, so reusing the storage of s would destroy data that is needed later. This illustrates why reuse opportunities for signals in Rizzo are limited: they mainly arise in combinators that terminate, switch, or otherwise stop forwarding the original signal.

2.3 AST structure requirements

To implement the reference counting technique proposed by Ullrich et al. [9], the shape of the abstract syntax tree (AST) must satisfy the following properties:

- Eta-expanding constructor applications to ensure that all constructors are fully applied.
- Splitting over-applied constant applications into multiple applications to ensure that all constant applications are fully applied.
- Splitting variable applications into multiple applications to ensure that all variable applications take only one argument.

- Lambda-lifting all function abstractions to the top level to ensure that all functions are defined at the top level of the program.
- Eliminating trivial bindings of the form `let x = y` through copy propagation, to simplify the AST and reduce unnecessary bindings.
- Removing all dead let bindings to clean up the AST and remove unused variables.
- Ensuring that all parameter and let names of a function are mutually distinct to avoid name capture issues during transformations.

We define two separate abstract syntaxes: one for the Rizzo language, described in Section 3, and another for the intermediate representation (IR) used in the insertion of reference counting instructions, described in Section 4. To ensure the properties above and to convert from the Rizzo AST to the IR, we define a series of transformations in Section 4.1.

3 RizzoLang Syntax and Semantics

For our implementation of Rizzo, we have designed an ML-like surface syntax on top of the syntax presented by Bahr [2], extending the syntax with let-bindings, function definitions, and pattern matching.

We chose an ML-inspired syntax because it is familiar to functional programmers and it is syntactically close to the representation used by Ullrich et al. [9], allowing us to easily translate between the two.

In this section we first present the concrete syntax of Rizzo. We then describe the type system with a focus on the modal types `Later` and `Delayed`. After that, we describe the operational semantics of Rizzo, the constraints the semantics put on the choice of heap representation, and finally how we merge the semantics with the reference-counting semantics of Ullrich et al. [9].

3.1 Syntax

We have extended the base ML-like syntax with constructs specific to Rizzo, such as the signal constructor (`::`) and the `Later`/`Delayed` constructors. The syntax supports top-level definitions, local let bindings, lambda expressions, pattern matching, and type annotations.

3.1.1 Top-level Declarations

Programs in Rizzo consist of a sequence of top-level declarations. There are three forms: value bindings, function definitions, and type definitions.

```
let x = e                              (* value binding *)
fun f x1 x2 ... xn = e                 (* function definition *)
type TypeName x1 ... xn =
| C (t1, ..., tn) ...                 (* type definition *)
```

The function definition `fun f x1 ... xn = e` is syntactic sugar for `let f = fun x1 ... xn -> e`. Functions marked with the `@effectful` annotation are allowed to perform side effects such as registering output signals. This annotation also prevents the compiler transformations of Section 4.1 from removing calls to such functions.

3.1.2 Expressions

The core expression forms are:

```
e ::= x                                (* variable *)
    | c                                (* constants *)
    | fun p -> e                       (* lambda abstraction *)
    | e1 e2                            (* application *)
    | C (e1, ..., en)                  (* constructor application *)
    | let x = e1 in e2                 (* local binding *)
```

```
    | (e1, e2)                    (* tuple *)
    | e1 :: e2                    (* signal/list constructor *)
    | e1 |> e2                    (* pipe operator *)
    | match e with | p1 -> e1 ... (* pattern matching *)
    | if e1 then e2 else e3       (* conditional *)
    | (e : t)                     (* type annotation *)
    | delay e | ostar e1 e2       (* Delayed constructors *)
    | wait e | watch e | tail e
    | sync e1 e2 | laterapp e1 e2 (* Later constructors *)
```

Constants can be booleans, either false or true, integers, strings, and the never constructor of the `Later` type. The pipe operator `|>` is left-associative and desugars to an instance of the later application `laterapp (delay e1) e2`. The signal constructor `::` is right-associative and constructs a signal from a head value and a tail computation. For example, 5 :: never creates a constant signal with value 5.

3.1.3 Patterns

For patterns we have included the necessary forms to support matching against tuples, signals, and constructors:

```
p ::= _                         (* wildcard *)
    | x                         (* variable binding *)
    | c                         (* constant pattern *)
    | (p1, p2)                  (* tuple pattern *)
    | p1 :: x                   (* signal/cons pattern *)
    | c :: x                    (* string pattern *)
    | C (p1, ..., pn)           (* constructor pattern *)
```

Like other head and tail structured datatypes, we have chosen to use the $x :: xs$ pattern for signals, to bind the head of the signal to x and the tail

(a `Later` computation) to xs. We also added it for strings, to allow the head character to be bound separately from the tail string. This is useful for functions that need to process strings, such as when parsing user input.

3.2 Types

Rizzo's type system includes primitive types, compound types, and the `Later` and `Delayed` modal types that are central to functional reactive programming. In this section we describe the types available in our implementation.

3.2.1 Primitive Types and compound types

The language includes a small set of primitive types: `Int`, `Bool`, `String`, and the unit type `Unit`. These are simple by themselves, but can be combined using tuples and function types to create more powerful abstractions.

```
t ::= Int | Bool | String | Unit   (* primitive types  *)
    | TypeName                     (* named types *)
    | t1 * t2                      (* tuple types      *)
    | t1 -> t2                     (* function types   *)
    | Later t                      (* later type       *)
    | Delayed t                    (* delay type       *)
    | Signal t                     (* signal type      *)
    | t1 (t2, ..., tn)             (* type application *)
```

3.2.2 Modal Types: Later and Delayed

The modal types `Later` and `Delayed` are what distinguish Rizzo from ordinary functional languages. As introduced in Section 1.2, these types encode temporal properties of values. Bahr [2] introduce a set of constructors for these modal types. For the `Later` type, the constructors are:

- `never` : `Later` a is the delayed computation that never produces any value.

- wait c : `Later` a waits for an input on channel c : `Chan` a to produce a value.
- watch s : `Later` a watches a partial signal s : `Signal` (`Option` a) and will produce a value when the head of s is updated to a value of the form Some(x) for some $x : a$.
- tail s : `Later` (`Signal` a) builds a delayed computation to access the future value of a signal s : `Signal` a.
- laterapp f x : `Later` b is the applicative action on `Later` values. It builds a delayed computation to apply f : `Delayed` $(a \to b)$ to x : `Later` a in the future time step when x produces its value.
- sync x y : `Later` (`Sync` (a, b)) is the primitive for handling the asynchronous nature of Rizzo. This constructor builds a delayed computation that will produce a value when either x : `Later` a or y : `Later` b, or both produce a value. This either-or behaviour is captured by the `Sync` (a, b) type with its constructors Left(x'), Right(y'), and Both(x', y').

and for the `Delayed` type the constructors are:

- delay e : `Delayed` a delays the expression $e : a$ to a future time step.
- ostar f x : `Delayed` b builds a delayed computation that applies f : `Delayed` $(a \to b)$ to x : `Delayed` a in any future time step.

3.3 Semantics

For our language implementation we combine the semantics of Rizzo [2] and the reference counted semantics of Ullrich et al. [9]. This section aims to provide a description of how these two worlds merge to become RizzoLang, rather than a comprehensive explanation of each. To do so, we first provide

an abstract description of the semantics, then give a more concrete account of how this is implemented in the runtime of the language, and finally how reference counting is fitted in.

3.3.1 Semantics: Rizzo

In Rizzo [2], the operational semantics are split in two: the evaluation semantics for expressions, and the reactive semantics that describe how Rizzo programs react when time is stepped forward.

The evaluation semantics follow a standard eager evaluation strategy. The only exception is delay e, for which the argument e is lazily evaluated. We make one deliberate deviation from the semantics presented by Bahr [2]. We omit the `fix`-point construct, which is used to guarantee productivity by guarding recursion with the `Delayed` type. That is, our implementation makes no guarantees about productivity, a trade-off we are willing to make given the usefulness of general recursion in functional programming.

The reactive semantics define the reactive behaviour of Rizzo, i.e. what happens when any channel has produced a value and time is moved forward. These semantics are split into three parts: the *advance semantics*, which describe how a `Later` A value is advanced to produce a value of type A; the *update semantics*, which, using the advance semantics, describe how a signal is updated and brought forward in time; and the *step semantics*, which describe the process of applying the update semantics to every signal when time is *stepped* forward.

For the purpose of memory management, we will focus our attention on the update semantics but first we take a detour to understand the conceptual structure of the heap. In the typical way, the heap holds onto all dynamically allocated data such as tuples, signals, function closures, etc. Among these, signals are special because they must be traversed as part of the reactive semantics. Rizzo therefore distinguishes between the *heap*, storing all dynamically allocated data, and the *signal heap*, a subset of the heap con-

taining only signals. This signal heap is further divided into two parts: the `now heap`, storing only signals that belong to the current time step; and the `earlier heap`, which holds all the stale signals that need to be updated.

At runtime when some channel **C** produces a value, the step semantics describe that all signals must be moved to the `earlier heap`, marking their data as stale. Then, as described by the update semantics, each signal is moved one by one from the `earlier heap` to the `now heap`. To move a signal, we first check if its tail depends on **C** and if so, the tail is *advanced* to obtain a new signal whose data is copied into the original signal, and the original signal is moved to the `now heap`. Otherwise, the signal is moved to the `now heap` unchanged.

However, since chains of signals built using combinators form data dependencies, the move ordering is significant. Boiled down, the reactive semantics require that:

- Any new signal must only depend on other signals in the `now heap`.
- Signals must be updated based on a topological order of their dependencies. If signal B depends on signal A, signal A must appear before signal B in the heap and thus be updated first.
- If no reference exists to the intermediate signal produced by evaluating the tail, then the intermediate signal must be deallocated to prevent duplicate work.
- New signals must be inserted into the `now heap`, to prevent them from being evaluated in the current time step.

Based on these requirements, we have chosen to structure our heap as a linked list. Using a linked list has the following advantages:

- It represents the `now` and `earlier` heaps in one data structure, and expresses the split between them using a pointer (cursor).

- It moves signals from the `now` heap to the `earlier` heap in O(1) time by resetting the cursor to the start of the heap. Likewise, it moves signals back to the `now` heap by moving the cursor forward.
- It supports O(1) insertion that maintains the topological order of signals by adding them at the end of the `now` heap. This ensures that any signal can only depend on signals that were created before it.
- O(1) removal of signals when their reference count drops to zero.
- Stable references to signals, as insertions and removals do not move other signals in memory, so pointers to signals remain valid throughout their lifetime.

3.3.2 Piecing it all together

The evaluation semantics of Rizzo [2] and the operational semantics of Ullrich et al. [9] are both built on the same core: an eagerly evaluated lambda calculus. This common structure allows us to reuse the semantics of Ullrich et al. [9] for most of Rizzo, including both the evaluation rules and the reference counting.

In the semantics of Ullrich et al. [9], there are only two kinds of heap-allocated values: function closures, which represent partial applications, and constructor values of the shape $\mathbf{ctor}_i$, which represent values created using the i-th constructor of a tagged union. This is already enough to model most of Rizzo's data constructors. In particular, values of the `Sync`, `Later`, and `Delayed` types can be represented as regular tagged unions, where each constructor is assigned its own tag.

The exception is the `Signal` type. Although signals, like other constructors, are just containers of values, they must be treated differently because both their creation and erasure modify the signal heap. We therefore introduce a specialised **signal** value with five fields: the canonical *head* and *tail*, and three fields used by the runtime. These runtime fields consist of a flag U

required by the advance semantics and two pointers *prev* and *next* that link the signal into the signal heap. The **signal** constructor is evaluated like any other $\mathbf{ctor}_i$, except for an additional step that inserts the resulting **signal** value into the linked-list structure of the signal heap. When a signal is later erased, its fields are decremented and the signal is removed from the signal heap. Similarly, when a signal is reset, it is removed from the signal heap, and its fields are decremented. This makes reuse possible by overwriting the fields and reinserting the signal into the signal heap like a new signal.

There is one additional adjustment required for `Delayed` values. In particular, the constructor delay e should behave as call-by-name, but this is not enforced by the operational semantics since such values are treated as $\mathbf{ctor}_i$ values. Instead, the AST is rewritten so that expressions of the form delay e become delay $(\lambda x.e)$, effectively wrapping e in a thunk.

This leaves us with the interaction between reference counting and the reactive semantics. Although we treat the advance semantics as a black-box, its interface must consume an RC token of any later value it receives. When the update semantics call advance on a signal's tail, advance is therefore responsible for decrementing that tail once it is no longer needed. This ensures later values, and the data they reference, are released as early as possible. This is crucial for the success of **reset**/**reuse**. If advance only borrowed the later value, the update semantics would have to perform the decrement after advance, which would block the reuse of memory captured by the later value. We base these requirements on the result of running the program in Appendix B.3 through the reference counting algorithm of Section 4 to achieve the pseudo-code from Appendix B.4.

With this interface in place, reference counting in the update semantics proceeds as follows. If a signal does not produce a new value, nothing further happens. Otherwise, we first decrement the original head, since it is now stale and may still hold references that would block reuse. We then advance the tail, which consumes the reference and yields an intermediate signal.

The head and tail of that intermediate signal are incremented before being copied back into the original signal, after which the intermediate signal itself is decremented. Once the entire heap has been updated, we finally decrement the value that triggered the update process.

Having established the semantics of the language, we can make the distinction between the heap and the signal heap concrete. Figure 1 illustrates an example of the Rizzo heap as a sequence of mappings from locations l_i to values and reference counters C_i. Notice that regular constructor values and signal values exist in the same heap. The signal heap is the linked-list structure formed by the `prev` and `next` fields of signal values and it is rooted at the heap base. In this example, the heap base is l_2, whose `prev` field is empty, and following along the `next` pointers we get the signal heap l_2, l_n, l_m.

$$
\begin{array}{rcl}
l_0 & \mapsto & \mathbf{ctor}_0(),\ C_0 \\
l_1 & \mapsto & \mathbf{ctor}_0(42, 12),\ C_1 \\
l_2 & \mapsto & \mathbf{signal}(l_0, l_1),\ \texttt{prev} = \cdot,\ \texttt{next} = l_n,\ C_2 \\
l_3 & \mapsto & \mathbf{ctor}_0(),\ C_3 \\
l_4 & \mapsto & \mathbf{ctor}_1(5),\ C_4 \\
l_5 & \mapsto & \mathbf{ctor}_0(l_3, l_4),\ C_5 \\
\vdots & & \vdots \\
l_{n-1} & \mapsto & \mathbf{ctor}_6(l_{n-2}, l_{n-3}),\ C_{n-1} \\
l_n & \mapsto & \mathbf{signal}(\texttt{"Hello, World!"}, l_{n-1}),\ \texttt{prev} = l_5,\ \texttt{next} = l_m,\ C_n
\end{array}
$$

Figure 1: The Rizzo heap storing both constructor and signal values. The signal heap is linked together by the `prev` and `next` fields.

4 Automatic reference counting of Rizzo

The algorithms by Ullrich and Moura [9] are defined over the calculi λ_{pure} and λ_{RC}, where λ_{RC} extends λ_{pure} with reference counting instructions. Figure 2

shows the reference counting intermediate representation (RC IR) used in our compiler, where Rizzo-specific extensions are highlighted in green.

In this IR, **pap** $c\ \bar{p}$ denotes partial application of a global constant c, $\mathbf{ctor}_i\ \bar{p}$ constructs a value of some erased type with tag i, and $\mathbf{proj}_i\ x$ projects the i-th field from a constructor value x. Fully applied calls and partial applications may only be performed by global constants, and variable application is defined separately for only a single argument.

$$
\begin{array}{rcl}
w, x, y, z & \in & \mathit{Var} \\
c & \in & \mathit{Const} \\
l & \in & \mathit{Literal} \ ::= \ \mathbb{Z} \mid \text{true} \mid \text{false} \mid \text{never} \mid \text{unit} \mid \mathit{string} \\
p & \in & \mathit{Primitive} \ ::= \ x \mid l \\
C & \in & \mathit{Ctor} \ ::= \ \mathbf{ctor}_i\ \bar{p} \mid \mathbf{signal}(p, p) \\
e & \in & \mathit{Expr} \ ::= \ C \mid \mathbf{pap}\ c\ \bar{p} \mid x\ p \mid \mathbf{ctor}_i\ \bar{p} \mid \mathbf{proj}_i\ x \\
 & & \mid \mathbf{reset}\ x \mid \mathbf{reuse}\ x\ \mathbf{in}\ C \\
F & \in & \mathit{FnBody} \ ::= \ \mathbf{ret}\ p \mid \mathbf{let}\ x = e;\ F \\
 & & \mid \mathbf{case}\ x\ \mathbf{of}\ \overline{\langle \mathbb{Z}, F \rangle} \mid \mathbf{inc}\ x;\ F \mid \mathbf{dec}\ x;\ F \\
f & \in & \mathit{Fn} \ ::= \ \lambda\, \bar{y}.\, F \\
\delta & \in & \mathit{Program} \ = \ \mathit{Const} \to \mathit{Fn} \cup \mathit{FnBody}
\end{array}
$$

Figure 2: The reference counting IR by Ullrich and Moura [9] with modifications for Rizzo marked in green.

Observe that the IR is defined in A-normal form [6], where expressions are non-recursive and are only allowed on the right-hand-side of let-bindings. Our representation keeps this shape but introduces *Primitives* so that variables and literals (i.e. integers, booleans, strings, and never) can appear directly in returns and as arguments in function/constructor application. This makes translation easier and avoids bindings such as `let x = 20; ret x` when `ret 20` is enough. We additionally add a specialised **signal** constructor that can be used in places where $\mathbf{ctor}_i$ is expected. This allows the runtime to know that the fresh memory cell should be stored in the signal heap (see Section 1.2.1). Finally, we add top-level values as mappings from constants to

FnBody where the *FnBody* is run at program startup to initialise the value. This way, we can reuse the automatic reference counting of Section 4.2.

To perform automatic reference counting on a Rizzo program, we first apply the transformations of Section 4.1 to get the program into a shape that satisfies the syntactic constraints of the IR and the invariants from Section 2.3. We then lower the transformed program to the reference-counting IR and apply the three algorithms from Section 4.2 in the order: R, which inserts **reset**/**reuse** pairs; $collect_{\mathbb{O}}$, which infers the borrow signatures of functions; and C, which inserts **inc**/**dec** instructions [9].

4.1 Transformations

To achieve the correct AST shape, we apply the transformation pipeline shown in Figure 3. Most of these transformations are standard compiler steps, but all of them bring the AST closer to satisfying the invariants of the algorithm by Ullrich et al. [9]. Each transformation performs a separate pass over the AST, allowing us to apply them in the correct order.

The compiler performs consecutive lambda elimination, lambda lifting, copy propagation, alpha-renaming, pattern lowering, dead let elimination, and A-normalization. Consecutive lambda elimination is not strictly required but reduces the number of lambdas to be lifted.

The lowering of patterns into let-bindings may not directly contribute to any invariant, however it is required since there is no pattern matching in the IR. The result of translating a match such as `match s with x :: xs -> e` is shown in Listing 1. When translating signal patterns, we delay the binding of the tail until its first use in e, since binding it too early may lead to **reset** instructions placed such that they prevent reuse. The match expression must be kept around because it is the only point where **reset**/**reuse** pairs can be inserted; we will discuss this further in Section 4.2.

While some of the invariants are clearly fulfilled by the transformation pipeline, such as the requirement that all lambdas be lifted or that trivial

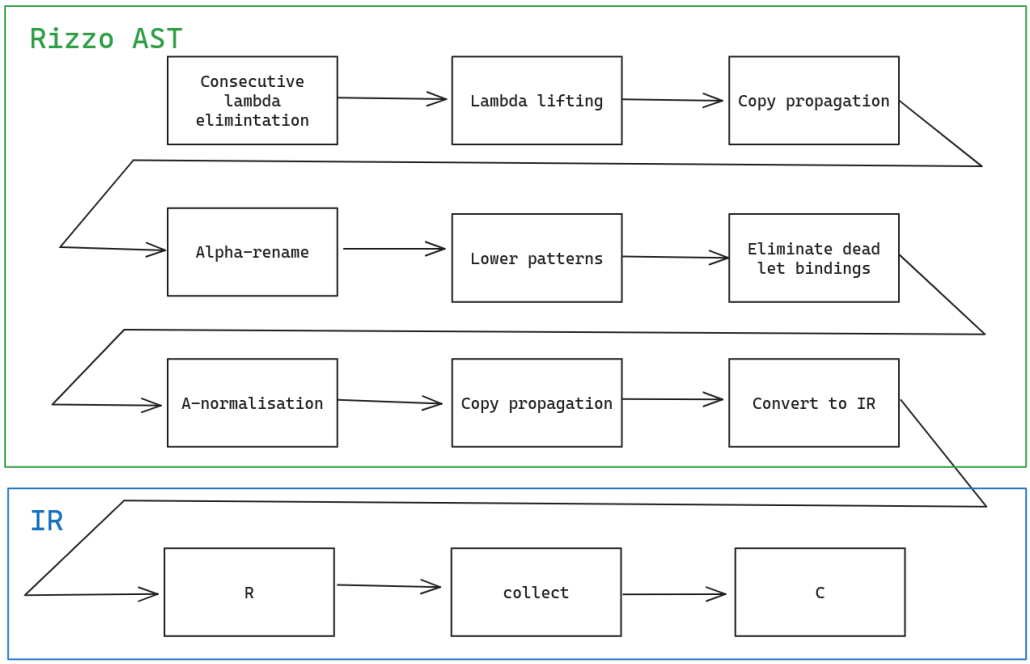

Figure 3: The compiler steps a Rizzo program goes through in order to be reference counted. Transformations inside the green (upper) box receive Rizzo programs as input, and in the blue (lower) box they operate on the RC IR.

bindings like `let x = y` have been eliminated, some of them are less obvious. In particular, the ones about application shape. The RC IR assumes unary variable applications and that applications on global constants are never over-applied. Therefore, when *converting to IR* variable applications are rewritten such that `x y z` becomes `(x y) z` and over-applied global constants are rewritten to a call followed by a series of unary variable applications.

Another non-obvious point is the *eta-expansion* of constructors. We do not introduce a dedicated transformation for invariant, since we assume a type-checker is run before the transformation pipeline and that it rejects programs containing partially applied constructors.

```
match s with
| _ ->
  let x = head s in
  let xs = tail s in e
```

Listing 1: Output of lowering a pattern match on a signal value

4.2 Counting Rizzo Beans

Once the Rizzo program has been transformed and converted into the IR form of Figure 2 it is time to insert RC instructions.

In this section, we describe the insertion of RC instructions and our adaptations of the algorithms by Ullrich et al. [9]. Note, these algorithms insert instructions but the counting of references and memory management is performed at runtime.

4.2.1 Reusing memory

The first step of this process is to insert **reset**/**reuse** pairs. The approach to inserting such pairs is based on the R algorithm by Ullrich et al. [9]. The algorithm will try to insert **reset**/**reuse** pairs for the scrutinee of each **match** expression. For each arm of the **match** expression, such pairs are inserted by first locating the point where the scrutinee is *dead*, i.e. not used for the remainder of the function body, and then by looking for a *suitable* constructor expression. If such a constructor is found then a **reset** is inserted at the point where the scrutinee is dead and the constructor expression is substituted with a **reuse**. Otherwise the code is left unchanged.

The shape of a constructor is *suitable* when the number of arguments $|\bar{p}|$ is equal to the number of fields of the scrutinee in the particular branch. Note the distinction between arguments and fields: here, an argument is the value given to the application of the constructor (creation), while a field is a place where a value is stored inside of a heap-allocated object.

Consider the example program shown in Listing 2. The *sync_fold* program applies a function f to a Sync, passing the arguments as optional values. The program is shown after inserting **reset**/**reuse** pairs. In the first branch s is a Left(x) value with a single field x, meaning s can be reused when wrapping x in a Some. In the third branch s is a Both(x, y) with two fields. The algorithm will try to reuse s in a constructor taking exactly two arguments but there is no such expression, so s isn't reused.

```
sync_fold f s : (Option 'a -> Option 'b -> 'c) -> Sync
   ('a,'b) -> 'c =
    match s with
    | #0 let x = proj_0 s in let s' = reset s in
      let x_just = reuse s' in Ctor1(x) in
      let nothing = Ctor0() in ret (f x_just) nothing
    | #1 (...)
    | #2 let x = proj_0 s in let y = proj_1 s in
      let x_just = Ctor1(x) in let y_just = Ctor1(y) in
      ret (f x_just) y_just
```

Listing 2: A fold on sync values. The program is shown using the IR but with a type annotation for readability.

The special-case constructor **signal** (p, p) is treated as a **ctor**$_i$ with the exception that when matched upon it makes five fields available and similarly it is only suitable when five fields are available from the scrutinee.

Listing 3 shows the map combinator from Section 1.2 after transformations and inserting **reset**/**reuse**. We see that the algorithm has added instructions for reusing the input signal.

```
fun map f s =
  match s with
  | #0 let x = proj_0 s in let xs = Ctor2(s) in
    let s' = reset s in let hd = f x in
    let map' = pap map(f) in
    let delayed_map = Ctor0(map') in
    let tl = Ctor6(delayed_map, xs) in
```

```
    let res = reuse s' in Signal(hd, tail) in
    ret res
```

Listing 3: The map combinator from Section 1.2 after transformations into IR form. The output has been simplified for readability, removing the thunk wrapper that would exist around `map'`.

However, this is not safe for *map*, since xs (the constructor tail s) needs the input signal s to live for its duration, so reusing the memory of s would be unsafe. We return to this in Section 4.2.3 when discussing the insertion of **inc**/**dec** instructions. For now, recall that a **reuse** x **in ctor**$_i$ $\bar{p}$ instruction may fail to reuse x when x is not unique at the time of **reset**. In the *map* function s must not be unique at the time of **reset** s because xs needs a valid reference to s.

4.2.2 Can I borrow that?

We distinguish between two kinds of references, $\mathbb{B}$orrowed and $\mathbb{O}$wned. Owned references participate in reference counting in the usual way, so the number of live references matches the counter stored in the object. Borrowed references, by contrast, are not counted; they are instead assumed to be kept alive by some surrounding owned reference.

The distinction exists to reduce the number of **inc**/**dec** instructions generated by the algorithm in Section 4.2.3. To insert those instructions, we first need a *borrow signature* for every function. A borrow signature tells us the type of reference that a function expects for each of its parameters. Signatures are captured in a mapping $\delta : Const \to \{\mathbb{O}, \mathbb{B}\}^*$ [9] which maps each function name to a sequence of reference kinds. Although these signatures could be declared by programmers, it is more convenient to infer them automatically and Ullrich et al. gives a heuristic for doing so.

The heuristic marks a variable x as owned if either x or one of its projections are used in a **reset**, appears in a partial or variable application, or is passed to a function that takes ownership. This heuristic fits with our

modifications and requires no further changes. Applying this heuristic recursively over the body of a function collects the set of all variables that must be owned. Once this set is computed, any parameter in the set is marked as owned in the function signature and the rest are marked as borrowed.

However, the borrow signature of a function depends on the signatures of the functions it calls. Recursive functions are also problematic, since their final signatures may depend on their own body. To handle this, we build a call graph that records, for each function f, the set of functions that call f. We then iterate over the functions, collecting owned variables and updating signatures until we reach a fixed point where no signature changes any further.

4.2.3 Learning to count

The algorithm for inserting **inc**/**dec** instructions is based on algorithm C by Ullrich et al. [9].

The insertion of **inc**/**dec** operations depends on the ownership of a variable and the ownership expected by its use site, which we call the context. Variables used in a *borrowed context* do not need to be incremented. By contrast, variables used in an *owned context* should be incremented before use, because the context consumes an RC token. Examples of owned contexts are constructor applications, variable applications, partial applications and calls to functions that expect ownership of a particular argument. Borrowed variables are never decremented while owned variables should be decremented after their last use. The algorithm goes one step further and will skip incrementing owned variables on their last use in owned contexts. This optimisation avoids unnecessary **inc**/**dec** pairs and leaves more opportunities for **reset**/**reuse** to succeed.

Adapting algorithm C to our IR, from Figure 2, requires handling the **signal** constructor, primitives, and global variables. The special case constructor **signal** is treated as any other **ctor** value, where all arguments are

considered to be used in an owned context. Primitives are either variables, which are handled according to the ownership rules above, or literals, which are never reference counted. Global variables are treated as borrowed for the purpose of **inc**/**dec** insertion, since they are assumed to be kept alive for the duration of the program. This choice keeps the implementation simple and safe but means global variables participate in reference counting when used in owned contexts.

Returning to the map example from earlier. Listing 4 shows the final compiled version of the map combinator. It also shows the result of borrow signature inference where both f and x are annotated as owned. We show the reference counts of the variables x, s, and f assuming they are all unique when map is called. Notice that because xs is a tail s value, it needs s to live for its duration, meaning s must be incremented before the implicit decrement performed by **reset** s. This increment sets the reference count of s to two, so s can never be unique at the time of **reset** s.

```
f: Owned, s: Owned
fun map f s =
  match s with
  | #0 let x = proj_0 s in      count(x) = 1
    inc x;                     count(x) = 2
    inc s;                     count(s) = 2
    let xs = Ctor2(s) in
    let s' = reset s in        count(s) = 1, s' = NULL
    inc f;                     count(f) = 2
    let hd = f x in
    let map' = pap map(f) in
    let delayed_map = Ctor0(map') in
    let tl = Ctor6(delayed_map, xs) in
    let res = reuse s' in Signal(hd, tail) in
    ret res
```

Listing 4: The map combinator with **inc**/**dec** instructions. The inferred ownership annotations for the arguments of map are shown on the first line. The output has been simplified for readability.

It may seem unfortunate that map can never reuse any memory but this is in fact necessary since the input signal is needed later when advancing the map-signal. To make this concrete, consider the program in Listing 5. The program uses the *map*-combinator to append exclamation marks to user input before echoing it to the console.

```
let c = "init" :: mk_sig (wait console) in
let f = fun x -> x + "!" in
let m = map f c in
let x = console_out_signal m in start_event_loop ()
```

Listing 5: A program that uses map to append exclamation marks to input read from the console.

After running the program through initialisation, the signal heap contains two locations: L_1 for the console signal c and L_2 for the mapped signal m:

	RC	Head	Tail
L_1	1	`"init"`	laterapp (delay (fun x -> x :: mk_sig (wait $console$))) (wait $console$)
L_2	1	`"init!"`	laterapp (delay ($map\ f$)) (tail L_1)

For this and the following visualisations we have unfolded the definitions of mk_sig, map and the $\triangleright$ operator as they were given in Section 1.2.

At some point the *console* channel produces a value v. This triggers a full heap update and so the heap cursor is set to point at L_1 and its tail is advanced. The laterapp is advanced by first advancing the wait *console* and getting the value v, then applying the delayed function to v. The result is a new signal `"Hello"` :: mk_sig (wait *console*) stored in memory at location L_3. This signal is then inserted into the signal heap just before L_1, which is currently the heap cursor. The heap looks like:

	RC	Head	Tail
L_3	1	v	laterapp (delay (`fun` x `->` $x :: mk\_sig$ (wait $console$))) (wait $console$)
L_1	1	`"init"`	laterapp (delay (`fun` x `->` $x :: mk\_sig$ (wait $console$))) (wait $console$)
L_2	1	`"init!"`	laterapp (delay ($map\ f$)) (tail L_1)

The data of L_3 is written back into L_1 and the reference count of L_3 is decremented by one, discarding it. Since the update of L_1 is finished, the heap cursor is moved forward so it points at L_2. The same process is applied to L_2, but this time the right operand of the laterapp is a tail value which, when advanced, returns the location L_1. This location is passed to the partial application of map, creating a new signal $f\ v :: map\ f \triangleright$ tail L_1 at some fresh location L_4. The heap now looks like:

	RC	Head	Tail
L_1	2	v	laterapp (delay (`fun` x `->` $x :: mk\_sig$ (wait $console$))) (wait $console$)
L_4	1	v + `"!"`	laterapp (delay ($map\ f$)) (tail L_1)
L_2	1	`"init!"`	laterapp (delay ($map\ f$)) (tail L_1)

Again, the data of L_4 is written into L_2 and L_4 is decremented, discarding it.

The example shows that if the *map* combinator were to reuse its input (in this case variable c and by extension L_1), then the only reference to the console would be overwritten, and it would no longer be possible to advance L_2. Therefore, when the algorithm decides to deny reuse in signal combinators such as *map*, it is making a correct and safe choice. However, this choice

also results in unnecessary reference counting overhead. In particular, for the IR of *map* in Listing 4, we see that the algorithm has denied the **reset** s by inserting an **inc** s instruction. Ideally, neither of these instructions should have been inserted. Assuming *map* takes an owned reference to the input s, it is sufficient to consume this owned reference by simply passing it to the $\mathbf{ctor}_2(s)$ expression. We return to this limitation in Section 6.1.

5 Evaluation

We originally set out to solve the memory management problem in Rizzo. With a pipeline for transforming Rizzo programs and inserting reference counting, we investigate the resulting programs and their memory performance. To do so, we have implemented the semantics described in Section 3.3 in a `C` runtime and built a code generator that emits the fully transformed, reference counted IR as `C`-code.

For this evaluation, our focus will be on the absence of space leaks and the proper timing of signal clean-up so that unnecessary work is avoided. In Section 5.1 we execute a small Rizzo program and inspect the signal heap at regular intervals and in greater detail. Then in Section 5.2 we zoom out and run a complex program for an extended period to observe the general memory usage. In Section 5.3 we show reuse in action by showing that we can reuse memory for values inside signals.

5.1 Signal heap behaviour

In order to evaluate the behaviour of the signal heap, we run a small Rizzo program. To show that our signal heap can grow and shrink in a predictable way, we make the program change which signals are actively used. For this, we have made a program with two modes. It has an English mode and a Danish mode, and echoes back any input given to it in the respective language. The full source can be found in Appendix A.

```
fun entry unused =
    let console_sig = mk_sig_of_channel console in
    let regular_inputs = filter_l is_regular_input
       console_sig in
    let language_sig = filter_map_l parse_language
       console_sig in
    let default =
        "Type en for English (Default).\nSkriv da for
           Dansk."
        :: language_signal regular_inputs EN in
    let mode_funcs = map_l (language_switch
       regular_inputs) language_sig in
    let switched = switch_r default mode_funcs in
    let _out = console_out_signal switched in
    start_event_loop ()
```

Listing 6: The entry point of the Rizzo program used to evaluate the heap behaviour.

When compiling the program we enable the `--debug-info` flag, which enables the logging of debug information. Specifically, the flag adds unique identifiers to signals and enables printing of the signal heap after each time step. We compiled the program with `rizzoc examples/eval_small.rizz --debug-info` and then ran the generated executable `./output(.exe)`. Running the program gives the output in Figure 4. We have coloured the user input in blue and the output from the program in green. The lines starting with 'Sig index' are the debug prints of the heap. The prints show the next available signal identifier, the size of the heap, and the unique identifiers of the signals in the signal heap. We also indicate if a signal was updated in the current time step by surrounding its identifier with brackets like $[i]$.

From the output of the program, we see that on initialisation, the heap contains four signals:

- Signal 0 is an internal signal created and watched inside the `filter_l`

```
Type en for English (Default).
Skriv da for Dansk.
After init, Sig index 4, (size: 4) | [0] [1] [2] [3] (|)
input 1
You typed: input 1.
Sig index 11, (size: 7) | [4] [0] [6] [1] [8] [2] [3] (|)
input 2
You typed: input 2.
Sig index 18, (size: 7) | [4] [0] [6] [1] [8] [2] [3] (|)
da
Skiftet til Dansk.
Sig index 26, (size: 8) | [4] [0] [6] [1] [22] [23] [24] [3] (|)
input 4
Du skrev: input 4.
Sig index 33, (size: 9) | [4] [0] [6] [1] 22 23 [30] [24] [3] (|)
input 5
Du skrev: input 5.
Sig index 40, (size: 9) | [4] [0] [6] [1] 22 23 [30] [24] [3] (|)
en
Switched to English.
Sig index 48, (size: 8) | [4] [0] [6] [1] [22] [23] [46] [3] (|)
input 7
You typed: input 7.
Sig index 55, (size: 9) | [4] [0] [6] [1] 22 23 [52] [46] [3] (|)
input 8
You typed: input 8.
Sig index 62, (size: 9) | [4] [0] [6] [1] 22 23 [52] [46] [3] (|)
```

Figure 4: The output of the language swapping program with debug info enabled. The blue lines are user input, the green lines are program output and the black lines are the heap prints. The brackets indicate which signals have been updated in that time step.

function.

- Signal 1 is an internal signal created and watched inside the `filter_map_l` function.
- Signal 2 is the initial English echo signal stored in the `default` variable.
- Signal 3 is an internal signal representing the output from the `switch_r` function.

The program starts in English mode, and the first two lines written are the initial greeting and the instruction for switching to Danish. This output is caused by the signal having been given an initial head value, which causes the output function to trigger on initialisation.

We then send two inputs, '*input 1*' and '*input 2*', causing the signal heap to grow as both the filters depend on the `Later` value of `console_sig`. Before the first input, the `console_sig` is not directly a signal yet, but a value of type `Later (Signal String)`. This means that if multiple filters depend on that `Later` value, each dependency on the `Later` can create a new signal from that `Later`. The new signals 4 and 6 represent these signals created from the `Later` of the console input. The double `Later` evaluation issue is described further in Section 6. The signal 8 is the English echo signal: the signal that maps the input to the form 'You typed: *input*'. After this first growth from '*input 1*', the next input '*input 2*' satisfies the same filter conditions, so the heap size remains unchanged.

When we input '*da*', the filter that watches for the language codes is satisfied, and outputs a new signal with index 22. This signal is then mapped over by the `language_switch` function, which creates a new signal 23 whose head is the function used to construct the new branch. Applying the new head of 23 to `switch_r` then creates the new Danish branch signal 24 replacing the old English branch signal 2. The signal 8 is not used in switching the language, so it gets removed until the next input that is not '*en*' or '*da*'.

Inputting '*input 4*' and '*input 5*' now only makes the heap grow by one signal. The new signal is the Danish echo signal 30, which is the replacement of the old English echo signal 8. Here we can see the signals 22 and 23 are still in the heap, as they are still needed for the language switching, but because the filter for switching languages is not satisfied, they are not updated in these time steps. When we switch back to English with '*en*', the echo filter is satisfied again, causing the signals 22 and 23 to be updated. This causes the Danish branch signal 24 to be replaced by the English branch signal 46. The Danish echo signal 30 is now not referenced by the deallocated signal 24 and is removed, until signal 52 is created by an input that satisfies the echo filter again.

As we can see from the heap prints, the heap grows and shrinks in a predictable way. The signals that are not needed anymore get removed from the heap, and the signals that are needed get created when they are needed. This shows that our signal heap is working as intended, and that it can handle dynamic changes in the topology of the signal graph.

5.2 No bit is forgotten

For this part of the evaluation, our focus is on space leaks. For this purpose, we use Valgrind [1], which is a framework for building dynamic analysis tools. Valgrind comes with a tool, *memcheck*, for detecting memory errors such as reading uninitialised values, use-after-free, and most importantly memory leak detection. Valgrind cannot prove we have no space leaks, but it can give us confidence that we do not have any obvious memory we are losing track of.

To get information on how many inputs the program has processed, we instrument the runtime to save the number of steps taken and the average time taken per step in microseconds. This information is logged at program exit when the program is compiled with the `--info-heap` flag.

The program we will run is an interactive console application ported from

Bahr et al. [3]. Our code can be found in Appendix B.1 and the full output can be found in Appendix B.2. The program maintains a counter that is incremented every time a clock ticks and can be printed to the screen when a user writes '*show*'. The counter can be manipulated by writing either '*negate*', to negate the current value, or a number n to increase the current value by n. To implement the desired behaviour, the program uses *clock* and *console*-input signals, and manipulates them with the head, tail, *map*, *filter*, *switch*, *trigger*, and *interleave* signal combinators. Through these combinators, the example makes use of Rizzo's core features: signals, sync construction, watch, wait, tail, and later applications (laterapp). The only Rizzo constructors not used by the example are never and ostar.

To mimic the program running for an extended period, we adjust the timer so the counter is incremented every 1 millisecond. We then have the program run for 200 seconds with input occasionally being given to the console, which results in about 200,000 steps taken. We run the program with Valgrind's `--leak-check=full` tool to check for memory leaks and other memory errors. For reproducibility, we made it into a script under `scripts/valgrind_evaluation.sh`.

```
Steps taken: 198480, Average step time: 154 us, Signals left in heap: 21
==3108==
==3108== HEAP SUMMARY:
==3108==     in use at exit: 5,811 bytes in 154 blocks
==3108==   total heap usage: 4,169,475 allocs, 4,169,321 frees, 174,719,546 bytes
==3108==
==3108== LEAK SUMMARY:
==3108==    definitely lost: 0 bytes in 0 blocks
==3108==    indirectly lost: 0 bytes in 0 blocks
==3108==      possibly lost: 0 bytes in 0 blocks
==3108==    still reachable: 5,811 bytes in 154 blocks
==3108==         suppressed: 0 bytes in 0 blocks
```

The output from Valgrind's leak summary has four categories: definitely lost, indirectly lost, possibly lost, and still reachable. Definitely lost means memory has been allocated but no remaining pointer refers to it, so it is a confirmed memory leak. Indirectly lost means memory is only reachable from another leaked block, so it is leaked as a consequence of the original loss. Possibly lost means Valgrind cannot determine with certainty whether the memory is still accessible, so it may indicate a leak. Still reachable means the memory was not freed before program termination, but valid pointers to it still exist, so it is not considered a true leak. Since the leak summary reports 0 bytes for definitely lost, indirectly lost, and possibly lost, this output shows that the program does not have a memory leak according to Valgrind.

5.3 Reusing memory at runtime

As mentioned in Sections 2.2.1 and 4.2.3, there are not many cases in which signals can be reused. However, reuse of other constructor values is more common. To show reuse in action, we use the Rizzo program in Figure 5, which strings that parse as integers from the console and stores them in a list. The program then outputs the list of integers to the console.

The `list_append` function recursively deconstructs the first list by separating the head cell from the tail. After this separation, the original cell is no longer referenced by the remaining input list, which makes it eligible for reuse. The generated code in Figure 6 illustrates the relevant part of `list_append`: `lst1` is reset before the recursive call on the tail, allowing its storage to be reused when constructing the resulting list cell from the original head and the recursive result.

We then run the program with the `--debug-malloc` flag enabled and provide it the input "4", "5", and "6" to the console. The program then shows debug information for each allocation, deallocation, and reuse of memory. To make it more convenient to analyse the debug output, we can filter the output for lines that contain "reuse" and extract the addresses of the reused memory.

```
fun entry _ =
    let x = ["1", "2", "3"] in
    let y = mk_sig_of_channel console in
    let state = scan_l (fun acc n ->
        list_reverse (match (parse_int n) with
        | None -> acc
        | Some(v) -> list_append acc [n])
    ) x y in
    let out = console_out_signal_l state in
    start_event_loop ()
```

Figure 5: A Rizzo program that reads strings that parse as integers from the console and stores them in a list. The program then outputs the list of integers to the console.

```
fun list_append(lst1, lst2) =
    match lst1 with
    | #0
        dec lst1;
        inc lst2;
        ret lst2
    | #1
        let ctor_field67 = proj_0 lst1 in
        inc ctor_field67;
        let ctor_field68 = proj_1 lst1 in
        inc ctor_field68;
        let var105 = reset lst1 in
        let var76 = list_append(ctor_field68, lst2) in
        let var96 = reuse var105 in Ctor1(ctor_field67,
           var76) in
        ret var96
```

Figure 6: The generated code for the `list_append` function, showing how the original head cell of the first list is reset before the recursive call and then reused when constructing the resulting list cell.

```
rizzoc .\examples\reuse.rizz --debug-malloc
"4‘n5‘n6" | .\output.exe |
    Select-String reuse |
    ForEach-Object {
    if ($_.Line -match '^reuse:\s+([0-9A-F]+)\b')
    { $matches[1] }
    } |
    Group-Object |
    Sort-Object Count -Descending |
    Tee-Object -Variable groups |
    Format-Table Count, Name -AutoSize

"TOTAL={0}" -f (($groups | Measure-Object Count -Sum).Sum)
```

Figure 7: The output of the program filtered for reuse events, showing the addresses of the reused memory and how many times each address was reused.

We can then group these addresses and count how many times each address was reused. The script to do this is shown in Figure 7.

Running the program, we see that we have a total of 12 reuse events. We see that three different addresses are reused three times, indicating the initial allocation of three list cells that are then reused for each new append to the list. The address reused twice would be the first input, and the address reused once would be the second input. This shows that a general operation like list append can benefit from reuse, and that the same memory can be reused multiple times. The output of running the script of Figure 7 is shown in Figure 8.

```
Count Name
----- ----
    3 000002064D9FE900
    3 000002064D9FE960
    3 000002064D9FE990
    2 000002064D9FEB10
    1 000002064D9FE9F0
TOTAL=12
```

Figure 8: The debug output showing the addresses of the reused memory and how many times each address was reused.

6 Limitations and Future Work

6.1 Reference counting overhead in signal combinators

As we described in Section 4.2.3, the reactive semantics limit reuse in signal combinators to non-recursive combinators such as *stop*. In recursive combinators, the **reset** instructions are still inserted even though they will never succeed, and these then force the insertion of additional **inc** instructions. The reason is that the reference counting algorithms always insert **reset**/**reuse** pairs whenever they encounter a suitable constructor expression, even if the memory will be needed again later. A natural way to fix this would be to avoid inserting **reset**/**reuse** in branches where the tail bound by the match is used in a recursive call. Preventing inserts at the level of branches would still allow reuse in combinators like *stop* that have non-recursive and recursive branches.

We are not the first to observe that the eager **reset**/**reuse** insertion can cause problems. Lorenzen et al. [8] present a program similar to our *map* combinator, but their concern is instead that the resulting code can hold onto memory for too long. Their solution is to reverse the order in which **reset**/**reuse** instructions and **inc**/**dec** instructions are inserted, so that **reset** instructions are only inserted when they can replace a **dec**. This

would also solve our issue of redundant and always failing **reset** instructions.

6.2 Specialised implementations of combinators

While all signal combinators can be implemented in RizzoLang, some of these may allocate more memory than necessary. As an example, consider the definition of *mk_sig* from Section 1.2. If we unfold the ▷ operator, its definition becomes:

$$
\begin{aligned}
&mk\_sig : \texttt{Later}\ a \to \texttt{Later}\ (\texttt{Signal}\ a) \\
&mk\_sig\ d = (\lambda x.x :: mk\_sig\ d) \triangleright d \\
&\Updownarrow \\
&mk\_sig\ d = \mathsf{laterapp}\ (\mathsf{delay}\ (\lambda a.\, a :: mk\_sig\ d))\ d
\end{aligned}
$$

From this unfolded definition, we see that every (recursive) call allocates a fresh **ctor** value for the laterapp, a **ctor** value for delay, and a closure to capture d for the lambda expression. Since the shape of the later values generated by *mk_sig* never changes, advancing such a later just reconstructs a new value that is structurally identical to the old one. This is wasted work. We suggest making *mk_sig* a primitive constructor for the `Later` type and adding a specialised implementation to handle such values in the advance semantics. This would allow us to avoid these repeated allocations. A similar optimisation could be made for the ▷ operator, which is used in most combinators.

6.3 The later applications forgotten to time

Later applications, values constructed using laterapp, can call arbitrary functions and are therefore not restricted by the operational semantics to return the same value each time. As a result, if n signals depend on the same later application, that application will be evaluated n times in a single time step,

once for each dependency. This is inefficient both in terms of time and memory, as later applications may be expensive to compute, and they may also allocate signals on the signal heap. In the latter case, evaluating the same later application n times will create n identical copies of any signal allocated by the function. This duplication will cause excessive computation for as long as the program depends on those signals.

In Section 5.1, the signals 4 and 6 arise from exactly this situation: the same later application is evaluated twice, once for each dependency. Ideally, the signal 4 would have been cached and returned for the second dependency, which would have prevented the creation of signal 6 and the associated memory allocation.

A current workaround is to turn the affected later computation into a signal and then apply the tail constructor to it:

$$\mathsf{tail}\ (\texttt{""} :: \mathit{mk\_sig}\ (\mathsf{wait}\ \mathit{console}))$$

This works because, according to the reactive semantics of tail, advancing a value tail s returns exactly s. This effectively means that the result of the later application is cached in the signal that we create as part of the workaround. The downside is that we abandon the lazy signal creation. If the signal never produces a value, the allocation is unnecessary. However, when the signal is used multiple times, we avoid repeated evaluation of the same later application and the associated memory allocations, which can significantly improve performance and reduce memory usage.

A proper fix would be to make laterapp values behave like lazy values, so that the result is cached after the first evaluation. However, the result may only be cached until a future time step when the laterapp produces a new value. One way to implement this is to extend the representation of laterapp values with two additional fields: one field for the cached value and one for the time step for which the cache is valid. Evaluation would then first check whether the cached value is valid. If so, then the cached value is returned.

Otherwise, a new value is computed, stored in the cache, and then returned.

6.4 To avoid allocating intermediate signals

Bahr [2] states that when an efficient implementation of Rizzo is updating a signal s, it can avoid the allocation of an intermediate signal by *simply* writing the data directly into s. This, however, does not seem to be so simple. When a laterapp value is advanced, the runtime cannot easily identify which of the allocated signals is going to be used to update s. Furthermore, even if the runtime could make this distinction, the signal s is not in scope when the allocation happens and is therefore not reusable.

This means that the optimisation would have to be performed at compile time using either a new static analysis step or, better yet, by reusing the **reset**/**reuse** insertion logic. To reuse this logic, we would need to inline the advance semantics into the tails of signals. However, simply inlining the advance semantics is not enough, since functions inside laterapp values are black boxes that prevent the insertion of **reset**/**reuse**, so those must also be inlined. Inlining these functions may mean unfolding recursive definitions, which could be problematic unless these are guarded by a delay.

To achieve the inlining, we would need some representation of the advance semantics that can be inserted during compilation. One such representation would be *code* written in either Rizzo itself or perhaps in the RC IR. Either way, we would need to add some language features. In particular, we would need mutable references or specialised private functions for manipulating the signals, and polymorphic recursion for type checking. A downside of this approach would be the increased program size caused by duplicating the advance semantics in the tail of every signal. This increase in size would cause longer compile times and potentially worsen runtime performance. However, the inlining would allow us to reuse the **reset**/**reuse** logic to avoid the allocation of intermediate signals.

6.5 The gain of indirections

There are some signal combinators that can leave the signal heap with duplicates. This is inefficient because the update semantics must then be applied to each duplicate, wasting both time and space. Consider the program of Listing 7. It defines a global signal c that produces the stream of values received from the console. Inside the entry point, it defines a signal s which, by using *switch*, initially behaves like the constant signal and then switches to match c once input is received on the console.

```
let c = "" :: mk_sig (wait console)
fun entry () =
    let s = switch (const "switch!") (tail c) in
    let _ = console_out_signal s in
    start_event_loop ()
```

Listing 7: Rizzo program using the switch combinator.

Compiling and running the program with the `--debug-info` flag produces the output shown in Figure 9. We follow the same colour scheme as earlier, where blue denotes user input and green denotes program output. After initialisation, there are three signals in the heap: 0 is the signal c; 1 is the constant signal of the string `switch!`; and 2 is the signal s. The first step is triggered by the user input `hi`. This produces a new value on signal c, which then also causes the tail c to produce a value, making s to switch its behaviour to that of c. The head of s, which is now identical to the head of c, is output to console. The constant signal, no longer needed by s, is discarded, and so the heap shrinks.

The last two lines of output in Figure 9, those beginning with `signal`, show the data of the live signals c and s. The signals store pointers to dynamic memory, so the values themselves are not interesting, but what matters is that they are identical across the two signals. This creates inefficiency, because future time steps must apply the update semantics to both signals even though they refer to the same underlying data.

```
switch!
After init | [0] [1] [2] (|)
hi
hi
step 0001 | [0] [2] (|)
signal 0 { head = 0x61b5e6e6cf00, tail = 0x61b5e6e6c440 }
signal 2 { head = 0x61b5e6e6cf00, tail = 0x61b5e6e6c440 }
```

Figure 9: The output of running the program of Listing 7 with the `--debug-info` flag enabled. Blue denotes user input and green denotes output.

A possible solution to this problem would be to change the representation of signals so that a signal is either a *concrete* signal, containing its head, tail, previous, and next fields, or an *indirect* signal, containing only a pointer to another signal. The main benefit of an indirect signal is that it does not need to be updated directly, since updates propagate through the pointer. Creating indirections would become part of the update semantics. Before any data is written, the runtime should check whether the intermediate signal is unique. If the intermediate signal is unique, the update should proceed as normal. Otherwise, an indirection should be created instead. In the example of Figure 9, this would allow the runtime to replace s (signal 2) with an indirection after it switches. However, this optimisation would require a change in semantics, since head, tail, and watch would then need to traverse indirections until reaching a concrete signal. The cost of such traversals can be reduced by performing *path-compression*, i.e. updating indirections during reads so that they point directly to the concrete signal once it has been found.

This solution would also partially solve the same issue discussed in Section 6.4, since indirections do not have tails to advance, thereby eliminating some of the intermediate signals allocated during a heap update.

6.6 Improving the structure of the signal heap

The current signal heap representation is quite simple but it is not very efficient. As a linked list the advance semantics must traverse the entire heap to find the signals that need to be updated. This causes the runtime complexity of the advance semantics to be linear in the number of signals in the heap. This means the more signals a program has, the more time each step takes to execute. We believe that for the language to be practical for big server applications, games, or user interfaces, the runtime complexity of the advance semantics should be sub-linear in the number of signals in the heap. Further work would therefore be needed to find a more efficient representation of the signal heap that allows the update semantics to skip or ignore signals that are not used in the current step.

7 Conclusion

We set out to investigate whether reference counting can be used to manage the intermediate signals created by the reactive semantics of Rizzo. More precisely, we wanted to ensure that signals which are no longer referenced are deallocated before future time steps, so that dead signals are not repeatedly updated by the step semantics.

To do so, we implemented the Rizzo language by Bahr [2], extended it with a reference-counting memory management system based on the work of Ullrich et al. [9], and built a transpiler from RizzoLang to C. The C backend implements the step, update, and advance semantics of Rizzo, which lets us evaluate the memory behaviour of compiled Rizzo programs directly.

To evaluate the memory management and signal heap, we created a set of example programs that we could compile and run. We used debug prints to show snapshots of the heap at runtime and used Valgrind to track if any memory was leaked. The results of these evaluations showed that the reference counting instructions properly deallocate unused signals, including intermediate signals, without signs of memory leaks or space leaks. Since these signals are removed from the signal heap, they are not updated in later time steps. This shows that we were able to solve the main issue of this thesis.

In addition to general reference counting, the approach of Ullrich et al. [9] made it possible to reuse objects. Our evaluation showed that this is effective for general objects like lists, and that values stored in the heads of signals can also be reused. However, we identified limitations in the reuse of the signals themselves.

One limitation comes from the semantics of Rizzo itself. Signals passed to recursive signal combinators cannot be reused in the same way as lists or other constructors. For example, the signal combinator $map\ f\ s$ results in a new signal s', where future values of s' depend on future values of s. This makes it impossible to reuse the signal s for s', as we would lose the

future values of s. A second limitation comes from our implementation of the update semantics and how it handles intermediate signals. These signals are created as part of the update semantics, but are often only created to store the updated values, before copying them back to the signal that was updated. This is often wasteful and should be avoidable, but the compiler does not have enough knowledge about the usage of the object in the advance semantics. To address this issue, we proposed a possible solution that gives the compiler more information, which should allow it to better reuse intermediate signals.

We also found that some combinators can leave duplicate signals in the heap. For example, after a combinator switches one signal for another signal, both signals may remain alive while containing the same underlying head and tail data. This is inefficient because future steps may update both signals even though they represent the same behaviour. As future work, we discussed using indirect signals, where a signal can point to another signal instead of duplicating its data.

Finally, we identified that the reactive semantics of Rizzo place strong constraints on the structure of the signal heap. In our implementation this led to a linked-list representation, which gives simple insertion, removal, and maintenance of the now/earlier split, but also means that each step must inspect the signal heap in its entirety. This motivates future work on heap structures that allow the runtime to skip signals that cannot be affected by the current step.

Altogether, we believe that we have successfully met the goals of the thesis. We implemented a working C backend for Rizzo, showed that reference counting can deallocate intermediate dead signals before they are reevaluated, and identified concrete limitations and future optimisations for both the compiler and the runtime representation of signals.

References

[1] *About Valgrind*. URL: https://valgrind.org/info.

[2] Patrick Bahr. *Simple Modal Types for Functional Reactive Programming*. 2025. DOI: 10.48550/arXiv.2512.09412. arXiv: 2512.09412 [cs.PL]. URL: https://arxiv.org/abs/2512.09412.

[3] 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. Vol. 14512. Series Title: Lecture Notes in Computer Science. Cham: Springer Nature Switzerland, 2023, pp. 18–36. ISBN: 978-3-031-52038-9. DOI: 10.1007/978-3-031-52038-9_2. URL: https://link.springer.com/10.1007/978-3-031-52038-9_2 (visited on 11/01/2025).

[4] Patrick Bahr and Rasmus Ejlers Møgelberg. "Asynchronous Modal FRP". In: *Proceedings of the ACM on Programming Languages* 7 (ICFP Aug. 30, 2023), pp. 476–510. ISSN: 2475-1421. DOI: 10.1145/3607847. URL: https://dl.acm.org/doi/10.1145/3607847 (visited on 09/12/2025).

[5] George E. Collins. "A method for overlapping and erasure of lists". In: *Commun. ACM* 3.12 (Dec. 1, 1960), pp. 655–657. ISSN: 0001-0782. DOI: 10.1145/367487.367501. URL: https://dl.acm.org/doi/10.1145/367487.367501 (visited on 03/09/2026).

[6] Cormac Flanagan et al. "The essence of compiling with continuations". In: *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*. PLDI '93. Albuquerque, New Mexico, USA: Association for Computing Machinery, 1993, pp. 237–247. ISBN: 0897915984. DOI: 10.1145/155090.155113. URL: https://doi.org/10.1145/155090.155113.

[7] Neelakantan R. Krishnaswami. "Higher-order functional reactive programming without spacetime leaks". In: *SIGPLAN Not.* 48.9 (Sept. 2013), pp. 221–232. ISSN: 0362-1340. DOI: 10.1145/2544174.2500588. URL: https://doi.org/10.1145/2544174.2500588.

[8] Anton Lorenzen and Daan Leijen. "Reference counting with frame limited reuse". In: *Implementation and Benchmarks for "Reference Counting with Frame Limited Reuse"* 6 (ICFP Aug. 31, 2022), 103:357–103:380.

DOI: 10.1145/3547634. URL: https://dl.acm.org/doi/10.1145/3547634 (visited on 03/05/2026).

[9] Sebastian Ullrich and Leonardo de Moura. "Counting immutable beans: reference counting optimized for purely functional programming". In: *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages*. IFL '19. New York, NY, USA: Association for Computing Machinery, July 15, 2021, pp. 1–12. ISBN: 978-1-4503-7562-7. DOI: 10.1145/3412932.3412935. URL: https://dl.acm.org/doi/10.1145/3412932.3412935 (visited on 01/27/2026).

A Heap Behaviour Evaluation

```
type Language = EN | DA

fun parse_language command =
    match command with
    | "en" -> Some (EN)
    | "da" -> Some (DA)
    | _ -> None

fun is_regular_input command =
    if command == "quit"
    then false
    else
        match parse_language command with
        | Some (_) -> false
        | None -> true

fun switch_message language =
    match language with
    | EN -> "Switched to English."
    | DA -> "Skiftet til Dansk."

fun echo_message language command =
    match language with
    | EN -> "You typed: " + command
    | DA -> "Du skrev: " + command

fun language_signal regular_inputs language =
    map_l (echo_message language) regular_inputs

fun language_switch regular_inputs language =
    fun _void ->
        switch_message language :: map_l (echo_message language) regular_inputs

// Console input is the only driver,
// but ‘switch_r‘ still swaps the active output signal.
fun entry unused =
```

```
    let console_sig = mk_sig_of_channel console in
    // 0 internal watched
    let regular_inputs = filter_l is_regular_input console_sig in
    // 1 internal watched
    let language_sig = filter_map_l parse_language console_sig in
    let default = // 2
        "Type en for English (Default).\nSkriv da for Dansk."
        :: language_signal regular_inputs EN in
    // 23
    let mode_funcs = map_l (language_switch regular_inputs) language_sig in
    // 3
    let switched = switch_r default mode_funcs in
    let _out = console_out_signal switched in
    start_event_loop ()
```

B Valgrind evaluation

B.1 Program

```
let console_sig  = mk_sig (wait console)
let quit_sig = filterL (fun x -> x == "quit") console_sig
let show_sig = filterL (fun x -> x == "show") console_sig
let neg_sig  = filterL (fun x -> x == "negate") console_sig
let num_sig  = filter_mapL parse_int console_sig
let clock_sig = clock 1
let nats = fun init -> scan (fun x y -> x + 1) init clock_sig
let interleaved =
  interleave
    (fun f g x -> f (g x))
    (map (fun x n -> 0 - n) ("" :: neg_sig))
    (map (fun m n -> m + n : Int -> Int -> Int) (0 :: num_sig))

fun nats' init =
    switchS (nats init) ((fun s n -> nats' ((head s) n)) |> tail interleaved)

fun entry x =
    let n = 0 in
```

```
    let nats_prim = nats' n in
    let show_nat = trigger_l (fun x n -> n) show_sig (nats_prim) in
    let _x = console_out_signal ("" :: map_l string_of_int show_nat) in

    let _x = quit_at quit_sig in
    start_event_loop ()
```

B.2 Program output

```
$ ./scripts/paper-example-valgrind.sh
Compilation successful! Executable generated at: output
==3108== Memcheck, a memory error detector
==3108== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==3108== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==3108== Command: ./output
==3108==

59
8404
-8403
12359079
12369087
-12369084
-12359073
Steps taken: 198480, Average step time: 154 us, Signals left in heap: 21
==3108==
==3108== HEAP SUMMARY:
==3108==     in use at exit: 5,811 bytes in 154 blocks
==3108==   total heap usage: 4,169,475 allocs, 4,169,321 frees, 174,719,546 bytes
==3108==
==3108== LEAK SUMMARY:
==3108==    definitely lost: 0 bytes in 0 blocks
==3108==    indirectly lost: 0 bytes in 0 blocks
==3108==      possibly lost: 0 bytes in 0 blocks
==3108==    still reachable: 5,811 bytes in 154 blocks
==3108==         suppressed: 0 bytes in 0 blocks
==3108== Reachable blocks (those to which a pointer was found) are not shown.
==3108== To see them, rerun with: --leak-check=full --show-leak-kinds=all
```

```
==3108==
==3108== For lists of detected and suppressed errors, rerun with: -s
==3108== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

B.3 Advance function

```
fun entry () = "see below"

type MyChan 'a = MChan

type MyDelayed 'a 'b =
    | MDelay(Unit -> 'b)
    // | MOstar(MyDelayed ('a -> 'b), MyDelayed ('a, 'b))

type MyLater 'a 'b =
    | Never
    | Wait (MyChan 'a)
    | Watch (Signal (Option 'a))
    | Tail (Signal 'a)
    | LaterApp( MyDelayed (('a->'b), 'b) , MyLater('a, 'b))
    | MSync(MyLater('a,'b), MyLater ('a, 'b))

fun advance_delay d : MyDelayed('a, 'b) -> 'b =
    match d with
    | MDelay(f) -> f ()
    // | MOstar(d1, d2) ->
    //     let x = advance_delay d2 in
    //     let f = advance_delay d1 in
    //     f x

fun or a b = if a then true else b

fun is_updated s = false

fun ticked l c v =
    match l with
    | Never -> false
    | Wait (c) -> c == c
```

```
    | Watch ((None) :: xs) -> false
    | Watch ((Some(_)) :: xs) -> false
    | Tail (s) -> is_updated s
    | LaterApp (_, x) -> ticked l c x
    | MSync (l1, l2) -> or (ticked l1 c v) (ticked l2 c v)

fun advance l c v =
    match l with
    | Wait (_) -> v
    | Watch ((Some(x)) :: xs) -> x
    | Tail (s) -> s
    | LaterApp (f, l) ->
        let x = advance l c v in
        advance_delay f x
    | MSync (l1, l2) ->
        if ticked l1 c v then Left(advance l1 c v)
        else if ticked l2 c v then Right(advance l2 c v)
        else Both (advance l1 c v, advance l2 c v)
```

B.4 Advance function lowered to IR

```
l: Owned, c: Borrowed, v: Borrowed
fun advance(l, c, v) =
  match l with
  | #1 dec l;
       inc v;
       ret v
  | #2 let ctor_field163 = proj_0 l in
       inc ctor_field163;
       dec l;
       match ctor_field163 with
       | #0 let sig_head170 = proj_0 ctor_field163 in
            inc sig_head170;
            dec ctor_field163;
            match sig_head170 with
            | #1 let ctor_field171 = proj_0 sig_head170 in
                 inc ctor_field171;
                 dec sig_head170;
```

```
                ret ctor_field171
            | default dec sig_head170;
                      let var303 = match_fail("Non-exhaustive pattern match") in
                      ret var303
        | default dec ctor_field163;
                  let var304 = match_fail("Non-exhaustive pattern match") in
                  ret var304
    | #3 let ctor_field164 = proj_0 l in
         inc ctor_field164;
         dec l;
         ret ctor_field164
    | #4 let ctor_field165 = proj_0 l in
         inc ctor_field165;
         let ctor_field166 = proj_1 l in
         inc ctor_field166;
         dec l;
         let x = advance(ctor_field166, c, v) in
         let var305 = advance_delay(ctor_field165) in
         let var306 = var305 x in
         ret var306
    | #5 let ctor_field167 = proj_0 l in
         inc ctor_field167;
         let ctor_field168 = proj_1 l in
         inc ctor_field168;
         let var492 = reset l in
         let var297 = ticked(ctor_field167, c, v) in
         match var297 with
         | #0 dec var492;
              dec var297;
              dec ctor_field168;
              let var298 = advance(ctor_field167, c, v) in
              let var309 = Ctor0(var298) in
              ret var309
         | #1 dec var297;
              let var299 = ticked(ctor_field168, c, v) in
              match var299 with
              | #0 dec var492;
                   dec var299;
```

```
                dec ctor_field167;
                let var300 = advance(ctor_field168, c, v) in
                let var308 = Ctor1(var300) in
                ret var308
            | #1 dec var299;
                let var301 = advance(ctor_field167, c, v) in
                let var302 = advance(ctor_field168, c, v) in
                let var307 = reuse var492 in Ctor2(var301, var302) in
                ret var307
    | default dec l;
            let var310 = match_fail("Non-exhaustive pattern match") in
            ret var310
```