Compiled Async RaTT

Awake when needed

Marcus Sebastian Emil Holmgaard
`seho@itu.dk`

Ahmad Sattar Atta
`ahsa@itu.dk`

Code: https://github.com/thehabbos007/ComRaTT

Supervisor Patrick Bahr
Submission 02. June 2025
Stads code KISPECI1SE

IT UNIVERSITY OF COPENHAGEN

Abstract

This thesis presents ComRaTT, the first implementation of a subset of Async RaTT that compiles directly to WebAssembly. Async RaTT is a functional reactive programming (FRP) language that uses modal types to allow different input channels to activate parts of a program independently. We develop a compiler that transforms ComRaTT programs into WebAssembly bytecode through multiple compilation passes, implementing essential features including signals, the stable modality, the later modality for delayed computations, clock tracking, and a reactive runtime system with custom heap management.

The main contribution is demonstrating the feasibility of compiling modal FRP languages to low-level bytecode while preserving reactive semantics. This represents a first step toward a complete implementation, with further work necessary to implement garbage collection, to incorporate more functional programming constructs, and to ensure that all well-typed programs produce valid WebAssembly code. ComRaTT serves as a proof-of-concept that reveals both the potential and challenges of targeting WebAssembly for functional reactive programming.

Contents

Introduction

This thesis explores compiling Async RaTT [3] to WebAssembly. Async RaTT is a functional reactive programming (FRP) language that uses modal types to track when values are available and which input channels can trigger computations. Unlike traditional FRP with global clocks, Async RaTT allows different inputs to activate parts of a program independently.

We present ComRaTT, a language that implements a subset of Async RaTT and compiles directly to WebAssembly bytecode. The goal is to see whether we can bring the reactive semantics of Async RaTT to WebAssembly platforms while working within the constraints of a stack-based virtual machine.

This work builds on a research project we completed in the fall of 2024 [18], but represents a substantial evolution. We introduce modal types with a bidirectional type checker, implement a proper reactive runtime system with heap management for delayed computations, and move from generating WebAssembly text format to binary format. In addition the implementation uses Rust instead of OCaml.

Compiling (a subset of) Async RaTT to WebAssembly is not a trivial endeavor. Functional language features like closures and higher-order functions d not map naturally to WebAssembly's stack machine. The reactive semantics require managing delayed computations across input events, which means we need custom memory layouts and a runtime that can activate the right computations when inputs arrive. WebAssembly's current limitations around garbage collection make this even more difficult.

Given these challenges, we narrowed the scope of the compiler significantly. ComRaTT focuses on the core reactive features including signals, the stable modality, the later modality, clock tracking, and basic functional constructs. We skipped many standard language features like polymorphism, pattern matching, and rich data types.

The result is ComRaTT, a language that compiles reactive programs with modal types into executable WebAssembly bytecode. Our implementation handles the essential reactive constructs—signals, modalities, and clock dependencies—proving that Async RaTT's core concepts can be realized on WebAssembly platforms. While this initial version establishes the foundational compilation techniques, it also identifies key areas for future development: implementing automatic memory management, expanding the supported language features, and resolving remaining compilation edge cases.

Chapter 1

Asynchronous Functional Reactive Programming

This chapter provides the background concepts needed to understand ComRaTT, our implementation of a subset of Async RaTT. We introduce functional reactive programming and examine the core calculus of Async RaTT by Bahr and Møgelberg [3].

1.1 Functional Reactive Programming

Functional reactive programming is a programming paradigm focused on modeling time-varying values and reactive behaviors in a functional setting. In FRP, these time-varying values are represented as signals, which model events within a program. Signals can be transformed with functional combinators to build computational graphs. For discrete FRP languages, time progresses in discrete steps called ticks, meaning the system only executes the program when input data is available. A reactive program is thus a function from input signals to output signals.

As outlined by Holmgaard and Sattar Atta [18], the FRP paradigm has seen several implementations and variations over the years. Fran [12] introduced FRP for animation, using continuous time-varying values. Yampa [19] built on these ideas, providing an arrow-based framework for hybrid systems. Elm [9] brought FRP to web development, though later versions moved away from classical FRP concepts. ReactiveML [26] explored synchronous programming concepts similar to FRP but with an ML-style approach.

Modal FRP is a family of languages [3, 24, 25] where *modal types* are used to ensure several key properties: causality, productivity, and absence of space leaks. In the literature, these modal types are presented as the modal type constructors $\bigcirc$ (*later*), which refers to values that are available at a later, global timestep; $\Box$ (*stable*), which refers to values that are always available; and a guarded fixed point operator with the type $(\bigcirc A \to A) \to A$, which allows guarded recursion.

Given these modal type constructors, we can define the concept of a signal as a value available now, along with a delayed computation to produce a new value at a later timestep: $\text{Sig } A = A \times \bigcirc (\text{Sig } A)$, where the head is now and the tail is later.

Types without modal type constructors represent values currently available regardless of their origin and are considered *stable*. Stable values are either primitive values (i.e.

types that are neither $\Box$ nor $\bigcirc$) that can be used immediately, or boxed ($\Box$) values that require unboxing to retrieve their inner value. However, the inner value of a boxed binding itself may not be immediately available—for instance, if we have a value of type $\Box(\bigcirc\,\mathbb{Z})$, the $\bigcirc\,\mathbb{Z}$ value after unboxing will not yield a value now. Values of type later cannot be immediately forced to produce a value within the program, but instead rely on being activated by a tick in the program's runtime environment. These temporal distinctions form the foundation for Async RaTT's approach to reactive programming.

1.2 Async RaTT

Async RaTT [3] extends the modal type constructors with concrete language constructs for asynchronous functional reactive programming. The core calculus provides `delay` to introduce the later modality ⓔ, `adv` to eliminate later types in specific contexts, `box` for the stable modality $\Box$, and `unbox` for stable elimination. The language also includes a guarded fixed point operator fix, that enables recursive definitions while maintaining productivity guarantees. The ⓐ modality is used to restrict fixed points to only unfold in the future.

Using the surface language of Async RaTT [3, section 3], we can explore an example map program presented by Bahr and Møgelberg [3, section 3.1]:

$$\begin{aligned}&\mathit{map} : \Box\,(A \to B) \to \mathrm{Sig}\ A \to \mathrm{Sig}\ B\\ &\mathit{map}\ f\,(x :: xs) = \mathrm{unbox}\ f x :: \mathrm{delay}\ (\mathit{map}\ f\,(\mathrm{adv}\ xs))\end{aligned}$$

This top-level declaration receives a boxed function f and a signal of type Sig A, producing a new signal of type Sig B. The transformation occurs by applying the boxed function to the head and recursively to the tail. Because the function f is boxed, we must unbox f before applying it to x. Signals are constructed and eliminated with the infix operator and pattern '::', where the left side is the current value of the signal, and the right side is the value of the signal at a later timestep. We can immediately apply the unboxed function f to the current signal value x to compute the new signal head. To understand how the tail is computed along with the recursive call to *map*, we must examine how the program desugars into the core Async RaTT calculus as shown by Bahr and Møgelberg [3, section 3.5].

$$\begin{aligned}\mathit{map} = \mathrm{fix}\ r.\lambda f.\lambda s.&\mathrm{let}\ x = \pi_1(\mathrm{out}\ s)\ \mathrm{in}\ \mathrm{let}\ xs = \pi_2(\mathrm{out}\ s)\\ &\quad\mathrm{in}\ \mathrm{into}\Big(\mathit{unbox}\ f\,x, \mathrm{delay}_{\mathrm{cl}(xs)}(\mathrm{adv}\ r\,f\,(\mathrm{adv}\ xs))\Big)\end{aligned}$$

The top-level definition of *map* desugars to a lambda abstraction using the fixed point constructor, providing us with r, which guards recursion on a future timestep. The type of r is ⓐ, which must be eliminated through `adv`. The tail of the signal calls this recursive binding r under a cl(xs) tick through adv r in the delay body. To ensure guarded recursion, recursive calls must occur under a tick, as we need to eliminate the $\textcircled{\forall}$ modality.

This example demonstrates how Async RaTT's temporal constructs work, but the language's key innovation lies in replacing global clocks with *input channels*. We can see this in the desugared form where $\mathsf{delay}_{\mathrm{cl}(xs)}$ explicitly states which clock governs the delayed computation. This clock-based approach enables the asynchronous behavior, where different parts of a program can be activated independently by different input sources, that distinguishes Async RaTT from traditional FRP systems.

1.3 Input channels and clocks

In Async RaTT [3], the notion of a global clock is replaced with discrete input channels κ, and delayed computations (values of type later) are associated with individual clocks θ.

Async RaTT has three kinds of input channels, *push-only* channels p, *buffered-only* channels b, and *buffered-push* channels bp. The channel kinds that are *push* refer to the fact that an active input will activate downstream computations of dependants. The buffered channels will persist their latest value, such that the values can be accessed by a `read` expression in programs regardless of what the active input channel is. An example of a buffered-only input channel is a *current time* channel, such a channel can be read at any point by the program.

A clock consists of a set of input channels, marking a delayed computation to be resumable by any active input channel $\kappa \in \theta$. This aspect of Async RaTT is what makes it "asynchronous"—inputs can activate parts of the program independently of each other, allowing efficient resumption of delayed computations. The property of independent handling of input data gives rise to a new property, *signal independence*, which Async RaTT [3, section 4.4.4] ensures. The outputs x_i of Async RaTT programs will have clock $\mathrm{cl}(x_i)$ consisting of input channels which may dynamically update at runtime.

Figure 1.1 depicts how Async RaTT tracks input channels throughout a program. The program receives input on channels shown on the left: $\kappa_1, \kappa_2, \kappa_3$. The middle nodes form the computation graph that transforms the input signals. Each clock shown inside the nodes represents a union of input channels. For example, the clock $\theta_1 = \{\kappa_1\} \sqcup \{\kappa_2\}$ means that the underlying delayed computation can be resumed by either input channel κ_1 or κ_2 becoming active.

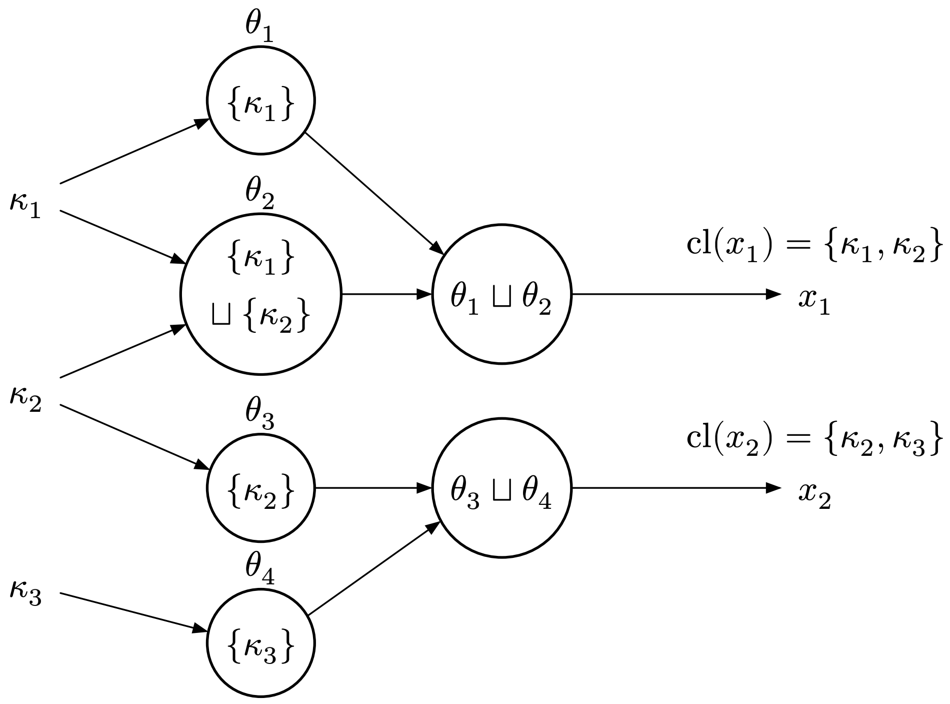

θ_1
$\{\kappa_1\}$
θ_2
$\{\kappa_1\} \sqcup \{\kappa_2\}$
θ_3
$\{\kappa_2\}$
θ_4
$\{\kappa_3\}$
κ_1
κ_2
κ_3
$\theta_1 \sqcup \theta_2$
$\theta_3 \sqcup \theta_4$
$\mathrm{cl}(x_1) = \{\kappa_1, \kappa_2\}$
x_1
$\mathrm{cl}(x_2) = \{\kappa_2, \kappa_3\}$
x_2

Figure 1.1: Input channel tracking across an Async RaTT program.

On the right-hand side, the resulting outputs x_1, x_2 contain all the upstream channels they depend on. The computation graph clearly demonstrates signal independence in the sense that input channel κ_1 will cause x_1 to update because $\kappa_1 \in \mathrm{cl}(x_1)$, but not x_2 because $\kappa_1 \notin \mathrm{cl}(x_2)$. Conversely, κ_2 will cause both x_1 and x_2 to update due to the overlap in their clocks: $\kappa_2 \in \mathrm{cl}(x_1)$ and $\kappa_2 \in \mathrm{cl}(x_2)$.

Chapter 2

WebAssembly

In this chapter, we provide background information on WebAssembly (WASM). We start by giving a brief overview of what WebAssembly is before we cover the relevant concepts of the WebAssembly specification [28].

2.1 Overview

WebAssembly [28] is a compact binary instruction format meant to be a compilation target for high-level languages, run by a stack-based virtual machine (VM). Browsers and existing JavaScript VMs can implement the machine and embed WASM, making it a viable alternative and/or supplement to JavaScript in web applications. The specification describes a sandboxed execution environment that isolates modules, the WASM unit of deployment, from their host environment and provides security properties[1].

WASM features a human-readable textual representation *WebAssembly Text Format* (WAT) [28, chapter 6] for experimentation and debugging purposes. The WAT representation is a higher level representation of the *WebAssembly Binary Format*, which is a byte-encoded instruction sequence containing metadata. The compiler presented by our former work [18] targets WAT directly.

WASM's close connection to browsers and web programming makes it a good fit for Async RaTT, since asynchronous input processing, reactive semantics, as well as guarantees of productivity, causality, and (runtime) space leak freedom map well to and complement the stateful Document Object Model of the web. Websites running on resource constrained environments (smartphones, basic laptops, or tablets) benefit from preserving memory and executing lower-level code that take up less compute. Prior work exists in the space of GUI programming with Async RaTT through the Async Rattus [2] extension Widget Rattus [10], though those GUIs are outside the context of the web.

2.2 Relevant WASM concepts

When using WASM as a compilation target, it is necessary to understand the instruction set exposed by the specification and how a WASM program is structured. To this end, we will briefly describe how WASM instructions are defined, followed by explaining the concept and structure of *modules* in WASM.

[1]https://webassembly.org/docs/security/, accessed 14/05/2025

2.2.1 Core instruction set

WASM has a core instruction set [28, section 2.4] covering operations for arithmetic, reading and writing memory, comparison, conditionals, calling functions, etc. These instructions are *stack machine* instructions, operating with an implicit operand stack. WASM instructions may consume (pop) and/or produce (push) values on the underlying machine's operand stack. Some instructions may require static immediate arguments in addition to consuming values from the operand stack.

A list of example instructions from the WASM spec can be seen in Equation (2.1), with the name of the opcode on the left, followed by static immediate, followed by the expected stack shape, and the resulting stack after execution.

$$\begin{array}{lll} \text{nop} & [\cdot] & \to [\cdot] \\ \text{i32.add} & [\text{i32}; \text{i32} \cdot] & \to [\text{i32} \cdot] \\ \text{i32.wrap\_i64} & [\text{i64} \cdot] & \to [\text{i32} \cdot] \\ \text{f32.div} & [\text{f32}; \text{f32} \cdot] & \to [\text{f32} \cdot] \\ \text{i64.const } \mathbb{Z} & [\cdot] & \to [\text{i64} \cdot] \\ \text{i32.store } h \, \{\text{offset}, \text{align}\} & [\text{i32}; \text{i32} \cdot] & \to [\cdot] \end{array} \tag{2.1}$$

We can construct a small example program, providing a trace of the operand stack. The program adds two 64-bit numbers and casts the result to a 32-bit number: $[\text{i64.const } 38; \text{i64.const } 4; \text{i64.add}; \text{i32.wrap\_i64}]$. The stack will look as follows:

$$\begin{array}{rr} \cdot \rightsquigarrow & [\,] \\ \text{i64.const } 38 \rightsquigarrow & [38_{\text{i64}}] \\ \text{i64.const } 4 \rightsquigarrow & [4_{\text{i64}}; 38_{\text{i64}}] \\ \text{i64.add} \rightsquigarrow & [42_{\text{i64}}] \\ \text{i32.wrap\_i64} \rightsquigarrow & [42_{\text{i32}}] \end{array}$$

The intermediary stacks are manipulated according to the instruction's specification. This is the basis for how WASM instructions interact, and the operational semantics are very thoroughly described in the WASM Specification [28] and formally verified by Watt et al. [34].

2.2.2 Feature extending proposals

The WASM instruction set is constantly evolving and being extended by so-called proposals. Proposals are needed for major changes like adding new instructions or changing semantics. Proposals are integrated through a standardization process[2], and

[2] https://github.com/WebAssembly/meetings/blob/main/process/phases.md, accessed 14/05/2025

the support status of different features can be tracked at https://webassembly.org/features/.

When considering opting in to a proposal, it is relevant to also assess its maturity, since support for the proposal may be lacking in browsers and WASM runtimes. Many proposals that were initially left out of the core WASM spec have already been standardized, with the March 2025 release of the WASM 2.0 spec adding even more[3]. Noteworthy features that started as proposals are *garbage collection*[4], *multiple memories*[5], and *tail call*[6].

We will make use of the *multiple memories* proposal in this project, as we want to segregate parts of a program's memory heap. The multiple memory proposal allows disjoint heaps that can be accessed and modified independently of each other. The *tail call* proposal is relevant for functional programming languages in general, which is why we will rely on it preemptively.

2.2.3 Embedding in a host environment

WASM is designed to be used in the context of a host environment—a VM that implements the semantics of the specification [28, section 1.2.1]. Having multiple implementations means that WASM binaries are portable across environments and gives way to using the same WASM binary in a browser as well as in a command line interface program. A WASM binary can import and export definitions to the host environment, allowing a foreign function interface between host and program, which is how portability is accomplished, as each environment must provide a program with functionality that affects the world outside the WASM virtual machine.

All modern browsers include a full WASM VM using different implementations. There are also a handful of WASM runtimes decoupled from browsers, including `wasmtime` [33], `wasmer` [32], `wasm3` [31], and many more.

2.2.4 Phases

There are three phases to running a WASM module according to the specification [28], validation, instantiation, and execution. The validation phase assesses the well-formedness of the module by type checking its contents. The instantiation phase handles initialization of tables and memories, ensures that imports match type declarations, initializes global variables and memory data segments, populates table elements, and finds and executes a start function. Finally, the execution phase begins when a function

[3]https://webassembly.org/news/2025-03-20-wasm-2.0/, accessed 15/05/2025
[4]https://github.com/WebAssembly/gc, accessed 14/05/2025
[5]https://github.com/WebAssembly/multi-memory/, accessed 14/05/2025
[6]https://github.com/WebAssembly/tail-call/, accessed 14/05/2025

is called, either the start function (via the instantiation phase) or an export (via the host environment).

Errors can occur in all three of these phases when attempting to execute a WASM program. Because of the strongly typed nature of WASM, the validation phase is able to pick up a lot of early errors, however runtime traps can still occur. The instructions in Equation (2.1) all include some contract for the implicit operand stack, which allows static checks of the program under validation.

2.2.5 Sections

A WASM module has, potentially empty, groupings of definitions (also referred to as *sections*) for different concepts like type signatures, functions, memory regions, global variables, and others. These sections must follow a specific order to preserve the validity of a module. The way a module is structured is thoroughly specified in the WASM specification [28, section 5.5.2].

A module consists of the sections seen in Figure 2.1, each of which has an explicit order (ID) and its own indexing space. This means that all types can be indexed by $0...n$ with $n+1$ being the number of types. Similarly, $m+1$ functions can be indexed by $0...m$. The indices are disjoint, so at index $i=0$, function_i and type_i are valid references to different entities in the module even though the same accessor i is used.

ID	**Section name**
0	Custom
1	Type
2	Import
3	Function
4	Table
5	Memory
6	Global
7	Export
8	Start
9	Element
10	Code
11	Data
12	Data count

Figure 2.1: Table of all WASM module sections, in order.

A high level description of the sections that are relevant for ComRaTT follow below. Each will have a small WAT example to go with it. The examples will omit the obligatory module declaration. The sections we have omitted are simply not of relevance for ComRaTT programs.

While WebAssembly definitions are referenced by index based on the order of their declaration, when dealing with WAT, each definition can optionally be given a name to be used instead of an index. This principle also extends to local variables and parameters within function definitions. The examples we will show below will all use a name for the definition. It is valid to refer to either the index or the name when writing WAT code. In WASM binary format, the names are simply debug information, meaning types (and any other entity in a module) are referenced by index only in code.

2.2.6 Type section

Type signatures in WASM have multiple purposes. Their main purpose is to provide static validation information for programs. An example is the use of a type index in the static immediate argument(s) to the `return_call_indirect {tbl_idx} {typ_idx}` and `call_indirect {typ_idx}` instructions. Providing a type statically allows the WASM validation phase to discern what stack obligations a function call puts on the operand stack, and what is left on the stack after the function call.

Types are defined in the module with the `type` constructor seen in Listing 2.1.

```
(type (func (param i32 i32) (result i32)))
(type $test (func (param $x i32) (result i32)))
```

Listing 2.1: Two type definitions within a WASM module

The first definition is an unnamed type, index `0` describing a function that transforms the stack $[\text{i32}; \text{i32} \cdot] \to [\text{i32} \cdot]$. Below type `0`, is the named `$test` type at type index `1`, which types a function that transforms the stack by $[\text{i32} \cdot] \to [\text{i32} \cdot]$.

2.2.7 Function section

Function definitions declare their parameters by type and optionally one or more return types. Local variables must be declared at the beginning of the function body. Functions can call themselves recursively and each other in a mutually recursive manner by default. Listing 2.2 shows an example of a single function definition.

```
(func $add (param $x i32) (param $y i32) (result i32)
  local.get $x
  local.get $y
  i32.add
)
```

Listing 2.2: A WASM module with a function definition

Similarly to types, the function `$add` has the function index `0` given by the order it appears in. Calling the `$add` function can be done with the `call` instruction `call $add` or `call 0`.

2.2.8 Table section

Tables are an array-like, resizable structure that store objects of WASM reference types `funcref`, or references to values in the host environment `externref` [28, section 2.5.4]. Tables of `funcref` are especially relevant for ComRaTT because they allow the use of `call_indirect` instructions, which can be called with a table index and accompanying type, essentially enabling function pointers and thus help treat functions as first-class citizens.

Tables are declared with an optional name, an initial/minimum size, an optional maximum size and the type of their contents.

```
(table $functions 0 128 funcref)
(table 10 funcref)
```

Listing 2.3: A WASM module with two table definitions

Listing 2.3 shows a named `funcref` table `$functions` (table index `0`), with an initial size of 0 and a maximum size of 128, and an unnamed, unbounded `funcref` table (table index `1`) of initial size 10.

2.2.9 Element section

To statically initialize entries in a WASM table, the Element section must be used. This section contains a mapping from table indices to function indices.

The table in Listing 2.4 will contain function `$id` in the `$fns` table at index `0`. This means that the table will conceptually have the following shape $[0\ \cdot]$. Having a table allows for indirect calls to functions in the table when the function type is known statically. `call_indirect $fns (type $id#typ)` if the stack has shape $[2\mathrm{i32}; 0\mathrm{i32}\ \cdot]$ where the first value on the stack corresponds to the function argument `$x = 2` followed by the index `0` of the function in the table `$fns`. We can thus call functions through tables if we know the function index and type index the call refers to.

```
(table $fns funcref)
(elem $fns (i32.const 0) $id)

(type $id#typ (func (param $x i32) (result i32)))
(func $id (param $x i32) (result i32)
  local.get $x
)
```

Listing 2.4: A WASM module populating two elements into a table

The WASM virtual machine will ensure that the type index provided statically matches the type of the function extracted from a function table at runtime. This is because `funcref` tables contain references to a heterogenous set of functions, requiring such runtime checks.

2.2.10 Memory section

WASM memories [28, section 2.5.5] are resizable linear arrays of raw uninterpreted bytes, just like a regular program heap. They can be both imported and exported allowing for sharing of data between WASM modules and host environments. They are declared with an initial size and an optional maximum size. The sizing arguments are given in pages of 64KiB.

The initial release of WASM only supported a single memory region but the now-standardized multiple memories proposal changes this. See Listing 2.5 for an example module containing two memory regions: one unnamed with a minimum page size of 1 (~64KiB) and maximum page size of 4 (~256KiB) and a named, unbounded region with a minimum page size of 1 (~64KiB).

```
(memory 1 4)
(memory $heap 1)
```

Listing 2.5: A WASM module with two memory regions

The heaps in Listing 2.5 can be read from and written to using the static immediate arguments in the relevant instructions touching memory. For example the `i32.load h {offset, align}` instruction takes an optional heap specifier that defaults to memory index `0`. To load a value from the `$heap` heap, we can issue the following instruction `i32.load (memory $heap) offset=0 align=2 (i32.const 0)`, which loads the memory at index 0 of `$heap`.

2.2.11 Global section

Global definitions are globally accessible variables that are optionally mutable. Listing 2.6 shows two named global variables of type `i32` initialized to 0, one immutable and one mutable.

```
(global $state (i32) (i32.const 0))
(global $mut_state (mut i32) (i32.const 0))
```

Listing 2.6: A WASM module with two global variables

Globals can be optionally exported such that the host environment is able to access them.

2.2.12 Import section

If a WASM program depends on foreign functions provided by the host environment, such functions need to be declared as imports. The name of an import consists of the name of a module and the name of an entity within the module providing function namespacing for any imported functions. All typed imported entities must explicitly state their types, such that they can be used in module validation. For example, an imported function also should provide its parameters and return type.

Valid import definitions are functions, tables, memories and globals. Listing 2.7 declares two imports from two separate modules: one of a function and one of a memory region.

```
(import "some_module" "test_function" (func $test (param i32) (result
i32)))
(import "other_module" "heap" (memory 0))
```

Listing 2.7: A WASM module with two imports

Anything imported in a WASM module can be used as if it was locally declared, but is initialized and passed to the WASM module at runtime.

2.2.13 Export section

To allow the host environment to interact with the WASM program, functions within the module can be exported (after instantiation). An export declaration consists of a name and a descriptor (type of definition and name/index), see Listing 2.8. The WASM definitions valid for export are functions, tables, memories and globals.

```
(export "test" (func $test))
(export "heap" (memory 0))
(export "functions" (table $functions))
(export "state" (global $state))
```

Listing 2.8: A WASM module with four exports

These will be available by their explicitly given name in the host environment.

2.3 Functional source vs. imperative stack machine

The imperative, stack-based execution model of WASM is fundamentally different from the pure, declarative, functional nature of Async RaTT. Language features such as partial application and implicit arguments are not directly supported in WASM.

WASM programs must be structured as top-level function definitions that can only access their parameters, local variables and global variables. Additionally, functions cannot nest, conflicting with the use of lambda abstractions (anonymous functions) which are a staple of the functional paradigm.

Furthermore, a WASM function must forward declare all of its local variables, which means any temporary or intermediary variables must be known up front and declared for them to be used in a function's body.

This means that a compiler for a high-level language targeting WASM must address these differences appropriately to generate code. The techniques used to bridge the gap are covered in detail throughout Chapter 4 Compiling ComRaTT (p. 29).

Chapter 3

The ComRaTT language

ComRaTT is an implementation of a subset of Async RaTT as a language that targets WebAssembly. In this chapter, we will provide descriptions and definitions of the language, forming the conceptual building blocks used throughout the thesis to explain the work we have done.

The chapter will present the syntax and the type system for ComRaTT followed by the language semantics of ComRaTT. We will be selecting parts of Async RaTT [3] that are essential for us to include in a functional reactive programming language. At the end, we will showcase an example program.

3.1 Language syntax

Async RaTT [3] presents a rich core calculus with many powerful features and concepts, but to keep ComRaTT small and minimal, we have picked out parts of Async RaTT that we found to be most essential.

The basic expression language of ComRaTT contains lambda abstractions, function application, let-bindings, and conditionals. It supports integer literals, boolean literals, and basic binary operations as well as tuples. This gives enough expressivity without overwhelming the implementation with features such as custom algebraic data types, parametric polymorphism, and lists. The first iteration of the language is described in Holmgaard and Sattar Atta [18], from which some of these core elements have been brought over.

Most importantly, ComRaTT supports the later ($\bigcirc$) modality introduction form `delay` and elimination form `advance` for delimiting and handling ticks, along with `box` and `unbox` for stable values that may persist across time. These give us the fundamental ability to describe values as being available "now" versus values available "later", as well as values that are available "forever" [3].

The grammar of ComRaTT is provided in Figure 3.1, where both the core expression syntax and the FRP constructs reside side by side. An important detail to point out is the fact that we provide explicit clocks in the syntax for `delay`. ComRaTT does not fully infer clocks the same way Async Rattus [5] or the assumed Async RaTT surface language [3, section 3] does, which means that we need to provide annotations for type checking.

$$
\begin{array}{rcll}
\text{Channel} & \kappa & \in & x \\
\text{Toplevel} & \mathrm{T} & ::= & \mathsf{chan}\ \kappa : t; \\
 & & \mid & x : t\ \ \mathsf{def}\ x\ x^{*} = e; \\
 & & \mid & x \leftarrow e; \\
\text{Type} & t & ::= & () \mid \mathbb{Z} \mid \mathbb{B} \mid t \to t \mid \bigcirc t \mid \square t \mid \mathsf{Sig}\ t \\
\text{Expression} & e & ::= & v \mid (e(,e')^{*}) \mid e\ .\ \mathsf{int} \mid e\ e' \mid e \oplus e' \mid \mathsf{fun}\ x^{*} \to e \\
 & & \mid & \mathsf{let}\ x = e\ \mathsf{in}\ e' \mid \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \mathsf{delay}\ \{\theta\}\ e \\
 & & \mid & \mathsf{advance}\ x \mid \mathsf{wait\ var} \mid \mathsf{box}\ e \mid \mathsf{unbox}\ e \mid \mathsf{never} \\
\text{Value} & v & ::= & \mathbb{Z} \mid x \mid \mathsf{true} \mid \mathsf{false} \mid () \\
\text{Clock expression} & \theta & ::= & \mathsf{cl}(x) \mid \mathsf{cl}(\mathsf{wait}\ x) \mid \theta \sqcup \theta \\
\text{Binary operator} & \oplus & ::= & :: \mid + \mid * \mid / \mid - \mid = \mid < \mid <= \mid > \mid >= \mid <>
\end{array}
$$

Figure 3.1: Syntax of the ComRaTT source language.

A ComRaTT program is made up of many top-level definitions, consisting of *channel definitions*, *function declarations*, and *output declarations*. It is not necessary for a user to explicitly define an entry-point for a program; instead, a program's outputs dictate what part(s) of the program can be executed.

```
chan some_channel : int; // unused channel

just_two : Sig int
def just_two = 2 :: never;

print <- just_two;
```

Listing 3.1: A simple ComRaTT program printing "2" once.

Programs must specify what channels they use, and for each channel, the type of values it produces. It is then the runtime system's responsibility to ensure that the channels and their types are valid and possible to provide.

An output definition contains a single expression which should have type `O Sig t` (later signal of type `t`), though we should also be able to support outputs of type `Sig t` (signal of type `t`). The code in Listing 3.1 shows such a simple example program. The `just_two` function declaration defines a signal producing the value `2`, and a later computation that will never be executed.

3.2 Type system

Whereas the initial version of ComRaTT [18] implements a Hindley–Milner style type system with principal inference based on Sestoft [29], ComRaTT now uses a bidirectional type system that expects explicit type declarations for every top-level function definition. This can be seen in the grammar, Figure 3.1, as well as in Listing 3.1 where the type of `just_two` is explicitly stated as `Sig int`.

Adding clocks to ComRaTT meant we had to consider how typing rules would be adjusted. We have been adamant about not inferring clocks in ComRaTT to reduce complexity of implementing the compiler, which led us to exploring alternative type systems that would give us the possibility to type check terms in addition to inferring types. Bidirectional typing, as presented by Dunfield and Krishnaswami [11], proved to be an extensible way for us to balance type inference and type checking.

It is important to point out that we do not propose that ComRaTT's type system is sound nor complete. As the Async RaTT type system [3, section 2] is provably sound, we are more interested in the entire "vertical slice" of a working, but not perfect, compiler being feasible. The vertical slice we are referring to is everything required to compile a ComRaTT program that uses the central constructs of Async RaTT, namely signals, delayed computations, clocks, input channels, and a reactive runtime to tie everything together.

The initial design of the bidirectional type system is based loosely on the work of Ethan Smith [7], where Hindley-Milner is utilized for inference, using constraint solving in a bottom-up fashion. The type system consists of rules that can make use of both modes as seen in Figure 3.2. The typing rules marked with $\uparrow$ are inference rules utilizing Hindley-Milner, while rules marked with $\downarrow$ are checking rules, expecting a type annotation. In these rules, the type judgements are split into inference judgements; $\Gamma \vdash e \Rightarrow \tau$ read as "the type of e can be inferred to τ under context Γ", and checking judgements, $\Gamma \vdash e \Leftarrow \tau$ read as "the type of e can be checked to have type τ in context Γ", as explained by Christiansen [8].

$$\frac{}{\Gamma \vdash_{\triangle} () \Rightarrow \text{Unit}}\,\text{Unit}\uparrow \qquad \frac{n \in \mathbb{Z}}{\Gamma \vdash_{\triangle} n \Rightarrow \mathbb{Z}}\,\text{Int}\uparrow \qquad \frac{b \in \{\text{true}, \text{false}\}}{\Gamma \vdash_{\triangle} b \Rightarrow \mathbb{B}}\,\text{Bool}\uparrow$$

$$\frac{\checkmark_{\theta} \notin \Gamma' \text{ or } \tau \text{ stable}}{\Gamma, x : \tau, \Gamma' \vdash_{\triangle} x \Rightarrow \tau}\,\text{Var}^{\dagger}\uparrow \qquad \frac{\Gamma \vdash_{\triangle} x \Rightarrow \bigcirc\,\tau}{\Gamma \vdash_{\triangle} \text{cl}(x) \Rightarrow \text{Clock}}\,\text{ClockCl}^{\dagger}\uparrow$$

$$\frac{\Gamma \vdash_{\triangle} \quad \Gamma \vdash_{\triangle} \theta \Rightarrow \text{Clock} \quad \checkmark_{\theta} \notin \Gamma}{\Gamma, \checkmark_{\theta} \vdash_{\triangle}}\,\text{ClockTick}^{\dagger}\uparrow \qquad \frac{\Gamma \vdash_{\triangle} \theta \Rightarrow \text{Clock} \quad \Gamma \vdash_{\triangle} \theta' \Rightarrow \text{Clock}}{\Gamma \vdash_{\triangle} \theta \sqcup \theta' \Rightarrow \text{Clock}}\,\text{ClockUnion}^{\dagger}\uparrow$$

$$\frac{\Gamma \vdash_{\triangle} e_1 \Rightarrow \tau_1 \ldots e_n \Rightarrow \tau_n \quad n \geq 2}{\Gamma \vdash_{\triangle} (e_1, \ldots, e_n) \Rightarrow (\tau_1, \ldots, \tau_n)}\,\text{Tuple}\uparrow \qquad \frac{\Gamma \vdash_{\triangle} e \Rightarrow (\tau_1, \ldots, \tau_n) \quad 1 \leq i \leq n}{\Gamma \vdash_{\triangle} e.i \Rightarrow \tau_i}\,\text{Access}\uparrow$$

$$\frac{\Gamma \vdash_{\triangle} e_{\text{val}} \Rightarrow \tau \quad \Gamma \vdash_{\triangle} e_{\text{later}} \Rightarrow \bigcirc\,\text{Sig}\ \tau}{\Gamma \vdash_{\triangle} e_{\text{val}} :: e_{\text{later}} \Rightarrow \text{Sig}\ \ \tau}\,\text{Sig}\uparrow$$

$$\frac{\Gamma \vdash_{\triangle} e_1 \Leftarrow \mathbb{B} \quad \Gamma \vdash_{\triangle} e_2 \Rightarrow \tau \quad \Gamma \vdash_{\triangle} e_3 \Rightarrow \tau' \quad \tau = \tau'}{\Gamma \vdash_{\triangle} \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Rightarrow \tau}\,\text{If}\uparrow$$

$$\frac{\Gamma \vdash_{\triangle} e_1 \Rightarrow \tau_1 \quad \Gamma, x : \tau_1 \vdash_{\triangle} e_2 \Rightarrow \tau_2}{\Gamma \vdash_{\triangle} \text{let } x = e_1 \text{ in } e_2 \Rightarrow \tau_2}\,\text{Let}\uparrow \qquad \frac{\Gamma, \checkmark_{\theta} \vdash_{\triangle} e \Rightarrow \tau \quad \Gamma \vdash_{\triangle} \theta : \text{Clock}}{\Gamma \vdash_{\triangle} \text{delay } \{\theta\}\ e \Rightarrow \bigcirc\,\tau}\,\text{Delay}^{\dagger}\uparrow$$

$$\frac{\Gamma \vdash x \Rightarrow \bigcirc\,\tau}{\Gamma, \checkmark_{\text{cl}(x)}, \Gamma' \vdash_{\triangle} \text{advance } x \Rightarrow \tau}\,\text{Advance}^{\dagger}\uparrow \qquad \frac{\kappa : \tau \in \Delta}{\Gamma \vdash_{\triangle} \text{wait } \kappa \Rightarrow \bigcirc\,\tau}\,\text{Wait}^{\dagger}\uparrow$$

$$\frac{\Gamma \vdash_{\triangle} e_1 \Rightarrow \tau \quad \Gamma \vdash_{\triangle} e_2 \Rightarrow \tau \quad \Gamma \vdash_{\triangle} e_1 \oplus e_2 \Rightarrow \tau'}{\Gamma \vdash_{\triangle} e_1 \oplus e_2 \Rightarrow \tau'}\,\text{BinOp}\uparrow$$

$$\frac{\Gamma, x_1 : \tau_1 \ldots x_n : \tau_n \vdash_{\triangle} e \Rightarrow \tau}{\Gamma \vdash_{\triangle} \text{fun } (x_1, \ldots, x_n) \rightarrow e \Rightarrow \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau}\,\text{Lam}\uparrow$$

$$\frac{\Gamma \vdash_{\triangle} e_1 \Rightarrow \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash_{\triangle} e_2 \Leftarrow \tau_1}{\Gamma \vdash_{\triangle} e_1\ e_2 \Rightarrow \tau_2}\,\text{App}\uparrow \qquad \frac{\Gamma^{\square} \vdash_{\triangle} e \Rightarrow \tau}{\Gamma \vdash_{\triangle} \text{box}\ \ e \Rightarrow \square\,\tau}\,\text{Box}^{\dagger}\uparrow \qquad \frac{\Gamma \vdash_{\triangle} e \Rightarrow \square\,\tau}{\Gamma \vdash_{\triangle} \text{unbox}\ \ e \Rightarrow \tau}\,\text{Unbox}^{\dagger}\uparrow$$

$$\frac{\Gamma, x : \tau_1 \vdash_{\triangle} e \Leftarrow \tau_2}{\Gamma \vdash_{\triangle} \text{fun } (x) \rightarrow e \Leftarrow \tau_1 \rightarrow \tau_2}\,\text{Lam}\downarrow \qquad \frac{\Gamma \vdash_{\triangle} e \Rightarrow \tau}{\Gamma \vdash_{\triangle} x \leftarrow e \Leftarrow \bigcirc\,\tau}\,\text{Output}\downarrow$$

$$\frac{\tau = \ldots \rightarrow \tau_r \text{ or } \tau_r \quad \Gamma, f : \tau, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash_{\triangle} e \Rightarrow \tau' \quad \tau_r = \tau'}{\Gamma \vdash_{\triangle} \text{def } f\ (x_1, \ldots, x_n) : \tau = e \Leftarrow \tau}\,\text{FunDef}\downarrow$$

Figure 3.2: Bidirectional Type Inference Rules.

Similarly to [3], we make use of the context Δ to be the reactive context containing the input channels for the program. Rules additionally marked with a dagger †, are ported over from the Async RaTT Type system [3, section 2].

Figure 3.2 covers a fair share of the rules from Async RaTT [3], where the main differences include fewer rules and lifting some restrictions. We do not restrict lambda abstractions to be outside of ticks—which according to Bahr [1] should not cause space leaks. Furthermore, we only allow input channels in the context to be *push-only* channels, explaining why we do not inspect the channel type for wait and entirely leave out read. The language design and type system naturally define a conceptual starting point for programs to be type checked: the top-level function definitions. These definitions are used to provide cross-program (*global*) type annotations for functions and start off the *local* Hindley-Milner constraint solving on a per-function granularity.

ComRaTT has no explicit fixpoint construct, it is expected that the host machine implicitly boxes top-level definitions and unboxes them implicitly once the top-level definitions are applied as functions as described by Async RaTT [3, section 3.5]. ComRaTT ensures that the body of top-level function definitions only access local variables or boxed parameters, similarly to how the fixed point operator in Async RaTT would behave.

The main advantage we found using bidirectional typing is that rules are syntax-directed, providing more algorithmic definitions when compared to the typing rules of the original ComRaTT type system [18]. This aspect of bidirectional typing is explained further by Dunfield and Krishnaswami [11].

3.3 Semantics

Well-typed ComRaTT programs are not executable on their own; they must be executed within the context of a runtime. When the runtime executes a program, it must obey the operational semantics of ComRaTT seen in Figure 3.3, which is based on Holmgaard and Sattar Atta [18] and Bahr and Møgelberg [3]. As ComRaTT is an implementation of Async RaTT with concessions made for the breadth of implementation, the semantics are mostly that of Async RaTT, with some restrictions and omissions. The rules marked with a dagger (†) in Figure 3.3 are more or less identical to their counterparts in Async RaTT [3].

Similarly to Async RaTT [3], ComRaTT uses locations ℓ to keep track of delayed computations. Each delayed computation consists of a clock (set of input channels) and a heap-allocated thunk. The location's clock is what enables the runtime to resume a computation if it matches the active input channel. The operational semantic rules E-Delay, E-Adv, Wait, E-Adv-Wait, and E-Never describe how delayed computations are handled. A user of ComRaTT may "suspend" a computation e with the delay construct, given a clock θ. This allocates a location ℓ and wraps the computation in a thunk $\lambda.e$. Resumed computations will be able to advance locations that match the current active clock, essentially forcing active thunks wherever it is required.

$$\text{E-Value}^{\dagger}\;\frac{}{\langle v;\sigma\rangle \Downarrow \langle v;\sigma\rangle} \qquad \text{E-Let}^{\dagger}\;\frac{\langle e_1;\sigma\rangle \Downarrow \langle v_1;\sigma'\rangle \quad \langle e_2[v_1/x];\sigma'\rangle \Downarrow \langle v_2;\sigma''\rangle}{\langle \text{let } x = e_1 \text{ in } e_2;\sigma\rangle \Downarrow \langle v_2;\sigma''\rangle}$$

$$\text{E-App}\;\frac{\begin{array}{c}\langle e_1;\sigma\rangle \Downarrow (\lambda(x_1,...,x_{n-1}).e;\sigma_1) \quad \langle e_i;\sigma_{i-1}\rangle \Downarrow \langle v_i;\sigma_i\rangle \\ \langle e[v_i/x_{i-1}];\sigma_n\rangle \Downarrow \langle v';\sigma'\rangle \quad i \in \{2..n\}\end{array}}{\langle e_1(e_2,...,e_n);\sigma\rangle \Downarrow \langle v';\sigma'\rangle}$$

$$\text{E-BinOp}\;\frac{\begin{array}{c}\langle e_1;\sigma\rangle \Downarrow \langle v_1;\sigma'\rangle \quad \langle e_2;\sigma'\rangle \Downarrow \langle v_2;\sigma''\rangle \\ \oplus \in \{+,*,-,=,<>,<,\leq,>,\geq,\&\&,||\}\end{array}}{\langle e_1 \oplus e_2;\sigma\rangle \Downarrow \langle v_1 \oplus v_2;\sigma''\rangle}$$

$$\text{E-If-True}\;\frac{\begin{array}{c}\langle e_1;\sigma\rangle \Downarrow \langle \text{true};\sigma'\rangle \\ \langle e_2;\sigma'\rangle \Downarrow \langle v;\sigma''\rangle\end{array}}{\langle \text{if } e_1 \text{ then } e_2 \text{ else } e_3;\sigma\rangle \Downarrow \langle v;\sigma''\rangle} \qquad \text{E-If-False}\;\frac{\begin{array}{c}\langle e_1;\sigma\rangle \Downarrow \langle \text{false};\sigma'\rangle \\ \langle e_3;\sigma'\rangle \Downarrow \langle v;\sigma''\rangle\end{array}}{\langle \text{if } e_1 \text{ then } e_2 \text{ else } e_3;\sigma\rangle \Downarrow \langle v;\sigma''\rangle}$$

$$\text{E-Delay}^{\dagger}\;\frac{\ell = \text{alloc}(\theta,\sigma)}{\langle \text{delay } \theta\ e;\sigma\rangle \Downarrow \langle \ell;\sigma,\ell \mapsto \lambda.e\rangle}$$

$$\text{E-Never}^{\dagger}\;\frac{\ell = \text{alloc}(\emptyset,\sigma)}{\langle \text{never};\sigma\rangle \Downarrow \langle \ell;\sigma\rangle}$$

$$\text{E-Adv}^{\dagger}\;\frac{\langle \eta_N(\ell);\eta_N\langle\kappa \mapsto v\rangle\,\eta_L\rangle \Downarrow \langle w;\sigma\rangle}{\langle \text{advance } \ell;\eta_N\langle\kappa \mapsto v\rangle\,\eta_L\rangle \Downarrow \langle w;\sigma\rangle}$$

$$\text{E-Wait}\;\frac{}{\langle \text{wait } \kappa;\sigma\rangle \Downarrow \langle \text{wait\_rt } \kappa;\sigma\rangle}$$

$$\text{E-Adv-Wait}\;\frac{\langle x;\eta_N\langle\kappa \mapsto v\rangle\,\eta_L\rangle \Downarrow \langle \text{wait\_rt } \kappa;\eta_N\langle\kappa \mapsto v\rangle\,\eta_L\rangle}{\langle \text{advance } x;\eta_N\langle\kappa \mapsto v\rangle\,\eta_L\rangle \Downarrow \langle v;\eta_N\langle\kappa \mapsto v\rangle\,\eta_L\rangle}$$

Figure 3.3: Big-step operational semantics of ComRaTT.

There is no effective use or interrogation of the clocks (θ) on locations in the semantics, Figure 3.3. In Async RaTT [3], the `select` combinator is the only language primitive that is interested in the clocks at runtime. For ComRaTT, it is the runtime system, described in Section 3.4 Reactive semantics (p. 24), that makes use of clocks attached to each location.

Exactly like in Bahr and Møgelberg [3], part of the program's heap can be filtered off to represent all of the delayed computations that have a clock containing the input channel κ.

$$\mathrm{Heap}^{\kappa} = \{\eta \in \mathrm{Heap} \mid \forall \ell \in \mathrm{dom}(\eta) \ .\ \kappa \in \mathrm{cl}(\ell)\}$$

Heap entries under Heap^{κ} are executable whenever an input channel is active. When data from the active input κ is requested in a program, the wait_rt construct will act as a marker for the runtime system to retrieve the data. Advancing a name that evaluates to a wait_{κ} should immediately retrieve the active input value for channel κ.

We borrow the store representation σ of Async RaTT [3, section 4.1], which specifies that σ can take two forms, either a single-heap store η_L or a two-heap store $\eta_L\langle\kappa \mapsto v\rangle\, \eta_N$. The two-heap representation is a virtual grouping and does not impose any direct requirements on the runtime system's underlying representation other than being able to address parts of the heap that is "now" (η_N) and "later" (η_L) under an active input channel tick carrying a value, $\kappa \mapsto v$.

New allocations using the $\mathrm{alloc}(\theta, e)$ function (provided by the runtime) are expected to allocate the expression e under a thunk, $\lambda.e$, for the resulting location ℓ to be a valid delayed computation. Adding allocated delayed computations to the heap, regardless of the kind of heap, is done by extending the store with $(\sigma; \ell \mapsto \lambda.e)$ with the constraint that $\ell \notin \mathrm{dom}(\sigma)$.

3.4 Reactive semantics

Similarly to Async RaTT [3], ComRaTT programs need to interact within an environment. We define the reactive semantics of ComRaTT programs in Figure 3.4, where the reactive semantics are mostly identical to that of Async RaTT, with minor differences to address the restricted subset of the language we are working within.

$$\textsc{Init}\ \frac{e \Downarrow \langle\langle \ell_1, \ldots \ell_n\rangle; \eta\rangle}{e \Longrightarrow \langle x_1 \mapsto \ell_1, \ldots, x_n \mapsto \ell_n; \eta\rangle}$$

$$\textsc{Input}\ \frac{}{\langle N; \eta\rangle \overset{k \mapsto v}{\Longrightarrow} \big\langle N; [\eta]_{k\in}\langle k \mapsto v\rangle[\eta]_{k\notin}\big\rangle}$$

$$\textsc{Output-End}\ \frac{}{\langle \cdot; \eta_N\langle k \mapsto v\rangle\, \eta_L\rangle \Longrightarrow \langle \cdot; \eta_L\rangle}$$

$$\textsc{Output-Skip}\ \frac{\kappa \notin \mathrm{cl}(\ell) \qquad \langle N; \eta_N\langle k \mapsto v\rangle\, \eta_L\rangle \overset{O}{\Longrightarrow} \langle N'; \eta\rangle}{\langle x \mapsto \ell, N; \eta_N\langle k \mapsto v\rangle\, \eta_L\rangle \overset{O}{\Longrightarrow} \langle x \mapsto \ell, N'; \eta\rangle}$$

$$\textsc{Output-Compute}\ \frac{\kappa \in \mathrm{cl}(\ell) \qquad \langle \mathrm{advance}\ \ell; \eta_N\langle k \mapsto v\rangle\, \eta_L\rangle \Downarrow \langle v' :: \ell'; \sigma\rangle \qquad \langle N; \sigma\rangle \overset{O}{\Longrightarrow} \langle N'; \eta\rangle}{\langle x \mapsto \ell, N; \eta_N\langle k \mapsto v\rangle\, \eta_L\rangle \overset{x \mapsto v', O}{\Longrightarrow} \langle x \mapsto \ell', N'; \eta\rangle}$$

Figure 3.4: ComRaTT Reactive semantics, similar to Async RaTT [3].

The main differences between ComRaTT and Async RaTT when it comes to reactive semantics, is the fact that we do not include an input buffer. This means that in ComRaTT the state of the reactive runtime will be either of shape e or of shape $\langle N, \sigma \rangle$ neither of which include an input buffer ι containing input values that are buffered across input ticks. Furthermore, ComRaTT only allows delayed computations when initializing outputs, whereas Async RaTT allows the use of signals directly. So a program e will evaluate to a series of locations ℓ_i rather than a series of signals $v_i :: \ell_i$. This only changes the fact that we do not allow start-values for outputs of programs at initialization.

Async RaTT [3, section 4.2] describes a program as being a pair of Δ, a context containing available input channels, and Γ_{out}, a context containing output channels. The pair $\Delta \Rightarrow \Gamma_{\text{out}}$ is a reactive interface that a program must implement. Thus the e form is a reactive program implementing the interface $\Delta \Rightarrow \Gamma_{\text{out}}$, where the output interface $\Gamma_{\text{out}} = x_1 : A_1, ..., x_n : A_n$, equivalent to all the output channels of the program. For the form $\langle N, \sigma \rangle$, N is a mapping from *outputs* to *locations* $x_1 \mapsto \ell_1, ..., x_n \mapsto \ell_n$ where $x_i \in$ dom (Γ_{out}). The forms do not deviate at all from Async RaTT [3], we simply omit the input buffer.

ComRaTT programs start their lifecycle at INIT, where each declared output x_i will be associated with a delayed computation ℓ_i. When an INPUT is received in the runtime, the heap is split into two, just as in Async RaTT [3, section 4.2]. The left-side of the heap contains delayed computations that match the active input channel, wheres the right-side contains delayed computations which are unaffected and to be executed later.

When executing the OUTPUT-* transitions, we iterate through all delayed computations in η_N (the now heap). Delayed computation clocks $\theta = \text{cl}(\ell)$ are interrogated sequentially through OUTPUT-COMPUTE when $\kappa \in \theta$. Outputs that do not contain the input channel in their clocks will transition through OUTPUT-SKIP when $\kappa \notin \theta$.

At the OUTPUT-END transition ComRaTT can collect garbage by removing η_N from the heap, transitioning back to a single heap η_L, similarly to Async RaTT [3, section 4.2]. The runtime repeats execution of input/output transitions indefinitely

3.5 Example program

To showcase how a ComRaTT program looks like and how it is type checked and executed, we present the program in Listing 3.2. This program is expected to react to keyboard presses and print back the value from the input channel. The program is somewhat trivial, but it will take us through the interesting parts of the ComRaTT language design.

```
chan keyboard : int;

kb : O Sig int
def kb =
  let key = wait keyboard in
  delay {cl(key)} (
    advance key :: kb
  );

print <- kb;
```

Listing 3.2: A ComRaTT program reacting to keyboard input by immediately printing.

The program contains a single input, `keyboard` producing integers, and a single output using the `print` output. Both input and output need to be supported by the underlying runtime. We define a function `kb` which has the type `O Sig int` (later signal producing integers) that binds `key` to the delayed computation `wait keyboard`.

This is a computation that has an implicit clock, cl(wait keyboard). `kb` returns a delayed computation constructed by `delay`, whose inner value is a signal (constructed with `::`) consisting of `advance key` as the head and `kb` as the tail. The return value assumes the clock of `key`, and the head of the signal is the runtime value of the keyboard input at the time of resuming the delayed computation. The tail is recursive, referencing `kb` itself, creating an infinite stream. The runtime manages extracting the head of the signal and printing it with the `print` output. We can type the `kb` top-level function by applying the rules of our type system starting from the initial top-level declaration rule.

$$
\text{1)}\dfrac{\text{2)}\dfrac{\text{3)}\dfrac{\dfrac{}{\text{keyboard} : \mathbb{Z} \in \Delta}}{\Gamma \vdash_{\Delta} \text{wait keyboard} \Rightarrow \bigcirc\, \mathbb{Z}} \qquad \text{4)}\dfrac{\text{5)}\dfrac{\dfrac{}{\Gamma \vdash_{\Delta} \text{key} \Rightarrow \bigcirc\, \mathbb{Z}}}{\Gamma \vdash_{\Delta} \text{cl(key)} : \text{Clock}} \qquad \text{6)}\dfrac{\text{7)}\dfrac{\dfrac{}{\Gamma \vdash_{\Delta} \text{key} \Rightarrow \bigcirc\, \mathbb{Z}}}{\Gamma, \checkmark_{\text{cl(key)}} \vdash_{\Delta} \text{adv key}} \qquad \dfrac{}{\Gamma \vdash_{\Delta} \text{kb} \Rightarrow \bigcirc\, \text{Sig}\ \mathbb{Z}}}{\Gamma, \checkmark_{\text{cl(key)}} \vdash_{\Delta} \text{adv key} :: \text{kb} \Rightarrow \bigcirc\, \text{Sig}\ \mathbb{Z}}}{\Gamma, \text{key} : \bigcirc\, \mathbb{Z} \vdash_{\Delta} \text{delay } \{\text{cl(key)}\} \text{ adv key} :: \text{kb} \Rightarrow \bigcirc\, \text{Sig}\ \mathbb{Z}}}{\Gamma, \text{kb} : \bigcirc\, \text{Sig}\ \mathbb{Z} \vdash_{\Delta} \text{let key} = \text{wait keyboard in delay } \{\text{cl(key)}\} \text{ adv key} :: \text{kb} \Leftarrow \bigcirc\, \text{Sig}\ \mathbb{Z}}}{\Gamma \vdash_{\Delta} \text{def kb } () = \text{let key} = \text{wait keyboard in delay } \{\text{cl(key)}\} \text{ adv key} :: \text{kb} \Leftarrow \bigcirc\, \text{Sig}\ \mathbb{Z}}
$$

Figure 3.5: Typing derivation for the example program in Listing 3.2.

In Figure 3.5, the derivation tree from recursively applying the typing rules from Figure 3.2 is shown. To present the tree, we will be referring to the labels at the left side of each rule for each step, explaining the tree bottom-up. We have omitted the outer-most leaf rules that would terminate variable bindings, and used `adv` in place of `advance` for brevity. Rule application is done under the assumption that the context Δ

has `keyboard` : $\mathbb{Z}$ as the only input channel.
Step `1)` uses the FunDef$\downarrow$ rule, which expects a top-level function definition. The contents of the function is a let-binding.
Step `2)` uses the Let$\uparrow$ rule, consisting of two parts, the right-hand side of the binding and the body expression of the binding.
Step `3)` uses the Wait$^\dagger\uparrow$ rule to the right-hand side of the let binding, this expression is a `wait` expression that terminates this part of the tree.
Step `4)` uses the Delay$^\dagger\uparrow$ rule to the body of the let expression. Here we must consider the clock of the `delay` expression and the body of the `delay` must be checked under a tick in its context.
Step `5)` uses the ClockCl$^\dagger\uparrow$ rule to establish that the `key` binding is a delayed computation (thus containing a clock).
Step `6)` uses the Sig$\uparrow$ rule, expecting a value of type τ as the head and a value of type $\bigcirc$ Sig τ as its tail. The Sig type is recursive which is why we refer to `kb` in the tail.
Step `7)` uses the Advance$^\dagger\uparrow$ rule, under the tick cl(key), resulting in advancing the `key` variable

To execute the program in Listing 3.2, we construct a reactive ComRaTT program instance with the following parameters

- $\Delta = \text{key} : \mathbb{Z}$ (shortened for brevity)
- Γ_{out} print : Sig $\mathbb{Z}$
- $e = \text{kb}$
- $\eta = \emptyset$

We can evaluate the program starting with INIT.

$$\text{kb} \Downarrow \langle \ell_1; \eta \rangle,\ \text{key} \in \text{cl}(\ell_1) \rightsquigarrow \langle x_1 \mapsto \ell_1; \eta \rangle$$

We have evaluated kb to a location (a delayed computation of type later signal of integers) that is assigned to the only output x_1. We then assume an input is received, the number 65 (ASCII code for the letter “A”) is received on input key, meaning we must apply INPUT

$$\langle x_1 \mapsto \ell_1; \eta \rangle \overset{\text{key} \mapsto 65}{\Longrightarrow} \left\langle x_1 \mapsto \ell_1; [\langle x_1 \mapsto \ell_1 \rangle]_N \langle \text{key} \mapsto 65 \rangle\, [\emptyset]_L \right\rangle$$

The runtime will filter out all locations that contain key in their clock and put them in the now heap on the left side. Since it is the only location in our program, the later heap is empty. Processing the input is done by OUTPUT-COMPUTE, as the map of output channels only contain ℓ_1, which has key $\in$ cl(ℓ_1).

$$\left\langle \text{advance } \ell_1; \left[\langle x_1 \mapsto \ell_1 \rangle\right]_N \langle \text{key} \mapsto 65 \rangle \, [\emptyset]_L \right\rangle \Downarrow \left\langle v_1 :: \ell_2; \left[\langle x_1 \mapsto \ell_1 \rangle\right]_N \langle \text{key} \mapsto 65 \rangle \, [\emptyset]_L \right\rangle$$
$$\rightsquigarrow \langle x_1 \mapsto \ell_2; \eta \rangle$$

As there are no more locations in the now heap, we will finish the computation loop by OUTPUT-END, which clears out the now heap (garbage collection) and puts the program's reactive state into waiting for inputs. For larger programs, many more outputs could be processed, and the input channel would dictate which heap locations are considered active.

Chapter 4

Compiling ComRaTT

This chapter covers the compilation of ComRaTT programs to WASM and the steps required to get there. Going from ComRaTT source code, as seen in Figure 3.1, to WASM bytecode requires careful planning for us to adhere to the semantics of the language while distilling the programs into WASM instructions.

The binaries resulting from compiling ComRaTT programs are meant to be executed in the context of a runtime. This section will make small references to that fact, but an explanation of the reactive runtime system is found in Chapter 5 Implementing the Reactive Semantics (p. 48).

The general WASM compilation process described in this chapter borrows many techniques and implementation details from our prior research project [18]. We have introduced new compilation techniques and methods, and switched the host language from OCaml to Rust.

We will cover the compiler's path from source code to a WASM binary, including descriptions of the methods and tools we use and the program transformation passes required to compile to WASM bytecode.

4.1 Methods & tools

Our initial research project [18] has provided a foundation for compiling a simple functional programming language to WASM. We will be basing the work in this chapter on the findings from that project. Most important are the program transformations we have made, which play a vital role in allowing us to compile ComRaTT to WASM.

Several CLI tools have been helpful in implementing and debugging the WASM code generation. Most of these come from *The WebAssembly Binary Toolkit* [30]. Noteworthy tools from this toolkit include `wasm-objdump` for dumping and inspecting a WASM binary, `wat2wasm`/`wasm2wat` for converting WAT to WASM and vice versa, `wasm-interp` for interpreting a WASM binary, and `wasm-validate` for validating a WASM binary.

4.2 Lexing and parsing

Whereas the ComRaTT compiler in our previous work [18] featured an LR(1) parser generated with OCaml's `menhir`, we now use the Rust crate `pest`[7] which uses parsing

[7]https://docs.rs/pest/latest/pest/, accessed 14/05/2025

expression grammars (PEG) [15] combined with Pratt parsing [27]. The main reason for using PEG is to ensure determinism and thus avoid any ambiguous parsing. PEG parsers do not allow backtracking, and the order that rules appear in alternatives is the order the rules are tried in. PEG parsing works well with Pratt parsing to allow succinctly defining operators and their precedence, which is the reason we use both techniques.

The choice of parsing library was influenced only by the perceived ease of use as well as the out-of-the-box error reporting ability. We briefly tried out the parser combinator library `winnow`[8] but found it difficult to generate clear error messages. The focus of this thesis lies entirely elsewhere, and considerations about the parser, parsing performance, and errors being reported did not weigh heavily in our decision making.

The grammar for the parser is found in Appendix A1 Grammar (p. 67). Note that there is no separate lexer specification given that the parser is based on PEG [15], which does not require a lexical analysis phase. The implementation of the parser is very similar to the syntax in Figure 3.1. The difference in the PEG implementation is that we ensure a certain shape for parts of the language that require operator precedence; these include expressions, clock expressions, and types. The `pest` library expects the shape of such rules to be

```
prefix* ~ primary ~ postfix* ~ (infix ~ prefix* ~ primary ~ postfix*)*
```

in order for the rule to be eligible for Pratt parsing. The Pratt parser receives parsed tokens and applies simple precedence tables for the different kinds of operators we support. That means we have one Pratt parser for expressions (with binary operators `+`, `-`, etc.), one for clock expressions (with binary operator `U`), and one for types (with operators `->`, `O`, and `Sig`, etc.).

4.3 Typing ComRaTT programs

Type inference in ComRaTT uses a bidirectional type checker with a constraint-based Hindley-Milner algorithm as described in Figure 3.2. The typing judgments translate directly into Rust pattern matching, which we demonstrate using the delay expression rule.

$$\frac{\Gamma, \checkmark_\theta \vdash_\vartriangle e \Rightarrow \tau \qquad \Gamma \vdash_\vartriangle \theta : \mathrm{Clock}}{\Gamma \vdash_\vartriangle \mathrm{delay}\ \{\theta\}\ e \Rightarrow \bigcirc\, \tau}\ \mathrm{Delay}^\dagger \uparrow$$

Figure 4.1: The delay type judgement.

The delay typing judgement seen in Figure 4.1, which is used to type the expression `delay {cl(_)} e`, can be implemented in our compiler via pattern matching. The resulting type of the expression is expected to be $\bigcirc\, \tau$ by extending the context with

[8]https://docs.rs/winnow/latest/winnow/, accessed 13/05/2025

clock `cl(_)` and inferring the type of `e` under that tick. In this case `cl(_)` is a symbolic clock expression, and *not* a concrete clock, meaning the clock itself is evaluated at runtime and is simply passed on to the typed expression when constructing the $\bigcirc\tau$ type. In Listing 4.1 the construction of $\bigcirc\tau$ with the clock `cl(_)` can be seen at the highlighted line 11.

```
1  Expr::Delay(e, clock) => {
2      // Introduce a tick given the clock
3      let context = context.promote_tick(&clock);
4      // Call recursively, propagate constraints
5      let (ty, type_output) = self.infer(context, *e);
6      (
7          Type::TLater(ty.clone().b(), clock.clone()),
8          TypeOutput::new(
9              type_output.constraints,
10             TypedExpr::TLam(Vec::new(), type_output.texp.b(),
11                 Type::TLater(ty.clone().b(), clock.clone()),
12                 Some(clock)),
13         ),
14     )
15 }
```

Listing 4.1: Rust translation of the delay judgement in Figure 4.1

The `promote_tick()` operation extends the typing context to reflect that we are now under a tick, allowing variables—bound before the tick—to be advanced with `advance` expressions. The resulting `TLater` type captures both the delayed value type and its clock dependency. The rest of the type judgements from Figure 3.2 are implemented in a similar fashion, with exception of the checking rules which will, in addition to inferring recursively, constrain the resulting type to be equivalent to some given expected type.

Type inference completes before compilation passes, ensuring all subsequent transformations operate on well-typed expressions. The Async RaTT core calculus type system [3, section 2] does not present inference of clocks, though the Async Rattus Haskell plugin [5], and the transpiled Async RaTT implementation by Berg and Jäpelt [6] both infer clocks. We do *not* infer clocks in ComRaTT.

4.4 From WAT to WASM binary format

Whereas we originally generated WAT strings in our research project [18], ComRaTT targets the WASM binary format directly. We utilize the Rust crate `wasm_encoder`[9] that provides an abstraction layer for encoding WASM. The `wasm_encoder` library does not require using a specialized WASM-focused abstract syntax or an SSA IR, it instead directly writes WASM instructions to byte buffers. Because mature tools for converting between WAT and WASM exist, there was very little reason for us to look elsewhere for alternatives with higher abstraction levels[10].

What we gain from `wasm_encoder` is mostly a simplified API to generate syntactically sound operations. Though, we do not have any kind of type safety guarantees in terms of the generated WASM output using the library.

```
(func
  i32.const 33
  i32.load offset=4 align=2
  i32.popcnt
)
```

Listing 4.1: Simple WAT function counting the number of one bits in a value loaded from the heap at index 37.

Instead of concatenating strings together to generate the WAT code in Listing 4.1, we can construct a function with the same operations using the API from the `wasm_encoder` library as seen in Listing 4.2.

```
let mut func = Function::new(vec![]);
func.instructions()
  .i32_const(33)
  .i32_load(MemArg {
    offset: FUNCTION_INDEX_OFFSET,
    align: WASM_ALIGNMENT_SIZE,
    memory_index: 0,
  })
  .i32_popcnt()
  .end();
```

Listing 4.2: Generating instructions for the function in Listing 4.1 using `wasm_encoder`

[9]https://docs.rs/wasm-encoder/latest/wasm_encoder/, accessed 14/05/2025

[10]Alternatives include `binaryen`, `walrus`, and `waffle`, `llvm`, all accessed 15/05/2025

This abstraction allowed us to move fast and not worry about well-formedness of our individual instructions and constantly looking up WAT syntax. Having editor suggestions for WASM operations and typed arguments for each operator saved us a lot of time.

4.5 Passes and transformations

Before generating code, the typed ComRaTT abstract syntax tree undergoes several transformation steps. These transformations are also referred to as “compilation passes”. Besides the new *Consecutive lambda elimination* and *ANF conversion*, all of them also exist in Holmgaard and Sattar Atta [18].

Figure 4.2 illustrates the pipeline of passes and transformation steps including code generation.

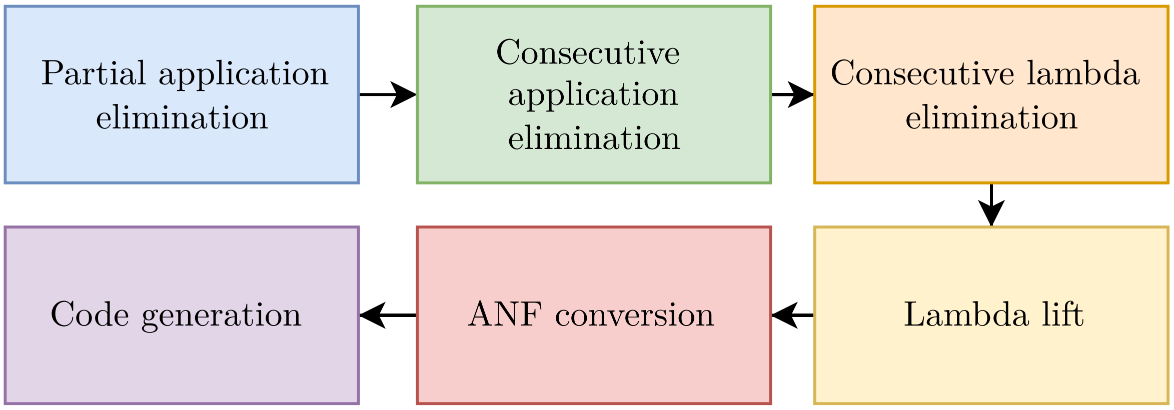

Partial application elimination
Consecutive application elimination
Consecutive lambda elimination
Lambda lift
ANF conversion
Code generation

Figure 4.2: Visualization of the compilation pipeline after parsing and typechecking.

4.5.1 Partial application elimination / η-expansion

The η-expansion pass serves the purpose of bridging the gap between functional programming and WASM by transforming partial function application to function application without leftover arguments.

As an example, this pass transforms Listing 4.3 into Listing 4.4.

```
main : int
def main =
  let add = fun x -> fun y -> x+y in
  let add2 = add 2 in add2 40;
```

Listing 4.3: ComRaTT code example with a partial application

```
main : int
def main =
  let add = fun x -> fun y -> x+y in
  let add2 = fun #part_elim_lam1 -> add 2 #part_elim_lam1 in add2 40;
```

Listing 4.4: ComRaTT code example in Listing 4.3 with partial application eliminated

The code in Listing 4.3 partially applies the `add` function in the `let add2 = add 2 in ..` binding. Through η-expansion, we explicitly declare a function wrapping `add 2` such that the second argument is explicit, as seen in Listing 4.4.

4.5.2 Consecutive application elimination

ComRaTT supports functions applied to multiple arguments at once. This pass collapses application expressions where the function being applied is also an application, i.e., the form `((f x) y) z`, to accommodate the non-functional nature of WASM. The code in Listing 4.5 transforms into the code in Listing 4.6 with this pass.

```
main : int
def main =
  let add_three = fun x y z -> x+y+z in
  ((add_three 37) 3) 2;
```

Listing 4.5: ComRaTT code example with consecutive applications

```
main : int
def main =
  let add_three = fun x y z -> x+y+z in
  add_three 37 3 2;
```

Listing 4.6: ComRaTT code example in Listing 4.5 with consecutive applications eliminated

Instead of partial application (which could lead to η-expansion and closure allocation), we consolidate the applications such that `add_three` is applied directly to the multiple arguments `37 3 2` at once. This gives us a single function call to `add_three`.

4.5.3 Consecutive lambda elimination

This pass merges consecutive lambdas, so that e.g. `fun x -> fun y -> x + y` becomes `fun x y -> x + y` in cases where a closure is not required and no partial application takes place in the nested lambdas. We do this transformation to reduce the amount of lifted lambdas, and thus also the amount of closures in a program.

4.5.4 Lambda lifting

As outlined in Section 2.2.7 Function section (p. 13), the WASM specification declares functions within a module as top-level functions. These functions are not able to nest, meaning any nested functions in ComRaTT must be moved to the top. This can be done by a technique called lambda lifting. Jones et al. [23] describes the process of lambda lifting as a way to produce 'supercombinators', which have the exact same properties as top-level functions, namely that they contain no free variables, they contain no nested lambdas, and they can have zero or more arguments.

Lambda lifting will recursively lift all nested lambdas from a function definition, leaving behind only supercombinators that are easily compiled to WASM.

To lift all lambdas, we start by going through every top-level definition, traversing through definition bodies, and lifting any nested lambdas to the top level. When a function definition (`fun .. -> ..`) is encountered in a top-level definition, a handful of steps are taken to lift the lambda. First, the free variables of the function are collected and combined with the existing function arguments. The function is then removed and declared as a top-level definition, leaving behind a call-site in its place. This call-site will be partially applied, fixing all variables that were free variables under the function scope.

To see this in action, we will consider a ComRaTT function definition `add5`

```
add5 : int -> int
def add5 =
  let value = 5 in
  fun x -> x + value;
```

We lambda lift the function found at the return position by first identifying its free variables, which consist only of the `value` variable in `fun x -> x + value`.

```
#lambda_1 : int -> int -> int
def #lambda_1 value x = x + value;

add5 : int -> int
def add5 =
  let value = 5 in
  #lambda_1 value;
```

After lifting to `#lambda_1`, the function is replaced by a call to the lifted lambda, `#lambda_1 value`, a partially applied function that is semantically identical to the original lambda. Notice that the explicit `value` argument in `#lambda_1` affects the type of the `#lambda_1` top-level definition accordingly by adding `int -> ...` to the type.

The program left after the transformation is more appropriate for compiling WASM code from, as we do not have to consider any high-level language constructs in a definition body that do not have equivalent low-level WASM constructs.

4.5.5 A-normalization

The final transformation before generating WASM code is converting the AST to A-normal form (ANF). In the research project [18], we considered A-normalization to help bridge the gap between the functional nature of Async RaTT and the sequential nature of WASM, but did not implement it.

When prototyping A-normalization in ComRaTT, we very quickly found that it was a useful intermediate representation for our compiler. With ANF, all intermediary computations are named and linearized. Such direct style forces the order of intermediary computation to match that of imperative machine code/bytecode Flanagan et al. [13]. To show what ANF looks like, we can consider the expression `f 1 2`, which will be transformed into a form where all intermediary partial applications are explicitly bound in what could be considered a natural order for most target machines:

```
let x0 = f 1 in
let x1 = x0 2 in
x1
```

Here the intermediary `f 1` application is bound to `x0`, which is a partially applied function used in `x1` for the full function application. This order of bindings matches the order we would like to generate code in, as it is linear and all required intermediary calculations are bound to an explicit variable before we need to use them.

There is no canonical ANF definition, although the literature Flanagan et al. [13] does provide a starting point for A-normalization. ComRaTT generates WASM bytecode directly from its ANF representation.

In ComRaTT ANF, expressions are delineated into two categories. We have either “atomic expressions” or “complex expressions”. In our ANF representation, atomic expressions consist of constants, binding names, lambda abstractions, and `wait` expressions. Complex expressions compose atomic expressions to form if expressions, function application, primary operations, tuple introduction, and tuple elimination. Let bindings in ANF can bind either category of expressions.

$$\begin{array}{rrcll}
\textsf{AnfExpr} & e & ::= & \text{AExpr} & \\
 & & \mid & \text{CExpr} & \\
 & & \mid & \text{let } x = e \text{ in } e' & \textit{let binding} \\
\textsf{AExpr} & a & ::= & v & \textit{constant} \\
 & & \mid & \lambda x.e & \textit{closure} \\
 & & \mid & \lambda^{\bigcirc} x.e & \textit{later closure} \\
 & & \mid & \text{wait var} & \textit{wait expression} \\
\textsf{CExpr} & c & ::= & a \oplus a' & \textit{binary operation} \\
 & & \mid & a\ a' & \textit{function application} \\
 & & \mid & (a(,a')^{*}) & \textit{tuple introduction} \\
 & & \mid & a\ .\ \mathbb{Z} & \textit{tuple elimination} \\
 & & \mid & \text{if } a \text{ then } e_1 \text{ else } e_2 & \textit{conditional} \\
\text{Value} & v & ::= & \mathbb{Z} \mid x \mid \text{true} \mid \text{false} \mid () & \\
 & \oplus & ::= & + \mid * \mid / \mid - \mid = \mid < & \\
 & & \mid & \leq \mid > \mid \geq \mid <> &
\end{array}$$

Figure 4.3: Syntax of the A-normalized ComRaTT source language.

ComRaTT's ANF syntax is seen in Figure 4.3, where the two expression categories are atomic expressions `AExpr`, and complex expressions `CExpr`, composed together by `let` expressions through the `AnfExpr` construct.

In ComRaTT ANF, there are two ways to construct lambdas (or technically closures at this point). Normal lambdas (λ) represent the kind of lambdas used for higher-order functions, and later lambdas ($\lambda^{\bigcirc}$) provide more explicit code paths in code generation for how delayed computations are handled. The ANF conversion pass picks out partially applied lifted lambdas (as the lifted lambdas are applied to any free variables after lifting) and converts them to the ANF closure representations. The ComRaTT ANF representation is mostly similar to the syntax in Figure 3.1 for the rest of the language constructs.

4.6 Structure of a ComRaTT WASM binary

Our research project [18] presented ComRaTT with 64-bit integers, 32-bit booleans and unit values and no pointers. There was three memory regions, one for stable types as well as two separate “now” and “later” heaps.

We will now be presenting ComRaTT with changes made to the representations and module structure. ComRaTT binaries follow the ordering of section within a module as seen in Figure 2.1.

4.6.1 Value representation

ComRaTT features two primitive types, boolean and integers. All values and pointers are represented by 32-bit integers to simplify compilation, as dealing with uniformly-sized heap allocations is more or less trivial compared to heterogenous sizes. In practice, this means using the WASM value type `i32`, which aligns well with memory regions being 32-bit in WASM (without a 64-bit memory proposal enabled).

4.6.2 Memory regions

ComRaTT features two memory regions. This decision was a step towards introducing a garbage collector (GC) into the runtime system. Because of implementation complexity, a GC for ComRaTT has not been implemented, see Section 6.7 Garbage collection (p. 55) for our GC considerations and plans.

One memory region, referred to as the *shared heap*, is for storing heap allocated values (tuples) and closures produced by delayed computations and higher-order functions. A closure is represented in WASM memory by $2 \cdot 4 + n \cdot 4$ bytes. The first $2 \cdot 4$ bytes reserve two `i32` fields, one for a function table index (function pointer) and one for the number of arguments left to be applied in the closure. The remaining $n \cdot 4$ bytes contain parameters required to call the underlying function, again as all ComRaTT values are represented by 32-bit integers, we can use a uniform 4-byte length for every parameter. These last $n \cdot 4$ bytes can be considered the closure environment.

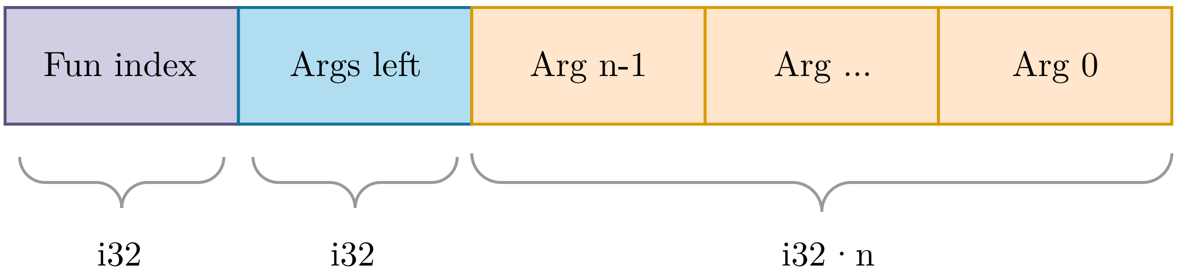


Figure 4.4: Illustration of how closures are represented in memory.

In Figure 4.4 an illustration of the described layout is depicted. The first box represents the function pointer, second box is the number of arguments remaining in the environment before the closure is fully applied and callable, followed by n arguments. It's important to point out here that the arguments are *in reverse order*. This is a simple decision that allows us to use the "args left" value as the offset into the n arguments when populating arguments. If, for example, we have a closure with an environment size $n = 2$, and 1 argument left to be applied, we can calculate where we should put the last argument before executing the closure, saving us a i32 field containing the size of the closure.

The other memory region, referred to as the *location heap*, handles fixed size data by storing pairs of 32-bit values: a pointer to a closure in the *shared heap* and a clock. Each input channel in ComRaTT is represented as an `i32` that is a power of two, so for κ_i, $1 \leq i \leq 32$, a channel will have a distinct value 2^{i-1}. A clock is represented by packing bits in an `i32` using logical OR on all the input channels in the clock. This way ComRaTT supports up to 32 individual input channels. Checking whether a delayed computation depends on a specific input channel is handled by logical AND. Thus if an input channel κ_2 is active, all delayed computations with clocks $\theta \wedge \kappa_2 > 1$ will be eligible for resumption. This would result in the following:

$$\begin{aligned} 0110 &\quad \theta \\ \wedge\, 0010 &\quad \kappa_2 \\ 0010 &\quad = 2 \end{aligned}$$

which marks the delayed computation θ as resumable.

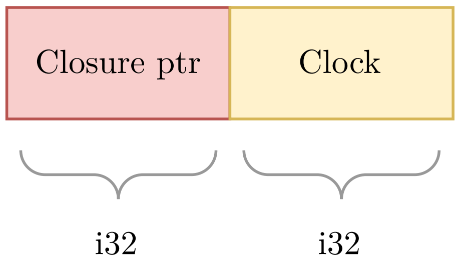


Figure 4.5: Illustration of how locations are represented in memory.

Figure 4.5 shows the layout of the location heap. Every entry is uniformly sized, with a repeating pattern of closure pointer followed by clock.

To relate the two heaps together, Figure 4.6 shows the interconnectedness between them. The heap state is based on calling `kb` from the code in Listing 3.2. We see two closures in the shared heap: one wrapping function 6 with no arguments and one wrapping function 7 with a single argument: 0. The 0 (last) argument of the second closure in the shared heap is a pointer to location 0—hence the arrow pointing back to the location heap. This represents the recursive call to `kb`.

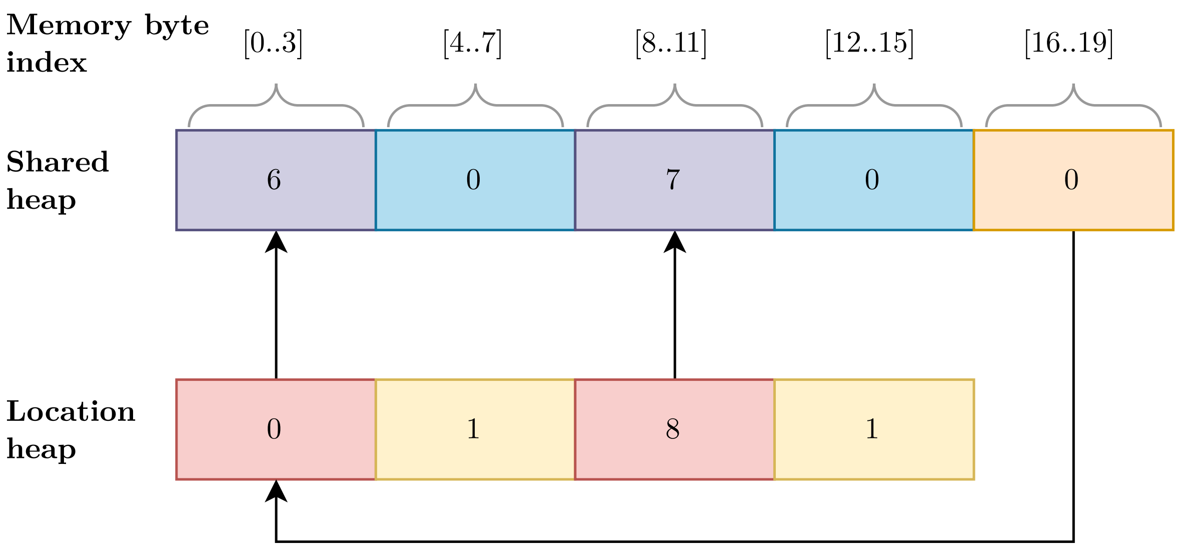


Figure 4.6: Illustration of how the heaps are connected.

Both heaps are 32-bit as this is the WASM default[11]. We saw no reason to change this given that we have no use for +4GB of memory or wider pointers for this proof of concept. Most browsers and runtimes do support 64-bit memory though[12].

4.6.3 Statically pre-generated code

All ComRaTT WASM binaries contain four statically generated functions, which are briefly described below.

`malloc` allocates space in the shared heap by incrementing a global variable containing the next free index in the shared heap and returning the old value. Since closures and general values can have varying sizes, this function is parameterized over an `i32` representing the size of the allocation.

`location_malloc` does the same for the location heap but does not take take an argument since locations are fixed in size.

`clock_of` looks up the clock of a location at runtime by taking a pointer to it and returning the clock part.

`location_dispatch` takes a location pointer and calls the program-dependant pre-generated `dispatch` function with the closure pointer extracted. See Section 4.6.5 Program-dependent pre-generated code (p. 41) for details on `dispatch`.

Furthermore all top level ComRaTT functions have an extra local variable `dupi32` added during compilation. This variable is used to store the pointer from calling `malloc`,

[11]https://webassembly.org/docs/portability/, accessed 16/05/2025
[12]https://webassembly.org/features/, accessed 16/05/2025

since it is needed multiple times when writing to memory and WASM has no instruction for duplication, meaning that the pointer will be gone after the first write to memory.

See Section 4.8 Compiled example program (p. 46) for a detailed WASM example that also shows these four functions.

4.6.4 Foreign function interface (FFI) functions

Each binary also imports two FFI functions which are to be supplied by the host environment and used for interacting with it: `wait` (equivalent to the `wait_rt` construct from Figure 3.3) and `set_output_to_location`. See Chapter 5 Implementing the Reactive Semantics (p. 48) for more information on the details of these functions.

4.6.5 Program-dependent pre-generated code

The binaries always contain the two functions: `dispatch` and `init`, the contents of which is generated at compile time based on the concrete ComRaTT program.

A WASM `call` instruction takes a function index as an immediate value encoded in the instruction itself. To achieve the function pointer-like behavior needed in ComRaTT closures, the `call_indirect` instruction proves useful, as it can take a numerical function table index as an argument on the stack instead of as an immediate argument. `call_indirect` still requires the type index of the function to be called as an immediate argument though, meaning that there is no way to look the type index up at WASM runtime.

`dispatch` solves this problem by providing a compile-time generated switch statement that has cases for all relevant functions in the module. `dispatch` takes a pointer to a closure, unpacks the function pointer part and then switches on that. Each case puts the relevant arguments of the function to be called on the stack and executes a `call_indirect` with the correct type index before returning.

`init` is the entry point from the perspective of the runtime, see Chapter 5 Implementing the Reactive Semantics (p. 48) for more details on the implementation of that. Delayed computations in ComRaTT are closures (locations) that wrap top level functions. If a top level function e.g. contains a `delay` it will allocate a closure and a location. The return value of calling the function will be a pointer to the location, to be used by the runtime. The body of `init` is generated to contain a sequence of calls to top level functions that return location pointers followed by a call to the FFI function `set_output_to_location` that pass these pointers to the runtime. See Chapter 5 Implementing the Reactive Semantics (p. 48) for a more detailed description of this procedure. `init` is generated based on the functions used in output channels. The state shown previously in Figure 4.6 is based on calling `init`.

Section 4.8 Compiled example program (p. 46) goes in depth with a compiled example and thus also shows examples of `dispatch` and `init`.

4.6.6 Debug info with custom sections

ComRaTT uses WASM custom sections to attach debug information to symbols, meaning that e.g. names are preserved when converting WASM to WAT. This has been a useful aid during development.

Listing 4.7 shows the WAT-translated partial output of a given program and Listing 4.8 shows the same output where custom sections have not been used to attach debug information. The snippets show type declarations but the concept applies to functions, variables etc.

```
(module
  (type $wait (func (param i32) (result i32)))
  (type $set_output_to_location (func (param i32 i32) (result i32)))
  (type $malloc (func (param i32) (result i32)))
  (type $location_malloc (func (result i32)))
  (type $clock_of (func (param i32) (result i32)))
  (type $add (func (param i32 i32) (result i32)))
  (type $main (func (result i32)))
  (type $dispatch (func (param i32) (result i32)))
  (type $location_dispatch (func (param i32) (result i32)))
  (type $init (func))
  ...
)
```

Listing 4.7: Partial WAT-translated output showing type declarations with debug information

```
(module
  (type (;0;) (func (param i32) (result i32)))
  (type (;1;) (func (param i32 i32) (result i32)))
  (type (;2;) (func (param i32) (result i32)))
  (type (;3;) (func (result i32)))
  (type (;4;) (func (param i32) (result i32)))
  (type (;5;) (func (param i32 i32) (result i32)))
  (type (;6;) (func (result i32)))
  (type (;7;) (func (param i32) (result i32)))
  (type (;8;) (func (param i32) (result i32)))
  (type (;9;) (func))
  ...
)
```

Listing 4.8: Partial WAT-translated output showing type declarations without debug information

4.7 Code generation

Apart from generating WASM bytecode instead of a WAT string, code generation is not fundamentally different from Holmgaard and Sattar Atta [18]. Appendix A2 Compilation schemes (p. 71) shows compilation schemes for the entire ANF syntax whereas this section will focus on compilation of delayed computations.

Like in Holmgaard and Sattar Atta [18] we use the syntax seen in Compilation Scheme (4.1, `example`) meaning that compiling the AExpr e results in `i32.const 42`. A `mono-spaced` font is used for WASM instructions whereas a regular font is used for necessary pseudocode and recursive compilation.

$$\text{AExpr}[[e]] = \texttt{i32.const 42} \qquad (4.1,\ \texttt{example})$$

In the following sections we use natural language names for the WASM instructions. Immediate arguments are provided to instructions like `call FUNCTION_INDEX` and are also `mono-spaced`.

All load and store instructions take a `memarg` consisting of an offset and an alignment. The alignment will always be 2 and is thus omitted for the sake of brevity. The offset is the offset from the base pointer provided to the load/store instruction i.e.

```
i32.const 4
i32.load offset=4
```

loads from index 8 in memory. Load and store also take an immediate argument representing the index of the memory section to be used e.g. `i32.store 0` and `i32.load 1`. The shared heap has index 0 and the location heap has index 1.

Calls to `malloc` will be presented as `malloc(size)` and represents the following sequence of instructions

```
i32.const size
call MALLOC_FUNCTION_INDEX
```

Similarly, calls to `gen_clock_of(clock)` is shorthand for a sequence of instructions that is generated based on the concrete `ClockExprs` given. In summary, the code generated is a sequence of `i32.const` and `i32.or` instructions that will leave the correctly represented clock on the stack.

Throughout the schemes we will also use various names like `WORDSIZE` and `INDEX_OF_TOPLEVEL`. These represent constants, pseudocode or some value that is available at the given point in the compilation that is also not relevant to fletch out the origin of.

4.7.1 Compiling delayed computations (`AExpr::LaterClosure`)

Compilation Scheme (4.2, `delayed computation`) below concerns delayed computations and require a detailed explanation, which will follow below.

Several invariants are assumed when generating code from a delayed computation. These are

- The expression is a `LaterClosure` ($\lambda^{\bigcirc}$) which should have an application immediately inside the body
- The function being applied should be a top level function
- The arguments given in the application are the arguments to populate the closure with
- The function is being fully applied

The code in Compilation Scheme (4.2, `delayed computation`) starts by allocating a closure large enough for the arguments provided in the application. The pointer for the new allocation is stored in `dupi32` and put on the stack immediately again with `local.tee`. Then the new allocation is populated with the index of the top level function, 0 arity (since it is being fully applied) and the arguments (the loop). A new location is allocated and populated with the closure pointer and the WASM representation of the clock. Finally the pointer to the location is returned.

When allocating locations, like when allocating closures, we also need the pointer on the stack several times. We cannot use the `dupi32` variable as it already holds the pointer for the closure at this point. Instead we can take advantage of the fact that locations are fixed in size and thus we can subtract from the address of the next free location to get the current locations address. See the last 8 lines of Compilation Scheme (4.2, `delayed computation`) for an example of this, where subtracting 8 and 4 from the address of the next free location produce the function pointer and clock parts of the current location, respectively.

```
if function = Expr::App(AExpr::Var(app_name, _var_ty), app_args, _app_ty)
        i.e. an application of a toplevel function name then
        AExpr[[LaterClosure(function, clock, lambda_type)]]
                              =
      malloc((2+arity)*WORDSIZE)
      local.tee index_of_dupi32_variable
      i32.const INDEX_OF_TOPLEVEL
      i32.store 0 offset=0
      local.get index_of_dupi32_variable
      i32.const 0
      i32.store 0 offset=4
      local.get index_of_dupi32_variable  \
      AExpr[[arg]]                         > for arg in app_args
      i32.store 0 offset=2+INDEX(arg)     /
      call LOCATION_MALLOC_INDEX                    (4.2, delayed computation)
      local.get index_of_dupi32_variable
      i32.store 1 offset=0
      global.get 1
      i32.const 4
      i32.sub
      gen_clock_of(clock)
      i32.store 1 offset=0
      global.get 1
      i32.const 8
      i32.sub
```

4.8 Compiled example program

To make the compiled output legible, it is run through `wasm2wat`[13] and displayed as WAT.

Considering the example program in Listing 3.2 again. It defines a single input channel `keyboard`, a function `kb` and finally an output to the channel `print` that depend on the delayed computation `kb`.

`kb` defines an infinite stream of keyboard inputs, represented as an asynchronously delayed signal of integers.

Listing 4.9 shows the code generated for the `kb` function converted to WAT. It omits everything else as we want to focus on the handling of delayed computations. The entirety of the code is viewable in Appendix A3 *sigrec* WAT code (p. 77).

The code contains three functions: `kb`, `#lambda_1` and `#lambda_2`.

`kb` contains delayed computations and thus its body is concerned with allocating in the shared and location heaps.

`#lambda_1` and `#lambda_2` are the lambda lifted closures for the `wait` and `delay` expressions respectively. These are the top level functions being pointed to by the allocations made in `kb`.

The former pushes the 32-bit integer constant 1 representing the first (and in this case only) input channel after which the FFI function `wait` is called. At runtime this will produce the actual value from the `keyboard` input channel.

The latter is the closure representing the delayed signal and thus more complex. It takes a single argument `key` which is a pointer to the location representing `#lambda_1`. This pointer is pushed on the stack and dispatched via `location_dispatch` resulting in the keycode from the `keyboard` input channel being left on the stack, after which it is stored in `tmp0`. Then a tuple (signal) is allocated by pushing 8 and calling `malloc`. The tuple pointer returned is stored in `dupi32` and put on the stack again. The keycode value is fetched from `tmp0` and stored in the first part of the tuple allocation, representing the head of the signal. The pointer to the tuple is pushed on the stack again, after which `kb` is called leaving a new pointer to a location representing `#lambda_2` itself on the stack, which is stored in the second part of the tuple. This second part represents the recursive tail of the signal. Finally the pointer to the tuple is returned, to be used in `init` and by the runtime.

[13] `wasm2wat --enable-multiple-memories --enable-tail-call`

```
(module
  (func $kb (type $kb) (result i32)
    (local $key i32) (local $dupi32 i32)
    i32.const 8
    call $malloc
    local.tee $dupi32
    i32.const 6
    i32.store
    local.get $dupi32
    i32.const 0
    i32.store offset=4
    local.get $dupi32
    drop
    call $location_malloc
    local.get $dupi32
    i32.store 1
    global.get 1
    i32.const 4
    i32.sub
    i32.const 1
    i32.store 1
    global.get 1
    i32.const 8
    i32.sub
    local.set $key
    i32.const 12
    call $malloc
    local.tee $dupi32
    i32.const 7
    i32.store
    local.get $dupi32
    i32.const 0
    i32.store offset=4
    local.get $dupi32
    local.get $key
    i32.store offset=8
    local.get $dupi32
    drop
    call $location_malloc
    local.get $dupi32
    i32.store 1
    global.get 1
    i32.const 4
    i32.sub
    local.get $key
    i32.const 4
    call_indirect (type $clock_of)
    i32.store 1
    global.get 1
    i32.const 8
    i32.sub)

  (func $#lambda_1 (type $#lambda_1) (result i32)
    (local $dupi32 i32)
    i32.const 1
    call $wait)

  (func $#lambda_2 (type $#lambda_2) (param $key i32) (result i32)
    (local $tmp_0 i32) (local $dupi32 i32)
    local.get $key
    i32.const 9
    call_indirect (type $location_dispatch)
    local.set $tmp_0
    i32.const 8
    call $malloc
    local.set $dupi32
    local.get $dupi32
    local.get $tmp_0
    i32.store
    local.get $dupi32
    call $kb
    i32.store offset=4
    local.get $dupi32)
  )
```

Listing 4.9: WASM output from compiling Listing 3.2, converted to WAT via `wasm2wat`

Chapter 5

Implementing the Reactive Semantics

In this chapter, we go over the implementation of ComRaTT's reactive semantics, also called the runtime. We start by covering the methods and tools used for the implementation before describing how state is represented and FFI functions are implemented. Then we discuss how the implementation is mapped to the transitions of the reactive semantics in Figure 3.4, describe implemented input and output channels, and finally present the execution of an example program.

5.1 Methods & tools

The implementation of the runtime uses the Rust crate `wasmtime`[14] to programmatically create and manipulate a host environment in which we embed ComRaTT WASM modules.

5.2 Implementing the runtime

The runtime is activated by the `--run` flag to the ComRaTT binary. This flag instantiates the runtime by providing the bytes representing the generated WASM module as well as collections of input and output channel names. Instantiation of the runtime also covers configuring the host environment and instantiating the WASM module.

5.2.1 Representation of internal state

The runtime keeps internal state to aid execution. Specifically, it maps indices for output channels[15] to a list of location pointers, representing all the delayed computations that output to the channel.

Furthermore, a mapping between input channels and the newest value received is kept. This implementation detail technically buffers the values despite ComRaTT only being concerned with push-only input channels—see Section 6.2 Async RaTT features (p. 53) for more details on this.

5.2.2 FFI functions

As described in Section 4.6.4 Foreign function interface (FFI) functions (p. 41), ComRaTT WASM modules import two functions from the host environment: `wait` and `set_output_to_location`.

[14] https://docs.rs/wasmtime/latest/wasmtime/, accessed 19/05/2025

[15] Output channels are named in ComRaTT source programs but referred to by index in WASM.

`wait` takes the index of an input channel and looks up the value received on the given channel in the internal state of the runtime.

`set_output_to_location` takes the index of an output channel and a location pointer. It is implemented by looking up the list of pointers associated with the output channel index and inserting the location pointer.

5.2.3 INIT transition

The execution of the reactive loop in the runtime starts by extracting the generated `init` function from the WASM module and calling it to populate the internal runtime state with pointers to the delayed computations depended on by output channels—corresponding to the INIT transition of the reactive semantics.

5.2.4 INPUT transition

The reactive loop uses a multi-producer single consumer (MPSC) channel to listen for events from input channel implementations. The loop uses asynchronous language features of Rust to achieve concurrency[16]. Such an event is associated with a specific input channel κ, and the runtime proceeds by updating the stored value v of κ and then looking up all output channels depending on delayed computations with clocks containing the given channel $\kappa \in \text{cl}(\ell)$, equivalent to Heap^{κ}.

5.2.5 OUTPUT-SKIP and OUTPUT-COMPUTE transitions

All delayed computations relevant for the current input are executed sequentially via FFI calls to `location_dispatch`. The ordering of these executions is decided by the order that they appear in the program, grouped by a static output order defined by the runtime.

Since output channels are restricted to depend on delayed signals, execution of locations results in the allocation of signals (in the form of tuples) where the first element is the value produced for the output channel and the second is the tail of the signal, represented by a pointer to a new location. The returned value is a pointer to the tuple representing the signal.

The runtime accesses the WASM memory regions and uses the returned pointer to index and pick out the signal head to be output and the signal tail for updating the location that the output depends on, before looping again.

These actions combined represent the transition OUTPUT-COMPUTE. The OUTPUT-SKIP transition is also implemented implicitly by virtue of the fact that only locations relevant for the current input channel κ are executed.

[16]Technically, the third-party library `tokio` is involved because Rust does not provide a runtime for asynchronous programming.

5.2.6 OUTPUT-END

ComRaTT does not implement garbage collection and thus does not implement the OUTPUT-END transition. The heaps will exhaust system memory (or the maximum 4GB supported by default) as no maximum size is set for the WASM memory regions. See Section 6.7 Garbage collection (p. 55) for a discussion of ideas for implementing garbage collection.

5.2.7 Supported input and output channels

ComRaTT supports a single `keyboard` input channel as well as two output channels `print` and `print_ascii`. See descriptions of each below.

5.2.7.1 `keyboard` input

Captures a single character from standard input, represented by its ASCII key code.

5.2.7.2 `print` output channel

Prints the raw 32-bit integer to standard output.

5.2.7.3 `print_ascii` output channel

Prints the ASCII representation of a 32-bit integer to standard output.

5.3 Demo program

This section demonstrates running the program in Listing 3.2, which is also found at the source folder path `examples/sigrec-report.cml`.

Run[17] the example with

```
./comratt --run examples/sigrec-report.cml
```

Initially the program being run as well as the representation after each pass will be shown. See this output in Appendix A4 Program representation output (p. 83).

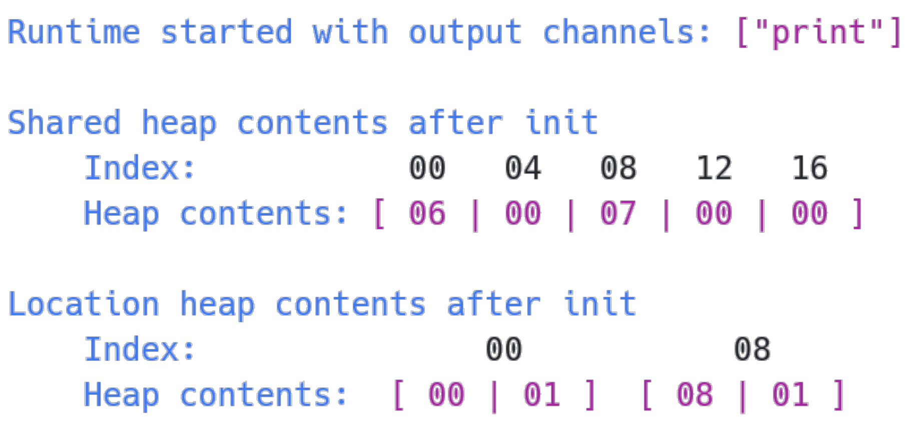
After that, a list of output channels used by the program as well as the contents of each heap is shown—see Figure 5.1. Note that the shared heap is represented with each individual memory cell while the indices for the location heap are for each `[ pointer | clock]` pair. To simplify the representation, elements in each heap are shown as 32-bit integers instead of the $4 \cdot 8$-bit cells they are made of. Likewise, the indices are scaled by 4 and 8 accordingly so that pointers make sense.

The shared heap consists of two allocations: one at indices 00 and 04 representing a closure for the function with index 6 with no arguments remaining to be applied and no arguments populated, and one at indices 08, 12, 16 representing a closure for function

[17]Assuming the existence of a `comratt` binary. Alternatively `cargo run -- --run examples/sigrec-report.cml` with a Rust toolchain

7 with 0 arguments still to be applied and a single argument populated: the (location) pointer 0.

In the location heap, we also see two allocations: the location at index 00 points to the closure at index 00 in the shared heap and has the clock represented by 1, while the location at index 08 points to the closure at index 8 and also has the clock represented by 1.



```
Runtime started with output channels: ["print"]

Shared heap contents after init
    Index:          00   04   08   12   16
    Heap contents: [ 06 | 00 | 07 | 00 | 00 ]

Location heap contents after init
    Index:               00           08
    Heap contents:  [ 00 | 01 ]  [ 08 | 01 ]
```

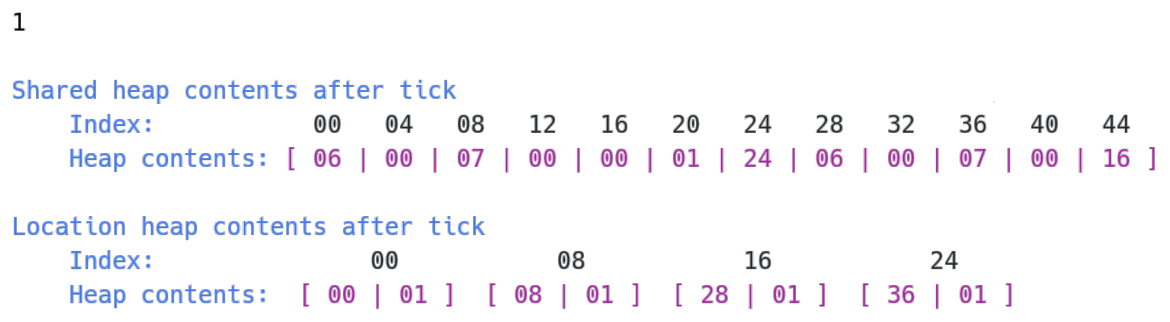
Figure 5.1: A list of output channels and the contents of each heap shown after initialization of the runtime

Pressing a button on the keyboard results in a tick (for this example, we use `CTRL+a` to produce the ASCII code 1). This outputs the key code and the heap contents, shown in Figure 5.2.

The shared heap has been expanded with indices 20 to 44. Indices 20 and 24 are the head and tail of the signal (tuple) produced by the tick where index 20 has the ASCII code 1 and index 24 has the pointer to the location starting at index 24. Indices 28 to 44 represent a repetition of the pattern initially seen in Figure 5.1 representing the recursive call to `kb`, where the only difference is the argument to the second closure now being the pointer 16.

Two new locations have also been allocated, the first pointing to the closure at index 28 and the second pointing to the closure at index 36 while both still have clock 1. Again a repetition of the already observed pattern, with updated pointer addresses.

Producing more ticks will repeat the patterns already observed.



```
1

Shared heap contents after tick
    Index:           00   04   08   12   16   20   24   28   32   36   40   44
    Heap contents: [ 06 | 00 | 07 | 00 | 00 | 01 | 24 | 06 | 00 | 07 | 00 | 16 ]

Location heap contents after tick
    Index:               00            08            16            24
    Heap contents:  [ 00 | 01 ]  [ 08 | 01 ]  [ 28 | 01 ]  [ 36 | 01 ]
```

Figure 5.2: Output from the runtime after a tick produced by pressing `CTRL+a`, it shows the ASCII code and heap contents

Chapter 6

Limitations and future work

While ComRaTT modestly implements the core ideas of Async RaTT, it is by no means a feature-complete language. This chapter addresses the functional limitations of ComRaTT and in some cases concrete directions for future work.

6.1 General functional programming features

Although ComRaTT is a functional programming language, it lacks some of the features one expects from a well-founded functional language. There is no type polymorphism[18], nor pattern matching, unions, records, boolean operators, or a rich library of built-in data types like strings or chars. Section 7.1 Evolution of project focus (p. 58) goes into detail about the reasons for the lack of these features.

6.2 Async RaTT features

ComRaTT implements essential parts of Async RaTT like signals, the stable modality, the later modality, clock tracking, allocating delayed computations, as well as a runtime needed for execution, but it does not implement the `select` construct. This construct is needed for synchronizing two delayed values and combining clocks ($\sqcup$ operator), which are important for dynamic dataflow graphs.

Furthermore, input channels in ComRaTT are limited to being *push-only* and thus not *buffered-push* or *buffered-only*. For that reason, we also do not implement `read` for reading the buffered value of a channel. From an implementation standpoint, implementing buffered and buffered-push inputs would not impose many challenges, though it would have taken time away from other parts of the implementation, so this limitation arises purely from constraining the scope.

6.3 Narrow slice

The primary goal of ComRaTT is to validate the feasibility of generating low-level bytecode from high-level asynchronous modal FRP. While working towards that goal, we reduced the generality of the compilation pipeline. As a result, certain sections of the compiler expects very specific shapes of expressions, particularly for higher-order functions and delayed computations.

[18]Technically `never` is implemented to be polymorphic, but the type system is generally monomorphic since we do not provide a way to give type variables (or omit type signatures for functions) in the syntax.

We do think it is possible to regain generality in the compiler if more time and effort is put into untangling the ComRaTT compilation pipeline.

Furthermore, output channels are restricted to expressions of type `O Sig A`, which is limited compared to the syntax presented by Bahr and Møgelberg [3]. Allowing for `Sig A` would not require substantial changes to the compiler.

6.4 WASM bugs

Some valid ComRaTT programs produce invalid WASM code. A subset of these issues stem from the fact that our pipeline lost generalization while we attempted to properly handle delayed computations, as described in Section 6.3 Narrow slice (p. 53).

Examples of this are the programs presented in Section 4.5 Passes and transformations (p. 33). Compiling the example from Listing 4.3 will produce WASM code that attempts to declare the closure represented by `add`, by issuing a call instruction to a top level function without setting up the stack correctly with arguments first.

6.5 Representation limitations

Because of how `dispatch` is generated, functions in ComRaTT are limited to always returning precisely one `i32`. Given that every ComRaTT value is represented by `i32`, the type does not present an issue. Instead, it is the missing possibility of dispatching functions with no or multiple return values.

An example is the FFI function `set_output_to_location`, which has a return value even though it is never used. A method for circumventing this could be to maintain state in the compilation pipeline that knows about the types of functions and thus could insert the correct type index in call-sites. There could, however, still be problems with tracking types in relation to partial application.

Adding to that, representing numbers with `i32` and using 32-bit memory regions implies obvious restrictions on the implementation. Numbers are limited in their size and programs are limited to using at most 4GB of memory. Changing this could be done in different ways and should not be challenging. One way is to make memory regions 64-bit and represent all pointers with 64-bit integers. ComRaTT could be extended to allow users to explicitly choose between 32-bit and 64-bit integers when working with whole numbers or everything could be represented by 64-bit integers, depending on the goals of the language.

6.6 Example limited program

Listing 6.1 shows a ComRaTT adaptation of the `scan` function presented by Bahr and Møgelberg [3, section 3.1]. There are notable differences because ComRaTT does not have polymorphism nor a way to constrain types to be stable.

```
scan : □ (int -> int -> int) -> int -> Sig int -> Sig int
def scan f acc siga =
    let a = siga.0 in
    let as = siga.1 in
    let acc_prim = (unbox f) a acc in
    acc_prim :: (delay {cl(as)} (scan f acc_prim (advance as)))
;
```

Listing 6.1: ComRaTT source program implementing `scan` from Bahr and Møgelberg [3]

The function type checks and is supported by the pipeline up until it fails in code generation because of how we handle boxed closures in the runtime. The compiler attempts unboxing `f` within a closure, which is not supported as closures expect only top-level definitions as function pointers. To remedy this, we would have to either special case how we unbox functions, or make closure conversion more general, to handle this without special cases.

6.7 Garbage collection

The version of ComRaTT we have presented in this paper does *not* feature a garbage collector. We have described how the reactive semantics in Section 3.4 Reactive semantics (p. 24) allow for efficient garbage collection of the η_N heap through the OUTPUT-END transition. In practice, the virtual splitting of heaps into η_N and η_L is not a constant time operation and requires traversal of the program heap to partition what can be kept and what can be removed. As such, garbage collection leaves the heap in a fragmented shape, so we decided to look for other viable options.

The heap structure defined in Section 4.6.2 Memory regions (p. 38), where we have a shared heap and a location heap, was designed with the intention of allowing efficient manipulation of location entries. Since we define the location heap as uniformly sized sequences of $2 \cdot 4$ bytes, we can swap entries as we wish. This gives way to a strategy of deleting η_N and defragmenting the location heap, which we will describe now.

In Figure 6.1, the shared heap and location heap of a program containing four delayed computations and input channels κ_1, κ_2 are given. κ_2 is the active input channel of the program, and we have transitioned through all of the OUTPUT-* transitions but OUTPUT-END from Figure 3.4.

Collecting garbage at the OUTPUT-END transition is accomplished by taking all active delayed computations η_N and removing them from the store, $\langle \cdot; \eta_N \langle k \mapsto v \rangle \, \eta_L \rangle \Longrightarrow \langle \cdot; \eta_L \rangle$. The active location heap entries in Figure 6.1, those marked with striped boxes, are eligible for garbage collection.

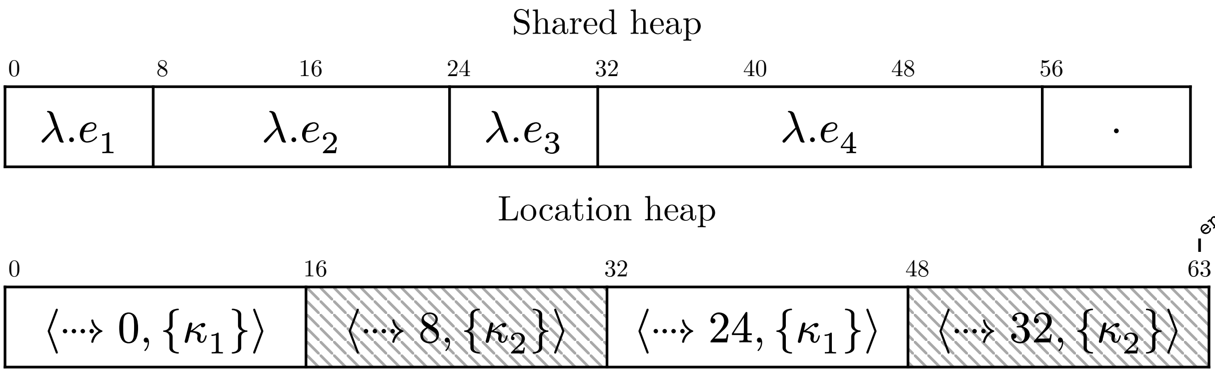

Shared heap
0
8
16
24
32
40
48
56
$\lambda.e_1$
$\lambda.e_2$
$\lambda.e_3$
$\lambda.e_4$
$\cdot$
Location heap
end
0
16
32
48
63
$\langle \dashrightarrow 0, \{\kappa_1\}\rangle$
$\langle \dashrightarrow 8, \{\kappa_2\}\rangle$
$\langle \dashrightarrow 24, \{\kappa_1\}\rangle$
$\langle \dashrightarrow 32, \{\kappa_2\}\rangle$

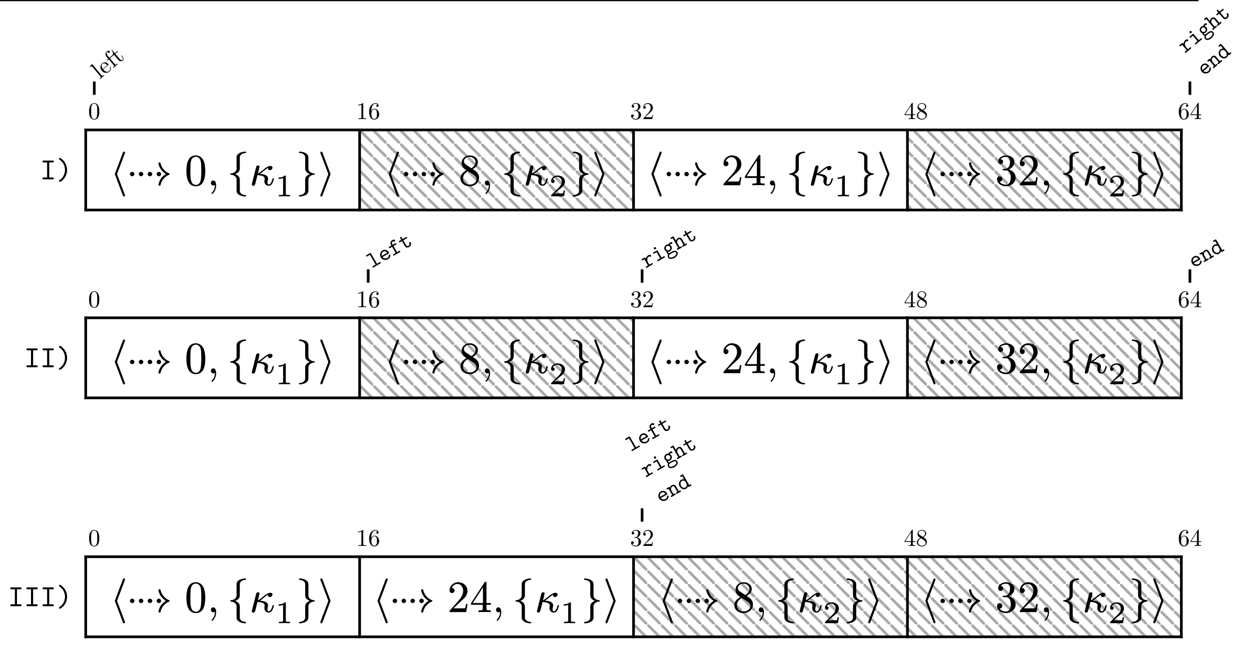
Figure 6.1: Heaps of a program with active input κ_2 before OUTPUT-END garbage collection.

The current pointer for the next memory allocation offset is at byte 64 in the location heap, marked with the `end` label. A single pass over the location heap is enough for collecting and defragmenting the heap given the following algorithm:

1. Prepare location traversal from both ends at the heap, moving $2 \cdot 4$ bytes at a time. `left` starting at location 0 and `right` starting at location `end`.
2. Traverse `left` until it meets a location i such that $\kappa_{\text{active}} \in \theta_i$
   a. Traverse `right` until it meets a location j such that $\kappa_{\text{active}} \notin \theta_j$
   b. swap location i with location j.
3. Repeat 2. until $i = j$
4. Reset heap location allocation offset `end` $= i$.

To showcase how the location heap collection looks, we will iterate through the algorithm in Figure 6.2, where $\kappa_{\text{active}} \equiv \kappa_2$. We start with subfigure `I)`, where we have set `left` to location offset 0 and `right` to the location of `end`. We progress `left` to the next location, i, and meet a clock where $\kappa_2 \in \theta_i$, which prompts us to move `right` towards `left` until it meets a location j such that $\kappa_2 \in \theta_j$ as depicted in subfigure `II)`. We then swap i and j, and move `left` until we either find another location i where $\kappa_2 \in \theta_i$ or `left` $=$ `right`, the latter of which is the case in subfigure `III)`. At the end of the algorithm, we update `end` to the location of `left`/`right`, leaving behind a collected location heap.

The algorithm is not sufficient for garbage collection on its own. As we are updating the indices of locations, we need to adjust any of the swapped locations j to point to i in the *shared heap* as well as the data structure(s) in the runtime that map outputs to locations. This could require introducing indirection layers such that location references are stable and more bookkeeping to defragment the indirection layers.



left
right
end
0
16
32
48
64
I)
$\langle \dashrightarrow 0, \{\kappa_1\}\rangle$
$\langle \dashrightarrow 8, \{\kappa_2\}\rangle$
$\langle \dashrightarrow 24, \{\kappa_1\}\rangle$
$\langle \dashrightarrow 32, \{\kappa_2\}\rangle$
II)
III)

Figure 6.2: Running conceptual GC algorithm on the location heap of Figure 6.1.

The bigger issue is handling transitive dependencies between locations and heap data. When we remove a location during garbage collection, we need to figure out what other locations and heap objects become unreachable as a result. This means implementing a proper marking phase that follows location pointers to the shared heap to identify which allocated elements are still reachable from the locations we are keeping. Any location that does not get marked during this traversal becomes garbage. But getting this right requires coordinating the location heap cleanup with the shared heap cleanup, and we need to make sure we do not accidentally follow stale pointers or miss valid references.

Finally, there is the question of when to actually run garbage collection. Running at every OUTPUT-END transition means that we pause everything and run collection constantly. Delaying/grouping collection over multiple input/output phases would amortize the cost of garbage collection, but increase implementation complexity. The upcoming WebAssembly GC proposal might help handling stable values, but using the GC would require restructuring the heap layout and code generation.

6.8 ComRaTT on the web

The runtime we have presented in this paper does not currently run on the web, but could easily be translated to a JavaScript implementation. ComRaTT's compilation target, WASM, is well equipped for the web, giving us the building blocks required to run graphical programs within a browser. The required next steps to accomplish this include defining inputs and outputs for web interaction, and integrating a JavaScript runtime with such primitives.

Chapter 7

Reflections

7.1 Evolution of project focus

We left off our preliminary research project [18] with identifying several limitations and future work candidates to potentially be part of the scope of this thesis, such as pattern matching, polymorhpism, and algebraic data types.

At that point we knew that heap allocating closures was one of the first steps to take towards implementing the reactive semantics and properly support the later modality and signals, giving way for causality and productivity. This, however, proved to be a bigger challenge than anticipated. Ultimately, achieving the main goal of supporting the modal types and reactive semantics became the sole focus, reducing the priority of features like polymorphism, built-in data types and GC. Specifically supporting programs like the one presented in Listing 3.2 that use the core ideas of Async RaTT was not straightforward while simultaneously attempting to keep regressions out of the pipeline and losing generality.

For those reasons, some of the points mentioned as limitations in the research project [18] have been repeated here.

7.2 Alternative compilation target

The decision to compile directly to WASM forced us to think very carefully about how the language should be structured and what restrictions we had to impose on ComRaTT programs. During implementation, it was clear that we could have moved faster if we did not have to generate WASM directly from our language, as many design decisions were made based on how the stack machine of WASM was designed. This means, for example, that closures had to fit a certain shape for allocation on the heap and that we were unable to provide a garbage collection strategy for the language.

We would have liked to explore the possibility of targeting our own custom stack machine that would give us more control over the execution environment. Implementing a virtual stack machine could trivially be done in any language that has WebAssembly as a compilation target, such as Rust or Zig, providing us with an abstraction over WASM while keeping the portability and (some of the) performance benefits. We believe that using an intermediary stack machine would establish a better foundation for exploring practical limitations and implementation requirements than targeting WASM directly, mainly because a reference implementation of compiled Async RaTT does not exist yet. Iterating at a higher abstraction level is beneficial for trying out things like garbage collection, compiler optimization passes, and the like.

Furthermore, a reference stack machine implementation is a way to establish an oracle for the operational semantics, similarly to how an interpreter can be a reference implementation for a programming language. As for performance, we cannot expect a virtual stack machine running on top of WebAssembly to have identical performance to directly targeting WebAssembly.

Given that the project set out to evaluate the feasibility of compiling an implementation of Async RaTT to WASM, it would not have been within the scope of the project to sidestep WASM with an intermediary stack machine.

7.3 Unit testing

Unit testing enabled us to iterate on different parts of the compiler with some level of certainty that we were not breaking underlying functionality and behavior. For example, when we transitioned from Hindley-Milner based type inference to the bidirectional type checker, our existing suite of tests allowed us to sanity check the results of the compiler, increasing confidence in and stability of the compiler.

Our testing approach had some limitations. We wanted to be able to express tests at a higher level than providing and comparing abstract syntax trees before and after a transformation. We also wanted to reuse the same test cases across different parts of the compiler transformations, which was not straightforward without substantial test infrastructure.

Additionally, our tests primarily verified compiler output rather than actual program execution—adding automated WebAssembly runtime tests would have caught issues in the generated code that only manifest during execution. Automated testing of reactive behavior in programs would have been particularly valuable for validating temporal semantics and ensuring that clock tracking worked correctly in practice. This would have required setting up a WebAssembly execution environment and testing framework, but could have provided stronger guarantees about the correctness of our implementation.

Chapter 8

Conclusion

This thesis has presented ComRaTT, the first implementation of a subset of Async RaTT [3] that compiles directly to a low-level machine language (WebAssembly [28]). Building on our prior research project [18], we have demonstrated the feasibility of targeting WASM as a compilation output for an asynchronous functional reactive programming language with modal types.

Our main contribution lies in successfully implementing the core reactive semantics of Async RaTT, including signal handling, the stable modality, the later modality, clock tracking, and heap-allocated delayed computations, all within the constraints of the WebAssembly stack machine. While previous work in functional reactive programming has focused on interpreters or higher-level compilation targets, ComRaTT represents the first direct code generation approach for Async RaTT to WebAssembly.

The implementation demonstrates both achievements and limitations. Our compilation pipeline transforms high-level functional constructs through multiple passes, including η-expansion, consecutive application elimination, consecutive lambda elimination, lambda lifting, and A-normalization, to produce WASM bytecode. But due to unresolved issues in our compilation pipeline, some ComRaTT programs generate invalid WASM.

We have developed a working reactive runtime system that implements key aspects of Async RaTT's operational semantics, managing delayed computations through a custom heap structure segregated into shared and location heaps. However, the technical challenges encountered proved more substantial than anticipated. Targeting WASM directly forced difficult design decisions that limited the language's expressivity and led to a brittle implementation. Most notably, we were unable to implement garbage collection, meaning programs will exhaust memory over time.

ComRaTT serves primarily as a proof-of-concept that demonstrates the potential and the challenges of compiling asynchronous modal FRP languages to WebAssembly. Future work would benefit from addressing the garbage collection challenge and reconsidering whether an intermediary virtual machine might provide a more practical foundation for such implementations.

Acknowledgement of generative AI use

In accordance with ITU's guidelines for use of generative AI in projects and theses [17], this section declares how we have used generative AI for this thesis.

We have used the AI models *Claude 3.5 Haiku*[19] and *Claude 3.7 Sonnet*[20] by Anthropic for small parts of the implementation of this project. The use was conducted through the AI model integration in the text editor *Zed*[21].

Specifically we have used it for generating abstract syntax trees in relation to unit tests, where this would have been cumbersome to do manually. In some cases the output from the model has been correct and as such has been used directly and in other cases it has been necessary to manually modify the output slightly or re-prompt to have it altered.

We have used prompts similar to "Given our AST definition … generate the AST of a primitive expression that adds the variable 'x' to the 3rd element of the tuple (42, true, 40)".

Furthermore, the cover-page mascot was generated using generative AI (OpenAI's ChatGPT image mode).

[19]https://www.anthropic.com/claude/haiku, accessed 13/05/2025
[20]https://www.anthropic.com/claude/sonnet, accessed 13/05/2025
[21]https://zed.dev/, accessed 13/05/2025

Bibliography

[1] Bahr, Patrick 2025. personal communication.

[2] Bahr, Patrick 2022. Modal FRP for all: Functional reactive programming without space leaks in Haskell. *Journal of Functional Programming.* 32, e15 (Dec. 2022). DOI:https://doi.org/10.1017/S0956796822000132.

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

[4] Bahr, Patrick, Graulund, Christian Uldal and Møgelberg, Rasmus Ejlers 2019. Simply RaTT: A Fitch-style Modal Calculus for Reactive Programming Without Space Leaks. *Proc. ACM Program. Lang.* 3, ICFP (Jul. 2019), 1–27. DOI:https://doi.org/10.1145/3341713.

[5] Bahr, Patrick, Houlborg, Emil and Rørdam, Gregers Thomas Skat 2023. Asynchronous Reactive Programming with Modal Types in Haskell. *Practical Aspects of Declarative Languages* (Cham, 2023), 18–36.

[6] Berg, Alexander and Jäpelt, Emil 2024. Implementing a Standalone Language for Asynchronous Modal FRP. (2024).

[7] Bidirectional Constraint Generation: 2023. *https://thunderseethe.dev/posts/bidirectional-constraint-generation/*.

[8] Christiansen, David Bidirectional Typing Rules: A Tutorial.

[9] Czaplicki, Evan and Chong, Stephen 2013. Asynchronous functional reactive programming for GUIs. *SIGPLAN Not.* 48, 6 (Jun. 2013), 411–422. DOI:https://doi.org/10.1145/2499370.2462161.

[10] Disch, Jean-Claude, Heegaard, Asger and Bahr, Patrick 2025. Functional Reactive GUI Programming with Modal Types. TFP 2025, to appear.

[11] Dunfield, Jana and Krishnaswami, Neel 2021. Bidirectional Typing. *ACM Comput. Surv.* 54, 5 (May 2021). DOI:https://doi.org/10.1145/3450952.

[12] Elliott, Conal and Hudak, Paul 1997. Functional reactive animation. *SIGPLAN Not.* 32, 8 (Aug. 1997), 263–273. DOI:https://doi.org/10.1145/258949.258973.

[13] Flanagan, Cormac et al. 1993. The essence of compiling with continuations. *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, USA, 1993), 237–247.

[14] Flanagan, Cormac et al. 1993. The essence of compiling with continuations. *SIGPLAN Not.* 28, 6 (Jun. 1993), 237–247. DOI:https://doi.org/10.1145/173262.155113.

[15] Ford, Bryan 2004. Parsing expression grammars: a recognition-based syntactic foundation. *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Venice, Italy, 2004), 111–122.

[16] Ford, Bryan 2004. Parsing expression grammars: a recognition-based syntactic foundation. *SIGPLAN Not.* 39, 1 (Jan. 2004), 111–122. DOI:https://doi.org/10.1145/982962.964011.

[17] Generative AI guidelines for students at ITU: 2025. *https://itustudent.itu.dk/-/media/ITU-Student/Study-Administration/Other/Generative-AI-guidelines-for-students-V3-pdf.pdf.*

[18] Holmgaard, Marcus Sebastian Emil and Sattar Atta, Ahmad 2024. Towards Compiled Async RaTT. (Dec. 2024).

[19] Hudak, Paul et al. 2003. Arrows, Robots, and Functional Reactive Programming. *Advanced Functional Programming: 4th International School, AFP 2002, Oxford, UK, August 19-24, 2002. Revised Lectures.* J. Jeuring and S.L.P. Jones, eds. Springer Berlin Heidelberg. 159–187.

[20] Jeffrey, Alan 2014. Functional reactive types. *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)* (Vienna, Austria, 2014).

[21] Jeffrey, Alan 2012. LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs. *Proceedings of the Sixth Workshop on Programming Languages Meets Program Verification* (Philadelphia, Pennsylvania, USA, 2012), 49–60.

[22] Jeltsch, Wolfgang 2013. Temporal logic with "Until", functional reactive programming with processes, and concrete process categories. *Proceedings of the 7th Workshop on Programming Languages Meets Program Verification* (Rome, Italy, 2013), 69–78.

[23] Jones, SL Peyton, Lester, DR and Peyton Jones, Simon 1992. *Implementing functional languages: a tutorial.* Prentice Hall.

[24] Krishnaswami, Neelakantan R. 2013. Higher-order functional reactive programming without spacetime leaks. *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA, 2013), 221–232.

[25] Krishnaswami, Neelakantan R. and Benton, Nick 2011. Ultrametric Semantics of Reactive Programs. *2011 IEEE 26th Annual Symposium on Logic in Computer Science* (2011), 257–266.

[26] Mandel, Louis and Pouzet, Marc 2005. ReactiveML: a reactive extension to ML. *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming.* (2005).

[27] Pratt, Vaughan R. 1973. Top down operator precedence. *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Boston, Massachusetts, 1973), 41–51.

[28] Rossberg, Andreas ed. 2022. *WebAssembly Core Specification.*

[29] Sestoft, Peter 2017. *Programming Language Concepts.* Springer International Publishing.

[30] wabt, 'The WebAssembly Binary Toolkit': *https://github.com/webassembly/wabt/.* Accessed: 2025-05-20.

[31] Wasm3, 'A fast WebAssembly interpreter and the most universal WASM runtime': *https://github.com/wasm3/wasm3/.* Accessed: 2025-05-20.

[32] Wasmer, 'Fast, secure, lightweight containers based on WebAssembly': *https://github.com/wasmerio/wasmer/.* Accessed: 2025-05-20.

[33] Wasmtime, 'A lightweight WebAssembly runtime that is fast, secure, and standards-compliant': *https://github.com/bytecodealliance/wasmtime/.* Accessed: 2025-05-20.

[34] Watt, Conrad et al. 2021. Two Mechanisations of WebAssembly 1.0. *Formal Methods: 24th International Symposium, FM 2021, Virtual Event, November 20–26, 2021, Proceedings* (Berlin, Heidelberg, 2021), 61–79.

[35] Ştefănescu, Alin, Esparza, Javier and Muscholl, Anca 2003. Synthesis of Distributed Algorithms Using Asynchronous Automata. *CONCUR 2003 - Concurrency Theory* (Berlin, Heidelberg, 2003), 27–41.

List of Listings

List of Figures

Appendix

A1 Grammar

```
program = _{ SOI ~ toplevel* ~ EOI }

toplevel = _{
    (function_def | channel_def | output_def) ~ ";"
}

function_args = { identifier* }

function_def = {
    identifier ~ ":" ~ type_expr ~ "def" ~ identifier ~ function_args ~ "=" ~ expr
}
channel_def  = { "chan" ~ identifier ~ ":" ~ type_expr }
output_def   = { identifier ~ "<-" ~ expr }

// Types
type_expr  =  { type_prefix* ~ type_primary ~ (type_infix ~ type_prefix* ~ type_primary)* }
type_infix = _{ arrow }

type_prefix = _{ later | signal | box_type }
later       =  { "O" }
signal      =  { "Sig" }
box_type    =  { "Box" | "□" }

type_primary = _{ int_type | bool_type | unit_type | parenthesis_or_tuple_type }
int_type     =  { "int" }
bool_type    =  { "bool" }
unit_type    =  { "()" }
arrow        =  { "->" }

```

```
parenthesis_or_tuple_type = {
    "(" ~ type_expr ~ ("*" ~ type_expr)* ~ ")"
}

tuple_access = { primary ~ ("." ~ integer)* }
term         = { tuple_access+ }
expr         = { (term ~ (infix ~ term)*) }

clock_union = { "U" | "⊔" }
clock_wait  = { "wait" ~ (!"cl(") ~ identifier }
clock_base  = { "cl(" ~ (clock_wait | identifier) ~ ")" | "(" ~ clock_base ~ ")" }
clock_expr  = { clock_base ~ (clock_union ~ clock_base)* }

let_expr     = { "let" ~ identifier ~ "=" ~ expr ~ "in" ~ expr }
fun_expr     = { "fun" ~ function_args ~ "->" ~ expr }
if_expr      = { "if" ~ expr ~ "then" ~ expr ~ "else" ~ expr }
delay_expr   = { "delay" ~ "{" ~ clock_expr ~ "}" ~ expr }
advance_expr = { "advance" ~ identifier }
wait_expr    = { "wait" ~ identifier }
box_expr     = { "box" ~ expr }
unbox_expr   = { "unbox" ~ expr }

primary = _{
    let_expr
  | fun_expr
  | if_expr
  | delay_expr
  | advance_expr
  | wait_expr
  | parenthesis_or_tuple
  | integer
  | true_lit
  | false_lit
  | unit_lit
  | identifier
  | box_expr
  | unbox_expr
}

parenthesis_or_tuple = {
    "(" ~ expr ~ ("," ~ expr)* ~ ")"
}

infix = _{
    sig_cons
  | equality_op
  | relational_op
  | add_op
  | mul_op
}

sig_cons      = { "::" }
equality_op   = { "<>" | "=" }
relational_op = { "<=" | ">=" | "<" | ">" }
add_op        = { "+" | "-" }
mul_op        = { "*" | "/" }

// Terminals
identifier = @{ !keyword ~ (ASCII_ALPHA | "_") ~
(ASCII_ALPHANUMERIC | "_")* }
integer    = @{ "-"? ~ ("0" | ASCII_NONZERO_DIGIT ~ ASCII_DIGIT*) }
true_lit   =  { "true" }
false_lit  =  { "false" }
unit_lit   =  { "()" }

keyword = _{
    "def"
  | "let"
  | "in"
  | "if"
  | "then"
  | "else"
  | "fun"
  | "chan"
  | "delay"
  | "advance"
  | "wait"
  | "true"
  | "false"
  | "box"
  | "unbox"
  | "Box"
  | "Sig"
  | "O"
}

WHITESPACE = _{ " " | "\t" | "\r" | "\n" }
COMMENT    = _{
    "/*" ~ (!"*/" ~ ANY)* ~ "*/"
  | "//" ~ (!("\n") ~ ANY)*
}
```

PEG parser grammar `pest.lang`

A2 Compilation schemes

A2.1 Code generation for *AExpr*

AExpr[[Const(CINt(n), _ty)]]
=
`i32.const n`

(`int constant`)

AExpr[[Const(CBool(b), _ty)]]
=
`i32.const` if b then `1` else `0`

(`boolean constant`)

AExpr[[Const(CUnit), _ty)]]
=
`i32.const -1`

(`unit constant`)

AExpr[[Const(CLaterUnit), _ty)]]
=
`i32.const -1`

(`laterunit constant`)

AExpr[[Const(Never), _ty)]]
=

```
call LOCATION_MALLOC_INDEX
i32.const -42
i32.store 1 offset=0
global.get 1
i32.const 4
i32.sub
i32.const 0
i32.store 1 offset=0
global.get 1
i32.const 8
i32.sub
```

(`never constant`)

AExpr[[Wait(channel_name, _ty)]]
=

```
i32.const index_of_channel(channel_name)
call INDEX_OF_WAIT_FFI_FUNCTION
```
(wait expression)

AExpr[[Var(name, _ty)]]
=

```
if name is a local variable then
    local.get index_of_var
else if name is toplevel function then
    call index_of_toplevel_function
else panic
```
(var expression)

---

if function = CExpr::App(AExpr::Var(app_name, _var_ty), app_args, _app_ty)
i.e. an application of a name
AExpr[[Closure(lambda_args, function, _lambda_type)]]
=

```
malloc((2+arity)*WORDSIZE)
local.tee index_of_dupi32_variable
i32.const INDEX_OF_TOPLEVEL
i32.store 0 offset=0
local.get index_of_dupi32_variable
i32.const LAMBDA_ARGS_LENGTH
i32.store 0 offset=4
local.get index_of_dupi32_variable  \
AExpr[[arg]]                         } for arg in app_args
i32.store 0 offset=2+INDEX(arg)     /
local.get index_of_dupi32_variable
```
(closure expression)

A2.2 Code generation for *CExpr*

$$\text{CExpr}[[\text{Prim(op, left, right, ty)}]] = \begin{array}{l}\text{AExpr}[[\text{left}]]\\ \text{AExpr}[[\text{right}]]\\ \text{B}[[\text{op, ty}]]\end{array} \qquad (\texttt{primitive expression})$$

where

B[[+, TInt]]	= `i32.add`
B[[-, TInt]]	= `i32.sub`
B[[*, TInt]]	= `i32.mul`
B[[/, TInt]]	= `i32.div`
B[[=, TBool]]	= `i32.eq`
B[[<, TBool]]	= `i32.lt_s`
B[[<=, TBool]]	= `i32.le_s`
B[[>, TBool]]	= `i32.gt_s`
B[[>=, TBool]]	= `i32.ge_s`
B[[<>, TBool]]	= `i32.ne`
B[[_, _]]	= panic

```
                CExpr[[App(f, args, _ty)]]
                            =
if f is a local variable representing a HOF pointer then
    $\left.\begin{array}{l}\texttt{local.get INDEX\_OF\_LOCAL}\\ \texttt{i32.load 0 offset=4}\\ \texttt{i32.const 1}\\ \texttt{i32.sub}\\ \texttt{i32.store 0 offset=4}\\ \texttt{local.get INDEX\_OF\_LOCAL}\\ \texttt{i32.load 0 offset=4}\\ \texttt{i32.const 4}\\ \texttt{i32.mul}\\ \texttt{local.get INDEX\_OF\_LOCAL}\\ \texttt{i32.add}\\ \text{AExpr[[arg]]}\\ \texttt{i32.store 0 offset=8}\end{array}\right\}$ for arg in args
else if f is a local variable representing a pointer then
    local.get INDEX_OF_LOCAL                                  (application expression)
    $\left.\text{AExpr[[arg]]}\right\}$ for arg in args
    if local is of type later then
        i32.const INDEX_OF_LOCATION_DISPATCH
        call_indirect
    else
        i32.const INDEX_OF_DISPATCH
        call_indirect
else if f is a top level function then
    AExpr[[arg]]} for arg in args
    call INDEX_OF_FUNCTION
else panic
```

CExpr[[IfThenElse(condition, then_branch, else_branch, ty)]]
=

```
AExpr[[condition]]
if
AnfExpr[[then_branch]]
else                                  (conditional expression)
AnfExpr[[else_branch]]
end
```

CExpr[[Tuple(expressions, _ty)]]
=

```
malloc(tuple_length * WORDSIZE)
local.set index_of_dupi32_variable
                                                          (tuple expression)
local.get index_of_dupi32_variable          }
AExpr[[e]]                                  } for e in expressions
i32.store 0 offset=WORDSIZE*INDEX(e)        }
local.get index_of_dupi32_variable
```

CExpr[[Access(expr, index, _ty)]]
=

```
AExpr[[expr]]
                                      (access expression)
i32.load 0 offset=index*WORDSIZE
```

A2.3 Code generation for *AnfExpr*

$$\begin{gathered}\text{AnfExpr}[[\text{AExpr}(\text{a})]] \\ = \\ \text{AExpr}[[\text{a}]]\end{gathered} \qquad (\texttt{ANF atomic expression})$$

$$\begin{gathered}\text{AnfExpr}[[\text{CExpr}(\text{c})]] \\ = \\ \text{CExpr}[[\text{c}]]\end{gathered} \qquad (\texttt{ANF complex expression})$$

$$\begin{gathered}\text{AnfExpr}[[\text{Let}(\text{name, \_ty, rhs, body})]] \\ = \\ \begin{array}{l}\text{AnfExpr}[[\text{rhs}]] \\ \texttt{local.set index\_of\_name} \\ \text{AnfExpr}[[\text{body}]]\end{array}\end{gathered} \qquad (\texttt{ANF let expression})$$

A3 *sigrec* WAT code

```
(module
 (type $wait (func (param i32) (result i32)))
 (type $set_output_to_location (func (param i32 i32) (result
i32)))
 (type $malloc (func (param i32) (result i32)))
 (type $location_malloc (func (result i32)))
 (type $clock_of (func (param i32) (result i32)))
 (type $kb (func (result i32)))
 (type $#lambda_1 (func (result i32)))
 (type $#lambda_2 (func (param i32) (result i32)))
 (type $dispatch (func (param i32) (result i32)))
 (type $location_dispatch (func (param i32) (result i32)))
 (type $init (func))
 (import "ffi" "wait" (func $wait (type $wait)))
 (import "ffi" "set_output_to_location" (func
$set_output_to_location (type $set_output_to_location)))
 (func $malloc (type $wait) (param $size i32) (result i32)
   (local $old i32)
   global.get 0
   local.tee $old
   local.get $size
   i32.add
   global.set 0
   local.get $old)
 (func $location_malloc (type $location_malloc) (result i32)
   (local $old i32)
   global.get 1
   local.tee $old
   i32.const 8
   i32.add
   global.set 1
   local.get $old)
 (func $clock_of (type $wait) (param $location_ptr i32) (result
i32)
```

```
        local.get $location_ptr
        i32.load 1 offset=4)
    (func $kb (type $kb) (result i32)
        (local $key i32) (local $dupi32 i32)
        i32.const 8
        call $malloc
        local.tee $dupi32
        i32.const 6
        i32.store
        local.get $dupi32
        i32.const 0
        i32.store offset=4
        local.get $dupi32
        drop
        call $location_malloc
        local.get $dupi32
        i32.store 1
        global.get 1
        i32.const 4
        i32.sub
        i32.const 1
        i32.store 1
        global.get 1
        i32.const 8
        i32.sub
        local.set $key
        i32.const 12
        call $malloc
        local.tee $dupi32
        i32.const 7
        i32.store
        local.get $dupi32
        i32.const 0
        i32.store offset=4
```

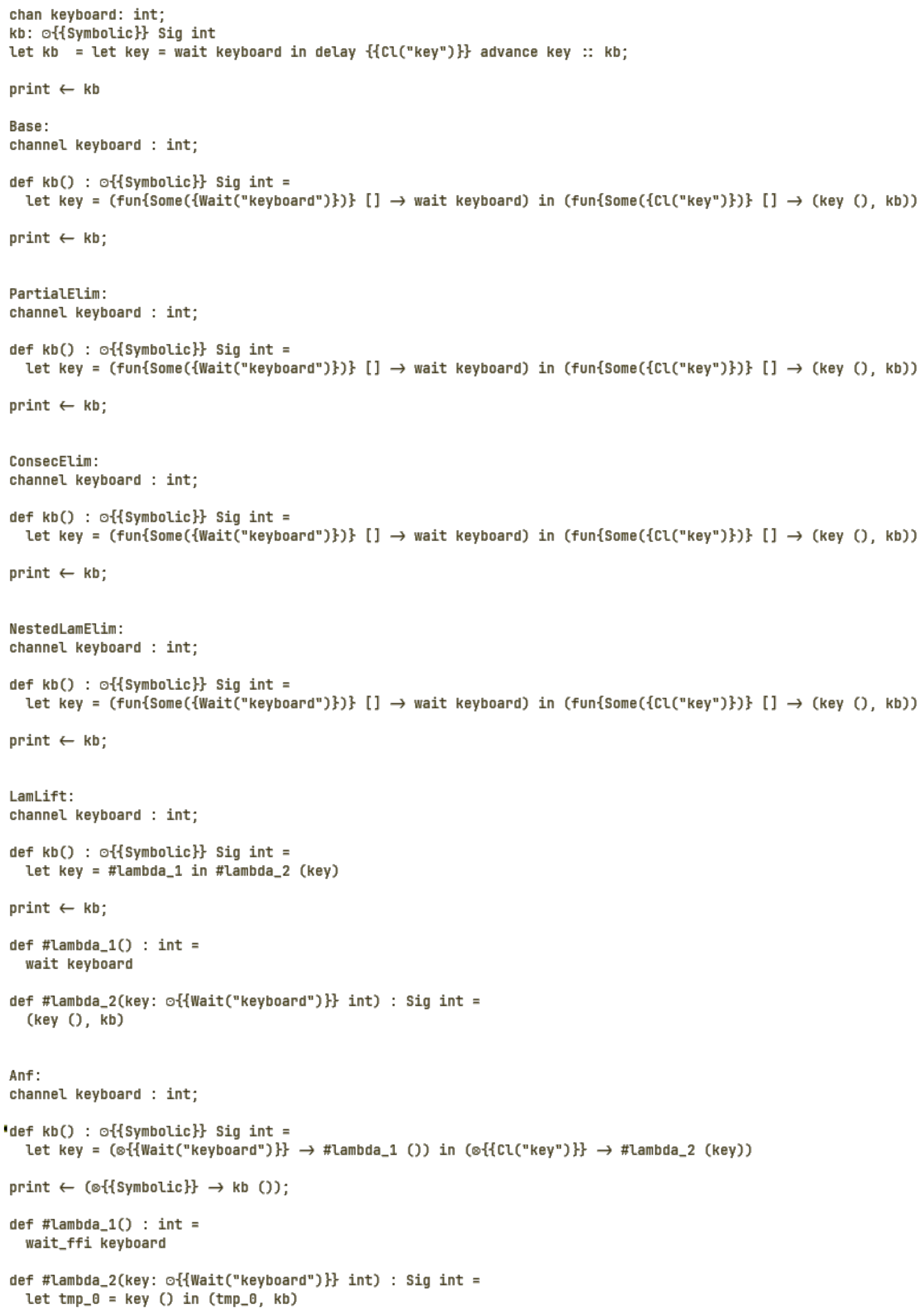
```
      local.get $dupi32
      local.get $key
      i32.store offset=8
      local.get $dupi32
      drop
      call $location_malloc
      local.get $dupi32
      i32.store 1
      global.get 1
      i32.const 4
      i32.sub
      local.get $key
      i32.const 4
      call_indirect (type $clock_of)
      i32.store 1
      global.get 1
      i32.const 8
      i32.sub)
  (func $#lambda_1 (type $#lambda_1) (result i32)
    (local $dupi32 i32)
    i32.const 1
    call $wait)
  (func $#lambda_2 (type $#lambda_2) (param $key i32) (result i32)
    (local $tmp_0 i32) (local $dupi32 i32)
    local.get $key
    i32.const 9
    call_indirect (type $location_dispatch)
    local.set $tmp_0
    i32.const 8
    call $malloc
    local.set $dupi32
    local.get $dupi32
    local.get $tmp_0
    i32.store
    local.get $dupi32
    call $kb
    i32.store offset=4
    local.get $dupi32)
  (func $dispatch (type $dispatch) (param i32) (result i32)
    (local i32)
    block  ;; label = @1
      block  ;; label = @2
        block  ;; label = @3
          block  ;; label = @4
            block  ;; label = @5
              block  ;; label = @6
                block  ;; label = @7
                  block  ;; label = @8
                    block  ;; label = @9
                      local.get 0
                      i32.load
                      br_table 0 (;@9;) 1 (;@8;) 2 (;@7;) 3 (;@6;) 4 (;@5;) 5 (;@4;) 6 (;@3;) 7 (;@2;) 8 (;@1;)
                    end
                    local.get 0
                    i32.load offset=8
                    i32.const 0
                    return_call_indirect (type $wait)
                  end
                  local.get 0
                  i32.load offset=12
                  local.get 0
                  i32.load offset=8
                  i32.const 1
                  return_call_indirect (type $set_output_to_location)
                end
                local.get 0
```

```
                    i32.load offset=8
                    i32.const 2
                    return_call_indirect (type $malloc)
                  end
                  i32.const 3
                  return_call_indirect (type $location_malloc)
                end
                local.get 0
                i32.load offset=8
                i32.const 4
                return_call_indirect (type $clock_of)
              end
              i32.const 5
              return_call_indirect (type $kb)
            end
            i32.const 6
            return_call_indirect (type $#lambda_1)
          end
          local.get 0
          i32.load offset=8
          i32.const 7
          return_call_indirect (type $#lambda_2)
        end
        unreachable)
      (func $location_dispatch (type $location_dispatch) (param i32)
      (result i32)
        local.get 0
        i32.load 1
        i32.const 8
        return_call_indirect (type $dispatch))
      (func $init (type $init)
        i32.const 0
        call $kb
        call $set_output_to_location
```

```
    drop)
  (table (;0;) 128 funcref)
  (memory (;0;) 1)
  (memory (;1;) 1)
  (global (;0;) (mut i32) (i32.const 0))
  (global (;1;) (mut i32) (i32.const 0))
  (export "heap" (memory 0))
  (export "location" (memory 1))
  (export "table" (table 0))
  (export "malloc" (func $malloc))
  (export "location_malloc" (func $location_malloc))
  (export "clock_of" (func $clock_of))
  (export "kb" (func $kb))
  (export "#lambda_1" (func $#lambda_1))
  (export "#lambda_2" (func $#lambda_2))
  (export "dispatch" (func $dispatch))
  (export "location_dispatch" (func $location_dispatch))
  (export "init" (func $init))
  (elem (;0;) (i32.const 0) func $wait $set_output_to_location
  $malloc $location_malloc $clock_of $kb $#lambda_1 $#lambda_2
  $dispatch $location_dispatch $init)
  (@custom "name" "..Debug info..")
)
```

Full output from WAT converting the result of compiling Listing 3.2

A4 Program representation output



```
chan keyboard: int;
kb: ⊙{{Symbolic}} Sig int
let kb  = let key = wait keyboard in delay {{Cl("key")}} advance key :: kb;

print ← kb

Base:
channel keyboard : int;

def kb() : ⊙{{Symbolic}} Sig int =
  let key = (fun{Some({Wait("keyboard")})} [] → wait keyboard) in (fun{Some({Cl("key")})} [] → (key (), kb))

print ← kb;


PartialElim:
channel keyboard : int;

def kb() : ⊙{{Symbolic}} Sig int =
  let key = (fun{Some({Wait("keyboard")})} [] → wait keyboard) in (fun{Some({Cl("key")})} [] → (key (), kb))

print ← kb;


ConsecElim:
channel keyboard : int;

def kb() : ⊙{{Symbolic}} Sig int =
  let key = (fun{Some({Wait("keyboard")})} [] → wait keyboard) in (fun{Some({Cl("key")})} [] → (key (), kb))

print ← kb;


NestedLamElim:
channel keyboard : int;

def kb() : ⊙{{Symbolic}} Sig int =
  let key = (fun{Some({Wait("keyboard")})} [] → wait keyboard) in (fun{Some({Cl("key")})} [] → (key (), kb))

print ← kb;


LamLift:
channel keyboard : int;

def kb() : ⊙{{Symbolic}} Sig int =
  let key = #lambda_1 in #lambda_2 (key)

print ← kb;

def #lambda_1() : int =
  wait keyboard

def #lambda_2(key: ⊙{{Wait("keyboard")}} int) : Sig int =
  (key (), kb)


Anf:
channel keyboard : int;

def kb() : ⊙{{Symbolic}} Sig int =
  let key = (⊗{{Wait("keyboard")}} → #lambda_1 ()) in (⊗{{Cl("key")}} → #lambda_2 (key))

print ← (⊗{{Symbolic}} → kb ());

def #lambda_1() : int =
  wait_ffi keyboard

def #lambda_2(key: ⊙{{Wait("keyboard")}} int) : Sig int =
  let tmp_0 = key () in (tmp_0, kb)
```

The output printed initially when running Listing 3.2. It shows the program as well as representations of it after each pass.