Towards a WebAssembly Backend for MicroHs

Apoorva Anand
Student Number - 2037610
apoorvaanand.work@gmail.com

First Supervisor: Dr. Marco Vassena
Second Supervisor: Dr. Wouter Swierstra

September 2025

Master's in Computing Science

Utrecht University

Contents

List of Figures

Acknowledgement

I would like to start by expressing my gratitude to my supervisor Dr. Marco Vassena for his constant guidance and support. His advisory was the right mix of autonomy and accountability. Thank you for letting me ask you the same question again and again - I have learnt a lot because of you. I am grateful to Dr. Wouter Swierstra, for acting as my second supervisor, and for teaching such a transformative course. I would also like to thank the Software Technology group at Utrecht University, their courses were enlightening and let me explore the topics I liked with lots of freedom.

Thank you to everyone in computer science who cares. In particular, special thanks to Lennart Augustsson for building MicroHs - He has given students a powerful base to experiment on. Your expertise is legendary. Special thanks also to Andreas Rossberg, for being the editor of the WebAssembly spec. Your attempt at explaining WebAssembly in as many ways as possible has not gone unnoticed.

I want to express my appreciation to all my employers. They gave me the opportunity to support myself while pursuing this crazy dream of moving across the world to study esoteric things.

Thank you to all my friends, whom I shall not name, in fear that I might forget to mention a few. Thank you for letting me be a bad friend, for letting me disappear for long instances and still welcoming me with open arms. I would like to thank my parents. Without their support I would have never been able to pursue this dream. Your presence in my life has been indispensable. I know I am not the perfect son and often make my own choices, thank you for bearing with that. The same goes to the rest of my family.

Eileen, I love you! Thank you for your constant support. Your acceptance of my whole self makes life beautiful. The universe has gifted me with a partner who is the complete package and I am grateful. Finally, I would like to thank myself, for sticking through all the trials and tribulations that finishing this master's degree required. Things might not have turned out the way I had planned but I am proud to have kept moving forward.

"I think that it's extraordinarily important that we in computer science keep fun in computing. When it started out, it was an awful lot of fun. Of course, the paying customers got shafted every now and then, and after a while we began to take their complaints seriously. We began to feel as if we really were responsible for the successful, error-free perfect use of these machines. I don't think we are. I think we're responsible for stretching them, setting them off in new directions, and keeping fun in the house. I hope the field of computer science never loses its sense of fun. Above all I hope we don't become missionaries. Don't feel as if you're Bible salesmen. The world has too many of those already. What you know about computing other people will learn. Don't feel as if the key to successful computing is only in your hands. What's in your hands I think and hope, is intelligence: the ability to see the machine as more than when you were first led up to it, that you can make it more."

— Alan J. Perlis

MSc Thesis - Towards a WebAssembly Backend for MicroHs[1]

Abstract - *This thesis presents the design and implementation of a WasmGC backend for MicroHs, a lightweight Haskell compiler. The backend targets modern WebAssembly runtimes with native garbage collection support, enabling functional programs to execute without manual memory management. To this end, we develop a WasmGC encoding of SK combinators, the theoretical foundation of MicroHs, and implement a reduction machine entirely in WasmGC. The runtime system is realized in two ways: as handwritten WasmGC code and through code generation using a Haskell-based DSL. Together, these components form our native WasmGC backend. While a complete mapping of all of Haskell to WasmGC was not achieved, the approach demonstrates promising potential, especially in binary size reduction, which warrants further exploration.*

1. Introduction

WebAssembly (Wasm) 1.0 introduced a compact, efficient, and secure execution format, but for many applications, it is not enough. While it provides a portable compilation target, Wasm lacks critical features that hinder adoption for high-level languages like Haskell and Java. Even low-level languages like C, C++, and Rust encounter limitations due to Wasm's current design constraints. The absence of built-in memory management and garbage collection (GC) forces developers to either rely on heavy runtime support or implement inefficient manual memory management. These then have to be bundled with the Wasm binary.

Even minimal Haskell programs produce binaries exceeding 1 MB - an unacceptable overhead for many use cases where lightweight distribution is essential. Furthermore, directly compiling C-based runtimes introduces security vulnerabilities into the Wasm ecosystem. The security guarantees of Wasm's sandbox are undermined by the inherent risks of unchecked memory operations from C. [1] Fortunately, WebAssembly evolves through a proposal system; the WebAssembly Garbage Collection (WasmGC) proposal offers a solution by integrating garbage collection directly into the Wasm runtime. This allows languages to leverage the browser's or runtime's built-in garbage collector, reducing binary size and improving

[1] **The Ethics and Privacy Quick Scan of the Utrecht University Research Institute of Information and Computing Sciences classified this research as low-risk with no fuller ethics review or privacy assessment required.**

performance. Initial experiments with WasmGC have already shown promising results: programs compiled using WasmGC can see up to a 30% reduction in binary size as demonstrated by the Fannkuch benchmark. Java binaries for the benchmark are only 2.3 KB, compared to C's 6.1 KB and Rust's 9.6 KB. This is because the latter two include `malloc` and `free` in their binaries. [2]

This thesis aims to reduce the size of Haskell binaries. Concretely, we extend MicroHs, a minimalistic Haskell implementation, with a WasmGC backend. MicroHs presents a unique opportunity: its simplicity makes it an ideal candidate for experimentation with Wasm, unburdened by the complexities of larger Haskell compilers like GHC. By targeting WebAssembly with a lightweight runtime tailored for functional programming, this project explores new strategies for efficient execution without excessive bloat.

Size reductions are only one aspect; WasmGC also enables optimisations that have led to a 1.9× speedup in Java programs compiled to WasmGC. [2] MicroHs's compact C runtime system, while currently compilable to Wasm 1.0, stands to benefit significantly from WasmGC. We propose to develop a native WasmGC backend for MicroHs, leveraging the new Wasm features to create a truly lightweight and efficient Haskell experience. By doing so, we aim to demonstrate the transformative potential of WasmGC, not just for Haskell, but for the broader landscape of language implementations targeting Wasm.

Code Repository All of the code used for this thesis project has been uploaded online, along with all of the binaries generated, and the test results.[3]

2. Background

2.1 WebAssembly Reference

WebAssembly is a low-level bytecode which is *stack-based*. Instructions are executed in order and manipulate values on an implicit *operand stack*. There are two kinds of instructions - *Simple* and *Control*. Simple instructions perform basic operations, they pop arguments and push results back on the operand stack. Control instructions alter the control flow.

WebAssembly has a textual format that takes the form of S-expressions. S-expressions can be represented in *folded form*. For example, let's look at the code for adding 2 and 3.

```
(i32.const 2)
(i32.const 3)
(i32.add)
;; Places 5 on the stack
```

`(i32.const 2)` pushes the 32-bit integer constant 2 onto the stack, which is then followed by `(i32.const 3)`. `(i32.add)` then consumes these two operands from the stack, first `(i32.const 2)` then `(i32.const 3)`, to produce the result `(i32.const 5)`.

Functions Functions are named sequences of instructions that take zero or more values as parameters and return zero or more values as results. They can be recursive. The parameters are *mutable local variables*. Functions may also declare additional variables. Consider the function `addOne` that adds 1 to its parameter.

```
(func $addOne (param $p i32) (result i32)
    (local $n i32)
    (i32.const 1)
    (local.set $n)
    (local.get $p)
    (local.get $n)
    (i32.add)
)
```

Several language constructs, including functions, are defined and referred to using *indices*. To make it convenient, WebAssembly allows these indices to be replaced with *identifiers* that are prefixed with a `$`. The

local variables in a function are also indices, in this case the 0th local variable is `$p` and the 1st local variable is `$n`. The latter is explicitly declared as a local variable. For convenience and clarity, we will refer to these indexed constructs with their identifiers instead of their index when applicable.

`(local $n i32)` declares a variable `$n` of type `i32`. `(i32.const 1)` pushes 1 to the stack. `(local.set $n)` then sets variable `$n` to 1. We then get both `$p` our parameter and `$n` using `(local.get)`. Finally, we return our result using `(i32.add)`. Functions are called by using `call`. For example,

```
(i32.const 2)
(call $addOne)
```

adds 1 to 2, placing the result 3 on the stack.

Validation WebAssembly is validated using a *type system*. Types are given to instructions or instruction sequences. An instruction (or instruction sequence) has a type $(t_1^* \rightarrow_{x^*} t_2^*)$ if it pops operands of type t_1^* from the stack and pushes results of type t_2^*. x^* represents the indices of the locals which have been set in these instructions.

The type of the function `$addOne` is $[i32] \rightarrow_1 [i32]$. The type system is used to make sure that Wasm programs are not ill-formed before they're run, i.e., validation is done before execution. Wasm programs often come from untrusted sources, validation ensures that programs do not break out of the Wasm sandbox.

Numeric Instructions Numeric instructions provide operations over number types - $i32, i64, f32, f64$. The following are the categories of numeric instructions - constants, unary operations, binary operations, tests, comparisons, and conversions.

We have already seen constants and binary operations (`(i32.add)`). Unary and comparison operations are self-explanatory. Test operations take one operand and produce a boolean result - `(i32.eqz)` is a test operation we will use to test if the operand on top of the stack is a 0. This is often used along with control instructions.

Finally, conversion instructions convert a value from one type to another. `(i32.extend8_s)` is an instruction that will be used to convert an 8-bit integer to a 32-bit integer.

Control Instructions Some of the Control Instructions in WebAssembly are `nop`, `unreachable`, `block`, `loop`, `if` and `br`. `nop` does nothing. `unreachable` causes WebAssembly programs to *trap*. Traps stop execution and cannot be handled by Wasm. They are reported to the environment the Wasm program is being run in.

`block`, `loop`, and `if` are *structured* control flow instructions. All three can be given *block types* that are similar to a type given to a function. Blocks and loops don't do anything by themselves, they have *labels* associated with them. Labels combined with `br`, give us the difference in semantics between blocks and loops.

Branching to a block causes a *forward jump* whereas branching to a loop causes a *backward jump*. Let us look at an implementation of the factorial function to see how they work.

```
(func $factorial (param $n i32) (result i32)
    (local $result i32)  ;; Variable to store factorial

    ;; Initialize result = 1
    (i32.const 1)
    (local.set $result)

    (block $exit  ;; Define a block to allow breaking out
        (loop $repeat
            ;; If n <= 1, exit loop
            ;; We reuse the parameter n as a counter variable
            (local.get $n)
            (i32.const 1)
            (i32.le_s)
            (if
                (then (br $exit))
                (else (nop))
            ) ;; Break out of the loop if n <= 1

            ;; result = result * n
            (local.get $result)
            (local.get $n)
            (i32.mul)
            (local.set $result)

            ;; n = n - 1
            (local.get $n)
            (i32.const 1)
```

```
            (i32.sub)
            (local.set $n)

            (br $repeat) ;; Jump back to the start of the loop
        )
    )

    ;; Return the result
    (local.get $result)
)
```

`(i32.le_s)` is a signed lesser-than-or-equal-to check to see if our counter variable has reached 1. Instead of the if-then-else, we could've also used `br_if`. `br_if` is a variation of `br` that only branches if the value on top of the stack is not 0. Branching to `$exit` breaks us out of the loop, whereas branching to `$repeat` starts the loop again by jumping backward.

We will now explain the new features introduced in WasmGC that will be used extensively in this thesis project.

Reference Instructions Reference instructions help work with *references* to values allocated during runtime. References work with *heap types*. We will primarily be concerning ourselves with aggregate types, which are a subtype of heap types.

`ref.null`, `ref.is_null`, `ref.eq`, `ref.test`, and `ref.cast` are some reference instructions. `ref.null` takes a heap type as an argument to create a null type. `ref.is_null` tests if the operand on top of the stack is null. `ref.eq` checks if two references are equal.

`ref.test` and `ref.cast` test the dynamic type of a reference, i.e., they check the concrete type of an object during runtime. They take a reference type as an argument. The former returns the result while the latter traps if the type does not match.

Aggregate Instructions Aggregate instructions work with references to aggregate types. *Aggregate types* consist of *structures* and *arrays*. Their use is going to be pervasive throughout our Wasm code. Structures can hold different types in their fields, while arrays are homogeneous.

Structures allow only static indexing, while arrays can be dynamically indexed. Both structures and arrays are optionally mutable.

Aggregate instructions are best illustrated with an example. The following code is used to create a *binary tree*.

```
(type $node
        (struct (field $left (mut anyref))
                (field $right (mut anyref))
                (field $name (mut i32))))
```

Like functions and locals, types are also represented using indices in WebAssembly. In this case, we create a type `$node` using the *type* keyword. `$node` is a structure with three fields - `$left, right`, and `$name`. `$name` is an mutable `i32`, whereas `$left` and `$right` hold mutable references. The mutability has to be explicitly declared (using `mut`) because a structure's fields are immutable by default.

`anyref` is short for `(ref null any)`, which means this is a reference that can be null and can hold any type. If we wanted it to point to only `$node`s then we could write it as `(mut (ref null $node))`.

Let us create a value of type `$node`, `get` from it and `set` values in it.

```
(local $x (ref null $node))
;; Declare a variable that we will store our node in

(i32.const 42) ;; Load integer constant
(ref.i31)
;; Make it into an unboxed scalar
;; This integer is now compatible with references
;; This will be our root's $left

(ref.null $node)
(ref.null $node)
(i32.const 0)
(struct.new $node)
;; Creates a $node whose $left and $right are nulls, with a name 0
;; A reference to the structure above is now on the stack
;; This will be our root's right

(i32.const 1) ;; Name of our root node
(struct.new $node)
(local.tee $x)
```

This creates the binary tree in *Figure 1*.

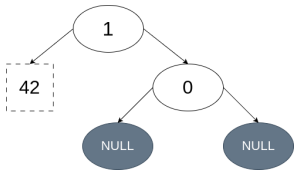


Figure 1: Binary Tree

`(local.tee $x)` sets the value of variable `$x` to our newly created binary tree and it also leaves that value on that stack instead of consuming it. Let us now get the name of our root node and change it.

```
;; The binary tree is on top of the stack
(struct.get $node $name)
;; puts 1 on top of the stack

(local.get $x)
;; The binary tree is back on top of the stack
(i32.const 100) ;; New name of root node
(struct.set $node $name)
;; Name of root node now changed to 100
```

This binary tree structure is going to be useful in the implementation of the SK Reduction Machine.

2.2 SK Reduction Machine

The SK Reduction Machine is an *abstract machine* that is used to implement lazy functional languages with no side-effects. It is the theoretical underpinning of MicroHs's runtime system.

Bracket Abstraction Programming language semantics are often defined in terms of *lambda calculus*. Lambda calculus terms are created using three rules - Abstraction, Application, and Variables. *SKI combinator calculus* removes the need for abstraction and variables. All SKI combinator terms are created using applications of SKI combinator terms. Other primitives (apart from SKI combinator terms) are added based on convenience. For example, one could encode numbers in terms of lambda calculus or SKI combinator calculus, but we choose to take them as primitives.

Terms of lambda calculus can be converted into SKI combinator calculus terms using an algorithm called *bracket abstraction*. There exist many versions of this algorithm, a simple one is as follows:

$$
\begin{aligned}
\lambda x.x &= I \\
\lambda x.c &= Kc && \text{(if } c \text{ does not depend on } x\text{)} \\
\lambda x.(y\ x) &= y && \text{(provided that } y \text{ does not depend on } x\text{)} \\
\lambda x.(y\ z) &= S(\lambda x.y)(\lambda x.z) && \text{(where the lambda terms are further recursively reduced)}
\end{aligned}
$$

Reduction Rules After converting all of our lambda calculus terms into SKI combinator calculus terms, we must evaluate these terms until they have been reduced to a primitive. This is where the reduction rules of SKI combinator calculus come in. These reduction rules can be thought of as *rewrite rules* that *simplify* a SKI combinator term into a smaller SKI combinator term. They are -

$$
\begin{aligned}
S\ x\ y\ z &= x\ z\ (y\ z) \\
K\ x\ y &= x \\
I\ x &= x
\end{aligned}
$$

In addition to the above three, more are introduced below. Theoretically, the S and K combinators suffice to represent all lambda calculus terms. In practice however, we introduce more combinators to reduce the number

of reduction steps. B and C are introduced by David Turner in his paper [4] and the remaining are used by MicroHs. These are added to MicroHs's backend based on experimentation and the set is quite *ad hoc*.

$$
\begin{aligned}
B\ x\ y\ z &= x\ (y\ z) \\
C\ x\ y\ z &= x\ z\ y \\
S'\ x\ y\ z\ w &= x\ (y\ w)\ (z\ w) \\
B'\ x\ y\ z\ w &= x\ y\ (z\ w) \\
C'\ x\ y\ z\ w &= x\ (y\ w)\ z \\
A\ x\ y &= y \\
U\ x\ y &= y\ x \\
Z\ x\ y\ z &= x\ y \\
P\ x\ y\ z &= z\ x\ y \\
R\ x\ y\ z &= y\ z\ x \\
O\ x\ y\ z\ w &= w\ x\ y \\
Y\ x &= x\ (Y\ x) \\
K2\ x\ y\ z &= x \\
K3\ x\ y\ z\ w &= x \\
K4\ x\ y\ z\ w\ v &= x
\end{aligned}
$$

Graph Representation The reduction rules are implemented using *graph transformation*. The graph represents an SKI combinator term. We start with our original SKI combinator term, which is represented as a graph. Step-by-step, starting with the *leftmost reducible expression*, we apply the aforementioned reduction rules as a graph transformation, until our graph is reduced to a single node. This graph representation and the transformation rules form the heart of our abstract machine. The machine can be seen as a substitution machine that substitutes parts of our expression with equivalent expressions.

Each node is a binary node where the `left` field is a combinator or a pointer to a graph that reduces to a combinator and the `right` field are its arguments. If a combinator takes multiple arguments then its nth argument is present on the right field of the nth node above it. To gain efficient access to the leftmost reducible expression and the arguments of a combinator, we maintain a *left ancestor stack* (LAS). Reducing expressions from the leftmost reducible expression is known as *normal*

order reduction. By maintaining the stack, we don't need to repeatedly travel down our tree to arrive at the leftmost reducible expression.

Primitive operations such as "plus" can also be implemented the same as a combinator; the only difference is that plus's arguments must be fully reduced before being applied to plus.

A single step of reduction in the case of the C combinator is shown in *Figure 2*. The dotted lines represent the change in the graph after a single step of reduction when the C rule is applied. The LAS (shown on the left) gives us efficient access to the arguments.

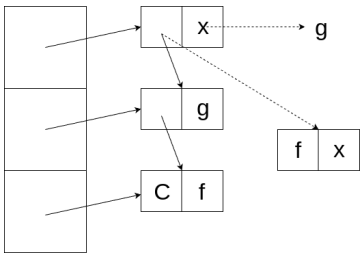


Figure 2: Reduction of C combinator term

Let us look at the graph transformation in detail by running through an example. Consider the following Haskell code,

```
let suc x = x + 1 in suc 2
```

This code is represented as

$$C\ I\ 2\ (plus\ 1)$$

in SKI combinator terms. The graph representation is shown in *Figure 3a*.

Then we apply the C reduction rule, transforming the graph into the graph shown in *Figure 3b*. This graph represents the SKI combinator term

$$I\ (plus\ 1)\ 2$$

The I reduction rule is then applied, giving us

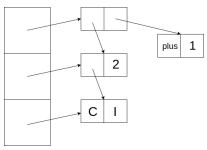
$$plus\ 1\ 2$$

shown in *Figure 3c*.

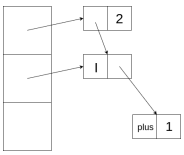
Finally, the rule for *plus* is applied. Plus is a primitive operation, so we make sure both the arguments of plus are fully reduced, which they are. Then we calculate the result and add an I combinator to our result to make sure the invariant of each node having a left field and a right field is maintained. This gives us the SKI combinator term

$$I\ 3$$

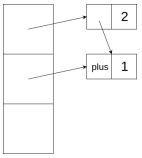
shown in *Figure 3d*. After this we stop our reduction since we have reached one node. The result is taken from this node.



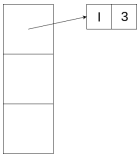
(a) Graph representation of $C\ I\ 2\ (plus\ 1)$



(b) After first step of reduction



(c) After second step of reduction



(d) After final step of reduction

Figure 3: All reduction steps

The representation for binary application nodes and the left ancestor stack that we will implement in our WasmGC RTS will directly follow from their description here.

Notable Combinator Reduction Rules

I combinator The I combinator's reduction rule states $I\,x = x$ which while easy to understand through a formula, does not talk about how we would reduce it in an SK combinator graph with binary nodes.

The value enclosed with an I could possibly be a `left` child of an node or the `right` child. When we apply I's reduction rule, the new value x must be set as which child?

We always set the value as the `left` child, any I-enclosed values that appear as the right child are left as is. This is because they will eventually have to be reduced due to strict primitives.

Strict Primitives Strict primitives are those operations that require all of their arguments to be fully reduced before they can be executed. An example would be ADD. When we come across a strict primitive, we are forced to reduce it's arguments fully before we can proceed further. We do this by recursively reducing nodes that are needed by our strict primitives.

This ties into the I combinator's reduction rule implementation. Everything on the `left` field has to eventually be reduced to a primitive operation or combinator. Everything on the right node has to eventually become a value. If we only reduce the I-enclosed values on the `left`, we eventually reduce to a strict primitive (even printing could be a strict primitive) thus forcing it's arguments, i.e., the `right` fields into reducing. Here, the arguments build their own LAS, where now the I-enclosed values appear on the left, and this continues our reduction.

Finally, with strict primitives, we can't just return a single value, because our program is held in binary nodes. So, our strict primitives always return values that are I-enclosed. Thus, through the entire process of reduction, we are constantly adding and eliding I combinators - these I-enclosed values can be thought of as *indirection nodes*.

Y Combinator The reduction rule for the Y combinator is $Y\,x = x\,(Y\,x)$, instead of implementing this directly we simply introduce a self-reference to the `appNode` containing the Y combinator in it's `right` field. [4]

2.3 MicroHs Compilation Pipeline

The SK reduction machine powers the runtime system (RTS) of MicroHs, but before our SK combinator terms are executed in the runtime system, there are several steps taken to reduce Haskell to SK combinator terms. This is shown in *Figure 4*.

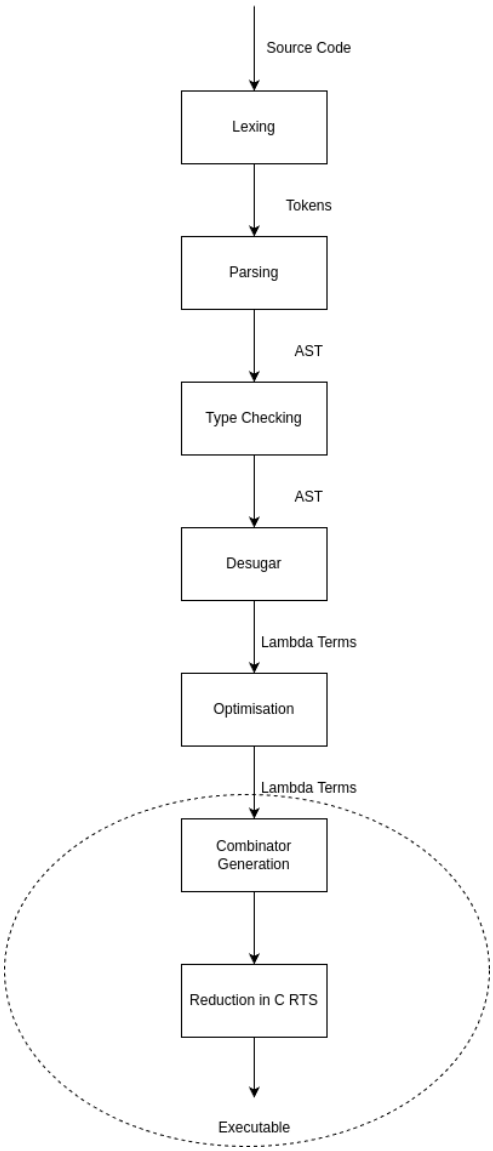


Figure 4: The MicroHs Compilation Pipeline

Haskell's source code is first broken down into tokens using the lexer. It is then parsed into an *abstract syntax tree* (AST) using *parser combinators*. This AST is type checked and desugared into lambda calculus. Opti-

misation is done to remove constants that are repeatedly passed during recursion and finally, we are left with lambda calculus terms that are ready to be converted into SK combinator terms.

The SK combinator terms are converted to C code by the *code generator*. The C code represents the SK combinator term's graph using an array of integers. These are processed by a C RTS. The RTS implements graph transformation to execute the code. The code generator and the RTS are encircled in *Figure 4*. It is these parts that are going to be our primary focus in this project.

3. Design

The design of the native WasmGC backend involves two parts:

1. **WasmGC RTS** - Generating a WasmGC RTS that reduces the encoded SK combinator graph.
2. **Extending MicroHs** - Making MicroHs emit WasmGC instructions that encode the SK combinator graph of our Haskell Program.

The RTS is an SK Reduction Machine created using a combination of handwritten Wasm code and code generated using our DSL. It has little support for literals other than integers smaller than 31 bits in size. The encoding step attempts to reuse and translate the combinators and tags used by the MicroHs C backend.

3.1 RTS Data Structures

Our WasmGC RTS is built around 4 main datatypes:

1. `appNode` - `appNode`s are binary nodes that contain two fields: `left` and `right`. The `left` field represents everything that will eventually reduced to a primitive operation, this includes combinators and intrinsic functions. The `right` field represents everything that can be used as a value.
2. `value` - `value`s follow a *tagged union* representation. We have a tag to represent the type of value and a payload that holds the actual data required by our RTS. Values are held in the `right` field of an `appNode`.
3. `stack` - The `stack` datatype is used to represent the left ancestor stack (LAS) which gives us efficient access to the arguments required by our primitive operations.
4. `table` - The `table` datatype is used to hold the `appNode`s of all function definitions used in a Haskell program. A function might call other functions as part of it's execution. The definition of the called function in terms of an SK combinator graph, which might or might not have already been reduced due to an earlier operation, can be found in the table. `appNode` references are held in the `table`.

In *Figure 5*, we see these data structures in use. The colour of a structure is used to depict the datatype that is allowed to inhabit it. `appNode`s allow `anyref`s because they can either be tags (`i31ref`s) or references to other `appNode`s. Unfortunately, Wasm does not have support for algebraic datatypes, so we must be content with `anyref`. Green is used to depict

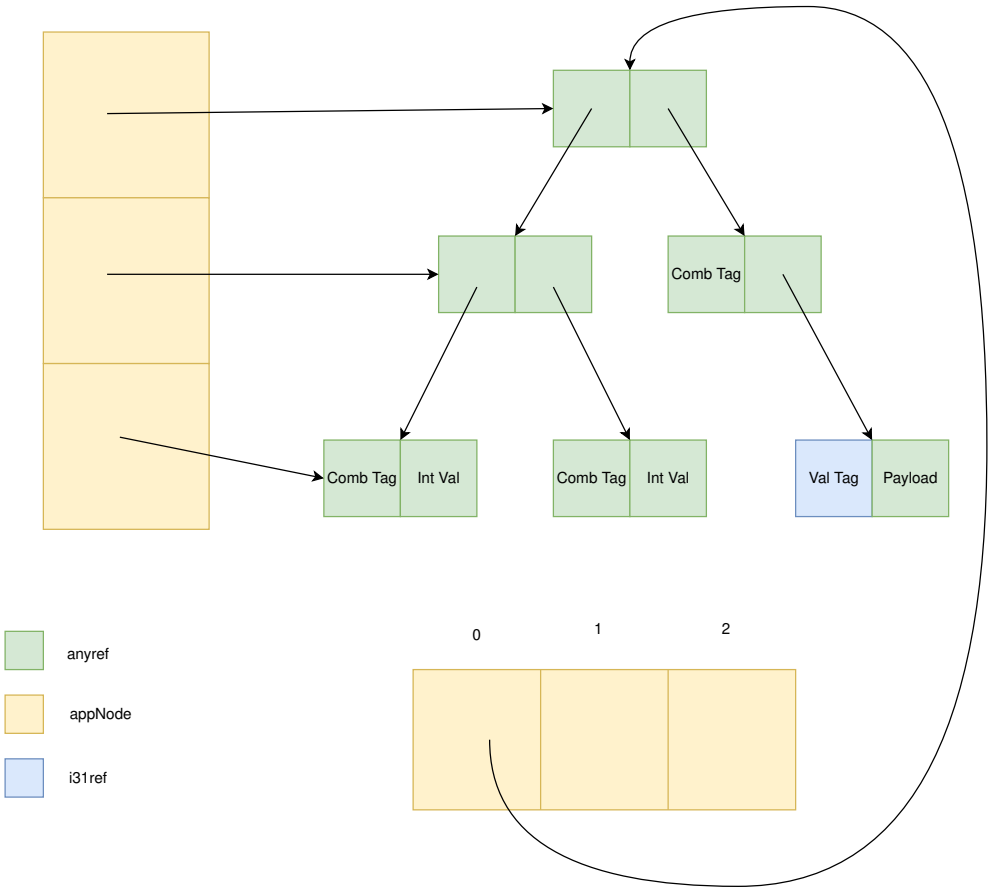


Figure 5: RTS Data Structures

`anyref`s which allow references to any type. This is used with `appNode`s as well as the `payload` part of a value. For the value-tag (as opposed to the combinator-tag), we can limit the values allowed to only `i31ref`s, and this is represented using the colour blue.

Finally, with the stack on the left and the table at the bottom (where we show the index, in which the index is the unique identifier for a function definition), we allow only references to `appNode`s.

Since we have only implemented `int`s in this thesis project, these values are treated specially by our RTS. No special tag-payload structure is required for them. We use an `i31ref` to represent `int`s as values. Other values are encoded as `(i32.const 42)(ref.i31)(i32.const 42)(ref.i31)(struct.new $value)`, i.e., any value that actually uses the `$value` type is unimplemented.

By using an `i31ref` we can avoid allocation of `int` values. The memory layout of `int` can be seen in *Figure 6* . An `i31ref` doesn't point to a GC object in the heap, instead it is a 31-bit integer that is stored unboxed in the reference itself. It's efficiency is why we have used them for representing both combinator and value tags.

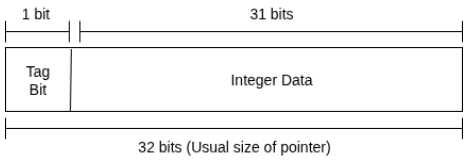


Figure 6: Memory representation of int values

Tables are used to take advantage of the self-optimizing properties of SK combinator graphs - Parts of the graph, once reduced, will not require recomputation if needed again.

Let's assume our Haskell program has function $f1$, $f2$, and $f3$. The SK combinator graphs, which are references, are stored in our table, indexed by the identifier of our function. The references held in this table are mutable and naturally support self-optimization.

Let's say $f1$ calls $f2$ as part of it's definition and the SK combinator expression for function $f2$ is $(((S\ K)\ K)\ 42)$. After calling it once, it reduces to $(I\ 42)$. If $f3$ then calls $f2$, $f2$'s value will be looked up in the table and a reference to $(I\ 42)$ will automatically be returned.

3.2 DSL for Code Generation

A big part of the RTS involves Wasm code for implementing reduction rules of each combinator. We often manipulate `appNode`s, and constantly use and modify the LAS. Writing this part by hand is both tedious and error-prone. Thus, we have designed a simple higher-level DSL that abstracts this and generates Wasm code automatically.

Reduction rules are built around metavariables. For example, in the reduction rule $C\,x\,y\,z = (x\,z)\,y$, we have 3 metavariables - x, y and z. The main observation is that we only need to know how many metavariables are used by a reduction rule and how these metavariables are manipulated to know what Wasm code needs to be generated.

Let us look at some common operations that we will often perform in the implementation of a reduction rule. These operations are not specific to Wasm but the usage of an SK Reduction Machine to implement graph reduction is a prerequisite.

1. Getting the *nth* ancestor of the LAS - We use the LAS for having efficient access to metavariables. The LAS points to `appNode`s; being able to immediately get an `appNode` that is n distance away from the top of the stack is highly useful.
2. Storing data into metavariables - Once we do have an `appNode`, we want to be able to store it's data, either from the `left` field or the `right` field, into a metavariable.
3. Setting `appNode`s - Overwriting a field of an `appNode` with the data stored in a metavariable. An example of this can be seen in the C reduction rule, where metavariable y will overwrite a field in an `appNode`.
4. Creating new `appNode`s - Simply overwriting an existing `appNode` is not enough for some rules, sometimes new nodes have to be created. In the C reduction rule, this can be seen when we use the metavariables x and z to create the new `appNode` representing $(x\,z)$.

If we were to represent the above operations using Haskell code, it would look like this

```
data GraphInstr = MkNode Comb Val
                | GetAncestor Int
                | Store Field Var
                | NodeSet Field Var
```

```
data Comb  = PrimComb String -- Primitive combinator or strict operation
           | CVar Var        -- Combinator created from data stored in Var
           | CRec GraphInstr -- Recursively create new combinator

data Val   = PrimVal Int     -- Create integer value
           | VVar Var        -- Same as CVar but for values
           | VRec GraphInstr -- Recursively create new value

data Field = LeftF | RightF
```

`GetAncestor` gets the *nth* argument starting from the top of the LAS. `Store` stores the data of a field (either `left` or `right`) in a variable. `NodeSet` sets a value into a field and `MkNode` is used to create new nodes. `Comb`s and `Val`s can be created using either primitives such as combinators or ints, and using variable identifiers.

Using these graph manipulating instructions (`GraphInstr`), let us look at two helper functions that would make encoding reduction rules even easier.

```
stores :: Int -> [GraphInstr]
stores i =
    let
        vars = map MV [X, Y, Z, W, V]
        f x  = [GetAncestor x, Store RightF (vars !! x)]
    in
        concatMap f [0..i-1]

modifyAncestor :: Int -> GraphInstr -> GraphInstr -> [GraphInstr]
modifyAncestor i ln rn = [ GetAncestor i, ln, NodeSet LeftF
                         , GetAncestor i, rn, NodeSet RightF ]
```

The function `stores` takes the number of metavariables required by a reduction rule and stores them in metavariables x, y, z, w and v. These metavariables are simply variabes of type `anyref` on the Wasm side. `modifyAncestor` takes the reduction rule itself and uses it modify the SK combinator graph as required by the reduction rule. `ln` and `rn` together make up the left node and the right node of the reduction rule respectively. We split them for convenience. `i` is the index of the LAS at which we would like the new nodes to be placed.

To encode the C reduction rule using our DSL, all we must do is

```
redRuleC :: [GraphInstr]
redRuleC = redRule n ln rn
```

```
    where
        n  = 3
        ln = MkNode (CVar (MV X)) -- MV = metavariable
                    (VVar (MV Z))
        rn = NodeSet RightF (MV Y)

redRule :: Int -> GraphInstr -> GraphInstr -> [GraphInstr]
redRule n ln rn = stores n ++ modifyAncestor (n-1) ln rn
```

Now we convert our `[GraphInstr]` into Wasm instructions, paste these instructions at appropriate parts of the handwritten RTS, for every reduction rule, completing the RTS. We do the pasting by splicing the handwritten RTS at annotations marked through comments (for example, `-- C Combinator Start` and `-- C Combinator End`), inserting generated Wasm code in between and voila, we have a fully functional RTS! Not quite.

Unfortunately, the above design, while a good start, does not do everything we would want it to do. Not every reduction rule can be encoded using the implementation above. Not only that, every time we create new nodes, we must also change our LAS to point to the new nodes. The actual implementation of the DSL does this automatically by statically looking at the `ln` part of our reduction rule and adjusting the pointers of the LAS - Thus, the current version of the `redRule` function is incomplete.

Nevertheless, the above principles and implementation explain the design behind our DSL well. In the next chapter, we will expand upon this to give a more accurate view.

4. Implementation

4.1 Expanding our DSL

We saw how `GraphInstr`s are a great way to automate the encoding of reduction rules, but they alone are not enough for our needs. We begin the explanation of our implementation by introducing the type `MixedInstr`. The idea behind `MixedInstr` is that we want to keep our `GraphInstr` type minimal, but at the same time there are cases where just having those 4 instructions is not enough. Using `MixedInstr`, we can inline Wasm code in our Wasm generating DSL.

```
data MixedInstr = GI GraphInstr | WI Instr
```

This might seem like an ugly solution to the problem, and it is, but one can see why we end up in this situation - If our RTS is a combination of handwritten and DSL auto-generated code, then our DSL, at times, might need to touch on the handwritten parts of the RTS, hence the introduction of inlined Wasm code.

4.1.1 Automatic LAS Modification

The first place we see the use of this in, is with the `modifyAncestor` function. The actual implementation looks like

```
modifyAncestor :: Int -> MixedInstr -> MixedInstr -> [MixedInstr]
modifyAncestor i ln rn = [ GI (Ancestor i), ln, GI (NodeSet LeftF)
                         , GI (Ancestor i), rn, GI (NodeSet RightF) ]
                          ++
                         map WI (lasModify i ln)
```

We have added the line `map WI (lasModify i ln)` to `modifyAncestor`. The function `lasModify` does the automatic fixing of the LAS when we add new nodes, i.e., new pointers to the new nodes are set in the LAS, and we automatically calculate and return the next index of the LAS that must be worked on for the next reduction step.

We do this by statically calculating the length of the left spine of our new node/graph (i.e. `ln`) that we have set in this rule. Starting from the root of the new node created for our current reduction rule, we keep going to the `left` field, setting these nodes' references into the LAS. The argument `i` that is passed to `modifyAncestor` is the index of the LAS that our new nodes will be set at. Thus, to calculate the return value of

the next index of the LAS that we must apply the reduction rule on, we calculate

$r = las - i + leftSpineLen$

where r is the new return value, las is the index of the current top of the LAS, i is the argument we pass to `modifyAncestor` and $leftSpineLen$ is the left spine length of `ln` (the other argument passed to `modifyAncestor`).

4.1.2 Y Combinator Another place where having access to `MixedInstr` is useful, is with the implementation of the the Y combinator. The Y combinator involves taking only 1 metavariable and making it the next tag to be processed with the argument to it, i.e., the `right` field a self-reference.

```
1 redRuleY :: [MixedInstr]
2 redRuleY = map GI (stores 1)                                    ++
3            [ GI (Ancestor 0), ln, GI (NodeSet LeftF),
4              GI (Ancestor 0), WI (LocalSet (MV Y)),
5              GI (Ancestor 0), WI (LocalGet (MV Y)), GI (NodeSet RightF),
6              WI (LocalGet LasIdx), WI (LocalSet ReturnVar) ]
7     where
8         ln = WI (LocalGet (MV X))
```

In line 2, we store data into 1 metavariable, i.e. x. Line 3 then sets this as the tag in the same node that we are processing, i.e., it sets the metavariable x in the `left` field of the current node. Line 4 sets the current node itself into another variable, and then in line 5 we set the `right` field of the node to the variable, thereby creating a self-reference. Finally in line 6, we return the new r, which remains the same.

4.1.3 Strict Primitives Finally, another interesting use case for having access to inline Wasm instructions is with the implementation fo strict primitives. When it comes to strict primitives, there are two things to keep in mind, we want to wrap our answer with an I combinator, and we also want to reduce the arguments needed by our strict primitives, before applying them. A classic example can be seen with the reduction rule implementation of `ADD`.

```
redRuleADD :: [MixedInstr]
redRuleADD = redRule' n ln rn
    where
```

```
        n  = 2
        ln = WI (prim "I")
        rn = map WI [ LocalGet (MV X), Call FnReduce,
                      LocalGet (MV Y), Call FnReduce,
                      I32Add, RefI31' ]
```

As you can see in `rn`, we can call the function `reduce`, that is in the handwritten part of the RTS, to be able to reduce our arguments fully before running them through the instruction `I32ADD`.

4.2 WAT for S Combinator Reduction Rule

We have seen how we can use our DSL to encode various reduction rules. It behooves us to see what the Wasm code for a reduction rule would look like. Let us look at the reduction rule for the S combinator. As a reminder, it is $S\ x\ y\ z = (x\ z)\ (y\ z)$.

The Haskell code for it is,

```
redRuleS :: [MixedInstr]
redRuleS = redRule n ln rn
    where
        n  = 3
        ln = GI (MkNode (CRef (MV X)) (VRef (MV Z)))
        rn = GI (MkNode (CRef (MV Y)) (VRef (MV Z)))
```

If we were write it by hand, the Wasm code would be

```
;; Set Metavariables
;; Automated by the stores function
(local.get $ascii)(i32.const 0)(i32.eq)
(if (then
(local.get $las)(local.get $n)(array.get $stack)
(struct.get $appNode $right)
(local.set $x)
(local.get $las)(local.get $n)(i32.const 1)(i32.sub)(array.get $stack)
(struct.get $appNode $right)
(local.set $y)
(local.get $las)(local.get $n)(i32.const 2)(i32.sub)(array.get $stack)
(struct.get $appNode $right)
(local.set $z)
;; New appNodes
;; Done using modifyAncestor
```

```
16 (local.get $las)(local.get $n)(i32.const 2)(i32.sub)(array.get $stack)
17 (local.get $x)(local.get $z)(struct.new $appNode)
18 (local.tee $temp)
19 (struct.set $appNode $left)
20 (local.get $las)(local.get $n)(i32.const 2)(i32.sub)(array.get $stack)
21 (local.get $y)(local.get $z)(struct.new $appNode)
22 (struct.set $appNode $right)
23 ;; Modify LAS
24 ;; lasModify does something similar
25 (local.get $las)
26 (local.get $n)(i32.const 1)(i32.sub)
27 (local.get $temp)
28 (array.set $stack)
29 (local.get $n)(i32.const 1)(i32.sub)
30 (local.set $r)
31 (br $combCase)))
```

In line 1 we check if we are in the S combinator case. We do this by looking at the tag at the top of the LAS (which is stored in the variable `$ascii`). If it is, we enter the `if` block. From line 4-12, we are setting the metavariables x, y and z. We look up the nth, $(n-1)th$ and $(n-2)th$ `appNode` pointed to by the LAS, look at it's `right` field and store it into the respective variables. The variable `$n` holds the index of the top of the LAS i.e., the length of the LAS.

Next, we must create the new `appNode` required by our reduction rule - $(x\ z)\ (y\ z)$. We first look up the `appNode` we would like as parent of our new `appNode`s. Our new LAS will be of length $(n-1)$ after the new `appNode`s have been placed, therefore the parents are at the $(n-2)th$ index. We get these `appNode`s, as seen in line 14 and line 18. Next, we construct our new `appNode`s as required by our reduction rule. $(x\ z)$ can be seen in line 15 and $(y\ z)$ in line 19. Finally, we set the pointers from the parent to the children in line 17 and line 20.

We are not done yet because for the next iteration of the `step` (which is used to implement a single instance of a reduction rule) function, we must modify the LAS so we point to the correct `appNode`, which in this case will be $(x\ z)$. We must also return the new length of the stack, which is stored in the variable `$r`. Finally, we break out of the block we are in and do a forward jump with line 28.

It is this snippet of code that our DSL automates the generation of, along with the other combinators used in our RTS. There are some differences

in the actual lines of code generated but in principle it does the same thing.

4.3 WasmGC RTS Outline

After the code generation of reduction rules in the WasmGC RTS using our DSL, and the encoding of our Haskell program into Wasm instructions, there is a two-step process that takes place to execute our Haskell program. This is very much akin to the SK combinator graph reduction described in the background section. We initialize the LAS and then apply the reduction rules of the top primitive operation or combinator to execute our program.

Let us look at some of the functions used in the WasmGC RTS's implementation. The RTS contains the SK Reduction Machine that executes our Haskell program. It is built on 4 functions:

1. A `createLAS` function that creates the LAS of our SK combinator graph
2. A `step` function that applies a single rewrite or reduction rule to our graph
3. A `reduce` function that continuously reduces our SK combinator graph until the end condition is reached, i.e., the LAS is of height 1 and the combinator on the `left` is I.
4. The `main` function which holds the SK combinator graph of our Haskell program

The SK combinator graph of our Haskell program encoded as Wasm code that is present in the `main` function is built using the methods described in the next section.

4.4 Wasm Encoding of Haskell Program

In the MicroHs compilation pipeline (*Figure 4*), we reduce our Haskell program into lambda calculus. After that, our program is converted into an SK combinator expression using bracket abstraction. Our implementation diverges from the original at this point of the compilation process. The SK combinator expressions are represented using the `Exp` datatype.

```
data Exp
  = Var Ident
  | App Exp Exp
```

```
  | Lam Ident Exp
  | Lit Lit
```

The most used constructor is `Lit` that stores various literals such as integers, doubles, and also our SK combinators. `Lam`s do not exist by the time bracket abstraction is done. The SK combinator calculus wants to remove variables introduced by `Lam`, hence everything is represented using only application. This is done using the `App` constructor.

The `Var` constructor is not used for variables introduced by `Lam`, instead it is used to represent the usage of another function in a given function's SK combinator expression. We do not inline the SK combinator expressions of functions that are dependencies of our given function. We do this to take advantage of *sharing*.

Thus, to encode our Haskell program, we must know how to encode an `Exp` into Wasm instructions.

4.4.1 Encoding `Exp` Let us look at how every `Exp` constructor is encoded in our WasmGC RTS. Combinator literals and integer values are encoded using `i31ref`s. Each combinator has a unique tag.

The difference between the `i31ref`s used to encode combinators and values is that they inhabit different fields. The former occupies (or will eventually occupy, after reduction) the `left` field and the latter, the `right` field.

`App`s are encoded recursively, the first `Exp` is converted into WasmGC instructions, then the second, followed by a `(struct.new $appNode)`, where the `appNode` type is defined as

```
(type $appNode
        (struct (field $left (mut anyref))
                (field $right (mut anyref))))
```

Finally, `Var`s are encoded by simply looking up the identifier in a table. Tables are an array of values of a particular reference type. To look up the 0th index, we do

```
(i32.const 0)
(table.get $fnDefs)
```

where `$fnDefs` is our table.

4.4.2 Table Initialization

Tables are declared with their size and the reference type they hold. Let's say our table size is 42, then our table declaration looks like

```
(table $fnDefs 42 (ref null $appNode))
```

While we know how `Exp`s are encoded, our program is not a single `Exp`. A program is a list of function definitions, therefore it is a `[Exp]`, one `Exp` per function definition. When encoding begins the tables are initialized with the `appNode` representing $(I\ I)$, as we encode our program, when we encounter a complete function definition, we replace the `right` field of that definition's table entry with an actual reference to the `appNode` that represents the current function definition.

For example, let's say function $f42$ is defined as $(((S\,K)\,K)\,42)$. Before we encounter the definition, the table `$fnDefs` has $(I\ I)$ at index 42. Every occurence to `Var 42` will become

```
(i32.const 42)
(table.get $fnDefs)
```

Later on, when we do encounter $f42$'s SK combinator expression, the reference stored in the table is modified with it's actual definition $(((S\,K)\,K)\,42)$. This works because the first function that we call during execution is the `main` function and that only happens after we declare all of it's dependencies. The function definitions are topologically sorted using *Depth-first Search (DFS)* to ensure this.

4.5 Summary

In this chapter, we saw how our DSL is used to generate Wasm code automatically for all pure combinators. We also saw what the generated Wasm code looks like and how it is part of a whole handwritten RTS. Finally, we saw how we encode Haskell programs that are converted to combinators into a Wasm SK combinator graph that is reduced by the RTS.

5. Evaluation

The main benefit of porting a runtime system to WasmGC is avoiding the inclusion of memory management code in the binary, reducing the binary size. A smaller binary size has the following advantages

1. Better user experience with web applications - Binary size is directly tied to initial load times. A Wasm binary has to be downloaded and parsed before it can be run. Lower latency increases user retention and also decreases user and environmental costs. A similar observation can be made for serverless/edge computing.
2. Embedded/IOT - With the WebAssembly Micro Runtime (WAMR)[5], one can run Wasm code in small footprint, high performance devices that require a sandbox. WAMR now supports WasmGC features.

Thus our concern primarily lies in the evaluation of binary size.

5.1 Methodology

We would like to measure and compare the binary size produced by the 3 different targets of the compiler -

1. The native C backend
2. The Wasm 1.0 backend (that uses `emscripten`)
3. The WasmGC backend that we implemented

We do this by compiling a corpus of programs to all 3 targets. Thankfully, the MicroHs compiler comes with it's own set of test programs[2] that we can use as a corpus. We want to get the smallest possible binary size from the C compiler as well as from `emscripten`, so for a fair comparison we want to pass the right compiler options. This is done by changing the `targets.conf` file that comes with MicroHs.

For the C compiler, we use the following compilation options: `-Os -ffunction-sections -fdata-sections -Wl,--gc-sections`.

- `-Os` tells the compiler to optimize for size rather than speed

[2]Specifically, the tests present in the MicroHs repository at commit f61410b846dcae51a9aa659eec40a5e9a998d3f7. The other two binaries have been compiled using commit 9c4ee662496fde8b0a948f89ab1ad73a30af4869, which is the latest commit at this time

- `-ffunction-sections -fdata-sections` puts function and data objects in their own section
- `-Wl,--gc-sections` tells the linker to discard unused data and funtions that live in their own sections

For the Wasm 1.0 binary, we use the following options: `-Oz -sSTANDALONE_WASM=1 -sEXPORTED_RUNTIME_METHODS=stringToNewUTF8 -sALLOW_MEMORY_GROWTH -sTOTAL_STACK=5MB -sNODERAWFS -sSINGLE_FILE -DUSE_SYSTEM_RAW -Wno-address-of-packed-member`

The only three options here that are different from the default are

- `-Oz`, which aggressively reduces binary size
- `STANDALONE_WASM=1` which generates a Wasm binary without any JS shims
- We also remove `-sEXIT_RUNTIME=1` which is present by default, to be able to compile our programs with the changed options

And for our WasmGC binary, we first compile all of the programs into a `.wat` file[3]. We then use `wasm-tools` (version 1.230.0) to validate and parse the `.wat` files into `.wasm` files. These files have the suffix `-wasmgc.wasm` to differentiate themselves from the Wasm 1.0 binary files.

5.2 Results

Of the 98 Haskell programs present, 8 failed to compile using our WasmGC backend. We then validated the `wasmgc.wat` files produced to find that 3 of them failed to validate. These 3 failed validation because the programs required support for `int`s greater than 31-bits which our backend lacks support for.

Figure 7 graphs a line graph of the binary sizes produced by compiling to our three targets. The straight lines parallel to the x-axis show us the size of the binary for the Haskell program

```
main = return
```

With this program, we aim to give a baseline binary size with (almost) only the RTS bundled into the binary, all of the rest of the programs are larger in size compared to this.

[3]Refer to the `Script.Try` module in the project's repository

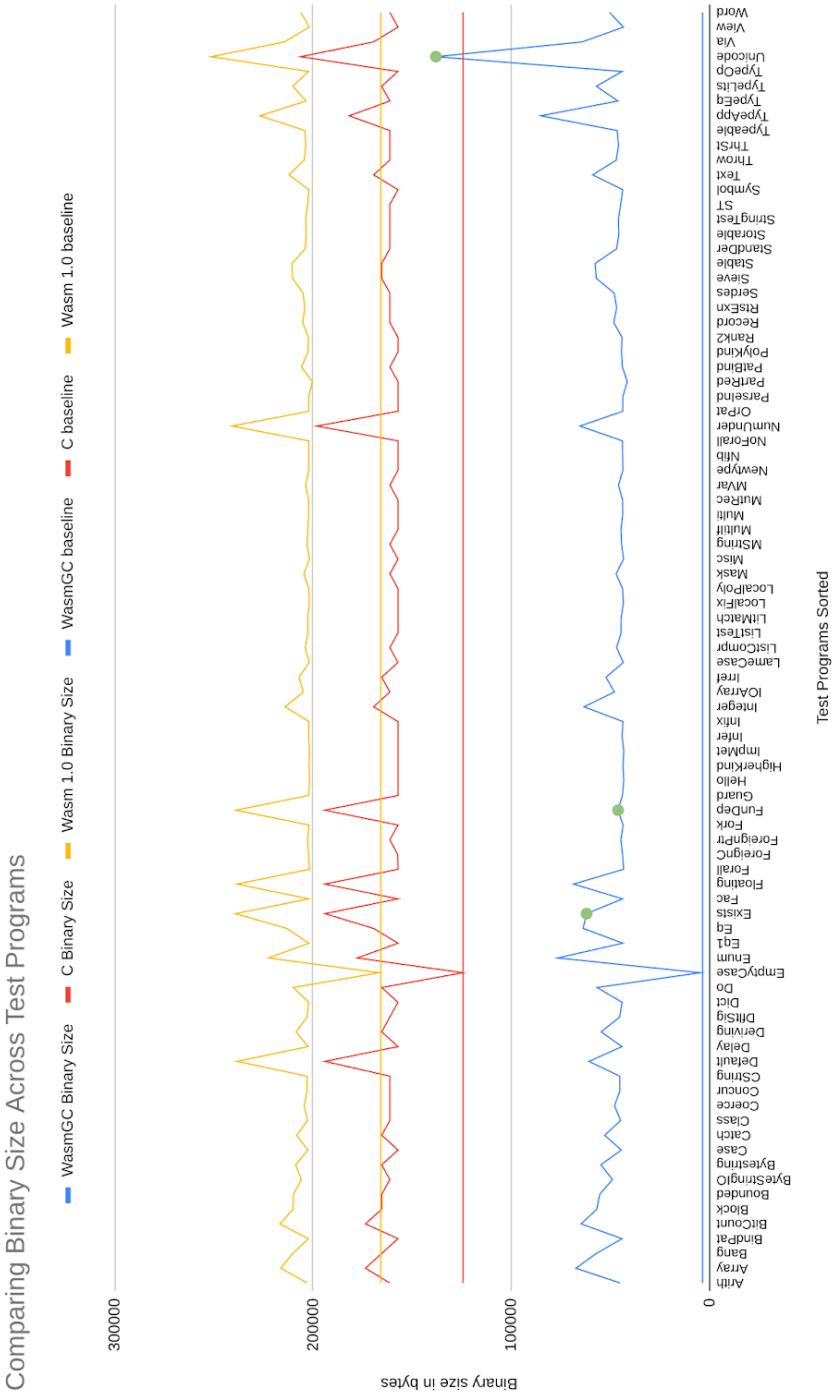


Figure 7: Jagged Line Graph of Binary Size

Targets	WasmGC	C	Wasm 1.0
Baseline binary size (in bytes)	3630	124408	165892

In *Figure 8*, we also show a grouped bar chart of the median binary size produced for each compilation target.

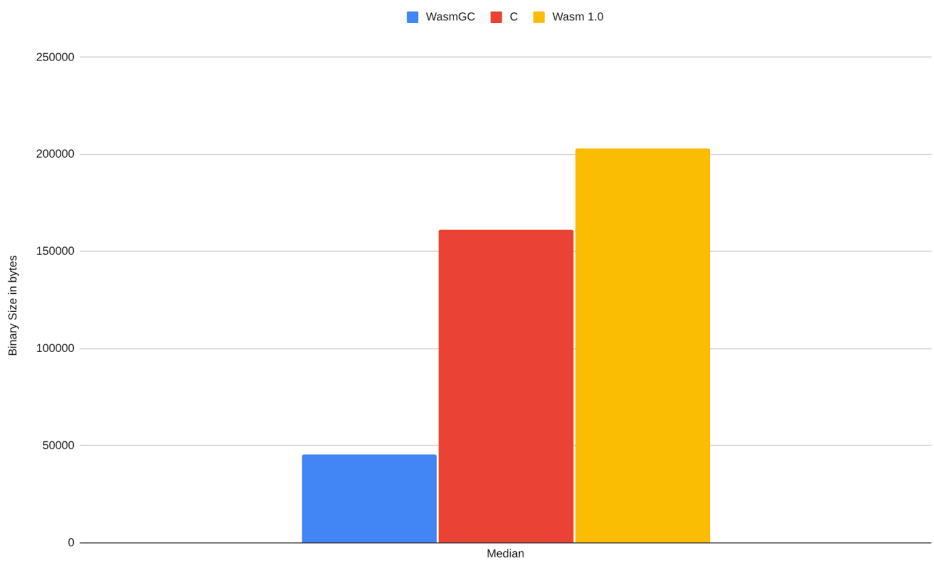


Figure 8: Bar chart of Median Binary Size

The astute observerer may notice that we have used a line graph (although jagged) for plotting discrete categories. This can be misleading but we want to emphasize and observe the jaggedness of the line graphs produced by our three compilation targets. For technical accuracy and posterity's sake, *Figure 9* contains a scatter plot version of our data.

5.3 Discussion

Our backend produces binaries that are categorically smaller than the C or the Wasm 1.0 binaries. This supports our initial hypothesis that native WasmGC binaries would be smaller. If our experiments showed that the binaries produced by the WasmGC backend were larger, this direction of research would not be worth pursuing, at least with the goal of minimizing binary size in mind.

Additionally, let's observe the line graph produced by our three compilation targets. The graph for the C and Wasm 1.0 backend are exactly

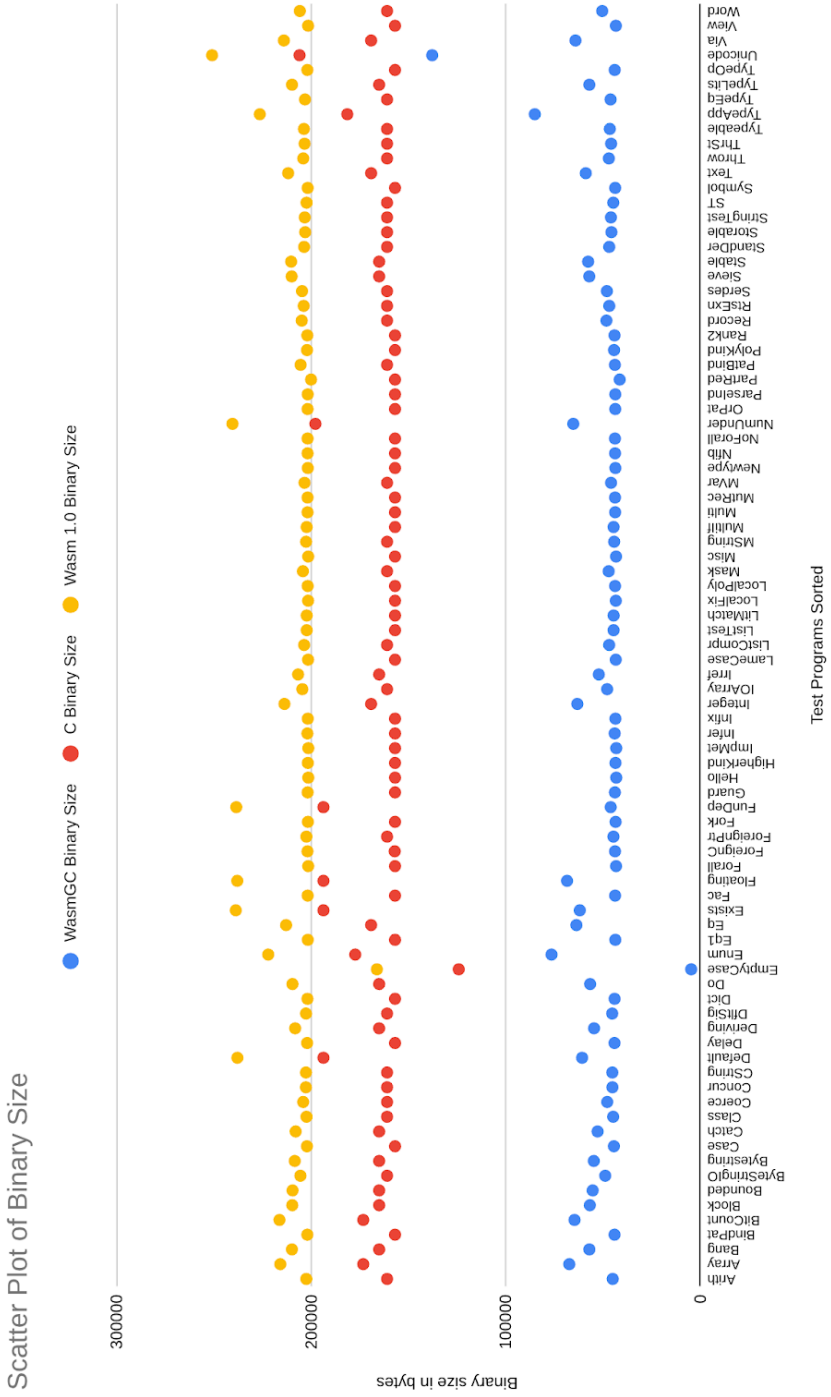


Figure 9: Accurate plot of Binary Size

the same. This makes sense given that the Wasm 1.0 backend does a one-to-one conversion of the C code into Wasm. Wasm 1.0 binaries have an additional overhead due to a lack of true dynamic linking. Common C libraries don't have to be included in the C binary, the same cannot be said about Wasm 1.0 binaries. `emscripten` also adds code for system call emulation which increases binary size.

While our native WasmGC backend produces the smallest binaries, one cannot trust this result to remain unchanged after we match the C backend in terms of feature support. Indeed, a major caveat of the data our backend has produced is that the backend's implementation remains incomplete. The program graph produced by the WasmGC backend has the same shape as the graph produced by the C backend but it is not populated by the same amount of tags or data that is present in the program graph of the C backend. Several features that a user might consider basic are deeply intertwined with several parts of Haskell's base library; the `Prelude` is needed by the simplest of programs and our binaries are not in a state that one would consider runnable due to their lack of support for the `Prelude`.

This raises a critical question: What is the basis for trusting the reported data? One could, for example, design a compiler that produces intentionally incomplete or non-functional binaries, yet report exceptional efficiency by virtue of the output's minimal size.

The answer lies in the jaggedness observed in the line graph indicating the binary size of programs produced by the WasmGC backend. They bolster our confidence in the progress made so far. For the most part, the data aligns with the other backends, i.e., the jaggedness of the graph remains the same at most data points Although the generated binaries are not yet fully functional, this alignment suggests a correct approach. We notice a few outliers, which are encircled in green.

Outliers The three programs we are going to discuss as outliers are `Exists.hs`, `FunDep.hs` and `Unicode.hs`. In the case of the former two, we have got a lower-than-expected binary size and in the latter, the opposite.

The hypothesis that the binary size is lower for `Exists` and `FunDep` because the program graph has a lot of missing data is falsified using an experiment involving static analysis. We note down the following for each program:

- The number of `appNode`s in each program - `anCount`

- The number of values that haven't been implemented (by searching for `(i32.const 42)(ref.i31)(i32.const 42)(ref.i31)(struct.new $value)`, which is the code our backend generates for unimplemented values) - `vCount`
- The number of unimplemented tags, that are not a part of an `appNode` that already contain an unimplemented value[4] (because these have already been counted) - `tCount`

Using them, we calculate 3 metrics:

- $\frac{vCount}{anCount}$
- $\frac{tCount}{anCount}$
- $\frac{vCount+tCount}{anCount}$

If the reason for the smaller binary size were unimplemented tags or values, then we would expect at least one of these 3 metrics to be high in said programs, but we noticed no such correlation.

The bigger factor in these outliers happens to be the compilation process itself. We confirm this by calculating $binFactor = \frac{wasmSize}{watSize}$, where $wasmSize$ is the size of the compiled Wasm binary and $watSize$ is the size of the respective `.wat` file, and indeed we find that `Unicode.hs` has the highest $binFactor$, `Exists.hs` has a lower $binFactor$ than `Eq.hs` causing the dip in the graph, and `FunDep.hs` has the 7th smallest $binFactor$. An interesting question to explore is why the other compilation targets don't have a similar factor in reduction of size, we leave this as future work.

[4]This can be done using a negative lookahead (`?!`) with a regular expression

6. Related Work

6.1 MicroHs Internals

We elucidate MicroHs' C backend in order to compare and contrast it with our own.

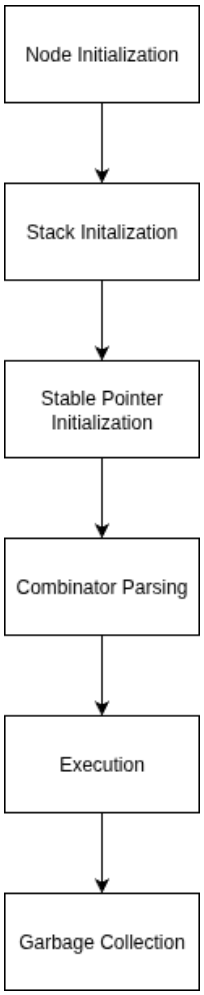


Figure 10: Pipeline of the C RTS

The C RTS begins by initializing commonly used nodes, such as the ones for the SKI combinators, so one can reuse the same node for every instance of a combinator. Our RTS unfortunately does not do this, we create a fresh reference with the same tag every time a combinator is used.

Stable pointer initialization and garbage collection are not needed by our RTS, so we can skip these. The C RTS uses a *mark-scan* garbage collector

and care has to be taken to not garbage collect FFI nodes. Nodes in the C RTS have a complicated structure[5]:

```
typedef struct PACKED node {
  union {
    struct node *uufun;
    intptr_t     uuifun;
    tag_t        uutag; // LSB=1 indicates that this is a tag
                        // LSB=0 that this is a T_AP node
  } ufun;
  union {
    struct node    *uuarg;
    value_t         uuvalue;
#if WANT_FLOAT32
    flt32_t         uuflt32value;
    uint32_t        uuint32value;
#endif  /* WANT_FLOAT32 */
#if WANT_FLOAT64
    flt64_t         uuflt64value;
#endif  /* WANT_FLOAT32 */
#if WANT_INT64
    int64_t         uuint64value;
#endif  /* WANT_INT64 */
    const char     *uucstring;
    void           *uuptr;
    HsFunPtr        uufunptr;
    struct ioarray *uuarray;
    struct forptr  *uuforptr;      /* foreign pointers and byte arrays */
    struct mthread *uuthread;
    struct mvar    *uumvar;
    struct weak_ptr *uuweak;
  } uarg;
} node;
```

Our node structure is far simpler, although whether it will stay as such after we add more runtime support for various features remains to be seen. In particular, MicroHs uses the same node structure for both values as well as tags, whereas we propose a separate type (`$value`) for our data/values.

By emitting WasmGC code that builds out our graph directly, we get to skip the parsing step entirely. The MicroHs frontend produces a

[5]Taken from commit 9c4ee662496fde8b0a948f89ab1ad73a30af4869

combinator file (`.comb`) that the C RTS has to parse first to assign tags and create the graph. It is then the execution step, that does the graph reduction and IO operations. For both reading the combinator file to build the graph and to reduce the graph, a stack is used. This is the one we see initalized in *Figure 10*. The execution stage handles multiple threads and takes care of exceptions - Neither of which we support.

Finally, we save function definitions in a table; while the C RTS uses a table to build the complete program graph, this is only temporary. The whole program is represented as one big graph with no indirection through a table. Using a table makes our implementation easier and cleaner but we foresee problems during garbage collection as references in a Wasm table never get garbage collected. Finally, given that indirection nodes play such a crucial role in the RTS and are constantly added and elided, the C RTS treats them specially to process them early. We do no such thing, simply reusing the I combinator.

6.2 The WebAssembly Landscape

The development of Wasm backends for functional languages has seen recent interest, with projects exploring various approaches to bridging the gap between high-level abstractions and the low-level Wasm target. For instance, in *Wasocaml: compiling OCaml to WebAssembly* [6], the Wasocaml project demonstrates the compilation of OCaml, a strict functional language, to Wasm. This work utilizes Flambda, an intermediate representation akin to lambda calculus, which is subsequently translated to Wasm. This approach highlights the viability of using intermediate representations for efficient Wasm compilation. In contrast, the Glasgow Haskell Compiler (GHC) employs a different strategy, as detailed in *Beyond Relooper: Recursive Translation of Unstructured Control Flow to Structured Control Flow* [7]. GHC translates Haskell's control flow graph, represented in Cmm, to Wasm. This process involves the removal of goto statements, a necessity due to Wasm's structured control flow requirements. Our work, which leverages SK combinators, avoids this specific challenge.

MicroHs: A Small Compiler for Haskell [8] introduces a small compiler for Haskell, serving as a direct point of reference for our work. It explains the compilation pipeline that we extend with a WasmGC backend. This extension is particularly relevant given the performance advantages demonstrated by WasmGC in *CertiCoq-Wasm: A Verified WebAssembly Backend for CertiCoq* [9]. This work provides empirical evidence that

WasmGC can lead to smaller binaries and superior performance compared to traditional Wasm 1.0.

Alongside performance considerations, security remains a critical aspect of Wasm development. *Everything Old is New Again: Binary Security of WebAssembly* [1] highlights potential security vulnerabilities in Wasm 1.0, particularly concerning the transference of insecure C/C++ code without traditional mitigations. The paper suggests that the introduction of multiple memories and the GC proposal will bolster Wasm's security. In a related vein, *MSWasm: Soundly Enforcing Memory-Safe Execution of Unsafe Code* [10] proposes the use of Handles, or fat pointers, to enforce memory safety and address Wasm 1.0's security concerns. However, these Handles lack type information and are not currently supported by mainstream browsers or Wasm runtimes.

7. Limitations and Future Work

The limitations of this thesis project can be understood at a glance by looking at an excerpt of the combinator file generated for the Haskell program

```
main = print 42
```

which generates the combinator file

```
v8.3
466
A _388 _464 @_191 _435 @_169 #42 @@@:465 @
A _130 _394 _463 @@_135 _464 @@_137 _464 @@:464 @
A _126 P _436 @_442 @@_443 _463 @@_444 _463 @@_182 _463 @@_183 _463 @@_46 ...
A B S I @@B Z @B B P _50 _447 @_448 @@@@B Z @B Z @C B @B B S U P _164 _16 ...
A S' S' P @@_292 @_460 @:461 @
A C S' S' @C _78 _81 @@#0 @@B P _41 @@Z B _460 @C _190 _96 @@#1 @@@@@@I   ...
A C' C @B C C' Y @B B S' C @B C S' @@B C P @@C O @K @@@@@@C' C'B @B B C   ...
A Y B S P @@C'B B @C _42 @@@@_41 @:458 @
A _126 P _456 @_288 @@_173 @_290 @_182 _457 @@_183 _457 @@_185 _457 @@_1  ...
A _127 P _254 @_235 @@B _170 @_275 @@:456 @
A B C S' C @B C' C @S' S' B @@B S' S @B S' S @B S C _427 @_432 @@@C'B B   ...
A P _357 fromUTF8 "last: empty list" @@@C' S @U @Z Z _454 @@@@:454 @
A _164 _163 @O #2 @K @@:453 @
A C' C @_462 @K @:452 @
A U A @:451 @
A B B U @@S' C' C'B @@B B B P @@@B B _405 @@_413 @@@_405 @@:450 @
A Y B B P _151 @@@B S' S' S @@B B U @@B B _152 @@C P @@@@@B B B Z @@@B B  ...
A P #3 @I @:448 @
A _51 P _445 @_446 @@_140 _447 @@_156 _447 @@_158 _447 @@:447 @
A _130 _353 _446 @@U S S' S' @C < @#3 @@C S' S' @C < @#4 @@C C' S' @C < @ ...
A _52 _351 fromUTF8 "Control.Exception.Internal" @@fromUTF8 "ArithExcepti ...
...
```

The combinator file tells us that our Haskell program (`print 42`) requires 466 function definitions. After using `gdb` to dynamically analyze the tags being processed by the C RTS, we discover that most programs use more than 50 unique tags in it's reduction process. Around half of these are RTS primitives that are not combinators or strict primitives, and this is where the crux of the challenge lies.

Even small Haskell programs require a lot of interconnected components to work together and everything in the C RTS is intertwined. It is hard to decouple what you want from what you do not think you need. Our attempt at dealing with this challenge was to reduce the scope of the WasmGC RTS, aiming to only support programs with integers in them. When that failed, we decided on a single benchmark program. This program would be the only program we would support but it would have still been useful, in comparing the runtime performance of our backend with the native one. Finally, we even attempted to run programs using just Church numerals (which only require support for SK combinators and their application, which we did have), but due to Haskell's laziness, these programs did not go through reduction without bringing in elements from the `Prelude`.

We are unable to run our binaries, thus a runtime and memory performance comparison is rendered moot. There are other smaller areas of improvement, such as implementing our own version of node initialization, so we can reuse nodes as often as possible (like MicroHs). `eval` being used recursively for strict primitives is also not optimal; in the future we would like to add special tags so our strict primitives reduce the same way as the others (also akin to MicroHs).

We have ideas for how we would like to implement some of the unimplemented tags and values. Let us take the tag `T_ARR_ALLOC` as an example. This tag is created when the RTS sees that the primitive "A.alloc" is present in the combinator file. Reduction/execution code can then can be generated from the following Haskell code:

```
redRuleArrAlloc :: [MixedInstr]
redRuleArrAlloc = redRule' n ln rn
    where
        n  = 2
        ln = WI (prim "P")
        rn = map WI [ prim "ARR",
                      I32Const 0,
                      LocalGet (MV Y), -- array is going to be filled with Y
{-of size X -}         LocalGet (MV X), Call FnReduce, ArrayNew IOArrayType,
                      StructNew ValueType ]
```

On the left node we assign the P combinator, following the implementation from MicroHs, and on the right node, we add a new tag `ARR` that represents the tag of the tagged union representation of values that are arrays, the rest of the code in the list creates a WasmGC array that is stored in the

`$payload` field of the `$value` type. The payload field is of type `$anyref`, so this works.

Other valid improvements that can be made are to the DSL itself. We got away with keeping the DSL untyped because we knew after the generation of WasmGC code, we could run it through a Wasm validation process, nevertheless, having a typed DSL that tells you if you have written invalid code during compilation itself or through an LSP sounds like a better programming experience.

The current strategy of generating the code for the reduction rules and primitives by splicing the handwritten RTS works, but is quite brittle. It would be great to be able to generate the entirety of the RTS in one go. We made the choice of splicing because we wanted to keep our DSL and Wasm AST minimal, in retrospect, even just storing the rest of the RTS as strings would have been a better option.

Finally, with the current state of WASI and the WebAssembly ecosystem, it is impossible to implement some language features. An example of this can be seen with multithreading. Since threads are not supported by WebAssembly yet, we cannot port them to our RTS. The C RTS benefits from the standardization of C and can make C-calls to implement features like `print`, it is not so straightforward with Wasm, until more standardization is established.

8. Conclusion

Our work contributes to the ongoing efforts in developing secure and efficient Wasm backends for functional languages, building upon the insights gained from various approaches. Our results show promise - The preliminary data indicates a vastly smaller binary size, and the graphs indicate that we are headed in the right direction. Despite how labour intensive creating a native Wasm RTS can get, we conclude that if one has the need for minimizing their binary size, it is worth it.

While the WebAssembly ecosystem is still nascent in some ways, we remain optimistic that the situation will improve.

The choices we made in this thesis that we are happy with are:

- The usage of the DSL - Even an untyped DSL such as ours is more friendly than writing macros in C.
- Emitting Wasm code directly - Instead of parsing a combinator file like the C RTS, we save on both binary size and runtime performance by moving the encoding and building of the program graph into the compiler.
- Running static analysis scripts - Decompilation of a Wasm module into a `.wat` file is not something we had to do in this project, but Wasm's ability to be easily decompiled is great for being able to perform experiments and test hypotheses on our Wasm binaries.

We noted about how one could continue implementing the tags established by the C RTS, to progress with the WasmGC RTS. Another direction to consider is to not use the combinator file generated by MicroHs directly. Since everything is so interdependent, an investigation into how much one could remove from the combinator file and change the design of RTS primitives, so they are more aligned with WasmGC primitives, is worth pursuing.

This change can happen if we stop thinking about Wasm as a second-class citizen in terms of a compilation target. There is merit to the idea that some languages should be designed with Wasm in mind.

9. References

[1] D. Lehmann, J. Kinder, and M. Pradel, "Everything Old is New Again: Binary Security of WebAssembly," in *29th USENIX Security Symposium (USENIX Security 20)*, USENIX Association, Aug. 2020, pp. 217–234. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann

[2] "A New Way to Bring Garbage Collected Programming Languages Efficiently to WebAssembly." Accessed: Feb. 27, 2025. [Online]. Available: https://v8.dev/blog/gc

[3] "WasmGC Backend." [Online]. Available: https://github.com/apoorvaanand1998/Thesis-MicroHs-WasmGC

[4] D. A. Turner, "A new implementation technique for applicative languages," 1979, 1. doi: https://doi.org/10.1002/spe.4380090105.

[5] "WAMR." [Online]. Available: https://github.com/bytecodealliance/wasm-micro-runtime

[6] L. Andres, P. Chambart, and J.-C. Filliatre, "Wasocaml: Compiling OCaml to WebAssembly," in *IFL 2023 - The 35th Symposium on Implementation and Application of Functional Languages*, Aug. 2023. Available: https://inria.hal.science/hal-04311345

[7] N. Ramsey, "Beyond Relooper: Recursive Translation of Unstructured Control Flow to Structured Control Flow (Functional Pearl)," *Proc. ACM Program. Lang.*, vol. 6, Art. no. ICFP, Aug. 2022, doi: 10.1145/3547621.

[8] L. Augustsson, "MicroHs: A Small Compiler for Haskell," in *Proceedings of the 17th ACM SIGPLAN International Haskell Symposium*, Association for Computing Machinery, 2024, pp. 120–124. doi: 10.1145/3677999.3678280.

[9] W. Meier, M. Jensen, J. Pichon-Pharabod, and B. Spitters, "CertiCoq-Wasm: A Verified WebAssembly Backend for CertiCoq," in *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs*, Association for Computing Machinery, 2025, pp. 127–139. doi: 10.1145/3703595.3705879.

[10] A. E. Michael *et al.*, "MSWasm: Soundly Enforcing Memory-Safe Execution of Unsafe Code," *Proc. ACM Program. Lang.*, vol. 7, Art. no. POPL, Jan. 2023, doi: 10.1145/3571208.

Leave the pain behind and let your life be your own again. There is a place where all time is now, and the choices are simple and always your own.

– Robin Hobb