Master's Thesis Software Science

Efficient WebAssembly GC Integration for a Functional Graph IR Language

Improving performance of the Clean-LLVM code generator by reducing memory operations

Dante van Gemert
dante.vangemert@ru.nl
s1032684

30 January 2026

First supervisor
dr. Mart Lubbers

Daily supervisor
dr. Steffen Michels (TOP Software)

Second reader
prof. dr. Sven-Bodo Scholz

TOP Software

Radboud Universiteit

Abstract

When using the functional programming language Clean on the web, the existing ABC interpreter for WebAssembly has poor performance for non-trivial programs. Instead, we work on Clean-LLVM, a code generator for Clean aiming to be more maintainable than the current one by using the LLVM compiler toolkit to generate target-specific assembly code and executables. At the same time it aims to achieve faster execution for WebAssembly than the interpreter. We integrate a garbage collector into Clean-LLVM, implementing solutions for WebAssembly's restrictive semantics. We then implement several improvements to the performance of this GC integration. Our generated Wasm code ends up running multiple times faster than the interpreter, and the LLVM output for native x86-64 code comes very close to the original compiler's speed.

Thank You

As you know, such a big project as a Master's thesis is not possible without help. Luckily, I got a lot of help—otherwise you would not be reading this. These awesome people deserve a thank you for making this a success.

Firstly, I want to thank my first supervisor, Mart Lubbers. For providing fortnightly feedback on my writing, having conversations about typography, and giving me confidence that everything is going to work out.

I am grateful to Steffen Michels for helping me with the practical side of things, aiding me in understanding Clean and Clean-LLVM, and also for his feedback. TOP Software has been a nice place to do both my research internship as well as this thesis, for which I also want to thank Rinus Plasmeijer.

Thank you to Camil Staps for taking the time to answer my questions about the Clean-LLVM project and the ABC interpreter, and to John van Groningen for his extensive explanations of the Clean compiler's internals.

I also thank Sven-Bodo Scholz for being the second assessor of my thesis.

During the months of working on my thesis, I have often sat next to Paul Tiemeijer, who kept me company and with whom I had good chats about programming language design. It was nice having someone to complain to about LLVM, too.

Although they understand not much of what I do (rightfully so!), my parents have supported me in other ways. They deserve a big thanks, and a warm hug.

Lastly, I want to thank you![1] For reading. Hopefully you find it interesting.

It's tempting to linger in this moment, while every possibility still exists.
But unless they are collapsed by an observer, they will never be more than possibilities.

— Solanum, Outer Wilds[2]

[1] If you are already mentioned above, thank you twice, I guess.

[2] I include a quote from a character in the game Outer Wilds at the start of each chapter.

Contents

1 Introduction 1
- 1.1 Background 2
  - 1.1.1 Clean and ABC 2
  - 1.1.2 LLVM 4
  - 1.1.3 WebAssembly 5
  - 1.1.4 Copying Garbage Collectors 6

2 Clean-LLVM 8
- 2.1 Pipeline Overview 8
- 2.2 Stack Simulation 9
- 2.3 Register Pinning 10
- 2.4 Node Entrypoints for Wasm 11
- 2.5 Descriptor Contents 11
- 2.6 Missing Functionality 11

3 Basic GC Integration 12
- 3.1 Custom Shadow Stack 13
- 3.2 GCRoot 13
- 3.3 Statepoints 14
  - 3.3.1 LLVM Pass Pipeline 15
- 3.4 Wrap-Up 15

4 Performance Improvements 16
- 4.1 Benchmarking 16
  - 4.1.1 Benchmark Tool 16
  - 4.1.2 Benchmark Programs 17
  - 4.1.3 Collection, Caching and Consistency 18
- 4.2 Static Nodes 20
  - 4.2.1 Benchmark Results 20
- 4.3 Alias Metadata 21
  - 4.3.1 Benchmark Results 21
- 4.4 Lazy Restoring 22
  - 4.4.1 Benchmark Results 22

4.5 Spilling After Optimisation 22
4.5.1 Benchmark Results 23
4.6 Results 23
4.6.1 Astack 24
4.6.2 List 25
4.6.3 Binarytrees 26

5 Related Work 27
5.1 WebAssembly for Functional Languages 27
5.2 Garbage-Collected Languages Using LLVM 28

6 Conclusions 29
6.1 Future Work 29

Bibliography 31

A Benchmark Programs 34
A1 `binarytrees` 34
A2 `list` 36
A3 `astack` 37

1 Introduction

I admire your curiosity, friend. Let's find out together.
— Solanum, Outer Wilds

The web is now more used than it has ever been. Many things that used to be installed as a program on a user's computer now run as a web app in the browser. From basic functionality like e-mail, calendar and instant messengers, to word processors, spreadsheets, code editors and even professional applications like 3D modelling software. The web with HTML, CSS and JavaScript has become such a popular platform that it has also been used outside the traditional environment: some installable programs are just websites shipped with a browser engine, LG's WebOS smart TV apps use it, and even SpaceX's space capsule control interface is made using web technologies.[1]

The introduction and development of WebAssembly (Wasm) [15] has made it feasible to use languages other than JavaScript for web development, without explicit browser support for that language. Wasm is, as the name suggests, a kind of assembly language for the web: any language (both high level and low level) can use Wasm as a compilation target. Recently, Wasm received support for proper tail calls. These are essential for functional languages, which do not use loops but recursion for iterating.

One project in a functional language that can benefit from Wasm is of special significance for this thesis: iTasks [24] is a framework for interactive web applications. It is the go-to basis for graphical and interactive programs for Clean [10], a pure and lazy functional programming language. iTasks is based on the paradigm of Task-Oriented Programming (TOP) [25], revolving around the concept of *tasks* as the primary abstraction. These tasks run server-side, but they also contain small pieces of Clean code running in the browser for creating the HTML elements of the UI. When these tasks interact with each other, an event gets sent to the server, which calculates how the UI should change in response. This back-and-forth incurs a hit in UI responsiveness, even though it is not always necessary. Large parts of an iTask application could theoretically run on the client side, which would not only improve responsiveness, but also availability, especially in case of a bad internet connection [31].

The small pieces of Clean code from iTasks running in the browser currently run in an interpreter [28]. However, doing anything computationally complex on the client side still requires writing JavaScript, since the interpreter then becomes prohibitively slow. To fix this, we work on Clean-LLVM [1], an alternative code generator for Clean that uses the LLVM [19] compiler toolkit to generate both WebAssembly code as well as executables for traditional (native) targets.

However, since any (untrusted) webpage can run Wasm code, it needs to have a much more secure design than for example x86-64 assembly. The result of that is that Wasm is more restrictive, and it does not support certain constructions that are used by the original Clean compiler and many other language implementations. One of the things which is trickier in Wasm is integrating a garbage collector (GC).

[1] https://www.reddit.com/r/spacex/comments/gxb7j1/comment/ft62781 (accessed: Nov. 18, 2025)

For most languages,[1] GCs look through the call stack to find all the pointers to the heap that are live; all other heap values are then discarded by the GC. However, this walking through the call stack is not supported in WebAssembly. In this thesis, we explore and evaluate ways to efficiently integrate a garbage collector into Clean-LLVM, with a focus on WebAssembly performance. While this case study works with Clean and its graph rewriting intermediate language, findings from this research may be applicable to other languages that use graph rewriting, or any other functional language.

The rest of this introductory chapter provides some preliminary knowledge in § 1.1. We introduce the Clean-LLVM project in chapter 2. In chapter 3, we look at some strategies for integrating a garbage collector into LLVM for Wasm, and we choose two for the rest of the thesis. After that, chapter 4 introduces a number of improvements to the performance of the chosen integration methods, along with a benchmark tool used for evaluating and comparing these improvements. Chapter 5 goes over previous publications that are related to this work, and we conclude the thesis and provide suggestions for future work in chapter 6.

1.1 Background

This section provides some knowledge required to understand the rest of this thesis. We explain Clean and its ABC bytecode (§ 1.1.1), LLVM (§ 1.1.2), WebAssembly (§ 1.1.3) and copying garbage collectors (§ 1.1.4).

1.1.1 Clean and ABC

Clean[2] [10] is a pure and lazy functional programming language developed at the Radboud University. The first version dates back to 1987, just a few years before the introduction of Haskell which bears many similarities. Listing 1 shows a Clean implementation of the `nfib` function, a simple test program that returns the number of calls that happened during its evaluation. We use it here as a running example to show the various languages involved in the process of generating Wasm from Clean.

Clean programs are executed using graph rewriting. It compiles to ABC bytecode [18], a stack-based intermediate language with special support for this graph rewriting. The ABC machine uses three stacks: **A** (argument), **B** (basic value) and **C** (control). The **B** stack contains booleans, characters and integers, among others; each of these values fit within one machine word. The control stack is for return addresses and such. In this thesis, we look at the **A** stack, which is where references to the heap reside. The heap is a region of memory where we can freely allocate and de-allocate values without being bound to the semantics of a stack.

During compilation, each Clean function gets turned into one or more ABC entrypoints (or zero, if the function is not used at all). Each kind of entrypoint has its own usage: the **strict** entrypoint contains the actual implementation, the **evaluate arguments** entrypoint is used for arguments that are possibly not yet evaluated, the **node** entrypoint is for evaluating thunks (sometimes referred to as closures), and the **lazy** entrypoint is used when currying. We only encounter *strict* and *node* entrypoints in this thesis.

We return to the `nfib` example program. The generated ABC code in listing 2 contains two labels: the strict function entrypoint `s2` and the jump target `else.1`. Lines starting with a dot (`.o`, `.r` and `.d`) are annotations. They indicate the number of values on the **A** and **B** stacks, as well as the types of the **B** values (`i`, an integer, in this case). The first part in the code (between `s2` and `else.1`) corresponds to the first line of the Clean function: lines 4 to 6 (highlighted blue) are the comparison `n < 2`, and lines 8 to 11 (also highlighted) return the integer `1`. Everything after the `else.1` label corresponds to the second line of the Clean function: notice the two `jsr` ("jump subroutine") instructions for the recursive calls.

[1] Clean with its ABC machine works a bit differently, see § 1.1.1.

[2] Available at https://clean.cs.ru.nl/Clean and https://clean-lang.org.

```clean
1 nfib :: Int -> Int
2 nfib n
3   | n < 2 = 1
4           = nfib (n-1) + nfib (n-2) + 1
```

Listing 1: Clean code for `nfib`.

```abc
1  .o 0 1 i
2  .r 0 1 i
3  s2
4    pushI 2
5    push_b 1
6    ltI
7    jmp_false else.1
8    pop_b 1
9    pushI 1
10 .d 0 1 i
11   rtn
12 else.1
13   pushI 2
14   push_b 1
15   subI
16 .d 0 1 i
17   jsr s2
```

```abc
18 .o 0 1 i
19   pushI 1
20   push_b 2
21   subI
22 .d 0 1 i
23   jsr s2
24 .o 0 1 i
25   update_b 1 2
26   updatepop_b 0 1
27   addI
28   pushI 1
29   push_b 1
30   update_b 1 2
31   update_b 0 1
32   pop_b 1
33   addI
34 .d 0 1 i
35   rtn
```

Listing 2: ABC code for the `nfib` function, generated from listing 1 by the Clean compiler.

Nodes on the heap can either be in head normal form (HNF, i.e. the node itself is evaluated, although its child nodes might not be) or a thunk (lazy; not yet evaluated). The first word of a node in head normal form is always a pointer to a *descriptor*, which contains meta-information about the kind of heap node: integers start with a pointer to the `INT` descriptor, etc. After that comes the actual data. For thunks, the first word does not point to a descriptor but to the node entrypoint to use for evaluating it. The next words contain the arguments of the thunk. Node entrypoints themselves also have metadata attached such as their arity, which is placed right before the entrypoint in memory. The GC needs to know the arity to determine the size of the allocated thunk.

Listing 3 contains another Clean example that creates a list containing the integer `1`, subsequently using pattern matching in `hd` to retrieve that item again. Lists in Clean are linked lists, consisting of a chain of *cons* nodes terminated by a *nil* node (*nil* is the empty list). We can see this in the generated ABC code in listing 4: create a node with `_Nil` descriptor and arity `0` (line 4), create an integer node with value `1` (line 5), and finally create a `_Cons` node using the top `2` values on the **A** stack (line 6). Figure 1 shows a visual representation of the **A** stack and heap for these three steps.

```clean
1 singleton :: Int
2 singleton = hd [1]
3
4 hd :: [Int] -> Int
5 hd [a:_] = a
6 hd []    = 0
```

Listing 3: Clean code for `singleton`.

```abc
1 .o 0 0
2 .r 0 1 i
3 s2
4   buildh _Nil 0
5   buildI 1
6   buildh _Cons 2
7 .d 1 0
8   jmp s3
```

Listing 4: ABC code for the `singleton` function (`s2`), generated from listing 3 by the Clean compiler. `s3` is the `hd` function (omitted here).

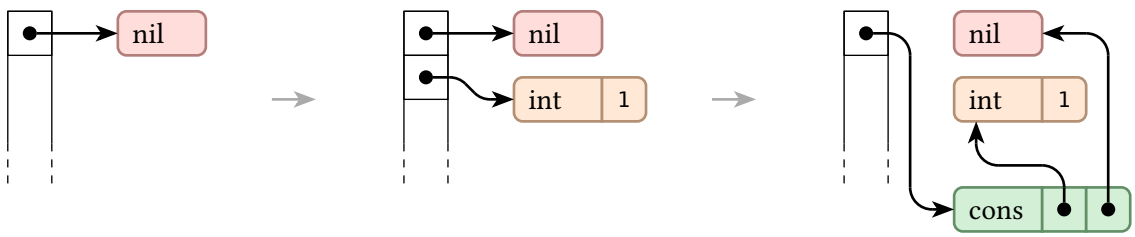


Figure 1: Schematic view of the **A** stack and heap for the three steps of creating a singleton list `[1]`.

1.1.2 LLVM

LLVM [19] is a compiler project featuring a custom intermediate representation (IR), an optimiser and several back ends like x86-64, Arm or WebAssembly. Also part of the project is the Clang compiler for C and C++, which uses the LLVM IR and optimiser. LLVM is meant as an easy compilation target for programming languages, functioning as an optimising code generator. It makes it easier for languages to support new targets such as WebAssembly, since most of the work needed to support such a target is done in the LLVM project.

The LLVM IR language is based on static single assignment (SSA). This means that LLVM IR consists of a sequence of instructions, each (optionally) with a name to refer them by. Unlike in many programming languages, there are no variables that can be assigned to multiple times: a name is bound to one single instruction and cannot be reassigned. This SSA form makes most analyses, optimisations and transformations easier. The LLVM back end translates these variables to registers[1] (or stack slots if there are not enough registers). LLVM IR is more high-level than typical assembly languages: it not only has SSA variables, but also functions and a type system with composite types like arrays and structs. The IR not only exists in-memory, but can also be saved to a file in both binary (`.bc`) and textual (`.ll`) form.

There are a number of features of LLVM IR that are relevant for this thesis.

Metadata. In LLVM IR, functions, other globals and instructions can all have metadata attached. This carries non-essential information like debug information (`!dbg`) and optimisation hints to the compiler such as which pointers can and cannot alias (`!noalias` and `!alias.scope`[2]).

Function attributes. Besides metadata, functions can also have attributes like `noinline`, `alwaysinline`, `noreturn` and `willreturn`.[3] Among these is the `gc` attribute,[4] which specifies which GC strategy to use for that function. This strategy can be provided by a plugin, but LLVM also provides some built-in ones like the *statepoint-example* strategy we look at in § 3.3.

Operand bundles. Call instructions can have something closely related to metadata called *operand bundles*.[5] As opposed to regular metadata, which can be dropped at any point, operand bundles cannot be removed from the instruction. This makes them suitable for transformations that are required for outputting correct code. Operand bundles are for example used for LLVM's statepoints (see § 3.3).

Intrinsics. LLVM includes a range of intrinsics,[6] which are called like regular functions in the IR. They differ from regular function calls in that they do not get compiled to a function call, but rather a series of machine instructions defined by the compiler. These include garbage collection intrinsics, C/C++ library functions (`llvm.abs.*`, `llvm.memcpy`, etc.), special arithmetics (bit manipulation, saturating or with overflow, fixed point, etc.), vector and matrix operations, debugging, and constrained floating point arithmetic.

Optimisation in LLVM is based on passes that run after each other in a pipeline, each of them working with the output of the previous pass. Each optimisation level (`-O0`, `-O1`, `-O2`, `-O3`) defines a pre-set pipeline of passes to run, but one can also manually specify a custom pipeline. There are three kinds of passes: **analysis** passes analyse the code and provide information for later passes, **transformation** passes transform and optimise the code in some way, and **utility** passes do neither of these things and are not used in a regular pipeline (they are for instance used for development or debugging). Passes are written in C++, like the rest of LLVM. In addition to the C++ API, there is an API for C with most common

[1] Or, to locations on the stack for languages without registers, like WebAssembly.
[2] https://llvm.org/docs/LangRef.html#noalias-and-alias-scope-metadata (accessed: Jan. 12, 2026)
[3] https://llvm.org/docs/LangRef.html#function-attributes (accessed: Jan. 12, 2026)
[4] https://llvm.org/docs/LangRef.html#gc (accessed: Jan. 12, 2026)
[5] https://llvm.org/docs/LangRef.html#operand-bundles (accessed: Jan. 12, 2026)
[6] https://llvm.org/docs/LangRef.html#intrinsic-functions (accessed: Jan. 12, 2026)

```llvm
define private i64 @nfib(i64 %n) {
entry:
  %n_lt_2 = icmp slt i64 %n, 2
  br i1 %n_lt_2, label %lt, label %else

lt:
  ret i64 1

else:
  %n_2 = sub i64 %n, 2
  %nfib_2 = call i64 @nfib(i64 %n_2)
  %n_1 = sub i64 %n, 1
  %nfib_1 = call i64 @nfib(i64 %n_1)
  %sum = add i64 %nfib_1, %nfib_2
  %result = add i64 %sum, 1
  ret i64 %result
}
```

Listing 5: LLVM code for the `nfib` function, generated from listing 2 by the Clean-LLVM code generator and manually cleaned up.

functionality, however not all features are available there. The C API is useful for interacting with LLVM from languages other than C++.

An example LLVM IR function is seen in listing 5. It is a slightly simplified form of the output of Clean-LLVM. A more in-depth description of the exact LLVM code that Clean-LLVM generates is found in chapter 2. An LLVM function is composed of basic blocks; this one has three (named `entry`, `lt` and `else`). A function has at least one basic block, although it does not need to be explicitly named like here. Each basic block contains a number of instructions (usually assigned to SSA variables), followed by a single terminator instruction (`br` or `ret` in this case).

1.1.3 WebAssembly

WebAssembly [15] is a stack-based intermediate language which aims to be safe, fast, portable and compact. For an assembly language, it is quite high-level: it has structured control flow, functions, exception handling and garbage collection. It describes both a binary format and a textual representation, and one can be converted to the other. The textual representation actually has two forms: a more traditional reverse Polish notation (RPN) form and a folded s-expression form; see listing 6 for a comparison.

```wasm
;; RPN form
local.get 0
i64.const 2
i64.sub
call $.Lnfib
```

```wasm
;; S-expr form
(call $.Lnfib
  (i64.sub
    (local.get 0)
    (i64.const 2)))
```

Listing 6: Comparison of Wasm in RPN form (left) and s-expression form (right). It is best to read the RPN form by simulating the stack in your head: push the local at index `0` onto the stack (line 2), push the number `2` (line 3), subtract them from one another (line 4), and call function `$.Lnfib` with the result (line 5, leaving the return value on the stack). The s-expression form looks more like a conventional programming language and makes the control flow and data flow clear, but it is just syntax sugar for the RPN form.

We continue the running example of the `nfib` function in listing 7. The function starts with a `block` instruction, which is Wasm's way of structured control flow. The `br_if` function inside this `block` will jump to the end of this `block` if the condition is false, thereby skipping lines 10 and 11.

```wasm
(func $.Lnfib (type 10) (param i64) (result i64)
  (block  ;; label = @1
    (br_if 0 (;@1;)
      (i32.eqz
        (i32.and
          (i64.lt_s
            (local.get 0)
            (i64.const 2))
          (i32.const 1))))
    (return
      (i64.const 1)))
  (return
    (i64.add
      (i64.add
        (call $.Lnfib
          (i64.sub
            (local.get 0)
            (i64.const 2)))
        (call $.Lnfib
          (i64.sub
            (local.get 0)
            (i64.const 1))))
      (i64.const 1))))
```

Listing 7: Wasm code for the `nfib` function, generated from listing 5 by LLVM.

1.1.4 Copying Garbage Collectors

We need two things for managing the heap: an allocator and a garbage collector. The simplest way of allocating space on the heap is called a *bump allocator*, which requires very little bookkeeping. We keep track of the first available heap location and call this the *heap pointer;* it initially points to the start of the heap. To allocate, we use this location and we increment the heap pointer by the size of the allocation. If the new heap pointer is pointing outside the available heap space, we invoke the garbage collector. This is different to how for example reference counting GC works,[1] where the GC is triggered when a heap reference is freed as opposed to when allocating. An advantage of such a simple bump allocator is that it is very quick, which is especially important in languages that allocate lots of small objects.

A copying GC is a kind of GC that allows for using such a bump allocator. One of its selling points is that it results in a very compact heap arrangement, but a big downside is that it requires twice the available heap size to work. It operates by dividing the heap up into two *semispaces*, of which only one is active at a time. When collecting garbage, all heap allocations that are reachable from the stack get moved to the other semispace and the heap pointer is updated accordingly. Dead heap allocations (i.e. no longer reachable from the stack) are ignored, in fact the GC does not know about them. This kind of GC is stop-the-world, meaning that with a naive implementation, no part of the program can run in parallel with the GC since heap values are being moved from one semispace to the other. Figure 2 shows a graphical example of garbage collection using a copying GC and subsequent allocation with a bump allocator.

Another allocation technique is the free list. It turns the unallocated parts of memory into a linked list: every free block of memory contains a pointer to the next free block. When allocating, this linked list is traversed until a spot is found that is large enough for the object, and the free list is updated accordingly. De-allocating memory using a free list is done by adding the memory section to the linked list. A free list also enables compacting garbage collectors, which move allocated regions together, filling the empty space between these regions that has appeared at de-allocation.

The original Clean compiler has two options for garbage collection: either using a copying collector, or a mark-scan collector. Both of these automatically switch to a compacting collector as well, when needed.

[1] Reference counting does not support cyclic data structures without modification or manual care from the programmer. Clean does not have reference counting, and implementing it for Clean may be impractical given these and other limitations.

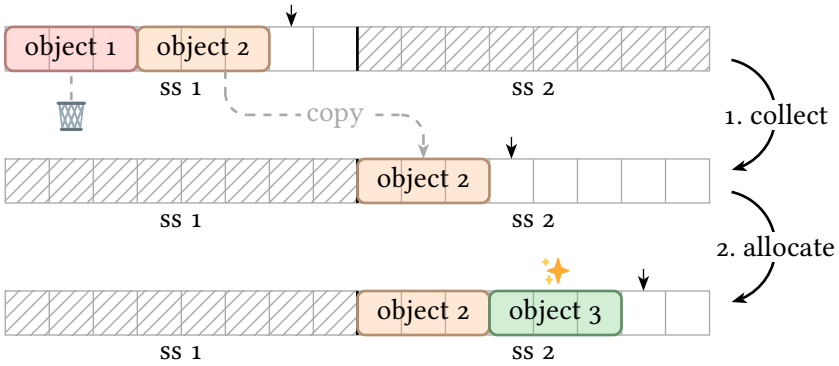


Figure 2: Schematic view of collecting and allocating. Semispaces are marked with 'SS 1' and 'SS 2', the heap pointer with an arrow. The heap consists of 8 available words in this example.

1. At the start, two objects are allocated, and we want to allocate another three words. There is not enough space available, so we run the GC. We only copy the active heap value to the other semispace.
2. After collecting, there is enough space, so we allocate space for a new heap value.

2
Clean-LLVM

As a child, I considered such unknowns sinister.
Now, though, I understand they bear no ill will.
The universe is, and we are.

— Solanum, Outer Wilds

We want to run Clean code on the web, for instance for use in iTasks. The ABC interpreter is fine for small pieces of code, but it is not fast enough to run more significant programs. That is why Clean-LLVM [1] is being developed. It plays a key role in compiling Clean to Wasm for fast execution on the web, bridging the gap between Clean's intermediate representation and LLVM, which can be compiled to Wasm.

In this chapter, we first explain the pipeline from Clean source code to the generated executable or Wasm code in § 2.1. We then introduce the basics of how LLVM code is generated from ABC code in § 2.2. After that, § 2.3 goes into detail on efficiently keeping track of some required global state. The rest of this chapter goes on to explain the Wasm-specific approach to node entrypoints in § 2.4 and descriptor contents in § 2.5. Finally, in § 2.6 we list some functionality that still needs to be implemented.

2.1 Pipeline Overview

We go through the steps in the compilation pipeline, from the Clean source file to the output executable. In a nutshell, we take in Clean's ABC bytecode and output LLVM IR. The Clang compiler, which is part of LLVM, then generates code for both WebAssembly and x86-64 targets. We use automatically generated Clean FFI (foreign function interface) bindings to the LLVM API for C to generate the output LLVM code. See figure 3 for a graphical overview. To clearly differentiate the different Clean implementations, we call the existing implementation *baseline* Clean.

1. *Clean → ABC.* We start with two files: a `.icl` file containing the implementation and a `.dcl` file with the exported definitions. The first step is done by the Clean compiler, specifically `cocl`. It compiles the `.icl` and `.dcl` file to ABC bytecode and outputs that to a `.abc` file. We only support single-module compilation in Clean-LLVM at this point, but in the future, this step runs for each of the imported files when we include other Clean modules.

2. *ABC → LLVM IR.* This is the part where we deviate from the original Clean compiler's compilation path. Our code generator (called `abc2bc`) takes this ABC bytecode and generates LLVM IR. Since we use the LLVM API, we can output both in the binary format (`.bc`) and in the textual format (`.ll`), without additional effort.

3. *LLVM IR → Executable / Wasm.* Lastly, we give this LLVM file together with the source code for the runtime system to `clang`, which runs the LLVM pipeline of optimisations and transformations. The output of this is either an executable file (a `.exe` on windows or an ELF on Linux), or a `.wasm` file for use in the browser.

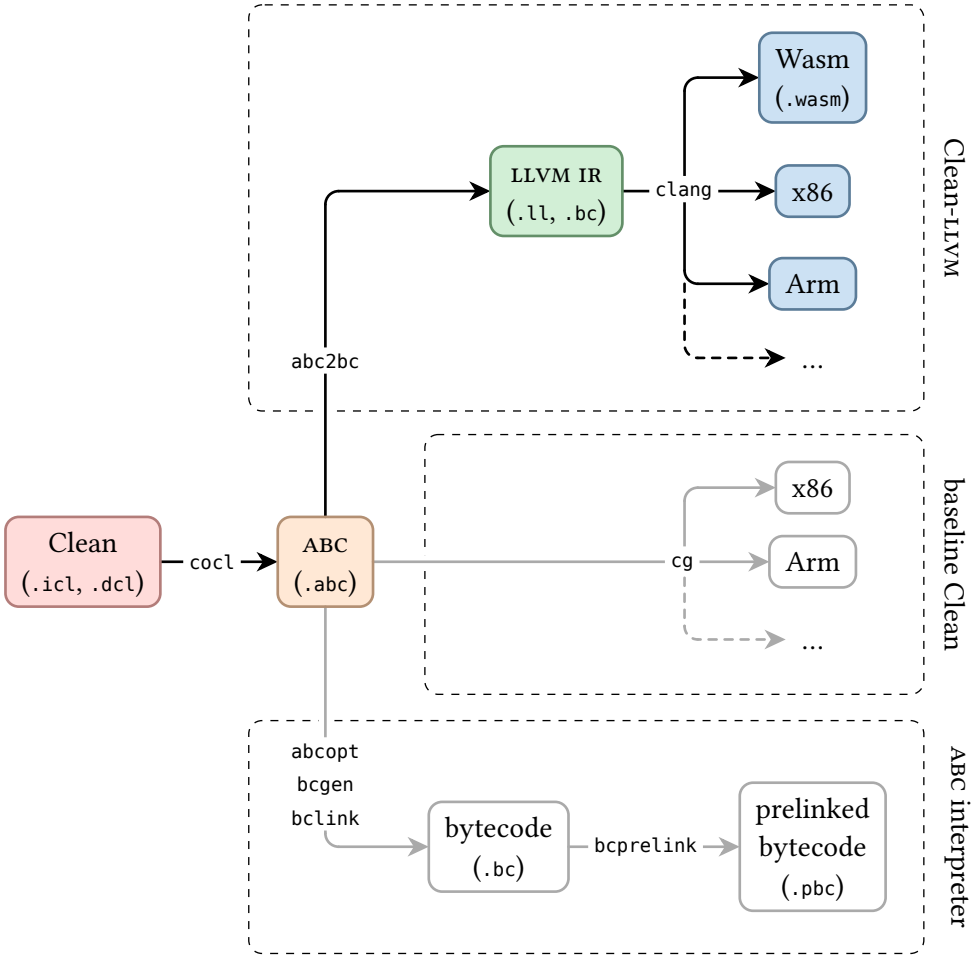


Figure 3: A graphical overview of the Clean-LLVM compilation pipeline. The ABC interpreter and baseline Clean pipelines are shown in grey. For brevity, not all targets of `clang` and `cg` are shown. Prelinked bytecode is fed as input to the ABC interpreter, and Wasm, files are run in the V8 JIT engine through Node.js.

2.2 Stack Simulation

From each entrypoint in the ABC input file, we generate an LLVM function. This means that the **C** stack is no longer needed as the control flow is handled by LLVM. Values on the **A** and **B** stacks are translated to LLVM SSA values. This is done by simulating these stacks during code generation, using LLVM values instead of real values. LLVM values can be a simple constant or a complex SSA instruction. This simulation of the stack is possible because every value's location on the stack within a function's stack frame is statically known at every point in time. To explain the simulated stack, take a look at the following example with ABC code resulting from the Clean function `answer = 37 + 5`.

ABC code	explanation	simulated **B** stack
`pushI 5`	Create LLVM integer constant `5` and push it to the simulated **B** stack.	`i64 5`
`pushI 37`	Create LLVM integer constant `37` and push it to the simulated **B** stack.	`i64 5`, `i64 37`
`addI`	Take the top two LLVM values from the **B** stack. Create an LLVM `add` instruction with these two operands, and push this instruction to the **B** stack (remember that LLVM instructions are also LLVM values).	`add i64 37, 5`
`.d 0 1 i`	Annotation indicating that the following return instruction returns `0` **A** stack values, and `1` integer (`i`) on the **B** stack.	

ABC code	explanation	simulated **B** stack
`rtn`	Take the top LLVM value from the **B** stack and create an LLVM return instruction with it: `ret i64 %_1` (we refer to the `add` instruction with `%_1`, its SSA variable name in this example).	(empty)

The resulting LLVM code would look like the following, if we ignore that LLVM automatically collapses the `add` instruction to a constant.

```llvm
define i64 @answer() {
  %_1 = add i64 37, 5
  ret i64 %_1
}
```

2.3 Register Pinning

For allocating objects on the heap, we need to know at all times where the next available heap position is and how much free space there is left on the heap. These values are essentially global state which we need to keep track of. For performance reasons, we do not want to put them in a global variable which is constantly read and updated, because that results in lots of memory accesses for allocation-heavy programs. These values should really be in registers at all times, i.e. they should each be *pinned* to a register. LLVM does not provide a way to do this, but the solution used by the LLVM code generators for GHC [29] and Erlang [26] is to pass these values as the first arguments to a function, using a calling convention that puts those in registers. One calling convention that does this is *fastcc*. This way, the values will always be in registers (at least across function call borders; LLVM is free to do anything with their registers within a function). Note that this only has an effect for native code generation; Wasm is too high-level and does not allow specifying a custom calling convention, nor does it have a concept of registers.

We manually removed these register-pinned values from the example in listing 5 for clarity since the `nfib` function does not use them (it does not allocate). In practice, we pin a couple values to registers and `nfib`'s function signature looks more like this:

```llvm
define private fastcc { ptr, i64, i64 } @nfib(ptr %hp, i64 %heap_free, i64 %n)
```

Function calls look like this (`%nfib_2` is the actual return value):

```llvm
  %_15          = call fastcc { ptr, i64, i64 } @nfib(ptr %hp, i64 %heap_free, i64 %n_2)
  %hp_new       = extractvalue { ptr, i64, i64 } %_15, 0
  %heap_free_new = extractvalue { ptr, i64, i64 } %_15, 1
  %nfib_2       = extractvalue { ptr, i64, i64 } %_15, 2
```

And returns look like this (corresponds to `ret i64 1` in listing 5):

```llvm
  %_10 = insertvalue { ptr, i64, i64 } undef, ptr %hp, 0
  %_11 = insertvalue { ptr, i64, i64 } %_10, i64 %heap_free, 1
  %_12 = insertvalue { ptr, i64, i64 } %_11, i64 1, 3
  ret { ptr, i64, i64 } %_12
```

In order to reduce stack usage for Wasm, Clean-LLVM accepts the `-ug` command-line flag. This changes the approach to use globals for the heap pointer and free heap space, instead of registers (`ug` stands for "use globals"). While this does reduce stack usage, it also incurs a performance cost of loading from and storing to system memory when accessing these values. Therefore, we do not use this flag now.

2.4 Node Entrypoints for Wasm

We briefly touched on node entrypoints in § 1.1.1. Every node entrypoint has its arity and other metadata placed right before it in memory, which the GC uses to determine the size of the allocated thunk. LLVM supports this with prefix data, which allows us to place arbitrary data before functions. WebAssembly's memory model complicates this: function addresses are consecutive integers, as if they are elements in a list of functions. This makes prefix data impossible, so we need a different solution for Wasm.

There are multiple ways to solve this, but we choose to implement it using an extra indirection. Instead of placing the metadata before the entrypoint, we place it in a global struct before a pointer to the entrypoint. The advantage of this is that the GC does not need to be adapted since it only looks at the metadata. At the point of evaluating a node entrypoint, the code generator inserts an extra load instruction to walk through the added indirection.

We selectively enable this extra indirection for node entrypoints only when generating code for Wasm, so there is no performance impact for native targets.

2.5 Descriptor Contents

As explained in § 1.1.1, nodes on the heap start with a pointer to a descriptor, which contains information about the type of node [2]. Descriptors for records contain the number of basic values and heap references, the types of those basic values, and the name of the record. ADT descriptors also contain the number of arguments and the name of the ADT constructor, in addition to a flag indicating if it is actually a record with lazy arguments, as well as a curry table. This curry table is used when not all arguments are supplied to an ADT or function. Because currying is not implemented in Clean-LLVM, we do not go into detail on the exact contents of this curry table.

For optimisation reasons, Clean's run-time system expects that the descriptors are laid-out in memory in a specific order. This is again a place where Wasm is restrictive: it does not provide us a way to specify the order of static data. We get around this by generating all descriptors in a single array. We then create LLVM aliases[1] for pointing to each descriptor.

Additionally, descriptors for ADTs include a static HNF node of arity 0 of that ADT, which is used by the garbage collector. This static node is located right before the descriptor itself in the descriptor array.

2.6 Missing Functionality

Being work-in-progress, Clean-LLVM lacks some functionality that is required for generating code for more complex programs. Some of them are listed here.

To start, not all ABC instructions are implemented. In fact, a large amount is not yet implemented, or only partially. For this thesis and in the preceding internship, we implement some of these missing instructions in order to be able to run the programs we want to. We also modify the programs so that they no longer compile to unimplemented ABC instructions.

Currently we compile a single Clean file with no import statements. Not even the standard library is included, so every file needs to contain its own implementation of for instance the + operator on integers.

Finally, a hurdle for test programs is that they use I/O, which is not implemented at this point. The only possibility for outputting text is that the integer returned from the main function is automatically displayed.

[1] https://llvm.org/docs/LangRef.html#aliases (accessed: Jan. 20, 2026)

3

Every decision is made in darkness.
Only by making a choice can we learn whether it was right or not.

— Prisoner, Outer Wilds

Basic GC Integration

We have now explained what Clean-LLVM is and why it is useful. Since Clean is a garbage-collected language, we need to implement a GC for Clean-LLVM. It is not possible to use the one from the existing Clean compiler because that one uses hand-written assembly (and we cannot compile that to Wasm). We choose to integrate the copying GC from the ABC interpreter, since it is already known to work for Clean, and because it is implemented in C (which Clang *can* compile to Wasm).

This collector is a moving GC, so the heap address of an object changes after each collection cycle. Therefore, the GC not only needs to *read* pointers in registers, but also *update* them again after a GC invocation, to reflect their new location. This needs to work without being able to inspect the run-time stack (which is not possible in Wasm) and without extra indirection (which would incur a performance penalty).

WebAssembly has its own solution for garbage collection, standardised in Wasm version 3.0.[1] It has instructions to create heap values, and all garbage is automatically taken care of by the Wasm runtime. This has numerous advantages compared to shipping a GC in Wasm code, and several language toolchains have (experimental) support for it [33]. However, the WebAssembly back-end of LLVM is not yet updated to support the Wasm GC instructions,[2] so we cannot use it. There are also difficulties stemming from Wasm's type safety, for instance when a thunk is evaluated to a head-normal form.[3]

There are a number of ways in which we can integrate our own GC into the Clean-LLVM code generator. LLVM has built-in functionality for helping with this: the older *gcroot* mechanism and the newer *statepoints*. gcroot is described as 'mostly of historical interest at this point'; it is no longer actively maintained, and is not bug-free.[4] Statepoints are intended to replace gcroot entirely, however the built-in GC strategies for statepoints do not support WebAssembly, since they are based on generating stack maps which WebAssembly does not support. Lastly, we can of course implement a custom LLVM pass that does something like what these LLVM features do, which requires no platform support and should therefore be portable.

This chapter goes into detail on three main ways to integrate a copying GC: by implementing a shadow stack manually (§ 3.1), by using LLVM's gcroot function (§ 3.2), or by using LLVM statepoints (§ 3.3).

[1] https://webassembly.org/news/2025-09-17-wasm-3.0 (accessed: Nov. 18, 2025)
[2] https://discourse.llvm.org/t/wasmgc-implementation-status/74821/2 (accessed: Nov. 18, 2025)
[3] https://gitlab.com/clean-and-itasks/clean-llvm/-/issues/5#note_2254379710
[4] See for example https://github.com/llvm/llvm-project/issues/31684 (accessed: Nov. 18, 2025), which we also encountered in preliminary testing.

3.1 Custom Shadow Stack

A way to let the garbage collector read and update heap pointers on the system stack, is to use a separate stack placed on the heap. We push *(spill)* each live heap pointer to this stack before a GC cycle and we pop *(restore)* the newly relocated pointer again afterwards. This construction is called a shadow stack. Where this stack is located, how it is structured, when to spill and when to restore are all implementation decisions.

We generate code for such a shadow stack directly, at the same time as the other code generation. It is also possible to do this later on, which provides some advantages but also introduces its own challenges. We elaborate on that in § 3.3. The advantage of doing this early on, during code generation, is that we still have all the information from the source ABC code, as well as the entire simulated **A** stack. There are never unused values on the **A** stack; the Clean compiler ensures this. This means that every value on the **A** stack is live.

Before each function call, we spill the entire **A** stack of the current function. We exclude any values that are passed as arguments to this call, since they will appear in the callee's **A** stack. Functions are thus responsible for spilling their received arguments themselves. Immediately after the call, we restore these values again. After restoring, we never use the old values again, since they no longer point to the same object if garbage collection has occurred during the call.

3.2 GCRoot

The older mechanism for describing heap references makes use of three LLVM intrinsics: `llvm.gcroot`, `llvm.gcread` and `llvm.gcwrite`. The latter two are read and write barriers which are not useful for a copying garbage collector. gcroot requires each heap reference to be stored in an allocation on the system stack. This is done with the `alloca` function / instruction, which works like `malloc` but for the stack instead of the heap. A reference to this stack-allocated spot is passed to the `llvm.gcroot` intrinsic call, which needs to happen in the first basic block of a function.

The LLVM shadow stack GC strategy makes use of the gcroot intrinsics to generate a shadow stack, implemented as a linked list of stack roots. It does not require any special treatment from the compilation target, so it is available anywhere. However, there are performance issues with `llvm.gcroot`:

- If garbage collection can happen before an **A** stack value is live (before it is created), its stack slot must be marked with a sentinel value (such as `null`) to prevent the GC from reading uninitialised data. When a value reaches the end of its lifetime, its stack slot should also be marked with a sentinel value again. This was also noted by the Erlang LLVM back end [26].
- Another trade-off with using `alloca` is that function arguments get an extra indirection: not the heap pointer itself, but the address of the `alloca` (which contains the heap pointer) is passed as argument. This forces the caller to spill every **A** stack value passed to a function call to system memory, even when that would otherwise not be necessary.
- Additionally, the code generator should take care to only allocate stack slots for **A** stack values that are actually live across function calls; LLVM does not rewrite `alloca` + `gcroot` instructions back to register instructions, since the `alloca` is used in the call to the `gcroot` intrinsic.

3.3 Statepoints

Garbage collection statepoints, usually called safepoints outside of LLVM, are a way to mark locations in the code where heap addresses can change due to the garbage collector. To mark a function call as statepoint, we replace it with a call to the `gc.statepoint` intrinsic, passing the original function and its arguments. We also note down which heap references are live across this call. The resulting statepoint is then used for retrieving the return value of the function, and for getting the new memory location of the live heap references. The specific implementation of the statepoint intrinsic ensures that the GC knows what values are live, and that the code calling the statepoint knows the new location of these moved heap values.

LLVM includes two GC strategies that use statepoints: *statepoint-example* and *coreclr*. These both make use of stack maps and stack walking over the native system stack, which are in theory more performant than a shadow stack for targets that support it. In order to support Wasm when using statepoints, we write a series of passes to transform code with statepoints to use a custom shadow stack as in § 3.1.

The advantage of generating code for a shadow stack in an LLVM pass is that we are able to run other optimisations before it, such as inlining and dead code elimination (see § 4.5). This results in less call sites in general, and potentially less live values at those call sites. If a later LLVM pass removes a use of an otherwise live heap reference (thereby making it no longer live) after a function call, or if it removes said function call entirely, earlier generated spills remain even though they are no longer required. That is why it is important to run these optimisations *before* generating spills.

We perform the following steps for generating statepoints:

1. All **A** stack values are marked with type `ptr addrspace(1)`, a pointer in address space 1. Address spaces are used by some LLVM targets to separate pointers to different (physical) memory regions from one another, but here we only use them to determine which pointers are heap references.
2. Functions that we generate are marked with `gc "statepoint-example"`. This signals to further passes that we want to transform code in these functions to use statepoints.
3. With this meta-information, we add calls to LLVM's statepoint intrinsics. The function call itself is replaced by `gc.statepoint`, which marks the actual statepoint. This intrinsic receives the callee and the call arguments. A list of live heap references is attached to this call with a `"gc-live"` operand bundle. Uses of the return value of the original call get replaced by calls to the `gc.result` intrinsic, and further uses of live references are replaced by the `gc.relocate` intrinsic.
4. As mentioned in item 1, address spaces have a meaning for the target platform. We need to transform code back to the default address space to avoid it being interpreted by the back-ends. For this, we use one of Julia's LLVM passes (`remove-addrspaces`[1]) which also works in our case.

This is enough for x86-64 and Arm targets, since they can use the built-in *statepoint-example* GC strategy which uses stack-walking.[2] For WebAssembly, we take this as a starting point to further transform the code:

5. Since we need to know the current **A** stack pointer (`asp`) for spilling to it, we add it as a register-pinned variable (see § 2.3), adding a function parameter and return value to each function.
6. We then transform all statepoint intrinsics back to regular calls with spills and restores.
   - `gc.statepoint` is replaced with a regular function call. Before it, we spill each value that is present in the `"gc-live"` operand bundle.

[1] https://github.com/JuliaLang/julia/blob/54fde7e012e6883a5bc9acd964593c8b9c9b5c74/src/llvm-remove-addrspaces.cpp (accessed: Jan. 21, 2026)

[2] The implementation of this is left as future work. Instead, the solution for Wasm is used for all targets.

- Uses of `gc.result` are restored to uses of the return value of the function call.
- Each `gc.relocate` is turned into a restore, i.e. a load from the spilled **A** stack at the known index.

3.3.1 LLVM Pass Pipeline

We use the following pipeline of LLVM passes. Those marked bold are custom passes: `remove-addrspaces` comes from Julia, while `place-asp-safepoints` and `add-asp` are contributions of this thesis.

```
globaldce,cgscc(inline,adce),function(place-asp-safepoints),remove-addrspaces,add-asp,globaldce,adce,gvn
```

An explanation of these passes:

`globaldce`	Global Dead Code Elimination. Removes unused (dead) functions and global variables.
`inline`	Replaces function calls with the callee's body. This not only eliminates function call overhead, but more importantly it allows further optimisations to be more effective since the function body is put in context of its call. Not all function calls are inlined; whether to inline or not is decided based on a heuristic. For example, large functions (with a lot of instructions) tend not to be inlined.
`adce`	Aggressive Dead Code Elimination. Removes all instructions whose output is not used.
`gvn`	Global Value Numbering. Further eliminates redundant instructions and memory loads.
`place-asp-safepoints`	Replace function calls with calls to LLVM's statepoint intrinsic functions. Corresponds to step 3 above. We do this for every call that is not in tail call position: there can never be active heap references after a tail call. At each call site, we collect a list of live heap references. We do this by looking at each operand of each instruction that comes after the call, adding it to the list if it is defined before the call instruction. Then, for each live heap reference, every use that comes after the call is replaced by the inserted `gc.relocate` call.
`remove-addrspaces`	Transform code using address space 1 back to the default. Corresponds to step 4 above.
`add-asp`	Replace statepoint calls with normal function calls and add spills and restores. Corresponds to step 5 and 6 above. Since LLVM does not allow for changing the signature of existing functions, we create new ones with an added **A** stack pointer parameter, and copy over the function body. We then transform each statepoint back into a regular call, making sure to call the new function which has the **A** stack pointer parameter when applicable.

3.4 Wrap-Up

This chapter introduced three ways of connecting a garbage collector with a shadow stack: either by manually generating spill and reload instructions, by using LLVM's gcroot feature or using statepoints. We implement both the custom shadow stack as well as the set of LLVM passes using statepoints. In the rest of this thesis, we improve these implementations further and we compare their performance.

4

Mission: Science compels us to explode the sun!
— Pye, Outer Wilds

Performance Improvements

The last chapter introduced the basic way of integrating the GC, which comes down to two different mechanisms for spilling the **A** stack. These implementations are far from optimal, and there are a number of improvements to the performance possible. In this chapter, we introduce the underlying issues, we explain what causes them, and we show how to fix them to generate more performant code. Specifically, we add two general optimisations that apply to both of these mechanisms: static nodes (§ 4.2) and LLVM's alias metadata (§ 4.3). For the custom shadow stack, we restore lazily in § 4.4, and we run LLVM optimisation passes before generating statepoint-based spills in § 4.5. We also evaluate the speedup gained by implementing each solution. At the end of this chapter, in § 4.6, we quantitatively compare these different improvements to both the original Clean compiler and to the ABC interpreter. Firstly though, we explain our benchmark methodology.

4.1 Benchmarking

In this section, we introduce the benchmark tool we use to measure the runtime (§ 4.1.1), we show the actual programs that we benchmark (§ 4.1.2), and we go over the impact of heap size on performance (§ 4.1.3). We are only interested in runtime performance—code size, memory usage and compilation time are of lesser importance for this thesis. All benchmarks are run on an AMD Ryzen 7 2700x with 16GB of memory,[1] running Pop!_OS 22.04 and using Clang 20.1.8 and Node.js 24.1.0.

4.1.1 Benchmark Tool

We use a shell script called `benchmark.sh` to compare runtime performance between different targets, code versions and compilation flags. It takes care of compiling, running, capturing run-times and displaying a statistical comparison of the results. The output formatting and statistical computation is inspired by Hyperfine [23], an open-source tool for benchmarking executables. Instead of using Hyperfine, we develop custom tooling for running benchmarks that directly integrates with the generated executable in order to remove as much variability from process creation and start-up time as possible.

With generated LLVM as well as for the ABC interpreter, we use a loop in C and JavaScript, respectively, that repeatedly calls the entry-point function. For baseline[2] Clean, this is done with a Clean program instead. This difference is unavoidable and should not result in any meaningful change in measured run-times.

[1] The benchmark programs do not get to use all of this memory. We run them with both 64kiB and 16MiB of memory as we explain in § 4.1.3.

[2] Remember that *baseline* Clean refers to the existing implementation, as opposed to Clean-LLVM.

The time each benchmark run takes is measured by calling the `clock_gettime` syscall with the `CLOCK_MONOTONIC` time. We do not use `CLOCK_REALTIME` since that clock experiences forwards and backwards jumps in time to align with wall-clock time. `CLOCK_MONOTONIC` does not jump, but its frequency is still adjusted to match network time. This makes `CLOCK_MONOTONIC_RAW` an even better fit for benchmarking since its frequency is not adjusted. However, the JavaScript standard library (which we use when benchmarking Wasm) only provides the `performance.now()` function, which uses `CLOCK_MONOTONIC` internally (in Node.js). It is more important to use the same clock everywhere than to use the most accurate one: `CLOCK_MONOTONIC_RAW` may have a slightly different frequency than `CLOCK_MONOTONIC`, so using both would skew benchmark results. At the end of each run, the measured time is written to the standard error output stream, to be read by the benchmark script.

Just-In-Time (JIT) interpreters, like the ones used for running Wasm code, most often experience a warm-up phase where it takes some time before run-times settle to a stable value [8]. To avoid that influencing the results, the benchmark tool first discards m warm-up runs after which n benchmark runs follow, where n and m are configurable and need to be set manually to suit the runtime of a single run. These warm-up runs are discarded for every executable: for the generated LLVM as well as for baseline Clean and the ABC interpreter.

Additionally, we run the `node` command with the `--no-liftoff` flag, which disables V8's Liftoff JIT compiler. This single-pass compiler is designed to reduce startup times, and the code it generates is used until the optimising TurboFan compiler is done compiling [7]. We want to benchmark the stable case (the TurboFan output), so we disable the Liftoff compiler. Interestingly, this flag sometimes makes the generated code *slower*, not faster. Regardless, we keep it enabled for our benchmarks.

The benchmark script has functionality to automatically replace functions in generated LLVM IR files. This patched function is for example hand-optimised to test different strategies. We also use this for testing with the GC disabled in § 4.1.3, by replacing the looping function with one that resets the heap pointer each loop.

We use two operating system features to improve inter-run variability. Firstly, we pin the program to a single CPU core with the `taskset` command to prevent it from being moved to a different core while it is running. Secondly, we disable CPU frequency scaling and run the core at a fixed clock speed. Since frequency scaling is OS and processor specific, it is not included in the benchmark script and needs to be set manually.

4.1.2 Benchmark Programs

In order to compare the speedup gained from the different implementations in the coming sections, we must use the same benchmark programs. Since Clean-LLVM does not yet support all ABC instructions, there is only a limited set of benchmarks we can support. Therefore, we only benchmark three programs. We choose one benchmark called *binarytrees*[1] from the Computer Language Benchmarks Game [13], an online repository of benchmark programs and results for a number of different language implementations. In the past, Clean was part of this list of languages, but it is not anymore at this time. Another benchmark we use is *list* from the ABC interpreter tests, and finally we use one program *(astack)* written with the purpose to show the effects of the optimisations we implement.

Binarytrees. This is an adaptation of Hans Boehm's GCBench[2]—a program to test GC performance. We use it because it is easily modified to not use any IO (which is not yet implemented in Clean-LLVM), yet it is not as small as other benchmarks and it still uses the heap. It first allocates one large binary tree

[1] https://benchmarksgame-team.pages.debian.net/benchmarksgame/description/binarytrees.html#binarytrees (accessed: Nov. 27, 2025)

[2] https://hboehm.info/gc/gc_bench/ (accessed: Dec. 2, 2025)

that is live until the end of the program, and then creates many shorter-lived trees of varying depth in a loop.

A Clean implementation of this program is available via the internet archive.[1] We use that implementation with a few modifications: we remove IO operations and instead add up all intermediate results, and we remove the integer from the tree node to avoid generating ABC instructions that are not yet implemented in Clean-LLVM. In addition, we manually add the definitions for the used operators, since the standard library cannot be imported yet. The resulting source code is found in appendix A1.

List. A very simple program that creates a long list of increasing integers and then sums it. See the code in appendix A2. Since this program does not do much with the **A** stack (which contains only the list), the optimisations in this chapter do not have much effect.

Astack. This program consists of a loop, repeatedly calling a function that creates 16 heap values. The function first creates a list of all of them and passes that to a dummy function (`isNotEmpty`, which simply returns `1` if the list contains at least one item, `0` otherwise). It then passes a list of just one heap value to the dummy function, followed by a list of another heap value, and lastly it passes a list of all the heap values again. The code is attached in appendix A3. Repeatedly using many values across function calls is meant as a stress-test for spilling and restoring.

4.1.3 Collection, Caching and Consistency

Ideally, we want each run of the benchmark to take an identical amount of time. Sadly, the real world makes this impossible, but we can come close to it. We already explained how we get rid of startup time and process scheduling differences by measuring individual runs with `CLOCK_MONOTONIC` time. Another source of inter-run variability is the impact of heap size on caching.

To illustrate that, we look at `astack`, which has very little heap space in use at a time since it runs the same function in a long loop. We compare the runtime of this program in a range of heap sizes, from 1 kiB (the smallest possible for this program) to 1 GiB. Additionally, we compare to a version where we reset the heap pointer between each iteration of the loop (named *no GC*), so it uses the least amount of heap space possible. We not only measure the runtime of the program, but also the time spent collecting garbage and the number of GC invocations. Lastly, we use the `perf` command[2] to gather

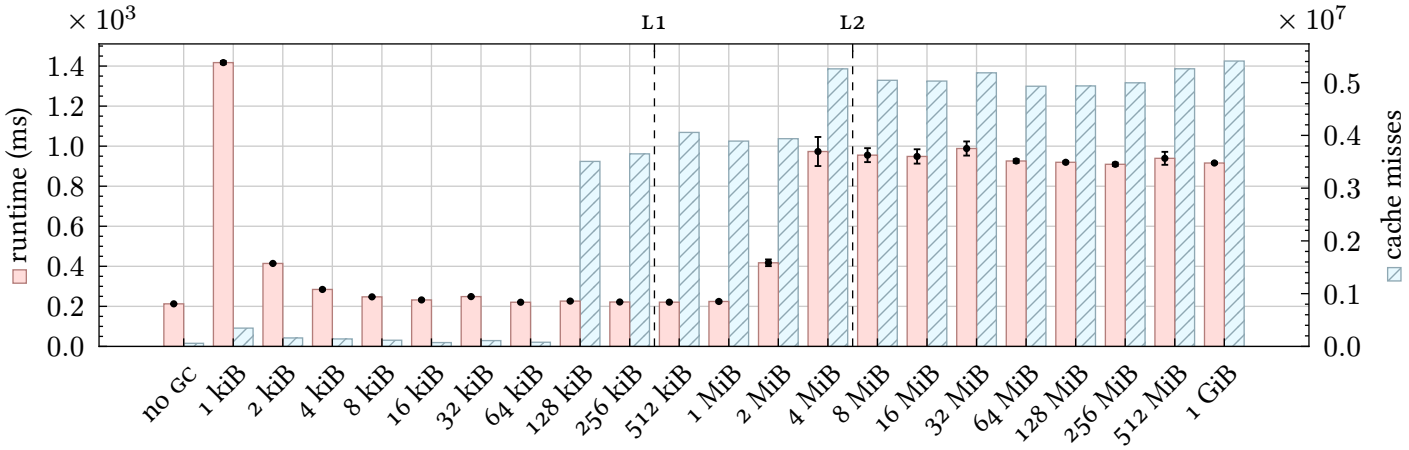


Figure 4: The runtime (left axis) and number of cache misses (right axis) for each heap size, for the `astack` program. The sizes of the L1 and L2 caches are marked with dashed lines. Note the sudden increase in cache misses at 128 kiB and 4 MiB, and the sudden increase in runtime around 2 MiB and 4 MiB.

[1] https://web.archive.org/web/20111118030741/http://shootout.alioth.debian.org/u64/program.php?test=binarytrees&lang=clean&id=3 (accessed: Nov. 27, 2025)

[2] `perf stat -e faults,cache-misses`

Table 1: Numbers per run of the `astack` program, for each heap size:
- the total runtime including GC,
- the time spent doing GC and the number of cycles,
- the amount of page faults and cache misses.

Observe that greater heap sizes correspond to less GC cycles and more page faults.

heap size	runtime	± stddev	(relative)	GC time	GC cycles	page faults	cache misses
no GC	212.59 ms	± 0.56 ms	(1.00×)	N/A	0	7	58 785
1 kiB	1 417.35 ms	± 7.81 ms	(6.67×)	978.59 ms	10 000 000	7	346 657
2 kiB	414.32 ms	± 2.13 ms	(1.95×)	104.96 ms	5 000 000	7	160 113
4 kiB	284.48 ms	± 3.69 ms	(1.34×)	35.22 ms	1 666 667	7	141 182
8 kiB	247.46 ms	± 1.41 ms	(1.16×)	17.63 ms	833 333	7	117 116
16 kiB	232.08 ms	± 3.06 ms	(1.09×)	8.82 ms	416 667	7	72 794
32 kiB	248.80 ms	± 2.14 ms	(1.17×)	25.11 ms	205 479	8	110 200
64 kiB	220.59 ms	± 0.29 ms	(1.04×)	2.24 ms	102 041	8	78 898
128 kiB	226.32 ms	± 2.91 ms	(1.06×)	5.36 ms	50 761	10	3 506 809
256 kiB	221.76 ms	± 1.16 ms	(1.04×)	0.56 ms	25 381	13	3 649 648
512 kiB	220.90 ms	± 0.65 ms	(1.04×)	1.45 ms	12 674	20	4 056 086
1 MiB	224.44 ms	± 4.50 ms	(1.06×)	0.14 ms	6 337	32	3 891 791
2 MiB	417.62 ms	± 16.70 ms	(1.96×)	0.07 ms	3 167	58	3 937 219
4 MiB	973.35 ms	± 72.57 ms	(4.58×)	0.21 ms	1 583	109	5 261 032
8 MiB	955.19 ms	± 34.84 ms	(4.49×)	0.48 ms	792	212	5 044 211
16 MiB	948.72 ms	± 35.78 ms	(4.46×)	0.03 ms	396	416	5 029 400
32 MiB	988.33 ms	± 35.31 ms	(4.65×)	0.16 ms	198	826	5 185 758
64 MiB	925.96 ms	± 8.93 ms	(4.36×)	0.07 ms	99	1 645	4 931 663
128 MiB	919.50 ms	± 5.15 ms	(4.33×)	0.01 ms	50	3 284	4 937 356
256 MiB	909.50 ms	± 8.30 ms	(4.28×)	0.01 ms	25	6 560	4 996 414
512 MiB	939.26 ms	± 32.41 ms	(4.42×)	0.00 ms	12	13 114	5 262 254
1 GiB	915.90 ms	± 5.36 ms	(4.31×)	0.00 ms	6	26 221	5 409 261

hardware statistics: the number of page faults and cache misses. The results of this are in table 1, and the resulting runtimes and cache misses are visualised in figure 4.

The figure clearly shows that a heap of 2 MiB causes slower runtimes, and even more so for 4 MiB and above. This is very likely related to the L2 cache size of the processor used, which is 4 MiB. If we account for the fact that the actual allocated memory is double the usable heap space because of the semispace allocator, the correspondence becomes clear between the L2 cache size and the point where runs start taking longer. We also see a sudden increase in cache misses starting at 128 kiB. A heap of that size still perfectly fits in the 256 kiB of L1 cache, but not together with the **A** stack and system stack.[1] The takeaway of this is that it is beneficial to use a small heap size. Not too small, however, since the time spent in the garbage collector then starts becoming significant: take a look at the GC time and number of cycles in the top few lines of table 1. A heap size of 64 kiB is close to ideal, both regarding the runtime as well as the standard deviation, which is important for benchmarking.

Sadly, not all programs fit completely in the L2 cache. `binarytrees` is one of those, requiring at least 16 MiB of heap. Because we still want to compare the programs with a smaller heap size for less variability, we benchmark with two heap sizes: 64 kiB (excluding `binarytrees`) and 16 MiB.

[1] We allocate 1 kiB for the **A** stack, which is enough for all benchmarks.

4.2 Static Nodes

Optimisation is often about *doing less*. That is why we first focus on reducing the number of garbage-collectable references to the heap. ADT constructors with no arguments (like nil; the empty list) are only ever one value. We store that value as a static node in the executable. This is something that is also done by baseline Clean. We also store some often-used nodes here: booleans, all characters and integers up to 32. This results in doing less in multiple ways:

1. Perhaps obviously, it reduces the time it takes to create such nodes—they do not actually need to be created, merely referenced.
2. By storing just one instance of these values, we free up space on the heap that these values would otherwise occupy. As an example, a (fully evaluated) list of small integers now only needs to store the cons nodes in the heap and not the integers themselves. As a result, the heap fills up less quickly, and the garbage collector needs to run less frequently.
3. Since the static nodes are not garbage collected, they do not need to be spilled. This reduces the time spent spilling and the time spent by the garbage collector walking through the shadow stack. The shadow stack is also smaller, which leaves more space in the processor cache.

An example of how the LLVM code as generated by § 3.1 changes as a result of using static nodes is seen in listing 8. Since using static nodes is a basic improvement that does not interfere with other optimisations, it is used in all further optimisations in the remainder of this chapter: all further improved versions also use static nodes where possible.

```llvm
1  store ptr getelementptr (i8, ptr @INT, i64 2), ptr %_2
2  %_3 = getelementptr i64, ptr %_2, i64 1
3  store i64 1, ptr %_3
4  store ptr %_2, ptr %_1, align 8
```

```llvm
1  store ptr getelementptr (i64, ptr @small_integers, i64 2), ptr %_1
```

Listing 8: Comparison of storing the integer `1` on the heap at address `%_1`, without (left) and with (right) static nodes.

- Without static nodes, we first store the integer descriptor on the heap (line 1), followed by the integer `1` (line 3). The heap address of the descriptor is put on the heap at the given address (line 4).
- With static nodes, we directly store the address of the static integer `1` node on the heap.

4.2.1 Benchmark Results

Using static nodes has a positive effect across the board. It is of course more apparent for programs that use a lot of zero-argument ADT constructors or small integers. `list` and `astack` do not use many of those, while `binarytrees` of course creates relatively many leaf nodes (which are ADTs with no arguments). We see the effect of this in table 2, where `binarytrees` is 12% faster for x86-64 and 21% faster for Wasm.

Table 2: Relative runtimes of the `list`, `astack` and `binarytrees` benchmarks. Each column is compared separately; the runtime on x86-64 with 64kiB of heap is relative to the x86-64 runtime before optimisations (§ 3.1) with 64kiB of heap, and Wasm results are compared to Wasm results.

static / base 64kiB	x86-64	Wasm
list	0.96×	0.99×
astack	0.97×	0.99×

static / base 16MiB	x86-64	Wasm
list	0.85×	0.96×
astack	0.97×	0.99×
binarytrees	0.89×	0.82×

4.3 Alias Metadata

LLVM sometimes does not optimise away some redundant `load`/`store` instructions because it does not have enough information to be certain that they really are redundant. A simple case is where a memory address gets written to twice, but in between there is another write to a different memory address. In principle, LLVM should be able to remove the first `store` instruction. It does not, however, since the two memory references may actually point to the same address (i.e. they may alias). We inform LLVM that these memory references do not alias by attaching `!alias.scope` and `!noalias` metadata to all memory access instructions. With these annotations in place, LLVM removes some redundant loads and stores, which primarily occur across inlined calls.

Specifically, we define three alias scopes: one for references to the heap, one for references to the **A** stack and one for references to globals. We annotate each `store` instruction with its corresponding `!alias.scope`, and add the other alias scopes to `!noalias`. The same happens with `load` instructions. This way, LLVM knows that reads and writes to the heap never alias those to the **A** stack.

Adding alias annotations is again a basic improvement that does not interfere with other optimisations, so it is used in all further optimisations in the remainder of this chapter: all further improved versions also have alias annotations (in addition to using static nodes).

```llvm
%_5 = load ptr, ptr %_2, align 8, !alias.scope !1, !noalias !7
; ...
store ptr %_41, ptr %_53, !alias.scope !1, !noalias !7

!0 = !{!"alias"}       ; alias domain
!1 = !{!2}             ; list of the heap scope
!2 = !{!"heap", !0}    ; heap scope (in the alias domain)
!3 = !{!4}             ; list of the globals scope
!4 = !{!"globals", !0} ; globals scope (in the alias domain)
!5 = !{!6}             ; list of the A-stack scope
!6 = !{!"a_stack", !0} ; A-stack scope (in the alias domain)
!7 = !{!4, !6}         ; list of all scopes except the heap scope
!8 = !{!2, !6}         ; list of all scopes except the globals scope
!9 = !{!2, !4}         ; list of all scopes except the A-stack scope
```

Listing 9: Example of how the alias metadata gets attached to a load from the heap and a store to the heap. These instructions operate in the `"heap"` scope (`!1`) and do not alias the `"globals"` and `"a_stack"` scopes (`!7`).

4.3.1 Benchmark Results

For the `astack` benchmark, adding alias metadata to memory access instructions gives a slight performance improvement. This example program spills the same values across multiple consecutive function calls. LLVM is able to detect that some spills are redundant, since the values in question have already been spilled to the same locations earlier.

When looking at the Wasm results, there is no performance improvement for `list` and `astack`. This seems to indicate that V8's JIT compiler already removes (some of) the redundant stores and loads.

Table 3: Relative runtimes of the `list`, `astack` and `binarytrees` benchmarks. Each column is compared separately; the runtime on x86-64 with 64kiB of heap is relative to the x86-64 runtime of *static* (§ 4.2) with 64kiB of heap, and Wasm results are compared to Wasm results.

alias / static 64kiB	x86-64	Wasm
list	1.00×	0.99×
astack	0.91×	1.00×

alias / static 16MiB	x86-64	Wasm
list	1.01×	1.00×
astack	1.00×	0.99×
binarytrees	0.99×	0.98×

4.4 Lazy Restoring

Many Clean functions contain more than two function calls. In a situation where **A** stack values created at the start of such a function need to live through to the end of the function, these values will constantly be spilled and restored for each call in between, even when they are not used between the function calls. LLVM optimises away some of these redundant spills and restores, but not all of them.

For each value on the simulated **A** stack, we keep track of whether it has been spilled or not. The internal representation stores the value itself together with its spilled address. We distinguish two cases: **active** for values which are not yet spilled, or values which are already restored and used, and **spilled** for values which have been spilled and have not been used yet. *Used* in this context means that the value itself was actually needed for some computation, not just to spill it. For **active** values, the spill location is optional. For **spilled** values, the value itself is optional. With this extra metadata, we make more educated decisions on when to spill and when to restore a value.

This primarily comes down to restoring lazily. At the moment, we emit `load` instructions directly after a function call, even though those values are not used before the next call (when they need to be spilled again). Instead of emitting `load` instructions, we internally mark the value as **spilled**, with the memory location of the spill. If we end up needing the value, we generate a `load` instruction right before and we update the internal representation to reflect that. At the next spill, we only generate `store` instructions for **active A** stack values that do not already have a spill location.

This has the added benefit that it automatically sinks `load` instructions down to their use. It is beneficial to restore values just before they are required, and not earlier, because an early restore means the value is sitting in a register without being used. Since there is a very limited amount of registers, that could lead to needing to use the system stack instead (which is significantly slower), if there are too many values for the amount of registers. Restoring values as late as possible minimises the amount of wasted register space. This is something that LLVM does not do on its own.

4.4.1 Benchmark Results

The `astack` benchmark is the only program to see any difference in performance for this optimisation. That is expected, because `list` and `binarytrees` do not make extensive use of the **A** stack, while `astack` does (it is designed to benefit from this optimisation). It is 17% as fast as non-lazy spilling, and 27% for Wasm (with a 64 kiB heap). With a larger heap, this improvement is no longer visible on native x86-64, but is still fully there for Wasm.

Table 4: Relative runtimes of the `list`, `astack` and `binarytrees` benchmarks. Each column is compared separately; the runtime on x86-64 with 64kiB of heap is relative to the x86-64 runtime of *alias* (§ 4.3) with 64kiB of heap, and Wasm results are compared to Wasm results.

lazy / alias 64kiB	x86-64	Wasm
list	1.00×	1.02×
astack	0.86×	0.79×

lazy / alias 16MiB	x86-64	Wasm
list	1.00×	1.02×
astack	0.99×	0.78×
binarytrees	1.02×	1.00×

4.5 Spilling After Optimisation

One of the advantages of generating spills in an LLVM pass as in § 3.3, is that other passes can run beforehand. The inlining pass is of special interest, since that removes function calls, thereby eliminating the need for spilling and restoring. In particular this should result in a notable performance increase when inlining makes it clear that the GC is never called (i.e. either the inlined function does not call

other functions, or those functions are inlined as well and none of them allocate space on the heap). See listing 10 for an example of where this happens in code where a single **A** stack value (`%a`) is live.

```llvm
%a = call ptr @create_heap_value()
%a.loc = spill %a
%res = call i64 @f()
%a.new = restore %a.loc
use %a.new
```

```llvm
%a = call ptr @create_heap_value()

%res = ; <body of @f>

use %a
```

Listing 10: Pseudo-code for the LLVM output without (left) and with inlining before generating spills (right). The instructions `spill`, `restore` and `use` do not actually exist and are only used as example.

4.5.1 Benchmark Results

In the `astack` program, the `isNotEmpty` function is inlined, so no spills are generated. As expected, this benchmark sees a significant decrease in runtime, for LLVM (26% faster) and even more for Wasm (41% faster). We expect that the V8 compiler is able to optimise away even more of the memory operations. This effect is still present, though not as clear, when running with a heap size of 16MiB. The other benchmarks do not see such a good result, or even a performance decrease for `list`. Why this happens is unclear, and we do not rule out a bug in the implementation of the custom LLVM passes.

Table 5: Relative runtimes of the `list`, `astack` and `binarytrees` benchmarks. Each column is compared separately; the runtime on x86-64 with 64kiB of heap is relative to the x86-64 runtime of *alias* (§ 4.3) with 64kiB of heap, and Wasm results are compared to Wasm results.

pass / alias	64kiB	x86-64	Wasm
list		1.09×	1.18×
astack		0.79×	0.71×

pass / alias	16MiB	x86-64	Wasm
list		1.01×	1.16×
astack		0.97×	0.77×
binarytrees		1.04×	0.99×

4.6 Results

We now compare all different versions with each other:

- the original Clean compiler *(baseline)*,
- the ABC interpreter *(interpreter)*,
- the un-optimised LLVM implementation as in § 3.1, for native x86-64 *(LLVM)*,
- LLVM with static nodes as in § 4.2 *(LLVM+static)*,
- LLVM with static nodes and alias annotations as in § 4.3 *(LLVM+alias)*,
- LLVM with static nodes, alias annotations and lazy restoring as in § 4.4 *(LLVM+lazy)*,
- using LLVM passes to spill and restore after inlining as in § 4.5, also includes static nodes and alias annotations *(LLVM+pass)*,
- WebAssembly versions of the above 5 implementations running in V8, the JIT engine used by Node.js.

The bar plots include baseline and x86-64 and Wasm versions of the Clean-LLVM output. We explicitly exclude the ABC interpreter from the plots since it is so much slower that the differences between other benchmarks would no longer be visible. While we do also compare between baseline Clean and Wasm, remember that those run in different environments (natively on x86-64 versus in the V8 JIT engine). These are the most relevant comparisons here:

- between the basic LLVM implementation for native x86-64 and Wasm, and the optimised versions *(lazy* and *pass)*, showing that the improvements from this chapter actually make a difference,
- between the ABC interpreter and the optimised Wasm versions *(lazy* and *pass)*, showing our advantage over the interpreter,

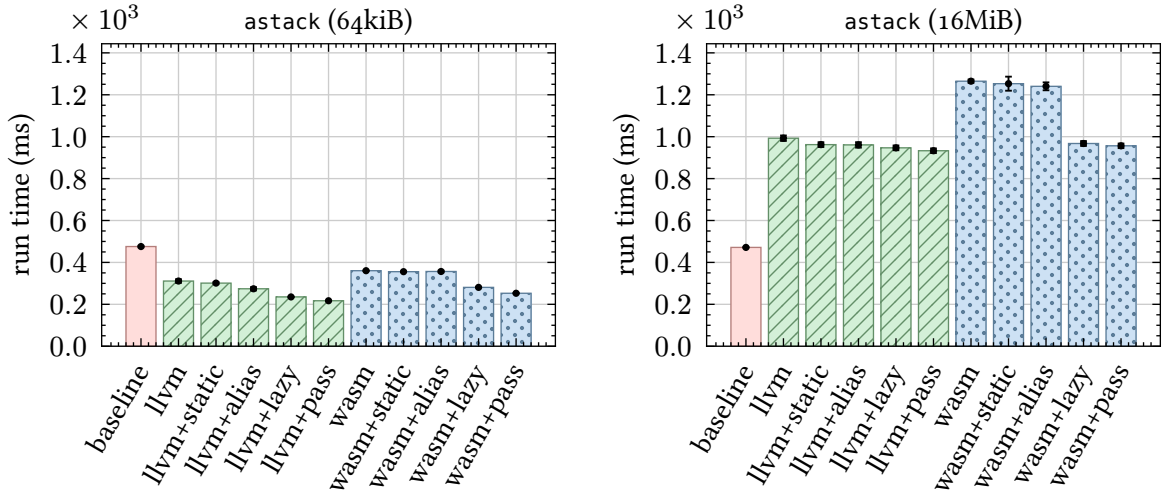
- between baseline Clean and the optimised LLVM versions for x86-64 and Wasm, showing how close we can get to the performance of the original Clean compiler. Keep in mind that the performance of baseline clean is hard to beat: it is the result of years of Clean-specific low-level optimisations and hand-written assembly code. Additionally, our LLVM output with statepoints on x86-64 still uses a shadow stack instead of walking through the system stack directly. This is unnecessary, but simply not implemented in this thesis.

We observe some general patterns. Without exception, Clean-LLVM (importantly also our Wasm output) is significantly faster than the ABC interpreter in all cases. While our generated code experiences a slowdown for larger heap sizes, both baseline Clean and the ABC interpreter do not care about the difference in heap size at all. Why those seem unaffected by this, remains a question for further research.

4.6.1 Astack

Written with the purpose to show the effects of the optimisations we perform, `astack` succeeds in its goal: for a 64 kiB heap, the basic version is already substantially faster than baseline. But the optimisations stack nicely for this benchmark: the LLVM pass version runs the program twice as fast as baseline Clean. However, as we already saw in § 4.1.3, this program is especially impacted by caching behaviour: for a heap of 16 MiB, the LLVM pass version is suddenly around twice as *slow* as baseline. Even then, Wasm is between 2 times and almost 8 times faster than the ABC interpreter, depending on the heap size.

`astack`	64 kiB			16 MiB		
baseline	475.76 ms	± 3.50 ms	(**1.00**×)	471.60 ms	± 1.25 ms	(**1.00**×)
LLVM	311.13 ms	± 8.77 ms	(**0.65**×)	992.73 ms	± 13.05 ms	(**2.11**×)
LLVM+static	301.24 ms	± 4.99 ms	(**0.63**×)	962.09 ms	± 11.46 ms	(**2.04**×)
LLVM+alias	274.16 ms	± 8.13 ms	(**0.58**×)	960.68 ms	± 13.00 ms	(**2.04**×)
LLVM+lazy	235.25 ms	± 1.43 ms	(**0.49**×)	946.69 ms	± 11.41 ms	(**2.01**×)
LLVM+pass	217.05 ms	± 0.77 ms	(**0.46**×)	932.87 ms	± 11.17 ms	(**1.98**×)
interpreter	1 991.22 ms	± 38.68 ms	(**4.19**×)	1 973.96 ms	± 6.87 ms	(**4.19**×)
Wasm	360.42 ms	± 1.16 ms	(**0.76**×)	1 264.76 ms	± 8.03 ms	(**2.68**×)
Wasm+static	355.77 ms	± 0.98 ms	(**0.75**×)	1 252.90 ms	± 33.51 ms	(**2.66**×)
Wasm+alias	356.73 ms	± 4.02 ms	(**0.75**×)	1 240.27 ms	± 18.87 ms	(**2.63**×)
Wasm+lazy	280.90 ms	± 1.21 ms	(**0.59**×)	967.43 ms	± 11.68 ms	(**2.05**×)
Wasm+pass	252.80 ms	± 1.46 ms	(**0.53**×)	956.36 ms	± 10.94 ms	(**2.03**×)

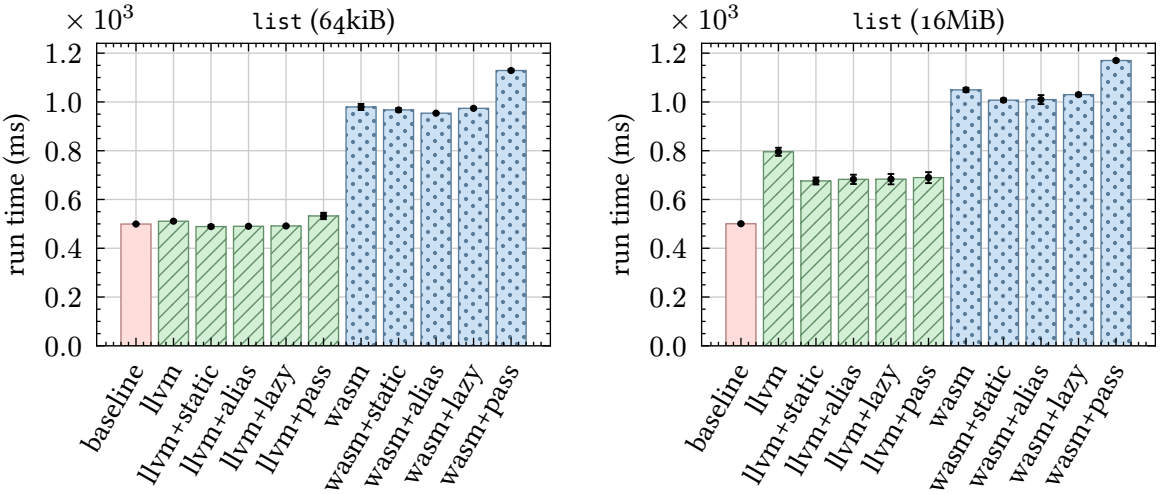


4.6.2 List

`list` is the least interesting benchmark of the three, since it does not do much with the **A** stack and it is a very synthetic benchmark. Nevertheless, LLVM on x86-64 is at worst 38% slower than baseline, but Wasm is at least 5 times as fast as the ABC interpreter, which turns into 6 times in the best case.

Wasm takes about double the time of both LLVM and baseline. We suspect that this may be caused by the amount of indirect function calls that this program does. Since lists in Clean are lazy linked lists, every step of traversing the list is an indirect function call to evaluate a thunk. Remember from § 2.4 that for Wasm, there is an extra indirection in evaluating thunks. This very likely influences the results we see here. In addition, indirect function calls in Wasm require run-time checks to ensure the safety and security guarantees set by the standard, which makes indirect calls a bigger issue for Wasm.

`list`	64 kiB			16 MiB		
baseline	499.37 ms	± 1.20 ms	(**1.00**×)	500.53 ms	± 3.66 ms	(**1.00**×)
LLVM	510.99 ms	± 1.97 ms	(**1.02**×)	795.55 ms	± 16.71 ms	(**1.59**×)
LLVM+static	488.95 ms	± 6.24 ms	(**0.98**×)	675.87 ms	± 14.38 ms	(**1.35**×)
LLVM+alias	490.15 ms	± 2.11 ms	(**0.98**×)	682.53 ms	± 19.09 ms	(**1.36**×)
LLVM+lazy	491.37 ms	± 3.42 ms	(**0.98**×)	683.38 ms	± 21.23 ms	(**1.37**×)
LLVM+pass	532.41 ms	± 13.07 ms	(**1.07**×)	689.66 ms	± 22.54 ms	(**1.38**×)
interpreter	5 687.92 ms	± 8.12 ms	(**11.39**×)	5 770.63 ms	± 34.68 ms	(**11.53**×)
Wasm	979.32 ms	± 12.56 ms	(**1.96**×)	1 049.58 ms	± 8.42 ms	(**2.10**×)
Wasm+static	967.17 ms	± 7.25 ms	(**1.94**×)	1 007.35 ms	± 6.40 ms	(**2.01**×)
Wasm+alias	953.94 ms	± 1.60 ms	(**1.91**×)	1 009.60 ms	± 18.51 ms	(**2.02**×)
Wasm+lazy	973.86 ms	± 2.48 ms	(**1.95**×)	1 029.96 ms	± 5.05 ms	(**2.06**×)
Wasm+pass	1 128.89 ms	± 4.23 ms	(**2.26**×)	1 169.92 ms	± 4.59 ms	(**2.34**×)

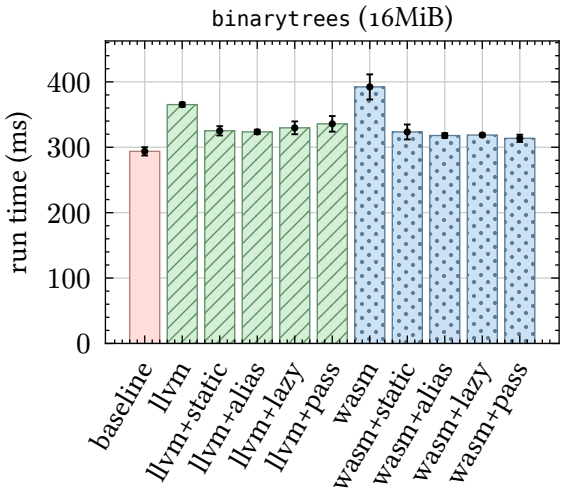


4.6.3 Binarytrees

Finally, we look at `binarytrees`. Out of the three benchmarks we have, this is the most realistic one. This program comes closest to how a 'real' program would behave, so it is important that its performance is good. Looking at the results, it seems that is indeed the case: while Clean-LLVM is slower than baseline across the board (which is to be expected), it stays within 14% for x86-64. As for WebAssembly, it manages to keep up with native code as well: the optimised Wasm versions running in Node.js are only 8% slower than baseline Clean on x86-64. This means that it is about as fast to run this program in the browser using Clean-LLVM as it is to run it natively using the original Clean compiler. Compared to the ABC interpreter, our Wasm output runs more than 7 times faster.

The different optimisations do not achieve any significant speedup, except for using static nodes which speeds up LLVM on x86-64 1.1 times and 1.2 times for Wasm.

`binarytrees`			16 MiB
baseline	293.71 ms	± 6.38 ms	(**1.00**×)
LLVM	365.19 ms	± 3.27 ms	(**1.24**×)
LLVM+static	325.11 ms	± 7.11 ms	(**1.11**×)
LLVM+alias	323.47 ms	± 3.35 ms	(**1.10**×)
LLVM+lazy	329.66 ms	± 9.81 ms	(**1.12**×)
LLVM+pass	335.75 ms	± 11.94 ms	(**1.14**×)
interpreter	2 282.92 ms	± 41.61 ms	(**7.77**×)
Wasm	392.26 ms	± 19.19 ms	(**1.34**×)
Wasm+static	323.37 ms	± 11.43 ms	(**1.10**×)
Wasm+alias	317.92 ms	± 3.68 ms	(**1.08**×)
Wasm+lazy	318.66 ms	± 2.37 ms	(**1.08**×)
Wasm+pass	313.64 ms	± 5.57 ms	(**1.07**×)



5 Related Work

It's the kind of thing that makes you glad you stopped and smelled the pine trees along the way, you know?

— Gabbro, Outer Wilds

Programming language development is a field where academic research, professional implementation and hobby projects come together. That is also seen in this chapter, which lists all three of these kinds of relevant earlier work.

The idea of a shadow stack as supported by LLVM with its gcroot mechanism is originally described by Henderson [16], although LLVM's implementation differs slightly. Henderson mentions some performance issues with the described implementation, as well as a couple of ways to somewhat (but not completely) alleviate them.

There has been lots of research on the performance of WebAssembly. When compared to native x86-64 code, Wasm is about 1–3× slower [17, 27]. However, it is found to usually run faster than the same program written in JavaScript [32]. There have even been benchmarks for using Wasm on embedded devices [20]. These show that Wasm is more performant than Micropython and Lua in that scenario, but native code still runs significantly faster.

5.1 WebAssembly for Functional Languages

There are several other functional languages that compile to WebAssembly. Some use LLVM, some use Emscripten (a tool to compile C to Wasm that also uses LLVM internally), and others directly generate Wasm code. They each have a different approach for integrating a garbage collector.

The (unofficial and in development) Elm Wasm compiler uses Emscripten. It has a custom garbage collector specifically for use with Wasm, for which they use a custom stack map [11], very much like we describe in § 3.1. Their GC is a mark-sweep collector with compaction, so it also moves heap values around in memory like the copying GC we use for Clean-LLVM.

Wasm_of_Ocaml is a compiler from Ocaml to Wasm. It uses multiple proposed WebAssembly extensions, including the GC extension [21]. This means that they do not have their own GC but instead make use of Wasm to manage their memory. There is also Wasocaml [6], which is very similar.

Haskell has a WebAssembly back end using LLVM with a custom calling convention that passes everything in registers [29, 30]. They use the same generational GC as for native targets, and they do not use LLVM to manage GC roots [3]. There is also research on generating Wasm from Haskell using the MicroHs lightweight Haskell compiler, using WebAssembly's GC extension [5].

Erlang has an LLVM back end called ErLLVM which uses LLVM's `gcroot` functionality for keeping track of heap values [26]. They note a performance issue about having to explicitly write a sentinel value when a heap reference is no (longer) live. To bypass this issue slightly, they implement more complex

code generation, using register colouring for the stack slots. They also use the register pinning method of threading the pinned values through function calls as arguments and return values.

5.2 Garbage-Collected Languages Using LLVM

Julia employs several custom LLVM passes for optimisation and integrating the GC [9]. They use four custom address spaces for tracking which values are garbage-collectable, together with operand bundles and non-integral pointers; the custom address spaces are removed in a later pass which we use in Clean-LLVM as well. They use the system stack (with `alloca`) to store heap pointers. However, their garbage collector is non-moving, saving them from having to restore after calls.

The GraalVM Java implementation can use LLVM as 'Native Image' back-end, among others. For its copying garbage collector (enabled by default[1]), it uses LLVM statepoints [22] to generate stack maps. GraalVM uses LLVM's built-in `rewrite-statepoints-for-gc` pass to generate statepoint instructions. Like we decided to do in § 4.5, they first run optimisations before generating the statepoints.

Google's Go language has multiple compilers based on LLVM: TangoLLVM [14] uses statepoints, TinyGo [4] is meant for microcontrollers and also supports Wasm compilation, and there is also LLGo [12].

[1] https://www.graalvm.org/latest/reference-manual/native-image/optimizations-and-performance/MemoryManagement/#serial-garbage-collector (accessed: Nov. 18, 2025)

6 Conclusions

If you've come here looking for answers, I hope you find them.
— Solanum, Outer Wilds

We have integrated the copying garbage collector of the ABC interpreter into Clean-LLVM, using both a custom shadow stack approach and using LLVM's statepoints. We improved the performance of both of these implementations by using static nodes and by using LLVM's alias metadata. The custom shadow stack is improved by restoring in a lazy way, and we minimise the amount of spilling further by running LLVM's inlining pass before generating spills. We have evaluated the effect of these optimisations using three benchmarks: `list`, `astack` and `binarytrees`. The `astack` benchmark shows that each of these optimisations have a measurable, positive effect on the runtime. For `binarytrees`, a realistic benchmark program, the output of Clean-LLVM manages to come very close to the performance of the existing Clean compiler when running natively on x86-64. More importantly, our Wasm output is more than 7 times faster than the ABC interpreter. These are very promising results for running non-trivial Clean programs on the web.

6.1 Future Work

In this thesis we have evaluated our efforts using three benchmark programs. This is of course not a lot of data points, and other programs will behave differently. More programs should be implemented and benchmarked, preferably ones that are academically well-known such as from the Nofib benchmark suite or the Computer Language Benchmarks Game. Most of these non-trivial programs require features or instructions that are not yet implemented in Clean-LLVM, so benchmarking these programs is only possible when those are present.

Besides benchmarking the runtime performance, one can also compare the size of the generated code, as well as the amount of memory used and the time taken to compile. Since LLVM is an optimising compiler and the baseline Clean compiler is not, it is to be expected that the compilation time of Clean-LLVM will be significantly longer.

We have shown that generating spill/restore instructions in an LLVM pass (using statepoints) can be faster than when done during code generation. However, one of the reasons to use statepoints for this is that it allows us to do away with spilling entirely for targets that support stack unwinding. This has not been implemented, and doing so should result in even better runtime performance for those targets.

The list of optimisations in chapter 4 is far from complete and there are very likely still some performance improvements possible. Further work could go into generating more efficient LLVM IR code and giving LLVM more information for optimisations. More ideas for improving runtime performance did not fit within the timeline of this thesis. One of them is that spilling does not need to happen around calls to functions that never run the GC. Another is based on the fact that the **A** stack pointer returned by

a function call is always the same value as the one given as argument to that function call. LLVM does not know that, which potentially hinders some optimising transformations.

Another possible optimisation is based on the fact that the **A** stack values may no longer be in registers at the time they are spilled. If there are not enough registers, some values will be spilled to the system stack by LLVM's code generation. When that happens, spilling one of these values for the GC requires reading from memory just to place it somewhere else in memory (so that the GC can read it). This can be solved by spilling as early as possible and restoring as late as possible. This keeps the registers free for other uses. Preliminary results (done by manually altering the generated LLVM IR) show that this significantly improves performance in cases with lots of local heap references.

We currently generate spills from statepoints in creation order of the spilled values: older values are spilled first, newer values last. This is not done for any particular reason, but the order is actually important. We want to prevent a situation where spilled values move around on the spilled stack between two function calls. This already works for the custom shadow stack approach since it bases the spill order on where values appear on the **A** stack, which is directly based on the baseline Clean compiler's output. Further research is needed on how to best sort the spills and restores generated from the statepoints.

As the results show, Clean-LLVM is sometimes still significantly slower than baseline Clean, also in scenarios where the **A** stack is not used intensively. Our generated code is also much more sensitive to different heap sizes (see § 4.1.3). Ultimately, it is necessary to find out where this difference in behaviour comes from and how to get the performance up to the level of baseline Clean and beyond, also with larger heap sizes.

Bibliography

[1] Clean-LLVM. Retrieved 21 October 2025 from https://gitlab.com/clean-and-itasks/clean-llvm

[2] The Clean Language. Retrieved 21 October 2025 from https://top-software.gitlab.io/clean-lang/

[3] How does GHC's garbage collection work on LLVM?. Retrieved 21 October 2025 from https://www.reddit.com/r/haskell/comments/3qix1e/how_does_ghcs_garbage_collection_work_on_llvm/

[4] TinyGo authors. TinyGo - Go compiler for small places. Retrieved 19 January 2026 from https://tinygo.org/

[5] Apoorva Anand. 2025. Towards a WebAssembly Backend for MicroHs. Master's thesis.

[6] Léo Andrès, Pierre Chambart, and Jean-Christophe Filliâtre. 2023. Wasocaml: Compiling OCaml to WebAssembly. Retrieved from https://inria.hal.science/hal-04311345

[7] Clemens Backes. 2018. Liftoff: a new baseline compiler for WebAssembly in V8. Retrieved 15 January 2026 from https://v8.dev/blog/liftoff

[8] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual machine warmup blows hot and cold. In *Proceedings of the ACM on Programming Languages*, October 12, 2017. Association for Computing Machinery. https://doi.org/10.1145/3133876

[9] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Working with LLVM. Retrieved 21 October 2025 from https://docs.julialang.org/en/v1/devdocs/llvm

[10] Tom H. Brus, Marko. C. J. D. van Eekelen, Maarten. O. van Leer, and Marinus. J. Plasmeijer. 1987. Clean — A Language for Functional Graph Rewriting. In *Functional Programming Languages and Computer Architecture*, 1987. Springer, 364–384. https://doi.org/10.1007/3-540-18317-5_20

[11] Brian Carroll. elm_c_wasm/src/kernel/core/gc/stack.c. Retrieved 21 October 2025 from https://github.com/brian-carroll/elm_c_wasm/blob/master/src/kernel/core/gc/stack.c#L5-L39

[12] Goplus. LLGo - A Go compiler based on LLVM. Retrieved 19 January 2026 from https://github.com/goplus/llgo

[13] Isaac Gouy. The Computer Language Benchmarks Game. Retrieved 27 November 2025 from https://benchmarksgame-team.pages.debian.net/benchmarksgame/

[14] Tianxiao Gu, Rui Feng, Fengkai Liu, and Nian Sun. 2025. TangoLLVM: An LLVM Backend for the Go compiler. Retrieved from https://llvm.org/devmtg/2025-10/slides/quick_talks/gu.pdf

[15] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. 2017. Bringing the web up to speed with WebAssembly.

In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 14, 2017. Association for Computing Machinery, 185–200. https://doi.org/10.1145/3062341.3062363

[16] Fergus Henderson. 2002. Accurate garbage collection in an uncooperative environment. In *Proceedings of the 3rd International Symposium on Memory Management*, June 20, 2002. Association for Computing Machinery, 150–156. https://doi.org/10.1145/512429.512449

[17] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. 2019. Not so fast: Analyzing the performance of WebAssembly vs. native code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, July 2019. 107–120.

[18] Pieter W. M. Koopman. 1990. Functional programs as executable specifications. Dissertation. Radboud University Press.

[19] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*, March 2004. IEEE, 75–86. https://doi.org/10.1109/CGO.2004.1281665

[20] Konrad Moron and Stefan Wallentowitz. 2025. Benchmarking WebAssembly for Embedded Systems. *ACM Trans. Archit. Code Optim.* 22, 3 (September 2025). https://doi.org/10.1145/3736169

[21] Ocsigen. Wasm_of_ocaml. Retrieved 21 October 2025 from https://github.com/ocsigen/js_of_ocaml/blob/master/README_wasm_of_ocaml.md

[22] Oracle. LLVM Backend for Native Image. Retrieved 21 October 2025 from https://github.com/oracle/graal/blob/master/substratevm/src/com.oracle.svm.core.graal.llvm/src/com/oracle/svm/core/graal/llvm/LLVMBackend.md#llvm-statepoint-support

[23] David Peter. 2023. hyperfine. Retrieved 21 October 2025 from https://github.com/sharkdp/hyperfine

[24] Rinus Plasmeijer, Peter Achten, and Pieter Koopman. 2007. iTasks: executable specifications of interactive work flow systems for the web. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, October 01, 2007. Association for Computing Machinery, 141–152. https://doi.org/10.1145/1291151.1291174

[25] Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter Koopman. 2012. Task-oriented Programming in a Pure Functional Language. In *Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming*, September 19, 2012. Association for Computing Machinery, 195–206. https://doi.org/10.1145/2370776.2370801

[26] Konstantinos Sagonas, Chris Stavrakakis, and Yiannis Tsiouris. 2012. ErLLVM: an LLVM backend for Erlang. In *Proceedings of the Eleventh ACM SIGPLAN Workshop on Erlang Workshop*, September 14, 2012. Association for Computing Machinery, 21–32. https://doi.org/10.1145/2364489.2364494

[27] Benedikt Spies and Markus Mock. 2021. An Evaluation of WebAssembly in Non-Web Environments. *2021 XLVII Latin American Computing Conference (CLEI)* (October 2021). https://doi.org/10.1109/CLEI53233.2021.9640153

[28] Camil Staps, John van Groningen, and Rinus Plasmeijer. 2021. Lazy interworking of compiled and interpreted code for sandboxing and distributed systems. In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages*, July 15, 2021. Association for Computing Machinery. https://doi.org/10.1145/3412932.3412941

[29] David A. Terei and Manuel M. T. Chakravarty. 2010. An LLVM backend for GHC. In *Proceedings of the Third ACM Haskell Symposium on Haskell*, September 30, 2010. Association for Computing Machinery, 109–120. https://doi.org/10.1145/1863523.1863538

[30] David Anthony Terei. 2009. Low level virtual machine for Glasgow Haskell Compiler. Bachelor's thesis. Retrieved from https://www.llvm.org/pubs/2009-10-TereiThesis.pdf

[31] Sofie Vos. 2025. Running iTasks tasks in the browser. Bachelor's thesis. Retrieved from https://cs.ru.nl/bachelors-theses/2024/Sofie_Vos___1068747___Running_iTasks_tasks_in_the_browser.pdf

[32] Yutian Yan, Tengfei Tu, Lijian Zhao, Yuchen Zhou, and Weihang Wang. 2021. Understanding the Performance of WebAssembly Applications. In *Proceedings of the 21st ACM Internet Measurement Conference*, November 02, 2021. Association for Computing Machinery, 533–549. https://doi.org/10.1145/3487552.3487827

[33] Alon Zakai. 2023. A new way to bring garbage collected programming languages efficiently to WebAssembly. Retrieved 13 November 2025 from https://v8.dev/blog/wasm-gc-porting

A

Benchmark Programs

A1 `binarytrees`

clean

```
/* The Computer Language Shootout
   http://shootout.alioth.debian.org/

   contributed by Isaac Gouy (Clean novice)
   corrected by John van Groningen
   modified for Clean-LLVM

   https://web.archive.org/web/20111118030741/http://shootout.alioth.debian.org/u64/program.php?test=binarytrees&lang=clean&id=3
*/

implementation module binarytrees

start :: Int
start
    # stretch`   = max` + 1
    # io         = 0
    #! io        = showItemCheck 0 (bottomup stretch`) io
    #! longLived = bottomup max`
    #! io        = depthloop min` max` io
    #! io        = showItemCheck 0 longLived io
    = io

min` = 4
max` = 16

showItemCheck i a io = io + itemcheck i a

depthloop d m io
    | m < d = io
            = depthloop (d+2) m (io + check)
where
    n = 1 << (m - d + min`)
    check = sumloop n d 0

sumloop :: !Int !Int !Int -> Int
sumloop n d sum
    | 0 < n = sumloop (n-1) d (sum + check + check`)
            = sum
    where
    check = itemcheck n (bottomup d)
    check` = itemcheck (-1*n) (bottomup d)

:: Tree = TreeNode !Tree !Tree | Nil

```

```
45 bottomup :: !Int -> Tree
46 bottomup 0 = TreeNode Nil Nil
47 bottomup d = TreeNode (bottomup (d-1)) (bottomup (d-1))
48
49 itemcheck i Nil = i
50 itemcheck i (TreeNode left right) = i + itemcheck (2*i-1) left - itemcheck (2*i) right
51
52 class  (+)  infixl 6 a :: !a !a -> a
53 class  (-)  infixl 6 a :: !a !a -> a
54 class  (<)  infix  4 a :: !a !a -> Bool
55 class  (*)  infixl 7 a :: !a !a -> a
56
57 instance + Int
58 where
59     (+) :: !Int !Int -> Int
60     (+) a b
61         = code inline {
62             addI
63         }
64
65 instance - Int
66 where
67     (-) :: !Int !Int -> Int
68     (-) a b
69         = code inline {
70             subI
71         }
72
73 instance < Int
74 where
75  (<) :: !Int !Int -> Bool
76  (<) a b
77     = code inline {
78             ltI
79     }
80
81 instance * Int
82 where
83  (*) :: !Int !Int -> Int
84  (*) a b
85     = code inline {
86             mulI
87     }
88
89 (<<) infix 7 :: !Int !Int -> Int
90 (<<) a b
91     = code inline {
92             shiftl%
93     }
```

A2 `list`

clean

```
implementation module list

start :: Int
start = sum (from_to 0 20000000)

from_to :: !Int !Int -> [Int]
from_to x y
    | y < x
        = []
        = [x : from_to (x+1) y]

sum :: ![Int] -> Int
sum xs = accsum 0 xs
where
    accsum :: !Int ![Int] -> Int
    accsum acc [] = acc
    accsum acc [x:xs] = accsum (acc+x) xs

class  (+)  infixl 6 a :: !a !a -> a
class  (<)  infix  4 a :: !a !a -> Bool

instance + Int
where
  (+) :: !Int !Int -> Int
  (+) a b
    = code inline {
      addI
    }

instance < Int
where
 (<) :: !Int !Int -> Bool
 (<) a b
  = code inline {
      ltI
  }

```

A3 astack

clean

```
implementation module astack

:: Node = A Int

start :: Int
start = loop 5000000 0

loop :: !Int !Int -> Int
loop 0 acc = acc
loop x acc = loop (x - 1) (acc + g x)

g :: !Int -> Int
g x =
    let x1  = A 1
        x2  = A 1
        x3  = A 1
        x4  = A 1
        x5  = A 1
        x6  = A 1
        x7  = A 1
        x8  = A 1
        x9  = A 1
        x10 = A 1
        x11 = A 1
        x12 = A 1
        x13 = A 1
        x14 = A 1
        x15 = A 1
        x16 = A 1
    in
        isNotEmpty [x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13, x14, x15, x16] +
        isNotEmpty [x1] +
        isNotEmpty [x2] +
        isNotEmpty [x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13, x14, x15, x16]

isNotEmpty [] = 0
isNotEmpty _  = 1

class  (+)  infixl 6 a :: !a !a -> a
class  (-)  infixl 6 a :: !a !a -> a

instance + Int
where
    (+) :: !Int !Int -> Int
    (+) a b
        = code inline {
            addI
        }

instance - Int
where
    (-) :: !Int !Int -> Int
    (-) a b
        = code inline {
            subI
        }

```