The Implementation and Evaluation of a Compiler from a Subset of Racket to WebAssembly as a New Back End for the 838e Compiler

Danna Doratotaj, Advised by: Professor David Van Horn

Department of Computer Science, University of Maryland

May 18, 2021

Abstract

In this project, we incrementally developed the WebAssembly back end and the compiler from a subset of Racket to WebAssembly following the methods for the development of our 838e compiler from a subset of Racket to the a86 back end (an abstraction of the x86 instructions and programs as data types in Racket) and then from a86 to x86. The Wasm compiler works correctly as demonstrated by running our tests for the subset of expressions covered by this compiler, which includes integers, booleans, chars, if expressions, n-ary let expressions, expressions for making heap-allocated data structures, including boxes, pairs, lists, and strings, expressions for doing input and output, including read-byte, peek-byte, and write-byte, unary operations, binary operations, function definitions, and function calls. We also implemented two runtimes for the WebAssembly back end: The first runtime is written in Node.js and uses the JavaScript API for WebAssembly for interacting with the compiled WebAssembly program via imports to the WebAssembly program and exports from the WebAssembly program. The second runtime is implemented using Wasmtime and WebAssembly code with the use of either the Wasmtime fd_write function or the function print_codepoint that we added to Wasmtime for printing the chars, the strings, and the boolean results, and the functions print_int and error_exit that we added to Wasmtime for printing the integers and printing of "err" followed by exiting the program, respectively. We compared the speed of the different combinations of our compilers, runtimes, and the methods of printing strings in the case of running the compiled code obtained from each method for the expression (make-string 1000000 #\λ) in the file str1000000lms.rkt. With our x86 backend with C runtime, the average times were real: 0.584s, user: 0.059s, and sys: 0.053s. With the Wasm back end with Node.js runtime, the average times were real: 0.988s, user: 0.650s, and sys: 0.161s. With the Wasm back end with the Wasmtime and WebAssembly runtime with the use of the Wasmtime fd_write function, the average times were real: 0.252s, user: 0.013s, and sys: 0.031s. With the Wasm back end with the Wasmtime and WebAssembly runtime with the use of the function print_codepoint, the average times were real: 1.089s, user: 0.234s, and sys: 0.053s.

1. Introduction

WebAssembly (contraction: Wasm, acronym: WA) is an open standard programming language with a low-level virtual instruction set for a stack-based virtual machine [1-4]. It has a portable binary-code format (.wasm) for executable programs, and a corresponding human-readable textual format (.wat) [1-3].

The main goal of WebAssembly was initially to enable high-performance, near-native code execution speed in a memory-safe and secure way for the applications on webpages in the web browser [1-3]. But it is designed as a portable compilation target for programming languages with its core language being independent of its surrounding environment and with the interactions with the outside world being exclusively through APIs [1-5]. Therefore, WebAssembly can be executed and integrated in other environments as well and can provide a fast, scalable, secure way to run the same code across different machines [3,6]. Given this, a standardization effort for a WebAssembly system interface (WASI) with the goal of designing clean and portable system-oriented APIs for safely exposing low-level interfaces to the filesystem, networking, and other system facilities to WebAssembly programs has been in progress, and standalone runtime environments including Wasmtime, Lucet, and Wasmer have been developed [3-7].

The open standards for WebAssembly are developed in a W3C WebAssembly Community Group (that includes representatives from all major browsers) and WebAssembly Working Group [1-4,8,9]. In March 2017, the design of the minimum viable product was declared to be finished and the preview phase ended. Since 2017, support for WebAssembly has been included in all major browsers [3]. In February 2018, the WebAssembly Working Group published three public working drafts for the Core Specification, JavaScript Interface, and Web API [3].

1.1 Design Goals of WebAssembly

WebAssembly addresses the problem of safe, fast, portable low-level code on the Web [2]. For this purpose, it is designed to have fast, safe, and portable semantics, and efficient and portable representation [1,2]. Its semantics is fast with execution with near native code performance, taking advantage of capabilities common to all contemporary hardware, and safe with code validation and execution in a memory-safe, sandboxed environment preventing data corruption or security breaches [1,2]. Its semantics is also designed to be well-defined, hardware-independent, language-independent, platform-independent, and open [1,2].

It is also designed to have an efficient and portable representation that is compact, with a binary format that is fast to transmit by being smaller than typical text or native code formats, is modular, such that programs can be split up in smaller parts that can be transmitted, cached, and consumed separately, is efficient with the ability to be decoded, validated, and compiled in a fast single pass, equally with either just-in-time or ahead-of-time compilation, and in addition is streamable, parallelizable, and portable, making no architectural assumptions that are not broadly supported across modern hardware [1,2].

A formal mechanized Isabelle specification for the WebAssembly language have been done, and a verified executable interpreter and type checker, have been implemented for WebAssembly [10]. In addition, a fully mechanized proof of the soundness of the WebAssembly type system with detection of issues with the specification, influencing its development have been done [10].

1.2 Language Tools Developed in Association with WebAssembly

1.2.1 Emscripten: Emscripten is a compiler toolchain for compiling C, C++, or any other language that uses LLVM to WebAssembly [11-13]. It uses the LLVM system, the LLVM WebAssembly backend, and Clang (a C/C++ compiler front end to the LLVM system) [11-14]. In addition, it uses Binaryen for optimizing the generated WebAssembly code [11,12]. Emscripten output can run on the Web, in Node.js, and in Wasm runtimes [11]. It can also compile the C/C++ runtimes of other languages into WebAssembly [11].

1.2.2 Binaryen: Binaryen is a compiler and toolchain infrastructure library for WebAssembly, written in C++ [15,16]. It aims to make compiling to WebAssembly easy, fast, and effective [15]. It has an internal IR (intermediate representation) that is almost identical to WebAssembly that allows only s-expressions and not linear format [15,16]. It has a C API to compile to this IR [15, 16]. Binaryen also has optional support for a more general control flow graph-style input IR for convenience [15,16]. It then compiles the code from the IR to a Wasm binary with optimizations [15,16]. Its optimizer has many passes that can improve code size and speed [15,16]. Its wasm-opt tool takes a WebAssembly .wasm or .wat file and runs the optimizer on it [15,17].

1.2.3 Cranelift: Cranelift is a low-level retargetable optimizing code generator written in Rust [18-21]. It translates a target-independent intermediate representation (Cranelift IR) into executable machine code [18-21]. It is designed to compile WebAssembly code to native machine code, but it is general enough to be useful elsewhere too [18]. The x86-64 backend for Cranelift is currently the most complete and stable; other architectures are in various stages of development [18]. Cranelift currently supports both the System V AMD64 ABI (Application Binary Interface) calling convention used on many platforms and the Windows x64 calling convention [18].

1.2.4 Lucet: Lucet is a native WebAssembly compiler and runtime developed by Fastly and is the engine behind Terrarium, Fastly's experimental platform for edge computation using WebAssembly [7]. It is built on top of the Cranelift code generator [7]. It has a compiler, which compiles WebAssembly modules to native code, and a runtime which manages resources and traps runtime faults [7]. The compilation is ahead-of-time [7]. Lucet supports the WebAssembly System Interface (WASI) [7].

1.2.5 Wasmtime: Wasmtime is a standalone runtime for WebAssembly that is built on the optimizing Cranelift code generator to quickly generate high-quality machine code at runtime [22,23]. It runs the WebAssembly code outside of the Web and can be used both as a command-line utility or as a library embedded in a larger application [23].

Wasmtime supports the WebAssembly System Interface (WASI) APIs [22-24]. For example, it has an implementation of WASI's fd_write function to write the characters of a string stored in the memory to stdout using the Rust write_vectored function [22,24,25].

Wasmtime command-line utility can take a WebAssembly module in both .wasm and .wat formats as an argument and execute the module [22,23]. There is also an experimental subcommand wasm2obj to compile a WebAssembly module to native code (with the command: $ wasmtime wasm2obj foo.wasm foo.o) [23].

Wasmtime also has an implementation for a C API which supports embedding in C [22,23]. This is based on the WebAssembly C and C++ API [26]. Embedding in Rust can also be done through the wasmtime crate which contains an API to interact with WebAssembly [23,27].

Wasmtime has support for the following platforms: Linux x86_64, Linux aarch64, macOS x86_64, Windows x86_64 [23]. A working ARM64 (AArch64) backend was added in 2020 and appropriate integrations have been made so that wasmtime can run basic Wasm tests correctly on ARM64 hardware [20,28].

1.2.5 The JavaScript API for WebAssembly: This API was the first API on which the WebAssembly Community Group reached consensus in 2017 [3,29]. Node.js is a back-end JavaScript runtime environment that runs on the Google's V8 JavaScript and WebAssembly engine and executes JavaScript code outside a web browser [30-32]. Node.js lets developers use JavaScript to write command line tools and for server-side scripting [30]. Node.js implements the JavaScript API for WebAssembly and in addition has support for WASI [33,34].

1.2.6 WABT (The WebAssembly Binary Toolkit): WABT is a suite of tools that includes wat2wasm for converting .wat to .wasm format and wasm2wat for doing the inverse, among other tools [35].

1.2.7 Wasmer: Wasmer is an open-source runtime for executing WebAssembly on the Server [36]. Its goals are to provide the tools to compile everything to WebAssembly and run it on any OS or embed it into other languages [36]. Wasmer runtime can be used as a library embedded in different languages, including Rust, C/C++, D, Python, Go, PHP, Ruby, Java, Elixir, and R [36,37]. Other languages are also being added [36,37]. Wasmer ships with three different backends: Singlepass, Cranelift and LLVM [38]. AArch64 (ARM64) support in Singlepass backend was added in 2019 [38,39]. As a result, Wasmer can run WebAssembly programs in ARM64 chipsets in addition to x86 [38,39].

1.3 Notes on Intermediate Representation

In this project, we developed a compiler from a subset of Racket directly to WebAssembly. However, the modern compilers in general, and the compilers from different programming languages to WebAssembly usually first compile to one or more layers of intermediate representations (with LLVM being the major one used) and then compilation is done from the IR to the target language. This is done because of the additional optimizations that can be performed with the intermediate representations, and because a well-designed IR such as LLVM can support translation of code to a nonspecific target machine allowing development of back ends for multiple instruction set architectures [40-46]. For example, as mentioned above, Emscripten, uses Clang and the LLVM system (with the LLVM WebAssembly backend) for compilation of code

from C/C++ to LLVM and then from LLVM to WebAssembly [11,45]. This will allow the optimizations afforded by LLVM to be performed on the code [44,45]. In addition, Emscripten uses Binaryen to do further optimizations on the generated Wasm code [11].

The LLVM system has several features that allow extensive optimizations that we briefly mention here [44]. It performs sophisticated transformations at link-time, run-time, and after the software is installed in the field [44]. The compiler front ends emit code in the LLVM virtual instruction set [44]. The LLVM optimizing linker combines these LLVM object files, optimizes them, and finally integrates them into a native executable which it writes to disk [44]. The LLVM virtual instruction set is designed as a low-level representation with high-level type information [44]. Its goal is to be low-level enough to allow significant amounts of optimization in the early phases of compilation, while being high-level enough to support aggressive link- and post-link time optimizations [44]. It provides extensive language independent type information about all values in the program, exposes memory allocation directly to the compiler, and is specifically designed to have uniform abstractions [44].

The executable written by the LLVM optimizing linker contains native machine code directly executable on the host architecture as well as a copy of the LLVM bytecode for the application itself [44]. As the application is executed in the field, a runtime reoptimizer may monitor the execution of the program, collecting profile information about typical usage patterns for the application [44]. Optimization opportunities detected from application behavior may cause the runtime reoptimizer to dynamically recompile and reoptimize portions of the application (using the stored LLVM bytecode) [44]. However, some transformations may be too expensive to perform directly at runtime [44]. For these transformations, idle time on the machine is used by an offline optimizer to recompile the application using aggressive interprocedural techniques and the accurate profile information detected from the end-user's actual usage patterns [44].

As mentioned above, there are toolchains for compilation from LLVM to WebAssembly [11]. Therefore, if we write a compiler from Racket to LLVM, the available tools can be used to do the rest of the compilation to WebAssembly, and in addition LLVM will do the above optimizations.

In addition, there is an intermediate language called MIL (a "mondadic intermediate language) that has been specifically designed for use in implementation of functional programming languages [46]. It can be used in between the source language as the front end and the LLVM as the back end and has an optimizer that can do additional optimizations specific to functional programming languages [46]. Therefore, first compiling from Racket to MIL, and then from MIL to LLVM may allow more optimizations to be performed.

1.4 Overview of WebAssembly Concepts

The computational model of WebAssembly is based on a *stack machine* [1,2]. Code consists of sequences of *instructions* that are executed in order [1,2]. Instructions manipulate values on an implicit *operand stack,* consuming (popping) argument values and producing or returning (pushing) result values [1,2]. In the text format, the instructions can be written in a linear instruction list or in s-expression form [1,16,34,47-51]. WebAssembly is structured around the following concepts [1]:

Values: WebAssembly provides only four basic *value types* [1]. These are integers and IEEE 754-2019 numbers, each in 32- and 64-bit width. 32-bit integers also serve as Booleans and as memory addresses [1].

Instructions: Instructions fall into two main categories [1]. *Simple* instructions perform basic operations on data [1]. They pop arguments from the operand stack and push results back to it [1]. *Control* instructions alter control flow [1]. WebAssembly does not offer simple jumps [1,2]. Control flow is *structured*, meaning it is expressed with well-nested constructs such as blocks, loops, and conditionals [1]. Branches can only target such constructs [1]. For the list of the different instructions and their details please see the WebAssembly Specification [1].

Traps: Under some conditions, certain instructions may produce a *trap*, which immediately aborts execution [1]. Traps cannot be handled by WebAssembly code, but are reported to the outside environment, where they typically can be caught [1].

Functions: Code is organized into separate functions [1]. Each function takes a sequence of values as parameters and returns a sequence of values as results [1]. Functions can call each other, including recursively, resulting in an implicit call stack that cannot be accessed directly [1]. Functions may also declare mutable *local variables* that are usable as virtual registers [1].

Tables: A *table* is an array of opaque values of a particular *element type* [1]. It allows programs to select such values indirectly through a dynamic index operand [1]. Currently, the only available element type is an untyped function reference [1]. Thereby, a program can call functions indirectly through a dynamic index into a table [1]. For example, this allows emulating function pointers by way of table indices [1].

Linear Memory: A *linear memory* is a contiguous, mutable array of raw bytes [1]. Such a memory is created with an initial size but can be grown dynamically [1]. A program can load and store values from/to a linear memory at any byte address (including unaligned) [1]. Integer loads and stores can specify a *storage size* which is smaller than the size of the respective value type [1]. A trap occurs if an access is not within the bounds of the current memory size [1].

Modules: A WebAssembly binary takes the form of a *module* that contains definitions for functions, tables, and linear memories, as well as mutable or immutable *global variables* [1]. Definitions can also be *imported*, specifying a module/name pair and a suitable type [1]. Each definition can optionally be *exported* under one or more names [1]. In addition to definitions, modules can define initialization data for their memories or tables that takes the form of *segments* copied to given offsets [1]. They can also define a *start function* that is automatically executed [1].

Embedder: A WebAssembly implementation will typically be embedded into a host environment [1]. This environment defines how loading of modules is initiated, how imports are provided (including host-side definitions), and how exports can be accessed [1]. The details of any particular embedding are provided by environment-specific API definitions [1].

2. The Implementation of the WebAssembly Back End and the Compiler from a Subset of Racket to this Back End

Following the methods for the development of our 838e compiler from a subset of Racket to the a86 back end (an abstraction of the x86 instructions and programs as data types in Racket), and then from a86 to x86, we incrementally developed the WebAssembly back end and the compiler from a subset of Racket to WebAssembly [52-54]. This implementation is submitted in a pull request as the artifact for this project.

2.1 Overview

The Wasm compiler takes an expression in abstract syntax tree form after the expression and its subexpressions are recursively parsed by our parser in parse.rkt to structures defined in our ast.rkt file, and compiles this abstract syntax tree form to WebAssembly by recursively compiling the code for the structures of the subexpression and the expression.

Compiling of the abstract syntax tree forms of the following expressions to WebAssembly were implemented:

Integer, char, and boolean literals, (add1 e), (sub1 e), (zero? e), (if e1 e2 e3), (integer? e), (integer-length e), (char? e), (char->integer e), (integer->char e), (write-byte e), (read-byte), (peek-byte), (eof-object? e), (void), (begin e1 e2), n-ary let expressions, variables, (+ e1 e2), (- e1 e2), (box e), (cons e1 e2), empty list, (empty? e), (unbox e), (car e), (cdr e), (eq? e1 e2), string literals, (string-length e), (string? e), (string-ref e1 e2), (make-string e1 e2), function definitions as parsed from the program of the form (begin (define (f0 x0 …) e0) (define (f1 x1 …) e1) … e), and function calls as parsed from the

expressions of the form (fi e0 …). In addition, errors in type, and in range for some integer values, are checked, and if encountered, an "err" message is produced, and the program is exited.

2.1.1 The Files Added and Changed

The following new files have been added and incrementally committed, and are included in the pull request for this project:

1 – For the version of the Wasm compiler that interacts with a Node.js runtime: compile-wasm.rkt, compile-wat-file.rkt, wtypes.rkt, jsmain.js, makewat, watjs2run, watjs2run.c, watjs2run_README.md, wasm/interp.rkt, wasm/printer.rkt, wtest/compile.rkt, wtest/interp.rkt, wtest/test-runner.rkt, wtest/test-progs.rkt, and wtest/test-programs/get-progs.rkt.

2 – For the version of the Wasm compiler with Wasmtime and WebAssembly runtime: compile-wasm_2.rkt, compile-wat-file_2.rkt, wasm/interp_2.rkt, str1000000lms.rkt, wtest/compile_2.rkt, wtest/test-runner_2.rkt, the-838e-compiler/.wasmtime/bin/wasmtime, the-838e-compiler/.wasmtime/Additions_README.md, and the-838e-compiler/.wasmtime/bin/demo.wat.

3 – Other files have been added which we will discuss below. These are for measuring the speed of different runtimes and for a simpler version of compiler to x86 that has a make-string function with nonpacked string representation, with each char in one word, and a simpler C runtime that prints a string accordingly.

In addition, changes were made to .github/workflows/push.yml and runtime.h which we will discuss below. As part of the testing, the folders and the test files in test/test-programs for the languages up to the fraud language were copied unchanged and added to a new wtest/test-programs folder.

2.2 The Version of the Compiler that Interacts with the Node.js Runtime

The following is an outline of the code in the version of the compiler in the **compile-wasm.rkt** file that interacts with the implemented Node.js runtime in **jsmain.js**. The complete compiler is available in the GitHub repository.

```
(define word-size 4)  ;; in bytes

;; Expr -> Wasm
(define (compile e)
 (match e
  [(Letrec fs ls e)
   (let ((main-prog-fn
          `(func $sendResult (result i32)
            (local $a i32)        ;; local var (used as virtual register)
            (local $b i32)        ;; virtual register
            (local $c i32)        ;; virtual register
            ,@(compile-e e '()))))
     `(module
       (import "writeBytejs" "writeByte" (func $writeByte (param i32)))
       (import "readBytejs" "readByte" (func $readByte (result i32)))
       (import "peekBytejs" "peekByte" (func $peekByte (result i32)))
       (import "errorjs" "error" (func $error))
       (memory 1024)  ;; 1024 pages of memory = 2^26 bytes
       (export "memory" (memory 0))
       (global $sp (mut i32) (i32.const 67108860))
       (global $hp (mut i32) (i32.const 0))
        ,main-prog-fn
        ,@(compile-defines fs ls)
       (export "sendResult" (func $sendResult))))]))
```

```
;; Expr CEnv -> Wasm
(define (compile-e e c)
 (match e
  [(Int i)          (compile-value i)]
  …
  [(Prim2 p e1 e2)   (compile-prim2 p e1 e2 c)]
  …
  [(Let x e1 e2)     (compile-let x e1 e2 c)]
  [(LCall e es)      (compile-call (symbol->wa-label (Var-x e)) es c)]
  [(Var x)          (compile-variable x c)]))
…

;; Op2 Expr CEnv -> Wasm
(define (compile-prim2 p e1 e2 c)
 (append
  `(,@(compile-e e1 c)
   local.set $a
   global.get $sp
   i32.const ,(* word-size (length c))
   i32.sub
   local.get $a
   i32.store)
 (match p   […]   …
  ['make-string
   `(,@(compile-e e2 (cons #f c))
```

```
      ,@(assert-char)
      local.set $b             ;; $ b = the char parameter
      global.get $sp    ;; get the first param (integer) from stack
      i32.const ,(* word-size (length c))
      i32.sub
      i32.load
      ,@(assert-integer)    ;; $b should not be used in assert-type
      local.set $a             ;; $a = the int parameter
      local.get $a
      i32.const ,(imm->bits 0)  ;; call $error if the int param < 0
      i32.lt_s
      if
      call $error
      end
      global.get $hp   ;; put the heap ptr on the operand stack.
      local.get $a
      i32.store     ;; store the length on the heap at str_addr = $hp
      global.get $hp       ;; create the tagged ptr to the str
      i32.const ,type-string
      i32.or
      local.set $c   ;; $ c = the tagged ptr to the start of str on heap

      block
      loop       ;; start of the loop
      global.get $hp    ;; advance the heap ptr by the word-size
      i32.const ,word-size
      i32.add
      global.set $hp
      local.get $a   ;; the cntr starts from len and counts down
      i32.eqz              ;; check if counter = 0
      br_if 1         ;; if so, break to the outer block (with index 1)
      global.get $hp
      local.get $b
      i32.store              ;; store the char in str_addr[i]
      local.get $a
      i32.const ,(imm->bits 1)  ;; decrement the counter
      i32.sub
      local.set $a
      br 0                   ;; continue with the loop (with index 0)
      end                    ;; end of the loop (with index 0)
      end                    ;; end of the outer block (with index 0)
      local.get $c)]))    ;; return the tagged ptr to the start of str
…
;; (Listof Id) (Listof Expr) Expr CEnv -> Wasm
(define (compile-let xs es e c)
  (append (compile-es es c)
          (compile-e e (append (reverse xs) c))))

;; [Listof Expr] CEnv -> Wasm
(define (compile-es es c)
  (match es
    ['() '()]
    [(cons e es)
     (append (compile-e e c)
      `(local.set $a
        global.get $sp
        i32.const ,(* word-size (length c))
        i32.sub
        local.get $a
        i32.store)
      (compile-es es (cons #f c)))]))
…
(define (compile-call f es c)
  (let ((h (* word-size (length c))))
    `(,@(compile-es es c)   ;; put the args on the user stack
      global.get $sp   ;; change stack ptr to point to top of stack
      i32.const ,h       ;; (to the addr after C env)
      i32.sub
      global.set $sp
      call ,f
      global.get $sp
      i32.const ,h
      i32.add
      global.set $sp)))   ;; return the stack ptr to before the call

(define (compile-defines fs ls)
  (match* (fs ls)
    [('() '()) '()]
    [((cons f fs) (cons (Lam l xs e) ls))
     `(,(compile-define f xs e)
       ,@(compile-defines fs ls))]))

(define (compile-define f xs e)
  (let ((wa-f (symbol->wa-label f)))
    `(func ,wa-f (result i32)
       (local $a i32)        ;; local var (used as virtual register)
       (local $b i32)        ;; virtual register
       (local $c i32)        ;; virtual register
       ,@(compile-e e (reverse xs)))))
```

The **(compile e)** function generates the compilation of a Racket program containing the expressions covered by our compiler as data in the form of an s-expression. It has a main program function called **$sendResult** which is exported and by its signature should return an i32 result (a 32-bit integer), which should be one i32 value remaining on the stack at the end of the function execution after all operations are carried out by the WebAssembly stack machine. The Node.js runtime will receive this result. The functions **$writeByte**, **$readByte**, **$peekByte**, and **$error** are imported from the Node.js runtime through an import object. These functions are implemented in the Node.js runtime file **jsmain.js**. This file also contains the implementation of the other Node.js functions needed for printing the different types of the result values. WebAssembly has a two-level namespace and so the imports have two-level names [47]. The details of the JavaScript API for WebAssembly that Node.js implements and the Node.js commands used in **jsmain.js** are available in the references [47,34,56-60].

We currently use three local variables named **$a, $b,** and **$c** as virtual registers.

Wasm currently supports one linear memory. The index of this memory is 0. We export this memory for use by the runtime (Node.js or Wasmtime for us). We allocate 1024 pages = 2^{26} bytes as the size of this memory. We divide this memory into a section for the heap and a section for the user stack. The heap starts from the address 0 and grows upward. We set a tentative unenforced high address limit of 2^{25} for the heap. The user stack grows downward from the address 67108860 (= $2^{26} - 4$). The user stack in memory is distinct from the virtual operand stack that the WebAssembly stack machine uses for carrying out the instructions.

We use a mutable global variable **$hp** for the heap pointer and a mutable global variable **$sp** for the user stack pointer. We maintain a C environment of variable bindings on the user stack. For the binding of a variable in a let expression, which leads to the extension of the C environment, we do not change the user stack pointer and place a variable's value on the stack based on the length of the C environment at the time of the binding [implemented in **(compile-let xs es e c)** and **(compile-es es c)** functions]. We retrieve the variable's value based on the position of the variable as a nonnegative offset from the stack pointer [implemented in **(lookup x c len-c)** and **(compile-variable x c)** functions].

For binary operations, such as **(cons e1 e2)** and **(make-string e1 e2)**, we use the user stack to save the result of **e1** and compile **e2** in an environment extended with one placeholder symbol (#f) not to overwrite this value.

For functions calls, the arguments are placed on the user stack, and before the call, the stack pointer is changed to point to the end of the current C environment. The expression in the function is evaluated in an environment of the reverse of the list of the identifiers of the parameters. Therefore, if the environment is extended in the function, because this is done with respect to the new stack pointer with the length of the environment being the number of parameters, then the arguments will not be overwritten. After the call, the stack pointer is returned to its position before the call to now allow the calculation of the position on the stack of the value of a variable of the original C environment based on the offset from the end of that C environment, and the next variable value will be written after this C environment, where the first argument is overwritten.

The parser parses the program of the form **(begin (define (f0 x0 …) e0) (define (f1 x1 …) e1) … e)** or **e** to a **(Letrec fs ls e)** structure, where **fs** is the list of the identifiers of the defined functions and **ls** is the list of lambda structures for these functions. The lambda structures are compiled to the WebAssembly functions by **(compile-defines fs ls)** and **(compile-define f xs e)**. The generated Wasm function has the label of the form **$label_...** formed from the function identifier by the **(symbol->wa-label s)** function and has the same local variables as those of the main function of the program, so that the compiled expression works in this function correctly, the same as it would in the main function. We use the user stack in the memory for passing the arguments of the function. As such, the body of the compiled Wasm function is the compilation of the expression **e** of the function body in an environment of the reverse of the list of the identifiers of the function parameters **xs**.

The current word-size for our Wasm compiler is 4 bytes, with values being i32 (Wasm 32-bit integer type) values for simpler interaction with the Node.js runtime, because i64 is not a legal type for JavaScript [55]. Historically, the best solution for this was "legalization" of the Wasm, meaning to convert Wasm imports and exports to use valid types for JavaScript [55]. In practice, this was done by replacing a 64-bit integer parameter or return value with two 32-bit ones [55]. But now there is another solution to use JavaScript BigInts [55]. With the Node.js runtime, for implementing 64-bit values for our compiler these methods need to be used.

We use some of the bits in the i32 values for type tagging in our Wasm compiler following the same method implemented in our compiler to x86 with the difference that the values are 32-bit, and thus our current word-size is 4 bytes. In addition, our current heap addresses can potentially go from 0 to 2^{26} (but in order to avoid overwriting the stack, which grows downward from the address 2^{26} - 4, we should not allow the heap addresses to go beyond 2^{25}). So, the bits 1 and 2 of the heap addresses, from the right (LSB), are 0, and the component of the addresses in the bits 3 to 25 can go from 0 to 2^{23}. Therefore, the bits 1, 2, and 26 to 31 of the addresses can be used as tag bits. We use the bits 1, 2, 29, 30, and 31 as tag bits. The bit 32 is the sign bit. Using these bits, the type tags for the Wasm compiler are defined in the new file **wtypes.rkt**.

We check the type (and for some operations the value) of the arguments of the operations to make sure they are valid, and if not the **$error** function imported from the Node.js **jsmain.js** runtime file is called, and this function prints "err" to stdout and exits the program.

A string is represented on the heap and its value is a pointer to its address on the heap tagged with the string type tag bits. The first word at this address has the integer value of the length of the string and the next words have the chars. Each char is in one word. Thus, for our Wasm compiler the representation of strings is not packed.

2.2.1 Converting the S-expression Generated by the Compiler to the WebAssembly Text Format Code, Writing the Code in a .wat File, Making a .wasm File from the .wat File, and Running the Code with a Node.js Program

The function **(wasm-string a)** in the file **wasm/printer.rkt** takes the s-expression generated by the compiler and generates the corresponding WebAssembly text format code in readable print form. In this file, it uses the function **(s-exp2string a),** which uses **(define op1 (open-output-string)), (write a op1),** and **(get-output-string op1),** for doing part of this work. The file **compile-wat-file.rkt** takes a **%.rkt** file name as an argument and compiles the Racket code in the file to WebAssembly text format code by using the **(compile e)** function in **compile-wasm.rkt** and the **(wasm-string a)** function. For compiling a Racket program covered by our compiler, the Linux command: **racket -t compile-wat-file.rkt -m %.rkt > %.wat** compiles the Racket code in **%.rkt** file to WebAssembly text format code and writes the code to a **%.wat** file. Then, we make the **%.wasm** file from the **%.wat** file using the **wat2wasm** tool.

We have made the executable **watjs2run** from **watjs2run.c** by **gcc**. This executable takes a **%.wrun** file name as an argument and creates a C file which contains the command **system("node [working directory of watjs2run]/jsmain.js %.wasm");**, and then makes **%.wrun** file from this file with **gcc**. Thus, **%.wrun** is an executable that runs the above Linux command, which runs our Node.js runtime file **jsmain.js** with the **%.wasm** file name as an argument. The **jsmain.js** program then does the interactions mentioned above with the Wasm code to provide the imports to the Wasm program and to get the result exported from the Wasm program and print the result (for details please see the **watjs2run_README.md** file)**.**

We have made the make file **makewat** to automate the execution of the above commands. Given a **%.rkt** file, the command **make -f makewat %.wrun** will compile the Racket code to WebAssembly text format code writing it to a **%.wat** file, makes the **%.wasm** file from **%.wat** file using **wat2wasm**, and then makes the **%.wrun** file using **watjs2run**. Then, using the command **./%.wrun** from the Linux command line, we can run the WebAssembly code with the Node.js runtime.

2.2.2 Running the tests

The file **wtest/test-runner.rkt** in the new folder **wtest** contains all the tests in the file **test/test-runner.rkt** for the expressions covered by the Wasm compiler in the **(test-runner run)** and **(test-runner-io run)** functions plus some additional tests. The file **wtest/compile.rkt** runs the **test-runner** function with the argument run being **(λ (e) (wasm-interp (compile (parse e))))** and the **test-runner-io** function with the argument run being **(λ (e s) (match (wasm-interp/io (compile (parse e)) s) ['err 'err] [(cons r o) (cons r o)])).**

The **(wasm-interp a)** and **(wasm-interp/io a input)** functions are implemented in the file **wasm/interp.rkt** following the methods in a previous compiler of our course with the addition of the functions **(result-str->value str), (string->result str), (cons->result s)** and reading of two lines, written to stdout by **jsmain.js** (one for the result value of the Wasm code and the other for the output that the Wasm code writes to stdout), in the case when **wasm-interp/io** function is used with an input string as the second argument. This case is communicated to **jsmain.js** via a second command line argument " io" [54].

The file **wtest/interp.rkt** is copied unchanged from **test/interp.rkt** and runs the tests in **wtest/test-runner.rkt** using the **interp** function in **interp.rkt** and the **interp-io** function in **interp-io.rkt** files.

The file **wtest/test-progs.rkt** is similar to **test/test-progs.rkt** and runs the function **(test-prog p.rkt)** from **wtest/test-programs/get-progs.rkt** for each of the Racket programs in the list obtained by the function **(get-progs lang)** from the same file with the lang argument being "fraud". The **(get-progs lang)** and **(lang-before lang ols)** functions in this file are the same as those in **test/test-programs/get-progs.rkt.** This list, therefore, will include all the test .rkt files in the folders for the languages up to the fraud language folder in the **wtest/test-programs** folder. These files are the same as those in the **test/test-programs** folder.

The **(test-prog p.rkt)** function has been modified from that in **test/test-programs/get-progs.rkt** in that it uses the .**wrun, .wat** and **.wasm** files and a modified **(make p.wrun)** function instead of the **.run** file and the **(make p.run)** function. The **(make p)** function has been modified to run the command **make -C .. -s -f makewat [full path of p.wrun]** instead of the command **make -C .. -s [full path of p.run].**

All these tests and the added tests work correctly indicating the correctness of the Wasm compiler, to the extent of our compiler to x86, for the subset of Racket expressions covered by the Wasm compiler and tested in these tests.

2.3 The Version of the Compiler that Uses WebAssembly Code and Wasmtime as Runtime

The **compile-wasm_2.rkt** file contains the version of the compiler that uses the Wasmtime and WebAssembly code as the runtime to print the result and execute external functions such as $error_exit. This is a partial implementation of this runtime as printing of chars, strings, integers, and booleans, and calling an error function with printing of "'err" and exiting the program have been implemented for the purpose of delineating the method to write the runtime in WebAssembly, but printing of boxes, pairs, and lists, and operations of read-byte, peek-byte, and write-byte have not been implemented.

The file **compile-wat-file_2.rkt** calls the **(compile e)** function in **compile-wasm_2.rkt** and the **(wasm-string a)** function in **wasm/printer.rkt** to generate the compiled WebAssembly text format code similar to what is done from the **compile-wat-file.rkt** file.

A make rule **%.wasm2** with dependency **%.rkt** has also been added to the make file **makewat** to automate the generation of a target **%_2.wasm** file of the compiled WebAssembly code by running the following commands:

racket -t compile-wat-file_2.rkt -m $< > $(patsubst %.wasm2, %_2.wat, $@)

wat2wasm $(patsubst %.wasm2, %_2.wat, $@).

Thus, running the Linux command **make -f makewat %.wasm2** will result in the creation of a WebAssembly file **%_2.wasm** in binary format from the compilation of the Racket code in **%.rkt** using the **(compile e)** function in **compile-wat-file_2.rkt** that enables this file to interact with Wasmtime. Then, the Linux command **wasmtime %_2.wasm** will run the code in this file in the Wasmtime runtime resulting in the printing of the result.

The following is an outline of the new code in **compile-wasm_2.rkt:**

```
(define (compile e)
  (match e
    [(Letrec fs ls e)
     (let ((prog-fn  `(func $program (result i32)  …
                            ,@(compile-e e '())))
           (main-fn  `(func $main (export "_start")
                            ;; Ref: WASI tutorial
                (local $a i32)
                i32.const 0   ;; iov.iov_base – prt to start addr of the data
                i32.const 8
                i32.store
                call $program
                call $process_result  ;; process the program result
                local.set $a
                i32.const 4       ;; iov.iov_len - The length of the
                local.get $a   ;; the output string = returned nw (number
                i32.store     ;; of bytes written to memory in data section)
                i32.const 1       ;; file_descriptor - 1 for stdout
                i32.const 0       ;; *iovs - ptr to iov array, stored at addr 0
                i32.const 1 ;; iovs_len - printing 1 str stored in an iov
                i32.const 8
                local.get $a   ;; addr to write num bytes written
                i32.const 4    ;; should be multiple of 4
                i32.div_s
```

```
      i32.const 4
      i32.mul
      i32.const 4
      i32.add
      i32.add     ;; this address will hold numbytes written
      call $fd_write
      drop
      i32.const 10          ;; line feed (\n)
      call $print_codepoint
      drop))
(process-result-fn
  `(func $process_result (param $result i32) (result i32)
     (local $nw i32)  ;; num of bytes written to memory
      …
      local.get $result  ;; the parameter 0 of the function
      i32.const ,ptr-bottom-mask
      i32.and
      i32.eqz
      if              ;; if the result is of an immediate type
      local.get $result
      call $process_imm
      return
      end
      …
      if              ;; if the result is of type string
      local.get $result
      call $process_string
      ;call $process_string_with_print_codepoint
      return
      end
      local.get $nw))
(process-imm-fn
  `(func $process_imm (param $result i32) (result i32)
      …
      if                ;; if the result is of type int
      local.get $result
      i32.const ,int-shift
      i32.shr_s
      call $print_int
      i32.const 0           ;; 0 bytes written to memory
      return
      end
      …
      if                ;; if the result is of type char
      …
      local.get $result
      i32.const ,char-shift
      i32.shr_s
      call $print_codepoint
      drop
      i32.const 0           ;; 0 bytes written to memory
      return
      end
      …
      i32.const 0))

(process-string-fn
  `(func $process_string (param $str-ptr i32) (result i32)
     (local $nw i32)  ;; num of bytes written to memory
      (local $a i32)
      …
      (local $c i32)
      …
      i32.const 8     ;; start addr of where to write data
      i32.const 34    ;; UTF-8 encoding of "
      i32.store
      …
      i32.const 9     ;; 1 byte already written to address 8
      local.set $c   ;; ptr to where to write data in memory 0.
      …
      block
      loop
      local.get $a
      i32.load        ;; load the char in str_addr[i] to op. stack
      i32.const ,char-shift
      i32.shr_s
      local.get $c     ;; second arg
      call $write_utf_to_memory  ;; fd_write in main will
                                 ;; use this data to print the output.
      …
      br 0            ;; continue with the loop (with index 0)
      end                ;; end of the loop (with index 0)
      end                ;; end of the block (with index 1)
      …
      local.get $nw
      i32.const 1
      i32.add         ;; return $nw++
      ))
(process-string-with-print_codepoint-fn
  `(func $process_string_with_print_codepoint
     (param $str-ptr i32) (result i32)
      (local $a i32)
      …
      block
      loop
      …
      local.get $a
      i32.load        ;; load the char in str_addr[i] to op. stack
      i32.const ,char-shift
      i32.shr_s
      call $print_codepoint   ;; print the char.
      …
      end
      i32.const 0))  ;; return 0, as 0 bytes written to memory

(write_utf_to_memory
  `(func $write_utf_to_memory (param $char i32)
     (param $addr i32) (result i32)
      …
      local.get $char
      i32.const 2048  ;; case codepoint < 2048
      i32.lt_s
      if
      local.get $addr
```

```
local.get $char  ;; form byte 1 from 5 MSBs of char
i32.const 6
i32.shr_s
i32.const 192
i32.or
i32.store      ;; store 1st byte in smaller address

local.get $addr
i32.const 1
i32.add
local.get $char  ;; form byte 2 from 6 LSBs of char
i32.const 63
i32.and
i32.const 128
i32.or
i32.store      ;; store 2nd byte in higher address

local.get $addr  ;; advance data address by 2
i32.const 2
i32.add
return         ;; return updated address on op. stack
end
```

```
        …)))
`(module
  (import "wasi_unstable" "fd_write"
        (func $fd_write (param i32 i32 i32 i32) (result i32)))
  (import "wasi_snapshot_preview1" "print_codepoint"
        (func $print_codepoint (param i32) (result i32)))
  (import "wasi_snapshot_preview1" "print_int"
        (func $print_int (param i32) (result i32)))
  (import "wasi_snapshot_preview1" "error_exit"
        (func $error_exit (result i32)))
  (memory 1024)  ;; 1024 pages of memory = 2^26 bytes
  (export "memory" (memory 0))
  …
  (global $hp (mut i32) (i32.const 16777216))
   ,main-fn
   ,prog-fn
   ,process-result-fn
   ,process-imm-fn
   ,process-string-fn
   ,process-string-with-print_codepoint-fn
   ,write_utf_to_memory
   ,@(compile-defines fs ls)))]))
```

The Wasmtime **fd_write** function takes as arguments a file descriptor (usually 1 for stdout), the pointer to iov array (0 for memory address 0), iov_len (the number of the strings printed), and the address to store the number of bytes written [24]. In the linear memory of the WebAssembly module, in addresses 0 to 3, as part of the iov array, the address of the start of the data section is stored as an i32 value type in 4 bytes. This is the address in memory where we start the writing of the UTF-8 encodings of the characters of the string and is the integer 8. In the 4 bytes of the addresses 4 to 7, the length of the string in the number of bytes is written. As such, here we write the number of the bytes that we write to the memory for the UTF-8 encodings of the chars of the string. We allow the data section to go up to the address of $2^{24} - 1 = 16777215$ (which can hold about 2^{23} chars whose UTF-8 encoding is 2 bytes [e.g., λ] before overflowing into the heap section). Then, the heap section starts from the address 2^{24} and can go up to the address 2^{25}, and therefore, this will allow strings of up to the length $(2^{25} - 2^{24}) / 4 = 2^{22} = 4194304$ to be stored on the heap (given that we do not use the packed string representation in the Wasm compiler).

We have two methods for writing the Unicode characters of the strings to stdout in our compiler. The first method, which we have implemented in the function **$process_string** with calls to the function **$write_utf_to_memory**, is to write the UTF-8 encodings of the chars of the string to the data section in memory, keeping track of the number of bytes written, and then call the Wasmtime **fd_write** function with this data in memory. The **fd_write** function then uses the Rust write_vectored function to write the chars of the string (using UTF-8 encoding) from memory to stdout.

The second method, (which we have implemented in the function **$process_string_with_print_codepoint**), is to use the **print_codepoint** function, which is a function that we have written in Rust and added to Wasmtime following the method described in the reference 61 with modifications needed for the new version of Wasmtime (for details of the method to add this function please see the file **.wasmtime/Additions_README.md**) [61]. Similarly, we have written the **print_int** and **error_exit** functions in Rust and added them to Wasmtime for printing of integers, and printing of "'err" and exiting the program when there is an error, respectively. We also have implemented the printing of chars and booleans using **print_codepoint**.

To toggle between the two methods for printing the strings, we comment out one of the calls to **$process_string_with_print_codepoint** or **$process_string** in the **$process_result** function.

The executable **wasmtime** in the **the-838e-compiler/.wasmtime/bin/** folder in the submitted pull request is a modified build of **wasmtime** after making the above changes in a git clone of the Wasmtime repository and doing a new build with cargo build --release [62]. The **push.yml** file is modified to install **g++, cmake, wabt,** and **nodejs,** and add the paths of **wabt/bin** and **.wasmtime/bin** to the $PATH environment variable, so that the tests can be run on GitHub.

For testing the **compile-wasm_2.rkt** code with the modified **wasmtime** and the added WebAssembly code as the runtime, the following files were added: **wasm/interp_2.rkt, wtest/compile_2.rkt,** and **wtest/test-runner_2.rkt**. The file **wtest/test-runner_2.rkt** includes the subset of the test expressions from **wtest/test-runner.rkt** covered by this system, and the file **wtest/compile_2.rkt** uses the **(compile e)** function from **compile-wasm_2.rkt** for running these tests. The file **wtest/interp_2.rkt** includes a modified **(wasm-interp a)** function adapted to running the tests with **wasmtime**. All these tests work correctly indicating the correctness of the code in **compile-wasm_2.rkt** and the modified **wasmtime** to the extent tested in these tests.

2.4 Comparing the Speed of Different Combinations of Compilers, Runtimes, and the Methods of Printing Strings for Running the Executable for the Expression (make-string 1000000 #\λ)

We tested the speed of the different combinations of our compilers, runtimes, and the methods of printing strings in the case of running the compiled code obtained from each method for the expression **(make-string 1000000 #\λ)** in the file **str1000000lms.rkt**, which results in the forming of a string of 1000000 λ's on the heap, traversing this string on the heap, sending the characters of the string to stdout in UTF-8 encoding, and displaying the 1000000 λ's on the screen. The 5 combinations are listed below. The average times over 5 measurements for each combination were calculated and are shown in Table 1 below.

We automated the measurement of these times with the make file **mkexeslms**, the shell script **timestrlms2screen**, and the Racket file **timestrlms2screen.rkt**. The shell script **timestrlms2screen** takes a first integer argument that specifies the choice of one of the combinations 1 to 5 below. For choice 3 versus 4, we need to set the calling of the appropriate function (**$process_string** for choice 3 and **$process_string_with_print_ codepoint** for choice 4) in the function **$process_result** in the file **compile-wasm_2.rkt** before running the script. The shell script first creates all the executables for the combinations 1, 2, (3 or 4), and 5 with executing the command **make -f mkexeslms all**. Then, it times one of the combinations 1, 2, (3 or 4), or 5 based on its first argument. It can take a second argument also. If this argument is the word "clean", then it also removes the created files with the command **make -f mkexeslms clean**. The file **timestrlms2screen.rkt** executes **timestrlms2screen** with the argument 1 by a system command.

1- **Compiler to x86 in compile.rkt with C runtime:** Each run of this test was done with running the executable **str1000000lms.run** made from linking the compiled x86 code of the program and the libarary functions (generated using our compiler to x86 in **compile.rkt**) with x86 code of other libraries and our **C runtime.** For this test, we increased the heap_size in runtime.h to 333400 to have enough memory to put this string on the heap. Because we use a packed string representation with 3 characters in each word in our compiler to x86, 333334 words is enough. Less than 333310 resulted in segmentation fault as expected from accessing parts of the memory not allocated for the program. If the string representation was nonpacked with each char in one word, then we would have needed more than 1000000 words for the heap size.

2- **Wasm compiler in compile-wasm.rkt with Node.js runtime:** Each run of this test was with running the executable **str1000000lms.wrun** which runs the WebAssembly binary format code in the **%.wasm** file with **jsmain.js**, our **Node.js runtime program**. The Wasm binary format code in %.wasm has been obtained by the WABT **wat2wasm** tool from the **%.wat** WebAssembly text code generated using our Wasm compiler in **compile-wasm.rkt.**

3- **Wasm compiler in compile-wasm_2.rkt with WebAssembly and wasmtime runtime with using the fd_write function for printing the string:** Each run of this test was done with the command **wasmtime str1000000lms_2.wasm** which uses the **wasmtime** runtime to run the **%_2.wasm** file, with **%_2.wasm** file created from the WebAssembly code compiled with the version of our Wasm compiler in **compile-wasm_2.rkt** that in the WebAssembly runtime code writes

the UTF-8 encoding of the characters of the string to memory and then uses the Wasmtime **fd_write** function for sending the encodings to stdout.

4- **Wasm compiler in compile-wasm_2.rkt with WebAssembly and wasmtime runtime with using the print_codepoint function for printing the string:** Each run of this test was done with a process similar to the process in number 3 above, except that the version of our Wasm compiler in **compile-wasm_2.rkt** was used that in the WebAssembly runtime code prints the strings with the function **print_codepoint.**

5- A simpler version of our compiler to x86 in the file 3_compile.rkt (with use of 3_parse.rkt, 3_ast.rkt, 3_types.rkt) without higher-order functions and library functions, and with a make-string function that puts each char in a word (nonpacked representation of a string), and with a simpler C runtime in the files 3_main.c, 3_char.c, 3_types.h, 3_runtime.h, 3_io.c, that prints a string accordingly: Each run of this test was done with running the executable **str1000000lms.3_run** made from linking the compiled x86 code of the program (generated using this version of compiler to x86) with this simple version of **C runtime.** The executable was made using the make file **3_make** with the command **make -f 3_make str1000000lms.3_run**. The files mentioned here are new files added to the repository in the pull request.

Table 1 - The average of real, user, and sys times over 5 runs, measured in seconds, for the 5 combinations of our compilers, runtimes, and printing methods.

		Comb. 1	Comb. 2	Comb. 3	Comb. 4	Comb. 5
W/ output to screen	Real	0.584	0.988	0.252	1.089	0.593
	User	0.059	0.650	0.013	0.234	0.066
	Sys	0.053	0.161	0.031	0.053	0.047
W/ output to temp file	Real	0.080	0.776	0.030	0.247	0.077
	User	0.060	0.653	0.012	0.222	0.063
	Sys	0.016	0.159	0.016	0.028	0.012

We also measured the above timings with redirecting the output of 1000000 λ's to the disc to be written to a temp file instead of the screen. We automated the measurement of these times with the shell script **timestrlms** and the Racket file **timestrlms.rkt**. The shell script **timestrlms** takes a first integer argument that specifies the number of times the loop for timing the combinations runs. It first creates all the executables for the combinations 1, 2, (3 or 4), and 5 above with executing the command **make -f mkexeslms all**. Then, it times the combinations with the output redirected to the temp file. It can take a second argument also. If this argument is the word "clean", then it also removes the created files with the command **make -f mkexeslms clean**. The file **timestrlms.rkt** executes the shell script **timestrlms** with the argument 1 by a system command.

With executing **./timestrlms 5**, the average times for 5 measurements for each of the above 5 combinations of compilers, runtimes, and printing methods with the output redirected to a temp file were calculated and are shown in Table 1 above.

3. Other Changes

We also added the files **compile-intmd-utils_2.rkt**, **compile-exprs_2.rkt**, **compile_2.rkt**, and **test/compile_2.rkt**, and added the instruction **(ICall x)** to **a86/ast.rkt**. The **(ICall x)** instruction is for abstracting out padding the stack before, and unpadding the stack after a call to an external C function to ensure alignment of the stack frame to 16-byte boundaries before the call (to avoid faults due to the software conventions for 128-bit SIMD technologies in Intel® 64 and IA-32 architectures [63, 64]). The function **(intmd-to-a86 c)** in **compile-intmd-utils_2.rkt** desugars the **(ICall x)** instruction to **(seq (pad-stack**

c) (Call x) (unpad-stack c)). In addition, we replaced the instructions of the form **(Jx (error-label c)),** where Jx is one of the a86 instructions Je, Jl, Jg, Jmp, Jne, Jle, or Jge, with **(Jx 'err).** The desugaring of **(Jx 'err)** to **(Jx (error-label c))** is also implemented in the function **(intmd-to-a86 c).** We did the replacements of the above intermediate instructions in a copy of the file **compile.rkt** and divided this file into two smaller files **compile_2.rkt** and **compile-exprs_2.rkt**, which makes the organization of the files better. Desugaring of these instructions is done in **compile-exprs_2.rkt** by making calls to **(intmd-to-a86 c)**. The file **test/compile_2.rkt** runs the tests in **test/test-runner.rkt** for this implementation. The tests work correctly with this implementation.

4. Discussion

We wrote a Wasm compiler and a Node.js runtime with the features discussed above that work together correctly to compile the programs and print the result of the execution of the programs. In addition, we implemented the interaction of the Wasm compiler with a second runtime that includes functions written in WebAssembly to process the result and write the data to the linear memory of the Wasm module and then interact with Wasmtime through the fd_write function of Wasmtime, and also through functions that we have implemented in Rust and added to Wasmtime, to print the result or to print "'err" and exit the program when there is an error.

In terms of performance, for making a string of 1000000 λ's on the heap, traversing the string on the heap, sending the characters of the string to stdout, and displaying the characters on the screen, our Wasm compiler with the WebAssembly and Wasmtime runtime, with the use of the fd_write function, performed better than our compiler to x86 with the C runtime, with an average real time of 0.252s and an average user time of 0.013s (column 3 in Table 1), compared with an average real time of 0.584s and an average user time of 0.059s for the compiler to x86 with the C runtime (column 1 in Table 1), over 5 measurements for each time. Although we did not formally calculate the standard deviations and the *p*-values for these differences, the variations observed for each time over the 5 measurements were small compared with the difference in the case of user time and practically negligible in the case of real time.

This difference was also seen in the case the output was redirected to the disc to be written to a temp file, with the average real time and user time for the combination number 3 being 0.030s and 0.012s, as compared with 0.080s and 0.060s for the combination number 1.

We also measured these times for the combination number 5, with a simpler version of our compiler to x86 without higher-order functions and library functions, and with a make-string function that puts each char in a word (nonpacked representation of a string), and with a simpler C runtime that prints a string accordingly, to see if any part of this difference was caused by these factors. The times are very close to each other for the combinations number 5 and 1, indicating that these factors have negligible effect. This includes the time that it takes for the additional linking of some of the libraries and loading a larger final executable in the memory, which is expected to be negligible compared with the time that it takes for I/O. Our make-string function for the packed string representation, and our traversal of the string on the heap for obtaining the characters are also efficient being of complexity *O(n)* similar to the implementation in the nonpacked representation case, and therefore this factor is also negligible.

The main factor that probably accounts for this difference is that in the C runtime for each character of the string, the function printf is called, whereas in the WebAssembly and Wasmtime runtime with using the fd_write function, the UTF-8 encodings of the characters of the string are all written to memory before the function fd_write is called and then fd_write uses the Rust write_vectored function to write from memory to either the screen or the file on the disk. This process is more efficient likely because of a lack of need for repeated scheduling for I/O and because of DMA (direct-memory-access) with a DMA command block with the source pointer, the destination pointer, and the count of the number of bytes to be transferred [65]. The CPU writes the address of the command block to the DMA controller and then goes on with other work [65]. The DMA controller operates the memory bus directly, performing the transfer without the help of the CPU [65]. This obviates the need for programmed I/O (PIO), using the CPU to watch the status bits and to feed data into a controller register one byte at a time [65]. Calling printf for each character likely results in the involvement of the CPU for each character with printing each

character becoming a scheduled I/O that the operating system scheduler needs to schedule. This results in time delays causing the large differences in real time observed between the combination number 1 and the combination number 3 both with output to the screen and to the disc for writing to the temp file.

Using the print_codepoint function [which uses Rust print! function, std::char::from_u32(i32 argument cast as u32), and a match with Some(x)] added to the wasmtime executable is also not efficient as compared with using the Wasmtime's own fd_write function, as demonstrated by the large differences between the times in column 4 and column 3, likely in part because of the same problems with I/O described above caused by calling the print_codepoint function for each character. The times indicate this is worse than the case of the combination number 1, maybe because the x86 code with C runtime is more efficient than the WebAssembly with Wasmtime runtime. Other possible factors may be that some optimizations associated with calls to printf and buffering of the characters in C are performed and that some inefficiency is introduced in the writing of a print function and making calls to it as opposed to using one implemented at lower level.

Using the Node.js runtime is less efficient as the times for the combination number 2 indicate. Part of this is likely because of calling the console.log() function for each character of the string causing the same problems with I/O as described above. In addition, part of this can be because JavaScript is an interpreted language and Node.js can still not be as efficient as C and the code compiled to an executable even with the just-in-time compilations, optimizations, and reoptimizations performed by the V8 JavaScript engine [31]. The distinctly long user time with both output to the screen and to the disc for writing to the temp file with the combination number 2, as compared with the other 4 combinations, may be because of this, as the program is likely busier running code in the user space in the case of the Node.js runtime [66].

Even though using the fd_write function with the WebAssembly and Wasmtime runtime is more efficient than using the print_codepoint function, especially for long strings, one advantage of print_codepoint is that it will not overflow the data section of the memory. We set the size of the data section in the linear memory of our Wasm module to be $16777216 = 2^{24}$ bytes from addresses 0 to 16777215. We write the characters to this section in memory in UTF-8 encoding, because the fd_write function uses this encoding to write the characters to stdout. Then, given that the UTF-8 encoding of λ is 2 bytes, we cannot use fd_write to print a string of longer than about 8388608 λ's, because this will overflow the data section into the heap section.

But because we do not allow the heap to grow beyond the address 2^{25}, and given that we do not use a packed string representation in our Wasm compiler, with the word-size of 4 bytes, this will allow strings of up to the length about $(2^{25} - 2^{24}) / 4 = 2^{22} = 4194304$ characters to be placed on the heap, and therefore this is the limit for the length of the longest string of λ's that we can print. If we set the stack section to be smaller and allow the heap to grow to close to the top address of $2^{26} - 4$ of our memory, then we can print a string of up to length about $(2^{26} - 2^{24}) / 4 = 3 \cdot 2^{22} = 12582912$ characters. Trying to put a longer string on the heap, even if we allow overflowing into the stack section, will hit the top address of our memory and result in WebAssembly generating a trap with the message "wasm trap: out of bounds memory access".

In the more general case that the size of the UTF-8 encoding of a character can be up to 4 bytes, a string of longer than about $2^{24} / 4 = 4194304$ characters will overflow the data section, and therefore this is the upper limit that we should allow for the length of a string. If we increase the size of the data section, then the heap section will become smaller, and this will decrease the length of the string that we can put on the heap to less than 4194304 characters. So, the size of the data section of 2^{24} obtained from the equation $(2^{25} - x) / 4 = x / 4$ is the size that maximizes the length of the string that we can put on the heap and print with this method. Writing the code for a method to call the fd_write function repeatedly may allow us to decrease the size of the data section and to increase the maximum length of the string that we can put on the heap and print, to the limit imposed by the heap.

Therefore, with the use of the fd_write function, overflowing of the data section can occur, and we need to set a limit for the length of the strings. We also need to carefully balance the size of our data section and heap section to allow the maximum length of strings to be printed. If we use a packed string representation, we can make the heap size smaller and print longer strings with the fd_write function.

With using the print_codepoint function to print the string, we cannot overflow the data section of the memory, because we do not write the characters of the string to memory, but still the string have been placed on the heap, and with our memory size of 2^{26}, with the nonpacked string representation, the maximum length of a string that can be placed in this memory is $2^{26} / 4 = 2^{24} = 16777216$, even if we set the sizes of data and stack sections to 0. With a more packed string representation, this length can be increased. This is an advantage of about $2^{22} = 4194304$ characters over the case when we use the fd_write function. Given that using the print_codepoint function is slower than using the fd_write function, this is not a major advantage, and using the fd_write function with a larger data section still is better. As mentioned above, one feature that can be added is to implement making repeated calls to the fd_write function for longer strings, and by this way, the problem of fd_write function overflowing the data section can be solved.

The processing of the rest of the value types, such as boxes, pairs, and lists, for writing the UTF-8 encoding of the appropriate characters to memory to be printed by the fd_write function can also be implemented in WebAssembly to complete the runtime of WebAssembly and Wasmtime. Printing of the integers can also be done in this way. The error_exit function added to Wasmtime can still be used for errors, or a more sophisticated way can be found or implemented in Wasmtime for this. Implementation of the interaction with Wasmtime for the read-byte, peek-byte, and write-byte functions can also be done by adding functions similar to error_exit to Wasmtime by the method in the reference 61 and as described in the file the-838e-compiler/.wasmtime/Additions_README.md [61]. Alternatively, this can be done by more sophisticated uses of the Wasmtime functions or implementation in Wasmtime.

References:

1- WebAssembly Specification, Release 1.1 (Draft 2021-04-21). WebAssembly Community Group, Andreas Rossberg (editor) (https://webassembly.github.io/spec/core/_download/WebAssembly.pdf).
2- Haas, Andreas; Rossberg, Andreas; Schuff, Derek L.; Titzer, Ben L.; Gohman, Dan; Wagner, Luke; Zakai, Alon; Bastien, JF; Holman, Michael (June 2017). Bringing the web up to speed with WebAssembly. Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. 185–200. doi:10.1145/3062341.3062363. ISBN 9781450349888.
3- Wikipedia WebAssembly web page (https://en.wikipedia.org/wiki/WebAssembly).
4- WebAssembly website (https://webassembly.org/).
5- WebAssembly System Interface (WASI) overview (https://github.com/WebAssembly/WASI/blob/main/docs/WASI-overview.md).
6- Lin Clark. Standardizing WASI: A system interface to run WebAssembly outside the web (https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/).
7- Pat Hickey. Announcing Lucet: Fastly's native WebAssembly compiler and runtime (https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime)
8- W3C WebAssembly Community Group (https://www.w3.org/community/webassembly/).
9- WebAssembly Working Group (https://www.w3.org/wasm/).
10- Watt, Conrad (2018). Mechanising and Verifying the WebAssembly Specification (PDF). ACM SIGPLAN International Conference on Certified Programs and Proofs. ACM. 7: 53–65. doi:10.1145/3167082. ISBN 9781450355865. S2CID 9401691.
11- Emscripten website (https://emscripten.org/).
12- Emscripten GitHub repository (https://github.com/emscripten-core/emscripten).
13- Emscripten Wikipedia webpage (https://en.wikipedia.org/wiki/Emscripten).
14- Clang Wikipedia webpage (https://en.wikipedia.org/wiki/Clang).
15- Binaryen GitHub repository (https://github.com/WebAssembly/binaryen).
16- Compiling to WebAssembly with Binaryen (https://github.com/WebAssembly/binaryen/wiki/Compiling-to-WebAssembly-with-Binaryen#what-do-i-need-to-have-in-order-to-use-binaryen-to-compile-to-webassembly).
17- Alon Zakai. Shipping Tiny WebAssembly Builds video. (https://www.youtube.com/watch?v=_lLqZR4ufSI).
18- Cranelift Code Generator (https://github.com/bytecodealliance/wasmtime/tree/main/cranelift).
19- Cranelift IR Reference (https://github.com/bytecodealliance/wasmtime/blob/main/cranelift/docs/ir.md).
20- Chris Fallin. A New Backend for Cranelift, Part 1: Instruction Selection (https://hacks.mozilla.org/2020/10/a-new-backend-for-cranelift-part-1-instruction-selection/).

21- Frank Denis. Memory management in WebAssembly: guide for C and Rust programmers (https://www.fastly.com/es/blog/webassembly-memory-management-guide-for-c-rust-programmers).
22- Wasmtime GitHub repository (https://github.com/bytecodealliance/wasmtime).
23- Wasmtime guide (https://docs.wasmtime.dev/).
24- WASI tutorial (https://github.com/bytecodealliance/wasmtime/blob/main/docs/WASI-tutorial.md).
25- Rust trait Write (https://doc.rust-lang.org/std/io/trait.Write.html).
26- WebAssembly C and C++ API repository (https://github.com/WebAssembly/wasm-c-api).
27- Rust crate wasmtime (https://docs.rs/wasmtime/0.26.0/wasmtime/).
28- Add new MachInst backend and ARM64 support. #1494 pull request on Wasmtime GitHub repository (https://github.com/bytecodealliance/wasmtime/pull/1494).
29- The WebAssembly JavaScript APIs (https://developer.mozilla.org/en-US/docs/WebAssembly).
30- Node.js Wikipedia web page (https://en.wikipedia.org/wiki/Node.js).
31- V8 engine Wikipedia web page (https://en.wikipedia.org/wiki/V8_(JavaScript_engine)).
32- V8 engine website (https://v8.dev/).
33- Node.js v16.1.0 documentation on WASI (https://nodejs.org/api/wasi.html).
34- Colin Eberhardt. Writing WebAssembly by Hand (https://blog.scottlogic.com/2018/04/26/webassembly-by-hand.html).
35- WABT (the WebAssembly Binary Toolkit) GitHub repository (https://github.com/WebAssembly/wabt).
36- Wasmer website (https://wasmer.io/).
37- Wasmer GitHub repository (https://github.com/wasmerio/wasmer/).
38- Running WebAssembly on ARM (https://medium.com/wasmer/running-webassembly-on-arm-7d365ed0e50c).
39- Add AArch64 support for singlepass. #713 on Wasmer GitHub repository (https://github.com/wasmerio/wasmer/pull/713).
40- Professor David Van Horn's IR design and new backends project idea on GitHub (https://github.com/plum-umd/the-838e-compiler/issues/20).
41- Wikipedia page for intermediate representation (https://en.wikipedia.org/wiki/Intermediate_representation).
42- David Walker Lecture slides on intermediate representation (https://www.cs.princeton.edu/courses/archive/spr03/cs320/notes/IR-trans1.pdf).
43- Wikipedia page for LLVM (https://en.wikipedia.org/wiki/LLVM).
44- Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization, M.S. Thesis (https://llvm.org/pubs/2002-12-LattnerMSThesis.pdf).
45- Emscripten and the LLVM WebAssembly backend (https://v8.dev/blog/emscripten-llvm-wasm).
46- Mark P. Jones, Justin Bailey, Theodore R. Cooper. MIL, a Monadic Intermediate Language for Implementing Functional Languages. Proceedings of the 30th Symposium on Implementation and Application of Functional Languages, September 2018 Pages 71–82 (https://doi.org/10.1145/3310232.3310238).
47- Understanding WebAssembly text format (https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format).
48- Guy Royse. An Indroduction to WebAssembly video (https://www.youtube.com/watch?v=3sU557ZKjUs).
49- Jay Phelps. The factorial function code in WebAssembly Demystified video, (https://www.youtube.com/watch?v=cRwUD5SxF4o).
50- Viewing WebAssembly code generated by Clang for example C programs.
51- Code generated by WebAssembly Explorer (https://mbebenita.github.io/WasmExplorer/).
52- CMSC838e Class lecture notes (https://www.cs.umd.edu/class/spring2021/cmsc838E/).
53- the CMSC838e compiler code (https://github.com/plum-umd/the-838e-compiler).
54- CMSC430 Fall 2019 Class Codes (https://classroom.github.com/a/t5KO9b5-)
55- Alon Zakai. WebAssembly integration with JavaScript BigInt (https://v8.dev/features/wasm-bigint).
56- Node.js v16.0.0 documentation, File System (https://nodejs.org/api/fs.html#fs_fs_copyfile_src_dest_mode_callback).
57- Node.js v16.0.0 documentation, Buffer (https://nodejs.org/api/buffer.html#buffer_buf_index).
58- Answer of Anthony Garcia-Labiad in https://stackoverflow.com/questions/20185548/how-do-i-read-a-single-character-from-stdin-synchronously
59- 7- Node.js v16.0.0 documentation, Process , https://nodejs.org/api/process.html#process_process_stdin_fd
60- https://stackoverflow.com/questions/4976466/difference-between-process-stdout-write-and-console-log-in-node-js
61- Radu Matei. A beginner's guide to adding a new WASI syscall in Wasmtime (https://radu-matei.com/blog/adding-wasi-syscall/)
62- Wasmtime web page (https://wasmtime.dev/)
63- https://stackoverflow.com/questions/612443/why-does-the-mac-abi-require-16-byte-stack-alignment-for-x86-32

64- Intel® 64 and IA-32 Architectures Optimization Reference Manual, Section 5.4.2, Stack Alignment For 128-bit SIMD Technologies. (https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html)
65- Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. Operating System Concepts, 10th Edition. Chapter 12 I/O systems, P 498.
66- The Wikipedia page on CPU time (https://en.wikipedia.org/wiki/CPU_time)