Douglas Boyle

Optimising compiler from OCaml to WebAssembly

Computer Science Tripos — Part II

Churchill College

May 13, 2021

Declaration of Originality

I, Douglas Boyle of Churchill College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

I, Douglas Boyle of Churchill College, am content for my dissertation to be made available to the students and staff of the University.

Signed

Date

Proforma

Candidate number:	**2361C**
Title:	**Optimising compiler from OCaml to WebAssembly**
Examination:	**Computer Science Tripos Part II, May 2021**
Word count:	**11957**[1]
Lines of code:	**8807**[2]
Project originator:	Dr T. Jones
Supervisor:	Dr T. Jones

Original Aims of the Project

The original aims were to produce a compiler from a subset of OCaml to WebAssembly and measure its performance against several alternatives for running code in browsers. Only the core OCaml language was to be considered, leaving out the class and module layers. Extensions were to extend the subset supported, implement and evaluate optimisations, and add a garbage collector to the runtime.

Work Completed

I implemented a compiler to WebAssembly, using the parser and type-checker of the official OCaml compiler, supporting all of the core OCaml language and several standard-library operations. I wrote tests and benchmark programs and evaluated my compiler against equivalent programs compiled with Js_of_ocaml, Grain and Clang/LLVM. I also implemented several optimisations and a garbage collector, and measured their impact on the benchmark programs.

Special Difficulties

None.

[1] Computed using `texcount` for the 5 main chapters

[2] Computed using `cloc`, excluding whitespace, comments and the OCaml compiler front-end. See `https://github.com/AlDanial/cloc`

Contents

List of Figures

Chapter 1

Introduction

Until recently, JavaScript was the only language available for writing interactive web applications, and it is still the most popular choice for developers. However, JavaScript has a number of downsides, which has led to the development of WebAssembly as a more efficient rival for these kinds of programs. In particular, WebAssembly is better suited as a target for compiling strongly typed languages, in order for them to run on the web.

The first disadvantage of JavaScript is that it is sent as source code and then compiled to bytecode and interpreted, or just-in-time compiled to native code. Therefore, the client downloads a relatively large text file, increasing the overhead in loading the page.

Another issue with JavaScript comes from its dynamic typing system. Variables can be arbitrarily assigned values of different types or have properties added or removed. This uncertainty restricts the optimisations that can be performed, and means that most operations require runtime type checks and potentially throw exceptions, all of which adds to the runtime overhead. The dynamic typing also has some flexibility, implicitly casting objects to the correct type where possible. This behaviour can be confusing in places, resulting in hard-to-detect bugs that slow development. For example:

```
if ([]) console.log(‘[] is true’);

if ([] == false) console.log(‘[] is false’);
```

Perhaps surprisingly, both statements above will be printed (the issue here is that casting `[]` to a boolean gives `true`, whereas the second case casts both sides to integers, giving `0 == 0`), and unexpected semantics like this can take a long time to debug.

An advantage of programming in a functional language, such as OCaml, is that it is statically typed, so compile-time type checking detects many potential errors. However, compiling OCaml to JavaScript, as performed by the Js_of_ocaml tool [1], results in these checks being unnecessarily repeated at runtime. It would be better to avoid these inefficiencies. Targeting WebAssembly is one possible choice, since it is strongly typed, so these checks are not needed at runtime, providing a more efficient way to run functional code on the web. In addition, WebAssembly is now supported by major browsers, such as Chrome, Firefox and Safari.

1.1 Overview of WebAssembly

WebAssembly has both a text format for `.wast` files and a binary format for `.wasm` files. This means that code can be easily inspected in the text format but downloaded and executed as binaries, reducing the size of files and the overhead to run them. Figure 1.1 implements factorial, as an example of how WebAssembly code is structured:

```
(module
  (func $fact (export  ‘‘fact’’) (param $n i32) (result i32)
    (if (result i32) (i32.eq (local.get $n) (i32.const 0))
      (then (i32.const 1))
      (else (i32.mul
        (local.get $n)
        (call $fact (i32.sub (local.get $n) (i32.const 1)))))
    )
  )
)
```

Figure 1.1: Factorial function written in WebAssembly

WebAssembly is a stack-based language, with control flow expressed by nested blocks, such as the two cases of the `If` statement above, rather than jumps in linear code as in most assembly languages. The other nesting in figure 1.1 is syntactic sugar allowed by the text format, which makes it clearer how arguments are supplied and consumed on the stack.

1.2 Related work

As already mentioned, the Js_of_ocaml tool translates OCaml to JavaScript for running on the web. There is now a wide range of languages with compilers to WebAssembly [2]. This includes Grain [3], a functional language based on OCaml, designed for running on the web with a compiler that outputs WebAssembly. C can also be compiled to WebAssembly using Clang/LLVM [4], and I compare my compiler against all three of these approaches in my evaluation. The only compiler I could find from OCaml to WebAssembly was another Part II project from last year, which targeted a smaller subset of the language than my project.

1.3 Project summary

This project implements a compiler from OCaml to WebAssembly, supporting the core OCaml language and the integer, boolean, comparison and list operations from the standard library [5]. This was then extended with floating-point and reference operations. I also implemented several optimisation passes in the middle-end of the compiler, and implemented a garbage collector as part of the runtime system.

Chapter 2

Preparation

Before implementing my compiler, I researched the structure of WebAssembly programs and the intermediate representations used by the OCaml compiler, as well as possible optimisations to implement. I also considered which best practices to use to ensure that each stage of the project was manageable.

2.1 Starting point

Starting the project, my experience with OCaml was limited to the IB Compiler Construction course and studying the OCaml compiler, looking at the data structures used, and the ASTs generated for some short programs. I decided to build my project on the front-end of the OCaml compiler, after type checking has been performed, since there would be no benefit in reproducing this code. This was chosen rather than the next lower representation in the OCaml compiler, the Lambda language, as it introduces unnecessary structure and operations that would not be needed for the subset of OCaml I aimed to support.

I had read parts of the WebAssembly specification to understand the instructions available, and compiled a short C program to WebAssembly to check that I was able to run some of the tools involved. Many languages can now be compiled to WebAssembly, although lots of tools are still works in progress [2]. I chose to compare my compiler against compiling C to WebAssembly, since this appears to be the most matured platform in terms of supporting compilation to WebAssembly [4]. I also decided to compare against Grain, as this is a functional language similar to OCaml, and its official compiler emits WebAssembly [3]. I had not heard of Grain before starting to prepare for the project, so learnt the language from its documentation pages while setting up the tools at the start of my project. Lastly, I considered the performance of the Js_of_ocaml tool, which instead produces JavaScript from OCaml code, as this is the most well-known way to run OCaml programs on the web [1].

2.2 Research undertaken

For implementing my compiler, I first looked at the approach taken with the OCaml compiler's Lambda intermediate language, and the structure of the Grain compiler. Looking

at how translation is performed in the OCaml compiler helped to verify that I understood the semantics of the layer I was translating from, and identify the best way to handle specific elements, such as the primitive operations OCaml provides. Grain uses a very different intermediate representation, where operations are linearised so that the arguments to operations are always constants or variables, rather than nested expressions. I decided that my intermediate representation should have a linearised structure similar to Grain, since this makes the evaluation order explicit and simplifies implementing optimisation passes.

2.2.1 Optimisations

For selecting optimisations to implement, I looked at both the Part IB Compiler Construction [6] and Part II Optimising Compilers [7] courses for explanations of a range of optimisations. I also inspected the intermediate code and WebAssembly produced for sample programs, highlighting inefficiencies in translation and identifying where certain optimisations would be beneficial.

The optimisations I chose to implement were:

- **Tail-call optimisation**: Rewriting tail-recursive functions to use a `while` loop, rather than making recursive calls, avoids activation frames building up on the call stack, resulting in an error if the available stack space is exceeded.

- **Function inlining**: Where the function being called can be identified, a function application can be replaced by substituting it with the body of the function. This can enable further optimisations, possibly at the cost of increasing code size.

- **Uncurrying**: Where a function is always applied with several curried arguments, these can be passed as a tuple rather than constructing a closure for each argument in turn, as described by Bloss, Hudak and Young [8].

- **Constant/Copy propagation**: When a constant or another variable is bound to a variable that will not be overwritten, subsequent uses of that variable can be replaced with its known value, propagating values through the program to identify further optimisations.

- **Dead-code elimination**: Where a variable is unused (possible due to the previous optimisation) and the value bound to it does not contain side-effects, that binding can be safely removed to reduce code size and execution time.

- **Constant folding**: Constant expressions can be evaluated at compile time, such as `1+2`. Branches with constant conditions can also simplified, such as replacing `if false then e else e'` with `e'`.

The first three optimisations perform relatively complex transformations, requiring analysis to identify where they are both safe and beneficial to perform. The last three optimisations then eliminate redundancy, where some analysis is still necessary to correctly handle expressions with side-effects. This redundancy primarily occurs due to inefficiencies in translation to the intermediate representation, and because of the other optimisations.

2.2.2 Pattern matching

The Grain and OCaml compilers differ in the techniques used for pattern matching. The OCaml compiler implements a backtracking algorithm, which aims to minimise the size of the pattern-matching code produced, whereas the Grain compiler implements a decision-tree algorithm, which ensures that each pattern is examined at most once, minimising execution time. Each implementation references papers on the technique it uses [9, 10], which were useful in choosing the approach to take and in understanding how to implement pattern matching for my intermediate representation. I decided to implement a backtracking pattern-matching compiler. When optimised, each approach typically has similar performance [10], however backtracking compilers appear to be the more common approach, and guarantee linear code size.

2.2.3 WebAssembly

The majority of my preparation before the start of the project was familiarising myself with the structure of WebAssembly. This was achieved by reading the WebAssembly specification [11], as well as some useful articles on how the different components of a WebAssembly module interact in practice [12].

WebAssembly is a strongly typed stack-based language. As shown by the example in the introduction (section 1.1), WebAssembly supports nested blocks of instructions, in the form of `Block`, `If` and `Loop` constructs.

WebAssembly's control flow is very restrictive, only allowing branches out of enclosing blocks rather than arbitrary jumps. A `Br i` or `Br_if i` instruction (conditionally) branches out of the `i`$^{\text{th}}$ enclosing block, jumping to the end of a `Block` or `If` block and to the start of a `Loop` block, allowing loops in the control flow. Exceptions were left out of the subset of OCaml to support, since these restrictions in WebAssembly control flow significantly complicate their implementation, to the extent that exception handling is also disabled by default when compiling C++ to WebAssembly using Clang [13].

A WebAssembly module is initialised with a linear memory of a specified number of 64KiB pages, and the size of memory can be queried and expanded during execution. The memory layout of a native-code program typically has the heap starting at a low address, growing upwards, and the stack starting at a high address and growing down. However, as the stack is managed implicitly in WebAssembly, the linear memory contains just the heap so looks more like figure 2.1:

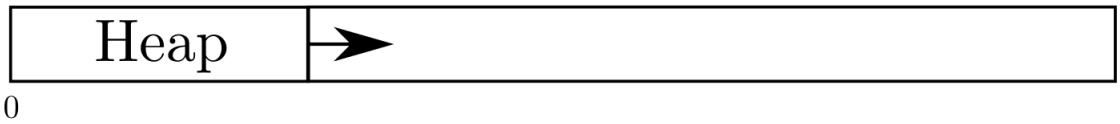

Figure 2.1: Typical WebAssembly memory layout

2.2.4 Garbage collection

OCaml is a garbage-collected language, so memory is managed automatically. However, memory is managed manually in WebAssembly, so a garbage collector must be implemented in the runtime system to manage objects allocated by compiled OCaml programs.

The two broad categories of garbage collectors are reference counting and tracing collectors. Reference-counting collectors keep a counter on each object for the number of pointers to that object that exist, freeing the object when the counter falls to zero. This has a high overhead, since a counter needs to be updated whenever a pointer is created or modified, and cyclic structures are never collected. Tracing collectors instead periodically scan a set of root objects that can be accessed directly, and use these to discover all objects indirectly reachable from them. Any object not marked as reachable is then safe to free.

WebAssembly's implicit stack cannot be accessed directly, which complicates garbage collection as the values on the stack are part of the root set and need to be scanned during garbage collection. Therefore, garbage collection was left as an extension due to the challenges surrounding it. One solution is to implement a shadow stack in the linear memory, which stores a copy of each local variable saved on the implicit stack, allowing these to be traversed during garbage collection. Shadow stacks are more common in the context of security, where a function's return address is copied elsewhere in memory and compared when the function returns, detecting attempts to overwrite the return address on the stack [14].

The shadow stack must extend upwards from the start of the linear memory, since WebAssembly memory grows over time, so the top of the address space is not initially valid. Therefore, the top of the stack is limited by the starting position of the heap, and the layout of memory is as shown in figure 2.2.

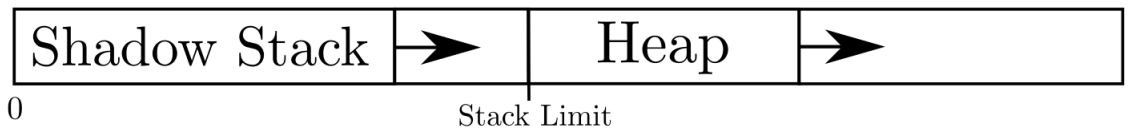

Figure 2.2: WebAssembly memory layout with a shadow stack

2.3 Requirements analysis

For this project to be successful, my compiler had to be able to compile OCaml programs that used just the integer, boolean, comparison and list standard-library operations, and did not use the module or object layers of the language, or exceptions. This defined the minimum subset of OCaml being supported. Achieving this required implementing multiple translation passes, first from the OCaml compiler's front-end output to my intermediate representation, and then from that to WebAssembly. I planned to verify the correctness of the translation passes by a set of test OCaml programs using the language features being supported. To be able to execute the resulting WebAssembly, I also needed to build a runtime system, to provide operations such as memory allocation.

I had to collect data measuring the performance of the code output by my compiler, comparing it against the three alternatives mentioned previously. Performance would be measured in terms of execution time, heap usage, and the size of the compiled code. This required writing a set of benchmark OCaml programs, with equivalent programs in C and Grain, and these benchmarks should have reflected a range of programming styles. Testing scripts were also necessary to compile each of these benchmarks, for each alternative,

and collect data for the three metrics identified. The aim was to demonstrate an improvement compared to the alternatives, particularly Js_of_ocaml, due to the inefficiencies of compiling OCaml to JavaScript mentioned in the introduction.

Implementing optimisations in the compiler was chosen as an extension, to help achieve this improvement. This required a set of analysis and optimisation passes on both the intermediate representation and WebAssembly stages of the compiler. I also aimed to extend the range of OCaml programs supported by my compiler, as floating-point and reference operations increase the range of benchmarks that could be considered.

Another extension was to implement a garbage collector, using a shadow stack to make local variables accessible during garbage collection. This would allow more memory intensive benchmarks to be considered and make the comparison of my compiler against the alternatives more realistic, by not ignoring the overhead of memory management.

2.4 Software engineering methodology

The project can be separated into several stages that had to be completed in sequence, following the stages of a typical compiler pipeline and outlined in the requirements analysis. Translation from the IR to WebAssembly was performed in two smaller steps, and the runtime system was designed in parallel with the back-end of the compiler. Optimisations were only added once I had a working compiler, and could each be implemented independently. Since I used the OCaml compiler's front-end, my compiler was also written in OCaml, which helped improve my familiarity with the language features being implemented. Similarly, the runtime system was written in WebAssembly, except for the garbage collector, written in JavaScript due to its complexity and to allow modifying it more quickly. WebAssembly was the natural choice for interacting with programs compiled as WebAssembly, and again helped improve my understanding of the language.

This structure of mostly independent translation/optimisation passes lends itself to the Kanban methodology [15], which is an incremental approach to software development I have used before on team projects. Work was broken down into manageable tickets, most of which could be completed in a couple of days, building on the project incrementally. I used the Jira project-tracking platform from Atlassian to maintain a board of tickets [16]. Having a list of tasks in progress or blocked by other components, updated regularly, meant that elements of the project were not forgotten about and I could track how long each task was taking, identifying when a task was more complex than anticipated and needed to be further subdivided. The tickets were also a means of grouping ideas or issues with the pieces of work they affected, and I added more precise tickets to the backlog as the project progressed and it became clearer which elements to implement next.

Both the project and dissertation were version-controlled using Git, backed up to GitHub daily and continually synced with OneDrive. At the start of the project, I also checked that I could run the tools required, such as the OCaml and Grain compilers, on the MCS machines as a backup.

2.4.1 Testing

Language features were added to the compiler following test-driven development. Before adding a new primitive, language construct or optimisation, OCaml programs would be

written using that feature, and the output of the OCaml interpreter on those files would be recorded. I wrote a script to compile and run each of these programs with my compiler, comparing their output with the expected values. This ensured that each new feature was implemented correctly, and that a change or new optimisation did not break any of the previous tests. Additionally, for tests added to verify that optimisations were performed safely, inspecting the output WebAssembly ensured that optimisations were being performed in the correct places.

Unit tests were written with OUnit [17] for several of the utility functions involved in translation and optimisation. For example, checking that the free variables in IR programs were computed correctly, or that removing an instruction from the flowgraph representation of a WebAssembly program correctly preserved the edges linking that node's successors and predecessors.

Testing whole translation passes was challenging, since translation often introduces several inefficiencies whilst still being correct, and the translation cannot be verified without running the full compiler pipeline and executing the output. Therefore, until the core stages of the compiler and runtime were built, I relied on pretty-printing code to display the intermediate representations for some sample programs. This allowed identifying any obvious errors by inspection, and was only relied on at the start of the project, before even a minimal pipeline had been written.

2.4.2 Tools used

I worked on the project on my Windows laptop, using Windows Subsystem for Linux to run OCaml. Opam, the OCaml package manager, was used to manage the OCaml compiler and packages needed for the project. I used the OUnit package for writing unit tests, the Wasm package for some of the back-end functionality, such as utility functions to output WebAssembly to a binary file, and the Js_of_ocaml package as an alternative to my compiler. The project used the front-end of version 4.11.1 of the OCaml compiler, which was the latest version at the start of the project, although version 4.12.0 has since been released.

I used the WebAssembly Binary Toolkit to translate between the binary and text format, both for producing a binary of the runtime system, and to inspect the output of my compiler on test programs. Node.js was used to run the output WebAssembly and perform benchmarking, and the project was developed using the IntelliJ IDE, available under its educational license.

2.5 Legality

I used of the front-end of the official OCaml compiler [18], which is distributed under the GNU Lesser General Public License. My use of the compiler and modifications to it are therefore permitted, provided that I retain the original licence in my project and highlight where modifications have been made.

Benchmark programs were adapted from two existing sources [19, 20], one of which is also distributed under the GNU LGPL, and the other under the BSD-3 licence. Therefore, modification is once again permitted, provided the original licences are retained.

Chapter 3

Implementation

In this project, I implement a compiler from OCaml to WebAssembly, primarily supporting the integer operations and all of OCaml's control-flow structures, besides exceptions. The front-end of the OCaml compiler produces a representation called the Typedtree. I translate this first to an intermediate representation I designed, and then to WebAssembly instructions. I also build a runtime system to execute the output code, and extend my project with several optimisations to the compiler and a garbage collector.

3.1 Linearised IR

From OCaml's Typedtree data structure, the first pass of the compiler translates programs to be closer to administrative normal form (ANF), which linearises the syntax tree so that operands are always constants or identifiers. Expressions are broken down into immediate, compound and linast (linearised AST) terms to achieve this. The general structure is given below, with some cases omitted for brevity.

```
linast = compound
  | Sequence (compound, linast)
  | Let(Identifier, compound, linast)
  | LetRec((Identifier * compound) list, linast)

compound = imm | unary_op imm | binary_op (imm, imm)
  | Block (tag, imm list)
  | Field (imm, index) | SetField (imm, index, imm)
  | If (imm, linast, linast) | Switch (imm, (tag * linast) list)
  | Try (index, linast, linast) | Fail index
  | Function (Identifier list, linast) | App (imm, imm list)

imm = Constant | Identifier
tag = index = int
```

Figure 3.1: Summary of IR structure

Temporary values are now explicit and the linear structure makes optimisations easier to implement. Code that may execute zero or many times cannot be completely linearised,

for example, the branches of an if statement, as only one branch will actually be evaluated.

Given an expression in the Typedtree to translate to an immediate or compound term, the expression is recursively decomposed, returning a list of let bindings for any temporary values required in its computation and an immediate or compound term for its final value. In this way, an initial call to translate an expression to a linast term builds up a list of setup operations and a final compound term, then combines these into a tree by chaining the setup operations together, terminated by the final compound term.

3.2 Pattern matching

Translation to the linearised IR also compiles patterns in the Typedtree to code that matches them. This includes combining the cases of a match expression or function into a single expression, making use of switch statements on integers where possible.

I chose to implement the backtracking pattern-matching compiler described by Le Fessant and Maranget [9], which is also implemented in the OCaml compiler for its intermediate representation. The idea is to have a vector of the values being matched, and a matrix representing the different cases and the action when each matches. An example of this is:

```
let f l1 l2 = match l1, l2 with
  | [], _ -> e1
  | _, [] -> e2
  | x::xs, y::ys -> e3
```

$$\text{values } \begin{pmatrix} \texttt{l1} \\ \texttt{l2} \end{pmatrix}, \text{ cases } \begin{pmatrix} \texttt{[]} & \texttt{\_} & \rightarrow & \texttt{e1} \\ \texttt{\_} & \texttt{[]} & \rightarrow & \texttt{e2} \\ \texttt{x::xs} & \texttt{y::ys} & \rightarrow & \texttt{e3} \end{pmatrix}$$

Figure 3.2: Representing a program as a pattern matrix and value vector

The algorithm given in the paper is made more complex by the range of syntax OCaml supports. First, OCaml patterns can have guard statements, which must be tested to see whether a pattern matches. Some of the compilation steps prefix additional variable bindings onto the actions, and the guard statement needs to be evaluated after these, but before any of the original action is evaluated. Therefore, each action $\rightarrow a$ is replaced with $\rightarrow a\ g\ b$ where g is the optional guard expression and b is the other bindings introduced.

The base case of this algorithm occurs when we reach a matrix with no rows. This indicates that we have run out of possibilities, so pattern matching fails and backtracks. If instead there are no columns, we have successfully matched the whole pattern. If the first row has no guard, we output the action of that row, otherwise the guard is tested and the action forms the `then` case of an if statement, with the remaining rows of the matrix considered in the `else` case.

There are four broad cases for when the matrix is non-empty:

- **Variable rule**: If the first column has just variable patterns, a corresponding binding of the first value in the values vector is added to each row's bindings, then the first column of the matrix is discarded, as is the now-matched value of the value vector.
- **Constructor rule**: If the first column has just constructor patterns, several specialised matrices are produced, each removing incompatible rows and deconstructing the fields of the chosen constructor. The result of compiling each of these then forms

the branches of a switch statement on the constructor tag. A default case captures any constructor with no corresponding rows, and fails so that matching backtracks.

$$\begin{pmatrix} c(q_1,\ldots,q_k) & p_2^1 & \ldots & p_n^1 & \to & \ldots \\ c'(\ldots) & p_2^2 & \ldots & p_n^2 & \to & \ldots \end{pmatrix} \xrightarrow{\text{specialise } c} \begin{pmatrix} q_1 & \ldots & q_k & p_2^1 & \ldots & p_n^1 & \to & \ldots \end{pmatrix}$$

Figure 3.3: Constructor specialisation on a pattern matrix

- **OR rule**: If there is just one row and it starts with an OR pattern `(p1|...|pk)`, a $k \times 1$ matrix is produced expanding out the OR pattern. The output is the result of compiling this matrix, sequenced with the result of compiling the rest of the original matrix. Simply duplicating the original row once for each case of the OR pattern would duplicate all patterns in later columns. That could lead to exponential growth in code size, which backtracking compilation aims to avoid.
- **Mixture rule**: If no other case is applicable, the matrix is split into an upper and lower matrix, where one of the other rules can be applied to the upper matrix. The matrices are compiled separately, with failure in the upper matrix resulting in a `Fail i` instruction. This is handled by the `try ... with (i) ...` construct, which backtracks to try the lower matrix instead.

The explanation above is how the paper describes the algorithm. In practice, OCaml has several different types that all fall under the "constructor" rule and must be handled separately:

- **Constants**: Constants can be viewed as a datatype with infinitely many 0-arity constructors, so compile to a switch statement on the constant's value. A default case that backtracks is always included, for all of the constant values not handled.
- **Arrays**: In OCaml, array patterns look like `[|p1, p2, ..., pk|]`, matching a specific length. The length is the tag of the object, so a switch statement on the tag is generated, extracting the corresponding number of fields in each case. A default case is always included for the array lengths not handled.
- **Records**: A record type is just a constructor with a single variant, and each pattern specifies a subset of the fields to match. To allow all rows of the matrix to be processed, the union of the fields mentioned in each row is extracted from the record, inserting the wildcard pattern `_` where a row does not match a field, so that the extracted value is ignored. In the example below, any unmentioned fields are ignored entirely.

$$\begin{pmatrix} \{x = q_1\} & p_2^1 & \ldots & p_n^1 & \to & \ldots \\ \{y = q_2\} & p_2^2 & \ldots & p_n^2 & \to & \ldots \\ \{x = q_3; y = q_4\} & p_2^3 & \ldots & p_n^3 & \to & \ldots \end{pmatrix} \to \begin{pmatrix} q_1 & \_ & p_2^1 & \ldots & p_n^1 & \to & \ldots \\ \_ & q_2 & p_2^2 & \ldots & p_n^2 & \to & \ldots \\ q_3 & q_4 & p_2^3 & \ldots & p_n^3 & \to & \ldots \end{pmatrix}$$

Figure 3.4: Handling of record fields in pattern matrices

- **Tuples**: Tuples are constructors with a single variant, just like records, except that a tuple pattern always specifies patterns to match against each element of the tuple.

3.2.1 Optimisations to pattern matching

The above process is made more efficient by including auxiliary information.

First, the mixture rule can reduce the number of splits needed by maximising the size of the first matrix, which is then handled by a single use of another rule. This is achieved by swapping incompatible rows. Patterns must be matched in top-to-bottom order, but this does not matter for two rows that match disjoint sets of value, allowing them to be swapped. For example, the two programs in figure 3.5 are equivalent, however the constructor rule can only be applied to the first row on the left, but to the first two rows on the right:

```
match l1, l2 with
  | [], _ -> 1
  | _, [] -> 2
  | x::xs, y::ys -> 3
```

```
match l1, l2 with
  | [], _ -> 1
  | x::xs, y::ys -> 3
  | _, [] -> 2
```

Figure 3.5: Example of reordering patterns not changing a program's semantics

Next, several data structures are added to the procedure. They are all related so I describe each of them before explaining their benefit.

The procedure is extended with a context, a pair of matrices of just patterns with no actions or guards. The first is called the prefix, remembering already matched structure, and the second is the fringe, describing the known structure of the values left to be matched. Together, these describe the possible values that may reach the point in the pattern-matching code currently being produced. Initially there is just one row, where the prefix is empty and the fringe is `_`, as nothing is known about the value being matched. The specialisation operations on matrices extend to contexts in the obvious way, for example with constructors ($\bullet$ separates the prefix and fringe):

$$\begin{pmatrix} p_1^1 \dots p_k^1 & \bullet & c(q_1, \dots, q_k) & \dots & q_n^1 \\ p_1^2 \dots p_k^2 & \bullet & \_ & \dots & q_n^2 \\ p_1^3 \dots p_k^3 & \bullet & c'(\dots) & \dots & q_n^3 \end{pmatrix} \xrightarrow{\text{specialise } c} \begin{pmatrix} p_1^1 \dots p_k^1 & c(\_, \dots, \_) & \bullet & q_1 & \dots & q_k & \dots & q_n^1 \\ p_1^2 \dots p_k^2 & c(\_, \dots, \_) & \bullet & \_ & \dots & \_ & \dots & q_n^2 \end{pmatrix}$$

Figure 3.6: Constructor specialisation on a context

The procedure is also passed a list of reachable handlers. When the mixture rule splits a matrix into upper and lower parts, P and Q, and introduces handler i, then P is complied with (i, Q) added to its handler list, although only the patterns of Q are stored in the handler list, not the actions or guards. Again, operations on matrices are naturally extended to each matrix in the list.

Correspondingly, in addition to the code to match a matrix, the procedure returns a jump summary, a mapping from indices of enclosing handlers to the contexts representing sets of values that backtrack to those handlers. When the algorithm produces `fail i` to backtrack, it returns a jump summary mapping `i` to the current context, indicating that such values trigger backtracking to handler `i`. At branches, the jump summaries of each branch are unioned by concatenating the rows of contexts for the same handler.

If compiling the upper matrix in the mixture rule returns jump summary ρ, then when compiling the lower matrix Q for handler i, the context $\rho(i)$ is used.

The reachable handler list remembers which patterns each handler will try to match if exited to, and the contexts obtained from the jump summary provide some information on the structure of values that will trigger that handler.

The first improvement is that, rather than always backtracking to the nearest handler, handlers guaranteed to fail can be skipped over. Since the matrices matched by reachable handlers are specialised as more specific patterns are matched recursively, at a point where backtracking occurs, some of those matrices may be empty, indicating that no values that reach this point in the code would be matched by those handers either. The procedure instead inserts a `fail i` to the first handler `i` with a non-empty matrix.

Second, the lower matrix for a handler may not cover all constructor cases if some are covered by the upper matrix matched before backtracking. Naively, this would result in additional cases for the missing constructors, which we know will never occur. By taking the context from the jump summary, we know the set of values that can actually reach the handler. If some constructor is incompatible with every pattern in the first column of the fringe of the context, the value being matched in the handler will never be that constructor, hence such cases can be discarded to reduce code size. In instances where this reduces the switch statement to a single case, the test can be removed entirely.

3.3 Compiling identifiers

After any optimisations on the IR, the linearised tree is lowered to WebAssembly in two stages. The first of these pulls function definitions out to the top level of the program, making closures and free variables explicit. It also converts the IR tree to lists of instructions (with nesting still present for `if`, `for` and `while` statements as WebAssembly supports nested blocks) and replaces string identifiers.

Identifiers are replaced with a datatype indicating the kind of variable being accessed, with free variables being stored in the closure of a function:

```
binding = Global of int | Arg of int | Local of int | Closure of int
```

Each let binding in a function's body generates a new local variable for that identifier. This results in a large number of local variables that is reduced after peephole optimisations on the generated WebAssembly are performed. The WebAssembly representation is stored with a flowgraph between instructions, a graph indicating which instructions may execute immediately before or after each instruction in a function. Live-variable analysis is performed on this, identifying the points in a function where each local variable's current value could be used later. These points indicate regions where a variable cannot be overwritten. From this, a clash graph is constructed with edges between every pair of variables that are ever simultaneously live, so must be stored separately. A graph-colouring heuristic then maps these local variables to a hopefully much smaller set of local variables, and unused local variables are removed from the function.

Using fewer local variables reduces code size, as a WebAssembly function must declare the type of each local variable. Also, the number of local variables used by a function directly affects the space it occupies on the shadow stack, needed for garbage collection.

Therefore, this is a useful optimisation over allocating local variables according to scoping, which will generally be much less efficient.

3.4 Code generation

Finally, the program is translated to WebAssembly code, with most operations mapping directly to fixed blocks of code that are concatenated together. Two additional "swap" local variables are introduced in each function, needed to compile some operations to WebAssembly. For example, if `ar` evaluates to an array pointer and `index` evaluates to the index being accessed, checking the array bounds and performing the load emits the code below. Some steps have been omitted for brevity, such as removing tags from values, or shifting the index to be a byte offset rather than a word offset.

```
ar
LocalTee 0       ;; swap variable 0
Load             ;; the size of the array is stored at its base address
index
LocalTee 1       ;; swap variable 1
Compare GtU      ;; size > index, unsigned comparison catches index < 0
if
  LocalGet 0
  LocalGet 1
  Add
  Load offset=8  ;; adjust for the start of the array storing its size
else
  trap
end
```

Figure 3.7: Code generated for accessing an array element

The expressions `ar` and `index` could be free variables being loaded from a closure, in which case storing their values in local variables avoids repeated loads from memory. The first local variables of a WebAssembly function are its arguments, so a free variable binding `Closure i` compiles to a `LocalGet 0` to get the closure pointer, then a `Load` with an offset to that variable.

I wrote a runtime in WebAssembly providing functions for memory allocation, polymorphic comparison, boxing of floats in memory, and list append, which are declared as imports into compiled WebAssembly modules. I also wrote a short JavaScript wrapper to interpret integers and pointers to data or closures. This means that JavaScript code can interact with the WebAssembly values without needing to understand the runtime representation, and can call functions using returned closure pointers.

Data must be tagged to identify its type, so that OCaml's `compare` function can be implemented correctly, which is defined on all types. As every operation is on either 32-bit integers or 64-bit floats, pointers are always aligned on 4-byte boundaries, so the last 2 bits are unused. I therefore distinguish the types of values by these last two bits. Closure pointers are tagged with 11, pointers to other objects in memory, such as constructors,

are tagged with 01, and integers are tagged with 10 or 00 by shifting them left one, so odd values now end in 10 and even values end in 00.

The layout of objects stored in memory is given by figure 3.8, where the "arity" field specifies the number of 32-bit values in "elements" / "free variables". Since arguments and return values in WebAssembly are strongly typed, and polymorphic OCaml functions need supporting, all values are 32-bit integers that are decoded as pointers to data or functions, as just described. In particular, floating-point values are "boxed" in memory and represented as pointers to data blocks storing their value. The variant tag on data blocks is non-negative, so -1 is used as a special value to distinguish blocks containing a float value.

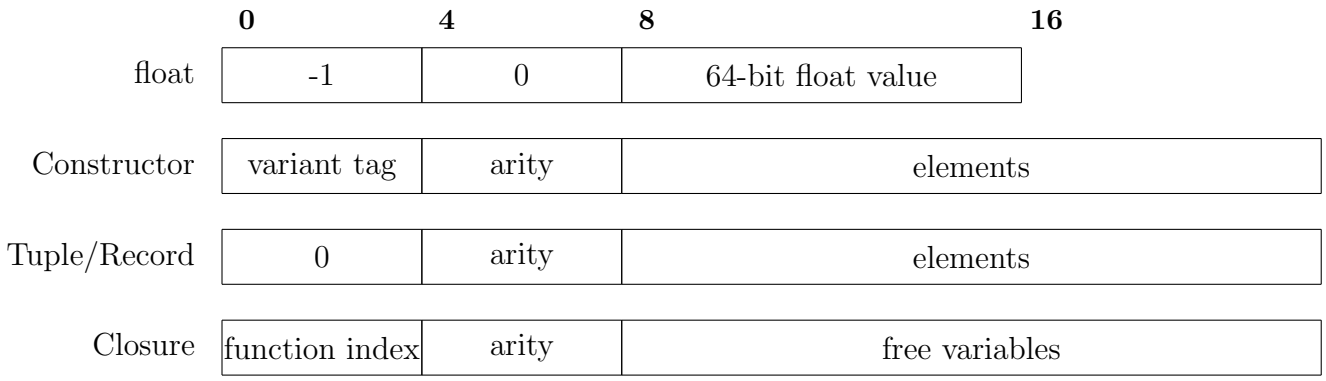

Figure 3.8: Memory layout of objects, column headers are byte offsets

The arity field in closures indicates the number of free variables. `compare` is undefined on function values, but the arity is still needed for garbage collection so that pointers stored as free variables in closures can be searched. The field is unnecessary for floating-point values, but is included and set to 0 so that garbage collection can search each of these blocks for pointers in the same way, by first looking at the arity field then checking that many subsequent fields.

3.5 Optimisations

One of the extensions for my project was to implement optimisations once I had a working compiler. Three main optimisation passes were added: function inlining, uncurrying and tail-call optimisation. Several cleanup passes were also implemented, propagating values to reveal further optimisations, and removing inefficiencies at both the IR and WebAssembly level by detecting dead code.

3.5.1 Tail calls

Tail-call optimisation is important in functional languages, since loops in a functional language are written as recursive functions, so a naive implementation for a large number of iterations can easily exhaust the available stack space. This is avoided by wrapping the body of the function in an imperative while loop and replacing the tail call with code that saves the arguments the function would have been called with and updates a variable to say the loop should not exit yet. This is extended to mutually tail-recursive functions by additionally storing which function should be called next when a tail call is reached.

There is first an analysis pass to construct a call graph of just the tail calls in recursive functions. Nodes in the graph correspond to recursive functions, with each node having edges to the functions it tail calls. This is performed conservatively, so if a function makes a tail call with a function variable that cannot be identified as a particular function, no edge is added. Tail-call optimisations only apply to fully applied curried functions, so we also discard any over or under-applied call sites.

As detailed below, tail-call optimising a recursive function increases its size, and the optimisation is more expensive for mutually tail-recursive functions, as each function must be split in two. Therefore, this call graph is iteratively pruned to identify functions worth optimising. Functions with no tail calls are removed from the graph, as are functions that only make tail calls to themselves, hence should be optimised in isolation. Once this process converges, we have the set of functions to make mutually tail recursive. Tail calls between these functions will then be rewritten to avoid making additional function calls.

```
while graph changing:
    for each f in graph:
        if f makes no tail calls:
            remove f and all edges to it
        else if f only tail calls itself:
            remove f and all edges to it
            optimise f on its own
```

Figure 3.9: Iterative algorithm to determine mutually tail-recursive functions

When deciding on this method for choosing which functions to tail-call optimise on their own or as mutually recursive functions, I studied how the Grain complier implements this analysis. In doing so, I identified a bug where it would wrongly optimise certain tail calls, and submitted an issue on GitHub[1] detailing this, which has since been corrected.

For a function `f` taking argument `x` that is to be made mutually recursive on its own, we create new variables `result`, `continue` and `x'`. The argument of `f` is changed to be `x'` and the body of `f` is replaced with an assignment `x = x'` and a while loop on the `continue` variable. This loop sets `continue` to false and `result` to the result of evaluating the original body of `f`. A tail call `f(y)` is replaced with `x = y; continue = true`. This will therefore execute the body of `f` again with the new argument each time a tail call occurs. When the loop eventually exits, `result` is the value the function returns. When `f` is a curried function with multiple arguments, a new variable is created for each argument.

[1] https://github.com/grain-lang/grain/issues/506

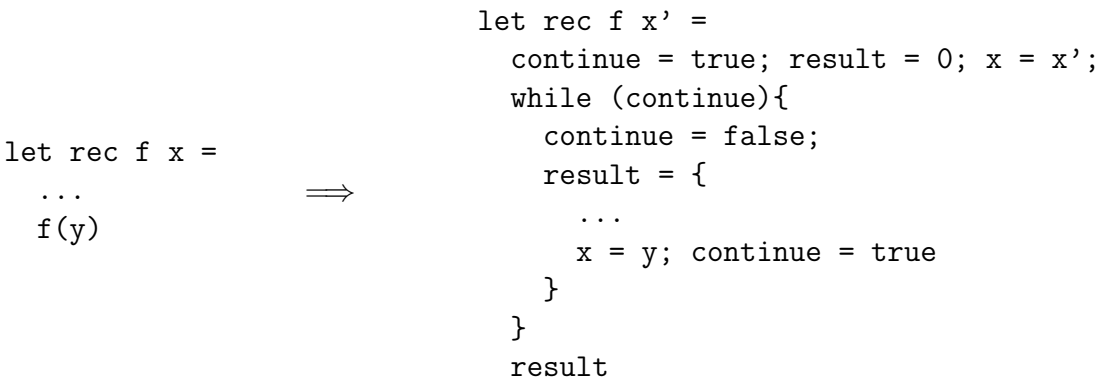

Figure 3.10: Example of how a single function is tail-call optimised

Above we were able to include the body of the function in the new function. For mutually tail-recursive functions, the while loop must execute one of a number of mutually recursive functions so this is not possible. Instead, the body of `f` is moved to a new function `_f` taking a unit argument and the new function `f` just executes a while loop, calling a function selected by a variable `next`, set at each tail call. `next`, `continue` and `result` are all shared global variables in this case, so they can be set by different functions. A shared set of global variables is also used to pass arguments at tail calls, and their values are bound to the original function variables when `_f` executes.

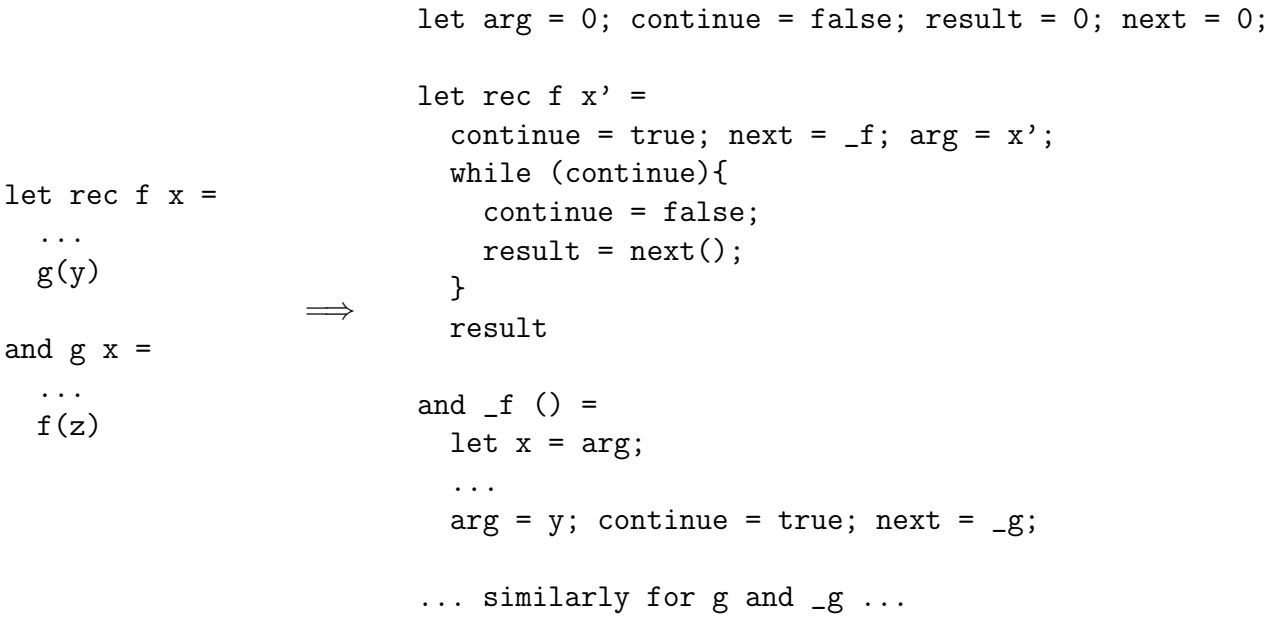

Figure 3.11: Example of how mutually recursive functions are tail-call optimised

3.5.2 Uncurrying

A curried function of many arguments is implemented in WebAssembly as several functions, where all but the last function just constructs the closure to the next function, copying across free variables in the closure and the curried argument.

This is made more efficient for functions that are not exported and are always called with all of their curried arguments. In this case, we can replace the set of curried functions with one function that takes all of its arguments as a tuple, and update the calling

sites accordingly. Rather than actually creating a tuple in memory and passing that to the function (as in a function like `f (x, y) = x + y`), the WebAssembly function takes multiple arguments directly, saving on memory usage. Functions and applications that are optimised in this way are annotated in the IR tree, so that translation to WebAssembly distinguishes them from functions called using curried arguments.

This optimisation requires that every call to the function can be identified precisely, since some other call that is not annotated would be compiled to a function call passing one argument at a time, which would be incorrect. For this reason, the optimisation is only done when every occurrence of the function variable is an application, so it is never bound to another variable or passed between functions. The compiler then relies on copy propagation, described below, to transform cases such as `let g = f in g(1,2)` to direct uses of `f` and removing `g`, allowing `f` to be optimised.

This could also be performed where a function always has more than one but not all of its arguments applied, but the benefit would not be as significant.

3.5.3 Inlining

Only non-recursive function applications are inlined, allowing values to be propagated inside the function body and some expressions to be evaluated by the compiler. This is achieved using some simple heuristics based on those described by Bonachea for the Titanium compiler [21].

A function is always inlined if its body contains at most 5 nodes in the IR tree, as the increase in code size is negligible and this avoids making function calls for simple operations. Also, if a function is only used in one place and not exported from the program, it is inlined, since the function definition can then be removed so code size does not increase. Larger functions are only inlined if doing so does not exceed the budget for code size, in terms of nodes in the IR tree. The resulting program cannot be more than 50% larger than the original program size. This limit was found to be enough that useful inlining occurred, without the output code size increasing significantly once further optimisations were performed.

3.5.4 Copying expressions, variables and constants

As OCaml is a mostly functional language, local variables in the IR are only assigned to once (excluding variables introduced specially for tail-call optimisation, which are marked as mutable and ignored), making these optimisations relatively straightforward.

First, the program is scanned and each `let x = Constant c` or `let x = Ident y` binding is added to a table. These are "immediate" values in the linearised IR and indicate bindings that do no real work. Whenever a variable `x` present in the table is used, it is replaced with its known value.

This is also extended to values passed through immutable memory, such as tuples, constructor fields or some record fields. During translation to the IR, these expressions in OCaml's `Typedtree` are translated to the `Block` expression in the IR, and tagged with an `ImmutableBlock l` annotation, where `l` is a boolean list indicating which of the block's fields are immutable.

When the IR tree is analysed, `Block` nodes are annotated with the variables or constants stored in the block's immutable fields. A table mapping variables to annotations records this when the block is bound to a variable, so this propagates to any uses of the variable too. This information is also approximated at branches, by taking the intersection of the known fields when each branch returns a block. For example, `if cond then (x, y) else (x, z)` would annotate the `if` node with that fact that its first field is `x`. When an immutable field of a block is accessed as `Field(b, i)`, if the known value of that field is annotated on `b`, the node is replaced with the known constant/variable, avoiding accessing it indirectly.

Common subexpressions are optimised similarly to constant and identifier assignments. First, side-effect-free expressions are identified by the effects analysis described below. When such an expression is bound to a variable `x`, subsequent occurrences of that expression are replaced with `x`, as it holds the same value and avoids recomputing the expression.

These passes enable further optimisations, by removing redundancy or indirect accesses in the program. They also result in unused variables, due to expressions being replaced with their known values. Therefore, after these passes have been performed, variable assignments that contain no side-effects and are never used are removed.

3.5.5 Effects analysis

OCaml is an impure language, so analysis of side effects is necessary to determine when expressions can be safely removed as above. This is achieved using the following annotations on nodes in the IR tree:

```
annotation = ... | Pure | Immutable | Latent of annotation list
```

`Pure` identifies expressions that do not modify program state, making them safe for dead-assignment elimination to remove. This includes creating a block in memory, as it could be immediately garbage collected if unused.

Of the pure expressions, `Immutable` identifies expressions that will return the same value each time they are evaluated, so common-subexpression elimination can remove repeated occurrences of such expressions. Creating blocks containing only immutable fields is treated as immutable, as is accessing an immutable field of a block, since it can never be overwritten so always has the same value.

The `Latent` annotation is used for functions, since a function definition itself is pure and immutable, but the body of the function may cause latent side effects when the function is called. Therefore, if a function body has annotations F, the function definition is annotated with {`Pure`, `Immutable`, `Latent`(F)}, and F is unwrapped at function applications.

The algorithm is summarised by the inference rules and functions in figure 3.12, abbreviating the three annotations as P, I and L. Γ is a mapping of both variables and handler indices to analyses. The algorithm safely over-approximates possible side-effects, by under-approximating the annotated properties in multiple places. Nothing is assumed about the latent effects of variables that have not been analysed, which occurs for function parameters or recursive function variables, and the intersection of properties is taken at branches.

The function `seq` encodes that the latent effects of `e; e'` are those of `e'`, but the expression is only pure/immutable if both `e` and `e'` are. The function `app` encodes the same for unwrapping the analyses of latent effects as a function is applied. This is necessary, rather than just returning the latent effects, since a partial application may perform some impure computation then return another function, in which case the overall application is impure.

$$\frac{\texttt{x} : F \in \Gamma}{\Gamma \vdash \texttt{x} : F \cup \{P, I\}} \qquad \frac{\texttt{x} \notin \Gamma}{\Gamma \vdash \texttt{x} : \{P, I\}} \qquad \frac{\Gamma \vdash \texttt{e} : F \quad \Gamma, x : F \vdash \texttt{e'} : F'}{\Gamma \vdash \texttt{let x = e in e'} : \texttt{seq}(F, F')}$$

$$\frac{\Gamma \vdash \texttt{e} : F}{\Gamma \vdash \texttt{fun x -> e} : \{P, I, L(F)\}} \qquad \frac{\Gamma \vdash \texttt{e} : F \quad \Gamma, f : \{P, I, L(F)\} \vdash \texttt{e'} : F'}{\Gamma \vdash \texttt{let rec f x = e in e'} : F'}$$

$$\frac{\Gamma \vdash \texttt{f} : F \quad \texttt{app}\ F\ \texttt{args} = F'}{\Gamma \vdash \texttt{f(args)} : F'} \qquad \frac{F = \{P, I\} \text{ if each field is immutable else } \{P\}}{\Gamma \vdash \texttt{Block (tag, args)} : F}$$

$$\frac{}{\Gamma \vdash \texttt{SetField (e, i, e')} : \{\}} \qquad \frac{F = \{P, I\} \text{ if field i of e is immutable else } \{P\}}{\Gamma \vdash \texttt{Field (e, i)} : F}$$

$$\frac{\Gamma \vdash \texttt{e} : F \quad \Gamma \vdash \texttt{e'} : F'}{\Gamma \vdash \texttt{if p then e else e'} : F \cap F'} \qquad \frac{\Gamma \vdash \texttt{e} : F \quad \Gamma \vdash \texttt{e'} : F'}{\Gamma \vdash \texttt{e; e'} : \texttt{seq}(F, F')}$$

$$\frac{\Gamma \vdash \texttt{e'} : F' \quad \Gamma, i : F' \vdash \texttt{e} : F}{\Gamma \vdash \texttt{try e with (i) -> e'} : F} \qquad \frac{i : F \in \Gamma}{\Gamma \vdash \texttt{fail i} : F} \qquad \frac{}{\Gamma \vdash \texttt{Const c} : \{P, I\}}$$

```
let seq (F1, F2) =  F2 \ ({P,I} \ F1)

let rec app F = function
  | [] -> F
  | arg::args ->
    if L(F') ∈ F:
      seq (F, app F' args)
    else: {}
```

Figure 3.12: Summary of the inference rules for effects analysis

Using a linearised IR has resulted in simpler rules in some cases. For example, the condition of an `if` statement is always a variable or constant so has no side effects, hence it is ignored.

3.5.6 WebAssembly peephole optimisations

At the WebAssembly level, in addition to being used for register allocation, live-variable analysis identifies assignments to variables that are unused so can be removed. Peephole optimisations also simplify several common cases resulting from compiling each operation independently. For example, `LocalSet i; LocalGet i` is replaced with `LocalTee i`, and `i32.const n; drop` is removed (`i32.const 0; drop` often occurs where a unit-valued expression is the first part of a sequence `e1; e2`). These changes enable a more precise analysis of which variables are actually used by a program, allowing further dead assignments to be removed.

3.6 Garbage collection

One of the last extensions to my project was to implement a garbage collector as part of the runtime system. Due to its complexity compared to the other runtime functions and going through several improvements, this was written in JavaScript and imported into the WebAssembly runtime rather than writing it in WebAssembly.

A tracing collector marks all objects reachable from pointers on the stack and global variables, then collects all allocated unmarked objects. Since the stack in WebAssembly is implicit, local variables are copied to the shadow stack in linear memory, allowing them to be explored during garbage collection. This additional store on every local-variable update is one of the main overheads of implementing garbage collection. The only bookkeeping required is a pointer to the top of the shadow stack, which moves up and down as functions are called and return.

3.6.1 Simple memory allocator

The garbage collector can be split into two components, one that provides `malloc` and `free` functions for manual memory allocation, and the other that implements the mark and sweep algorithm on top of this to decide which blocks to free.

I initially implemented the first part following the algorithm described in K&R [22]. A cyclic linked list, called the free list, is maintained, chaining together all of the free blocks in memory, and the runtime keeps a pointer to some block in the list. Each block has an 8-byte header containing the size of the block and a pointer to the start of the next block. As all values in compiled programs are 32-bit integers or 64-bit floats, blocks are always aligned to at least 4 bytes, so the least significant bits of each field are unused, leaving space for an allocated and marked flag.

`malloc` scans the free list until the first large-enough block is found. Either the whole block is allocated if it is a perfect fit, else the size of the block is reduced and just the tail end of it is allocated. If there are no large-enough blocks, the garbage collector is called. If that frees a large-enough block then the list is scanned again, otherwise the WebAssembly memory is grown and a block allocated from the new space.

`free` also scans the free list, until the last unallocated block before the block being freed is found. The freed block is then inserted here in the list, keeping blocks in the list in memory order. Existing free blocks adjacent to it in memory are also merged into one larger block to reduce fragmentation.

The mark and sweep algorithm then determines which blocks to free. When invoked, the shadow stack is scanned and each block pointed to is marked and put onto a stack. This stack is maintained using the pointer fields in the headers of blocks, which are otherwise meaningless for allocated blocks not in the free list. A depth-first search is then performed to mark all reachable blocks, using this stack to track which objects to explore the fields of next. Finally, the sweep part of the algorithm traverses all blocks in memory, clearing marked flags and freeing allocated blocks that were not marked and are therefore unreachable.

3.6.2 Optimised malloc and free

The implementation above requires `malloc` and `free` to both walk along the free list, either to find a large-enough block or to find where to insert a freed block. `free` can be made to always run in constant time, as can `malloc` for the majority of calls.

First, an additional 8 byte trailer is added to each block, used as shown below, with the flags A = allocated, M = marked. A block's size, and whether it is allocated or not, is now accessible from either end of the block. The free list is doubly linked but is no longer made circular.

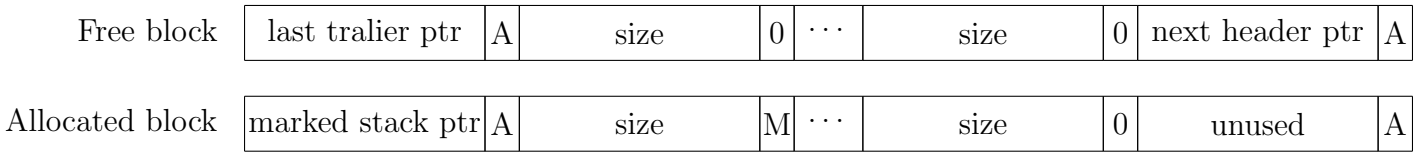

Figure 3.13: Memory layout of the header and trailer for optimised memory allocation

This is enough to be able to free a block without scanning the free list. From just the freed block's address, the procedure gets its size and checks the allocated flags of the blocks either side of it. This determines if either block should be merged with the one being freed. Also, since the list is doubly linked, each adjacent free block has a pointer to its successor and predecessor block so can be removed from the middle of the free list without traversing it. If neither block is merged, the newly freed block is inserted at the front of the free list, so the free list no longer keeps blocks in address order.

The cost of `malloc` comes from scanning the free list for a large-enough block. The solution to this is binning blocks into separate free lists based on their size, so scanning for a free block can skip searching blocks that will all be too small. All but the last bin fit exact sizes starting from the smallest possible allocation, then the last bin holds all larger blocks. The compiler supports limited array operations, so the size of most allocations is determined by user-defined datatypes and the number of free variables in functions. Often both are relatively small, hence most free blocks are placed in the fixed size bins. `free` must now insert freed blocks into the correct free list, and shrinking blocks when allocating, or merging blocks when freeing, means that free blocks sometimes move between lists.

A characteristic observed from tracing the behaviour of memory-intensive test programs is that, until garbage collection fails to free a suitable block and memory has to be grown, the garbage collector typically has diminishing returns each invocation as the set of long-lived objects grows. This results in there being less memory available over time and the garbage collector being called increasingly frequently. Memory grows in multiples of WebAssembly pages, which are 64KiB. Once the memory being freed drops below about 1KiB, a future garbage collection almost always grows memory eventually. Therefore, another optimisation was to grow memory as soon as this threshold is crossed, rather than waiting for several inefficient collections before the available memory is fully exhausted. This reduces the overhead of garbage collection by avoiding many inefficient collections, without growing memory unnecessarily.

3.7 Summary

This chapter has described the components of my compiler and the decisions made while writing it. A wide range of OCaml programs that do not use the module language can be compiled to WebAssembly, and executed using JavaScript. I have also described the extensions made to the project, adding a garbage collector and several optimisation passes, which will be evaluated in the next chapter. The original goal was to support the comparison, boolean, integer and list operations from the standard library, which has been achieved and extended with the reference and basic floating-point operations.

3.8 Repository overview

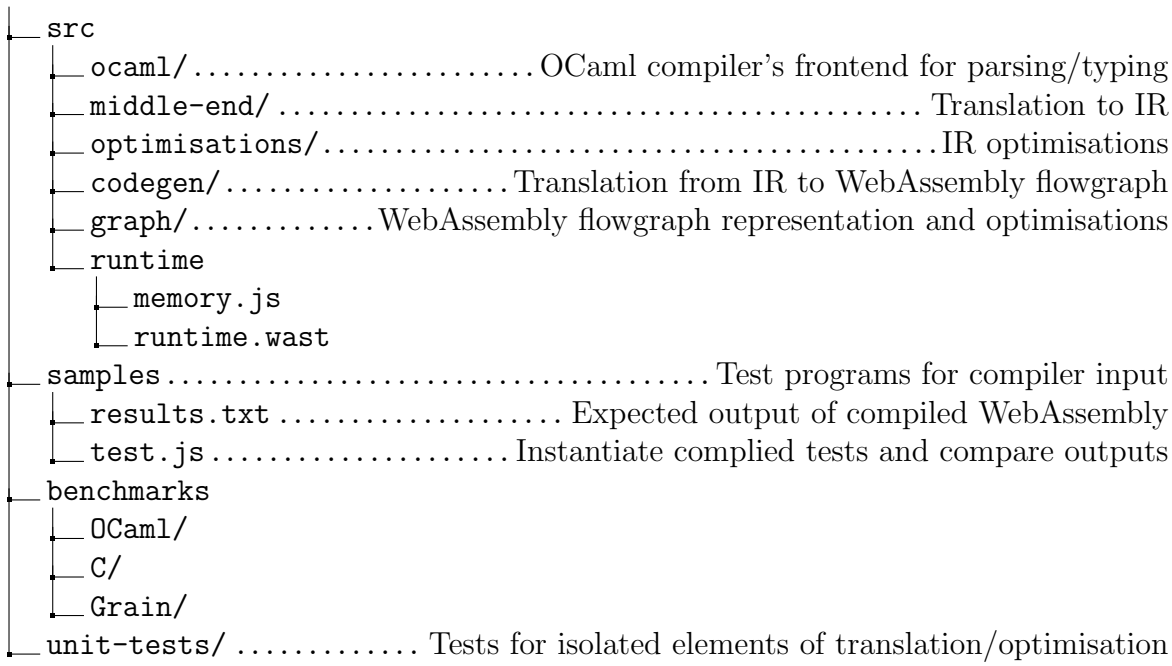

Figure 3.14: Overview of the structure of the repository. Most of the contents of directories are omitted for brevity, such as the example programs in the `samples` directory.

Chapter 4

Evaluation

This chapter compares the performance of my compiler with three existing alternatives for running code on the web: compiling C to WebAssembly with Clang/LLVM, compiling Grain to WebAssembly, and translating OCaml programs to JavaScript with the Js_of_ocaml tool. I also measure the impact of the optimisations implemented in my compiler. Performance is based on three metrics:

- **Execution time**: This is measured by a JavaScript program, run using Node.js, for each of the alternatives. Timing information is obtained from the `performance` interface, which has up to microsecond precision and is not affected by system time changes, so is guaranteed to be monotonic [23]. This is sampled over 20 runs and the mean and 95% confidence interval are recorded.
- **Heap usage**: The heap memory used is also collected by these scripts. Without garbage collection, my runtime allocates memory linearly so a call to allocate 0 bytes returns the total memory used. For the garbage-collected allocator, I created a separate version that tracks the peak memory allocated, updating this on each allocation.

  The Grain runtime's development build outputs similar memory-tracing statistics, including the amount of heap memory used. The overhead of tracing is significant so this is performed separately to collecting timing information. For C, where parts of `stdlib` are included in the WebAssembly output for memory allocation, `sbrk(0)` returns the size of the WebAssembly linear memory used. Lastly, for programs translated to JavaScript, the benchmarking script is run with the `--expose-gc` option. This allows calling the garbage collector before the program is executed, and obtaining the memory used from `process.memoryUsage().heapUsed` before and after the program runs. This is an approximate value, so is also averaged over 20 iterations.
- **Output file size**: This is easily obtained from the file system.

4.1 Microbenchmarks

I wrote a set of test programs, each aiming to represent a different programming style, to see how performance varied across applications. The programs were also parameterised so that comparisons could be made at different problem sizes. Where available, these programs were based on code from existing benchmarking libraries.

- `alltrees`: Constructs a list of all binary trees of a given size, which is very memory intensive.

- `arith`: Computes Euler's totient function for all integers from 1 to a given `n`, involving a large amount of integer arithmetic. This was based on problem 34 of the 99 Problems in OCaml [24].

- `composition`: Constructs a function that is the composition of a list of simple functions and maps it over a list, making heavy use of higher-order functions.

- `funcrec` (functions, records): Compares three forms of parameterisation: using functions defined at the top-level of a program, passing those functions as arguments, and passing those functions as fields of a record argument. This was based on an existing repository of OCaml benchmarks available on GitHub [19].

- `mergesort`: Implements mergesort, making heavy use of lists and pattern matching.

- `nbody`: Simulates the n-body problem, simulating the motion of planets for a number of time steps and making heavy use of floating-point arithmetic. This was adapted from the version in the Computer Language Benchmarks Game repository [20].

4.2 Comparison against alternatives

Although most of the runtime for my compiler is written in WebAssembly, the garbage-collected memory allocator is written in JavaScript due to its complexity and wanting to make several improvements to it. Therefore, the performance of my compiler is indicated with and without garbage collection (labelled as 'OCaml GC' and 'OCaml'), to distinguish the overhead of calling between WebAssembly and JavaScript for each memory allocation. The data given is also for the optimised version of the compiler.

In the following graphs, the output of compiling equivalent C programs to WebAssembly with Clang/LLVM is labelled as 'C', and the output of compiling Grain to WebAssembly is labelled as 'Grain'. Lastly, the JavaScript programs produced by translating the OCaml benchmarks with Js_of_ocaml are labelled as 'JS'. The subscript value on the names of benchmarks indicates the problem size, where multiple instances of the same benchmark were included.

4.2.1 Execution time

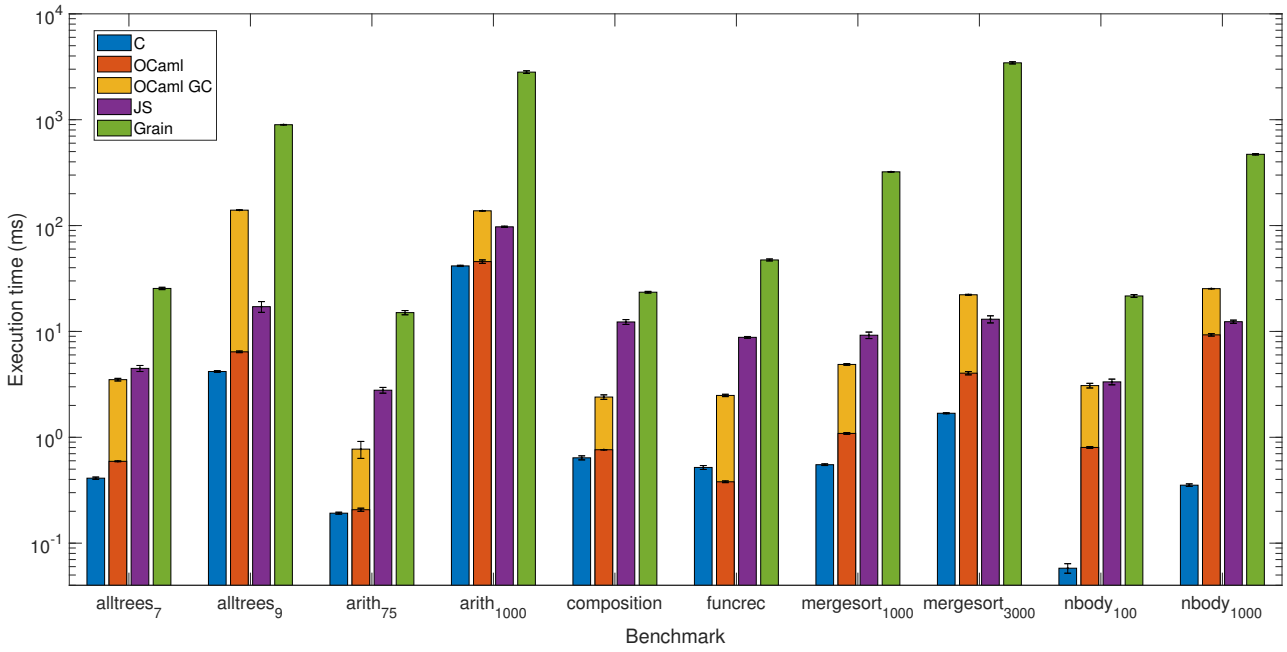

Figure 4.1: Execution times for each alternative (lower is better)

In figure 4.1, we see that the execution speed can vary by a couple orders of magnitude between the least and most efficient method. Unsurprisingly, the programs translated to C by hand and compiled to WebAssembly are the fastest. At the opposite end, Grain is always significantly slower than the other methods. My compiler is always faster than the equivalent JavaScript when run without garbage collection, however the overhead of garbage collection results in the two being more balanced, as shown further in figure 4.2.

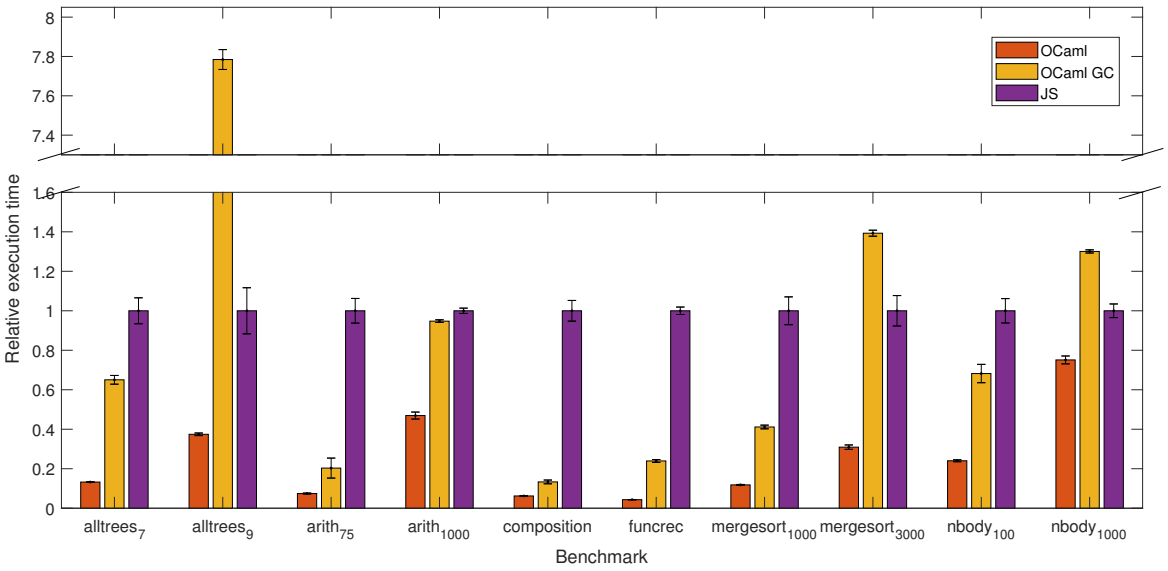

Figure 4.2: Comparison between my compiler and Js_of_ocaml

With garbage collection, execution time grows faster with problem size, so my garbage-collected implementation performs worse on the memory-intensive benchmarks. Considering the garbage-collected runtime is the fairer comparison here, since Js_of_ocaml converts OCaml programs to JavaScript, which then use the garbage collector of the environment the benchmark script is executed in. I ran the tests with Node.js, so this used the garbage collector implemented in Chrome's V8 JavaScript engine.

4.2.2 Heap usage

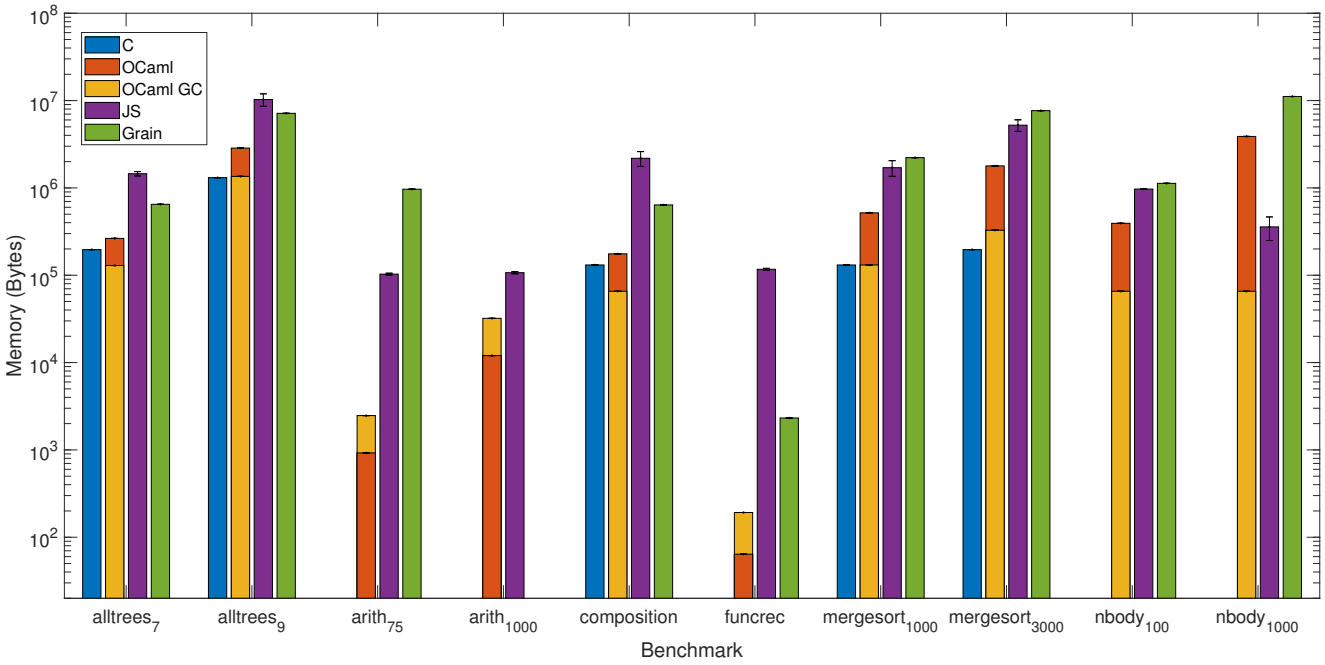

Figure 4.3: Heap usage for each alternative (lower is better)

There are no bars for C for `arith`, `funcrec` and `nbody` in figure 4.3, as these did not require the heap to implement. There is also no data for Grain for `arith_1000` as the program did not appear to terminate with tracing enabled.

For the compiled C programs that did use the heap, my garbage-collected runtime only uses more memory for `mergesort`, where my implementation uses 70% more memory at the larger problem size. This difference is due to C being an imperative language, with manual memory management, where it is natural to implement `mergesort` to perform operations in-place. Although manual memory management allows more precise control over memory, it also adds to the complexity of writing programs. Also, the WebAssembly memory is initialised as 2 pages (128KiB), which is the size recorded for `composition` and `mergesort`$_{1000}$ compiled from C, indicating that these programs may be using less memory than the WebAssembly module is initialised with.

Compared against JavaScript and Grain, my implementation uses significantly less memory than both alternatives. In every test, the garbage-collected runtime uses at most one third of the memory used by either the JavaScript or Grain version.

For my compiler, the garbage-collected runtime uses more memory for `arith` and `funcrec` than the version without garbage collection. These tests have fewer opportunities for

garbage collection so it saves little memory, but the implementation consumes more memory with garbage collection due to the overhead of headers and trailers on allocated blocks. In the worst case, `funcrec` has no opportunities for garbage collection and almost exclusively allocates small objects of 8 bytes. Since each header and trailer is 8 bytes, garbage collection triples the amount of memory used.

4.2.3 Output file size

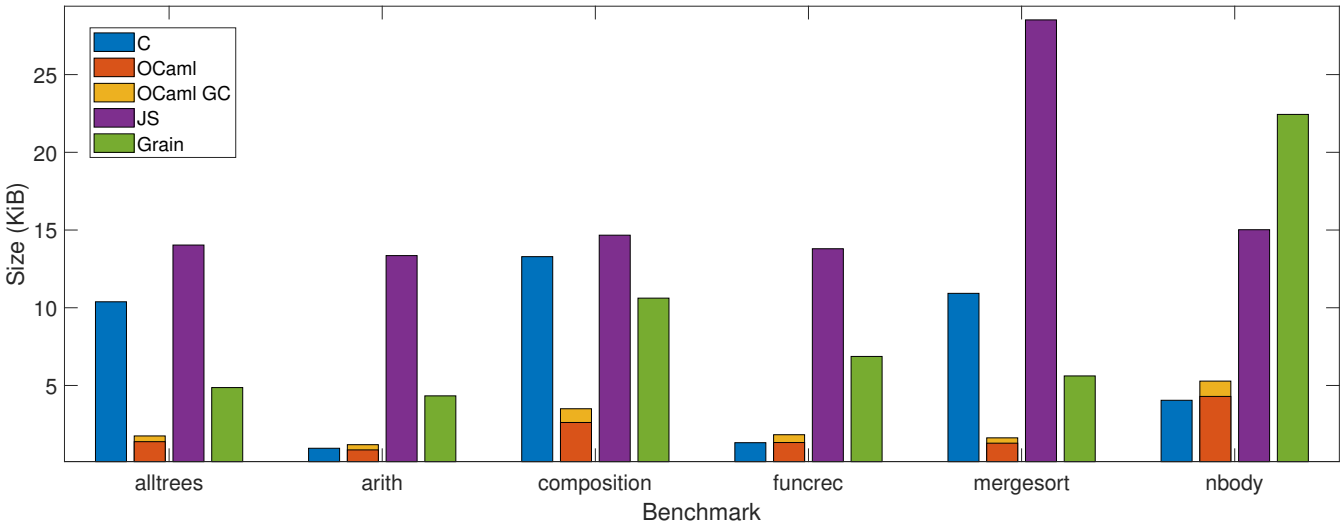

Figure 4.4: Output size for each alternative (lower is better)

As expected, figure 4.4 shows that the JavaScript output tends to be largest, as JavaScript is a text format rather than a binary format like WebAssembly. On average, it is 8 times larger than the output of my compiler with garbage collection enabled. Grain also produces much larger binary files, averaging about 3.5 times larger than my compiler's output. Inspecting the compiled output, there appear to be a few reasons for this. First, programs import a larger set of functions from Grain's runtime than used by my compiler. It also uses a reference-counting garbage collector, which adds more overhead since updating a variable requires both incrementing the new value and decrementing the old value, whereas my garbage collector just overwrites the old value on the shadow stack. Lastly, it does not appear to optimise the WebAssembly produced, compared with my compiler, which uses a register-allocation algorithm to reduce the number of local variables declared, and peephole optimises trivially useless statements such as `local.get i; drop`.

Compared with the output of compiling a C program, the sizes are similar except where parts of `stdlib` have to be included for memory allocation, which makes those programs about 9KiB larger. For comparison, my runtime without garbage collection is a 740B WebAssembly file, but the garbage collector is implemented in JavaScript, which when minified is a 4KiB file.

Figure 4.4 also demonstrates the overhead of garbage collection in terms of the extra bookkeeping operations added to maintain the shadow stack. On average, this adds about 30% to the size of the output WebAssembly.

4.3 Optimisations

I first compare the impact of the IR and WebAssembly optimisations described in section 3.5, and their combined effect on performance. After that, I look at the impact of specific optimisations at the IR level by seeing how performance changes when one is removed, and the benefit of optimising pattern matching. Finally, I show the benefit of repeating the optimisation passes multiple times.

Data is collected with garbage collection disabled to exclude the overhead it introduces, which depends primarily on the garbage-collection implementation used by the runtime system, rather than how programs are optimised.

4.3.1 IR and WebAssembly optimisations

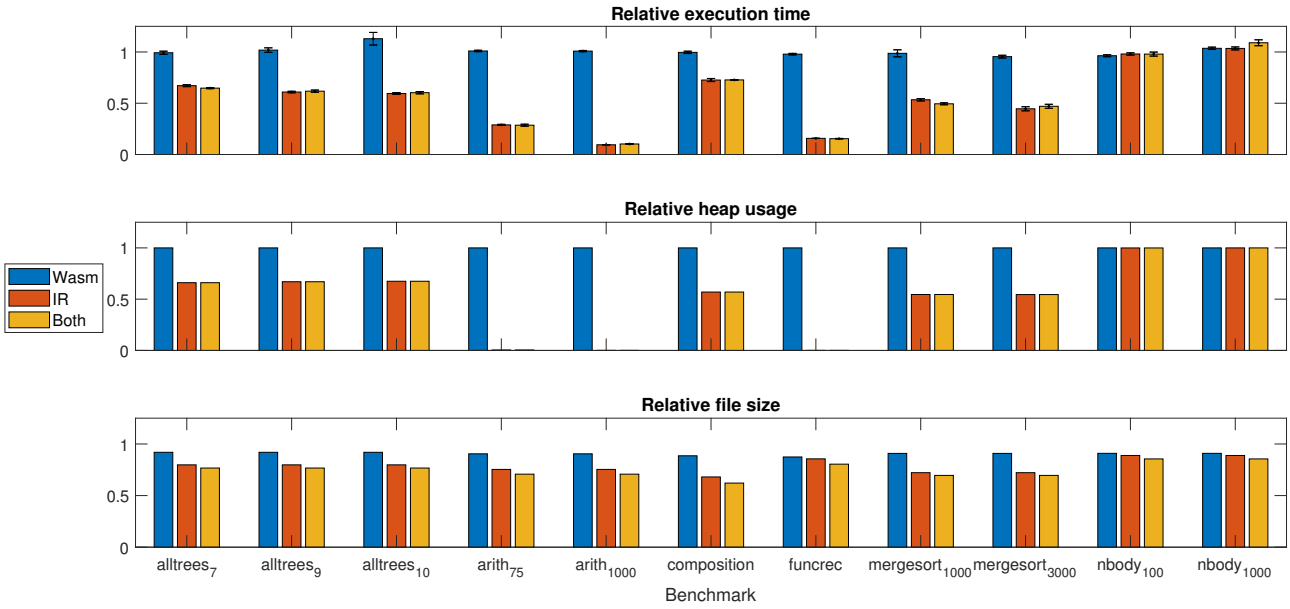

Figure 4.5: Performance with IR and WebAssembly optimisations

The peephole optimisations performed at the WebAssembly level have no impact on memory usage or execution time, but consistently reduce output size by about 10%, as shown in figure 4.5. The IR optimisations reduce execution time and heap usage by at least 30% for all microbenchmarks, except `nbody`. `nbody` is the only program that does not improve in all metrics, most likely because it is an imperative-style program and the optimisations implemented are unable to optimise uses of mutable variables. For `arith` and `funcrec`, inlining or rewriting functions has completely removed the need to construct closures recursively, resulting in near zero heap usage.

4.3.2 Impact of inlining, uncurrying and tail calls

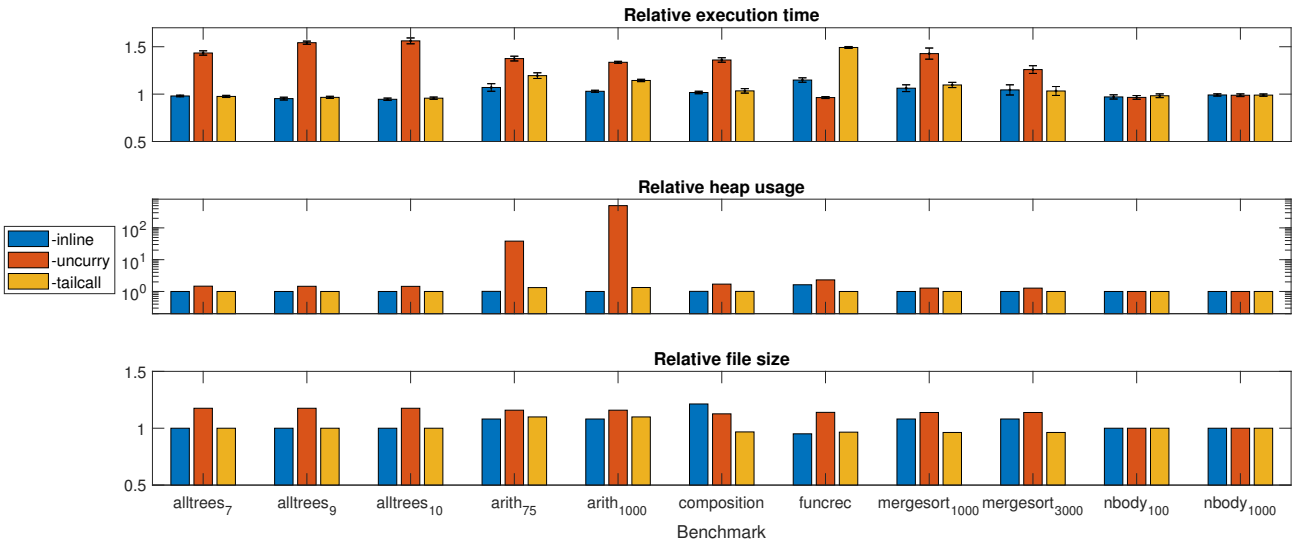

Figure 4.6: Impact of individual optimisations

Figure 4.6 shows the change when one optimisation is removed, so the magnitude of each bar can be viewed as the speed-up or size reduction an optimisation has, on top of the optimisations already present. First, inlining only increases file size in one case, `funcrec`, and only by 5%, so the optimisation does not cause significant code bloat. Instead, inlining reduces the size of the output in cases where it completely removes a function definition. Overall, it has a relatively small impact on performance, with the biggest change being a 15% speed-up for `funcrec`. This suggests that the heuristics used for inlining may be too conservative. However, these are all relatively small benchmark programs, which could also limit how significant a change can be achieved by inlining.

Tail-call optimisation does not affect most of the programs, but improves the execution speed of `funcrec` and the speed and file size for `arith`. A more important factor, not shown by figure 4.6, is that tail-call optimisation allows some programs to execute that would otherwise give an error, for example:
`let rec f x = if x = 0 then 0 else f (x-1)`
Despite being a very simple function, without tail-call optimisation, `f(30000)` exceeds the maximum call-stack size and the program fails. With tail-call optimisation, the function no longer makes recursive calls so can handle any input size.

Lastly, uncurrying fully applied functions has the most significant impact on performance. None of the benchmarks are negatively impacted by it and most speed up, by up to 50%. Additionally, not having to construct a closure for each separate argument reduces the amount of space used on the heap, and the number of functions defined in the code. In the case of `arith`, this removes the need to recursively construct any closures, so heap usage no longer increases with problem size. Once again, `nbody` is not improved, as it lacks opportunities for this optimisation to be applied.

4.3.3 Pattern matching

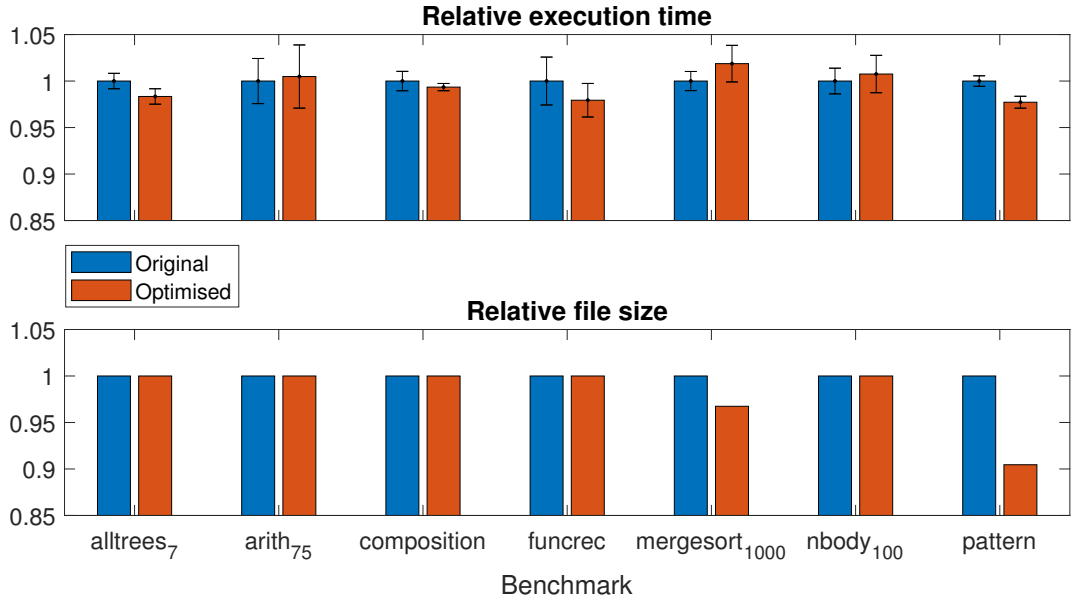

Figure 4.7: Impact of optimised pattern matching

For evaluating the optimisations to pattern matching, I consider an additional microbenchmark, `pattern`, which repeatedly calls each case of the function below, and benefits from the context information described in section 3.2.1.

```
type lst = Nil | One of int | Cons of int * lst

let f l1 l2 = match (l1, l2) with
  | Nil, _ -> 0
  | _, Nil -> 1
  | ((One _, _)|(_, One _)) -> 2
  | Cons _, Cons _ -> 3
```

As figure 4.7 shows, the optimisations have little or no impact on benchmarks with only simple patterns, but reduce output size for `mergesort` and `pattern`, which both involve more complex pattern matching. Execution time is also reduced slightly for `pattern`. This shows that the optimisations provide a slight benefit for complex pattern matching, without having a negative impact in simpler cases.

4.3.4 Impact of number of iterations

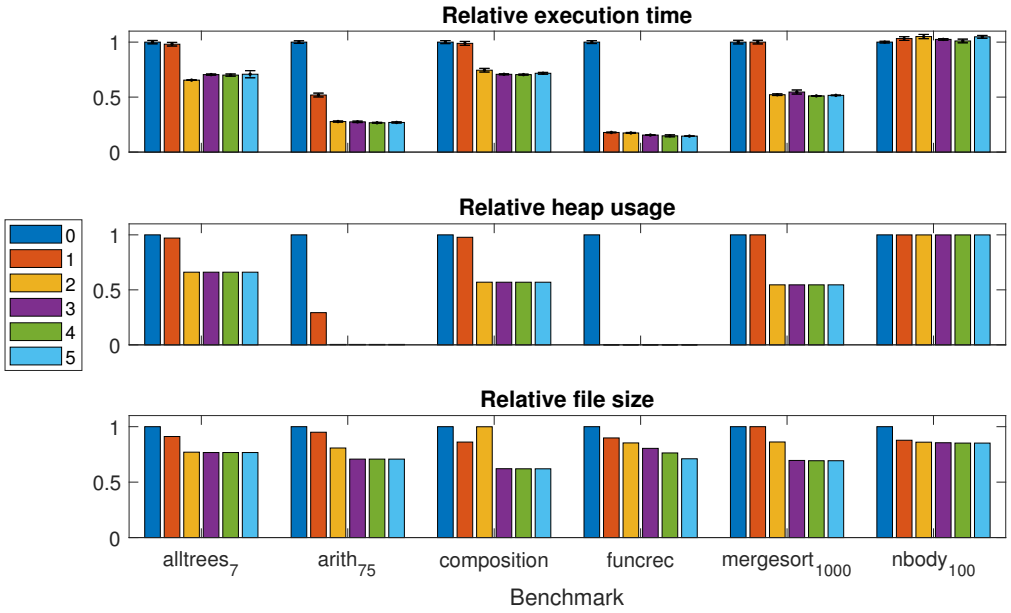

Figure 4.8: Effect of repeating optimisation passes

Figure 4.8 demonstrates that, in almost all cases, there is no further improvement after each optimisation pass has been performed three times, so this was chosen as the default number of iterations for all other tests. This ensures that every optimisation happens both before and after every other one, so the effects of phase ordering are reduced. Multiple iterations can also benefit one optimisation in isolation e.g. `let y = x in let z = y in f(z)`. The first pass of propagating variables would replace `f(z)` with `f(y)`, but another pass is needed to replace this with `f(x)`, allowing both `y` and `z` to be removed.

4.4 Garbage collection

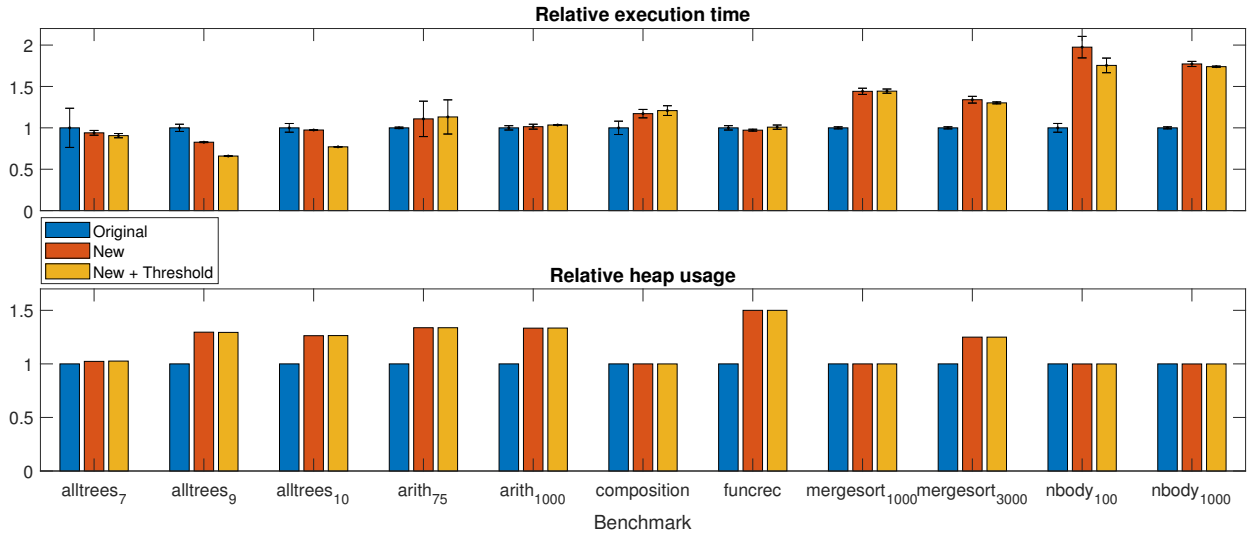

Figure 4.9: Effect of changes to garbage collection

Figure 4.9 shows that, due to adding a trailer to every memory allocation (described in section 3.6.3), the modifications to garbage collection increase both execution time and memory usage for most of the microbenchmarks, since these are small programs that generally only require simple memory management.

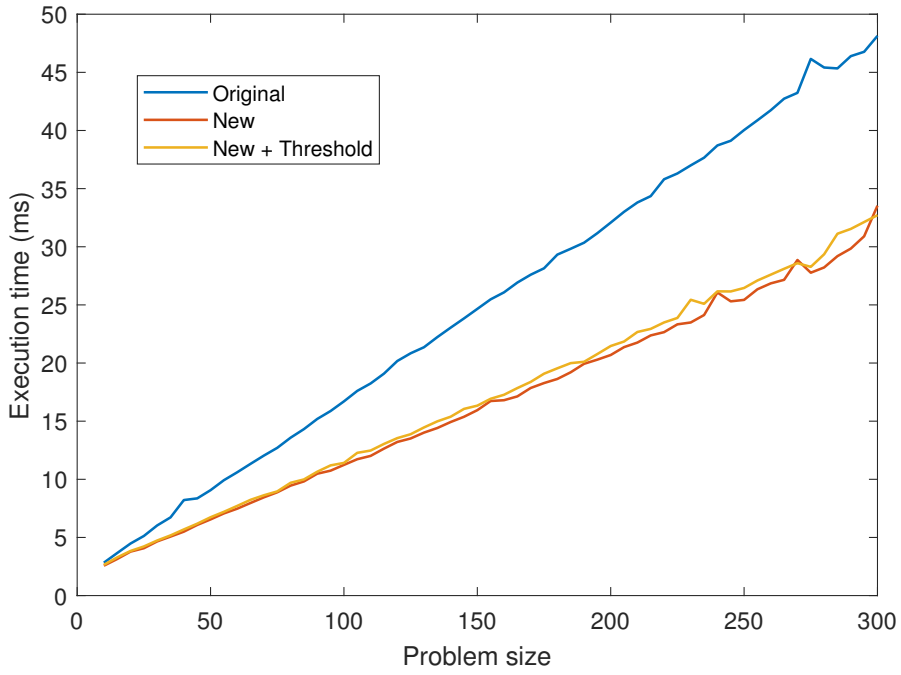

Figure 4.10: Performance on fragmented memory

Figure 4.10 demonstrates the benefit of these changes for a program allocating data after memory has been fragmented. The function below interleaves allocations to three different lists, one of which contains `Cons2` cells which are larger than `Cons1` cells. After the `shortFreedList` and `longFreedList` are discarded and garbage collected, the heap has lots of fragmented free blocks, with every tenth block being large enough for a `Cons2` cell.

```
type list = Nil | Cons1 of int * list | Cons2 of int * int * list

let longLivedList = ref Nil
let shortFreedList = ref Nil
let longFreedList = ref Nil

let rec buildLists = function
  | 0 -> ()
  | n ->
   (if (n mod 20) = 0
      then longFreedList := Cons2(n, n, !longFreedList)
    else if (n mod 2) = 0
      then shortFreedList := Cons1(n, !shortFreedList)
    else
      longLivedList := Cons1(n, !longLivedList));
    buildLists (n-1)
```

The problem size in figure 4.10 refers to how many times those larger blocks are reallocated. The original implementation scans the free list so considers nine small blocks for every one larger block it can allocate, and scans over all of the smaller free blocks before calling garbage collection. Binning free blocks by size avoids this, and the modified version only considers blocks that are large enough to allocate, running 50% faster.

Memory is initially one large free block, and fragmentation is only an issue once this large block is nearly all allocated, otherwise allocations continue to be taken from the large block rather than scanning the free list. The modified version adds a trailer to each block, so fragmentation is often an issue at different points for each version, since they consume a page of WebAssembly memory after a different number of allocations. Therefore, it was necessary to construct this artificial example of fragmentation, demonstrating the effect it has for both versions.

4.4.1 Threshold to not grow memory

Figure 4.11 demonstrates the benefit of having a threshold for garbage collection, increasing memory whenever garbage collection fails to free more than 1KiB of memory, not just when it fails to free a block of the required size. As problem size grows, `alltrees` gets more memory intensive and the modified version begins to narrowly outperform the original. However, the improvement with the threshold is much more significant, surpassing 40% at the largest problem size.

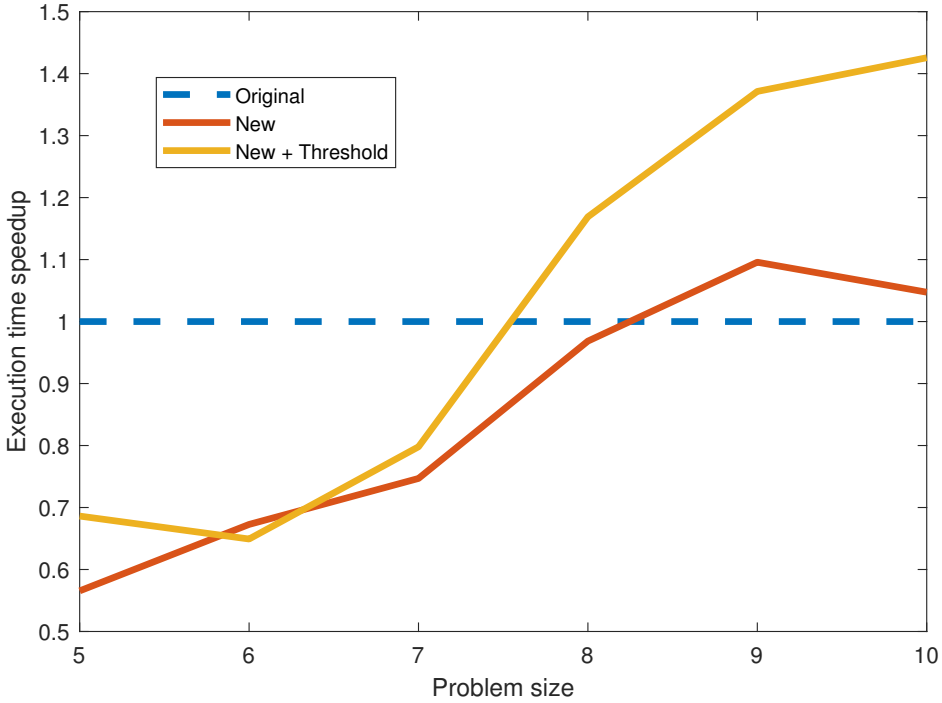

Figure 4.11: Performance on `alltrees`

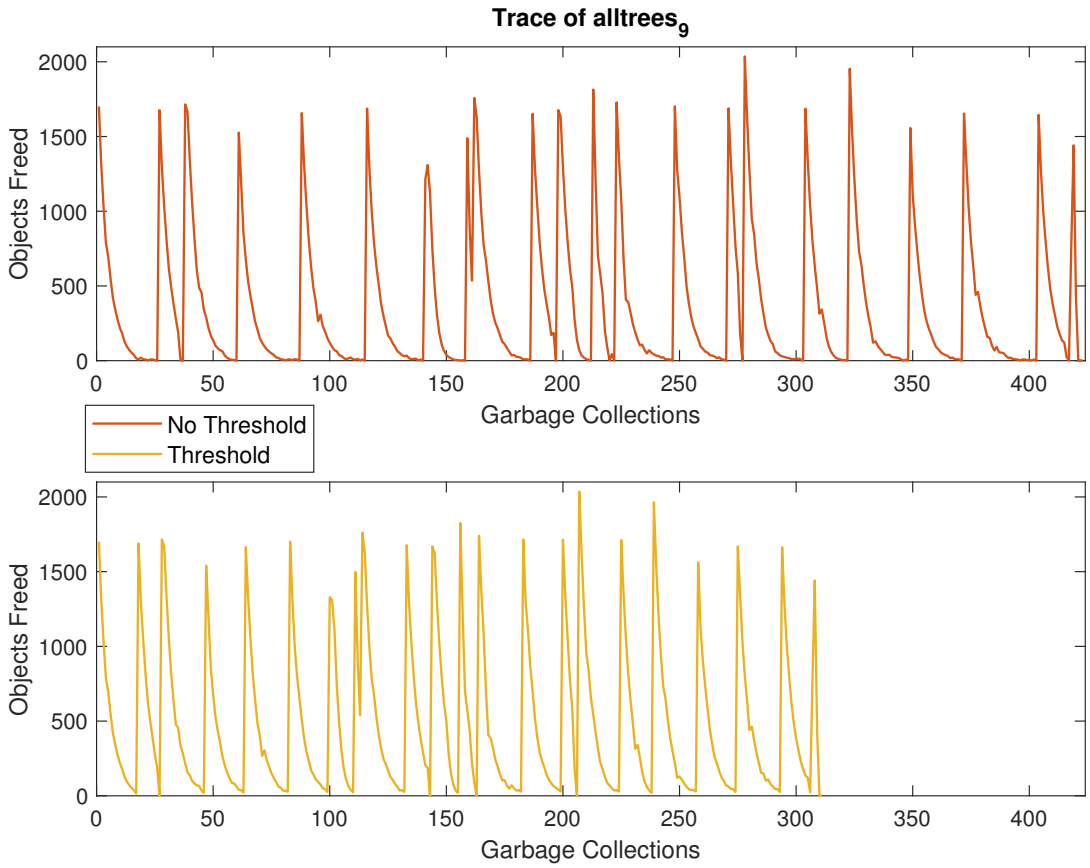

Figure 4.12: Objects freed during execution of `alltrees`

Figure 4.12 reveals why this is the case, by examining a trace of the number of objects freed each time the garbage collector runs, and the total number of garbage collections performed before the program terminates. There is a trend in the number of objects being freed to decrease over time, as the number of long-lived objects grows, until eventually memory has to grow and a large number of allocations occur before the next garbage collection, hence the spikes in the traces.

We see an identical pattern both with and without the threshold, but this pattern is compressed when the threshold is used, in which case the program performs about 25% fewer garbage collections in total. Rather than having tails of inefficient garbage collections, the threshold results in many objects all begin freed in one pass the next time garbage collection runs, after the newly-requested memory is used up many allocations later. This results in the garbage collector being invoked fewer times in total, without requesting memory in cases where it would otherwise not be necessary. The value of 1KiB was selected based on the amount freed by these tails of inefficient collections.

4.5 Summary

I have shown how my compiler compares to existing methods of running code on the web, outperforming Grain and Js_of_ocaml in many instances. I have also demonstrated the cases where my optimisations improve performance, and given possible explanations for the behaviour observed.

Chapter 5

Conclusions

This project met its success criteria, supporting the core OCaml language with integer, boolean, comparison and list operations, as well as all OCaml's control-flow and pattern-matching constructs, besides exceptions.

I also implemented extensions in each of the directions identified at the start of the project. The subset of OCaml supported was extended with references and basic floating-point operations, and the compiler supports using .mli interface files to hide parts of programs in the compiled WebAssembly. Several optimisations were implemented, including function inlining, uncurrying and tail-call optimisation, as well as clean-up passes on the intermediate representation and WebAssembly. I also extended the runtime system with a mark-and-sweep garbage collector, using a shadow stack due to the WebAssembly stack being implicit.

With these optimisations, my compiler consistently produces smaller output files than the Js_of_ocaml tool, and the compiled programs consume at most one third the amount of memory. This demonstrates the benefit of compiling a strongly typed language, such as OCaml, to WebAssembly, avoiding the inefficiencies of JavaScript. Execution time was also reduced, but only for benchmarks that did not make heavy use of the heap.

5.1 Personal reflection

This project allowed me to learn more about how complex language features can be implemented, such as pattern matching and mutually recursive functions, which were not discussed in the Part IB Compilers course. It has also reinforced the importance of careful planning and preparation for substantial software projects, which allowed me to complete the core compiler relatively quickly and explore a range of interesting extensions.

I was able to work much faster during the breaks than in term time, due to not having units of assessment and lectures to balance with the project. My initial plan accounted for the unit of assessment exam taking priority the week it was set, but could have allocated more work during the breaks. As a result, I implemented benchmarks and data collection during the Christmas break, a couple weeks ahead of schedule, and mostly focused on extensions during Lent term.

One challenge I faced was efficiently debugging my garbage-collection implementation.

Errors, such as incorrectly freeing an object, may only affect a program's output long after the error occurs, when that block of memory is reallocated and modified. To solve this, I eventually wrote several JavaScript functions to aid debugging, such as checking the structure of the free list, as well as logging data about pointers used during a program's execution and studying the resulting trace. Had I taken a more systematic view to debugging from the beginning, writing such utility functions in advance, I might have been able to debug some errors faster.

5.2 Future work

First, there are several features of OCaml not yet supported by my compiler. Implementing more of the standard library would allow supporting operations on additional types, such as 32 and 64-bit integers and strings. Similarly, although the basic array syntax is supported, not having the Array library implemented in WebAssembly severely limits the practical uses of arrays. This would likely require supporting the module layer of OCaml, which then creates the possibility of compiling multiple interacting OCaml programs.

Also, there are still many aspects where performance could be improved. My evaluation revealed that the optimisations implemented do not significantly affect imperative code, which could be improved by flow-directed analysis capable of inferring properties about mutable variables stored as references. One optimisation made in the OCaml compiler, which I did not get round to implementing, is identifying references being used as mutable local variables and not accessed outside of a function. These can be replaced with local variables of the function, rather than being stored on the heap. Also, control-flow analysis would more precisely identify where functions are used, enabling other optimisations, rather than relying on copy propagation to replace all indirect uses of a function variable.

Finally, the garbage collector is implemented in JavaScript rather than WebAssembly. I suspect that translating this to WebAssembly, avoiding the switch between WebAssembly and JavaScript for each memory allocation, would significantly reduce the overhead of garbage collection. More complex garbage-collection techniques could also be implemented, such as a generational collector, which identifies objects as short-lived or long-lived, and collects long-lived objects less frequently since they are more likely to still be in use.

Bibliography

[1] Js_of_ocaml. URL: https://ocsigen.org/js_of_ocaml (visited on 2021-03-20).

[2] Steve Akinyemi. A list of languages that compile to or have their VMs in WebAssembly. URL: https://github.com/appcypher/awesome-wasm-langs (visited on 2020-10-18).

[3] Grain. URL: https://grain-lang.org (visited on 2021-03-24).

[4] Richard Anaya. Write Web Assembly With LLVM. URL: https://richardanaya.medium.com/write-web-assembly-with-llvm-fbee788b2817 (visited on 2021-04-10).

[5] The OCaml Standard library. URL: https://ocaml.org/api/Stdlib.html (visited on 2021-03-27).

[6] Timothy Griffin. Compiler Construction. URL: https://www.cl.cam.ac.uk/teaching/1920/CompConstr (visited on 2021-03-24).

[7] Timothy Jones. Optimising Compilers. URL: https://www.cl.cam.ac.uk/teaching/2021/OptComp (visited on 2021-03-24).

[8] A. Bloss, P. Hudak, and J. Young. "An Optimising Compiler for a Modern Functional Language". In: *The Computer Journal* 32.2 (Jan. 1989), pp. 152–161. ISSN: 0010-4620. DOI: 10.1093/comjnl/32.2.152. eprint: https://academic.oup.com/comjnl/article-pdf/32/2/152/1445715/320152.pdf. URL: https://doi.org/10.1093/comjnl/32.2.152.

[9] Fabrice Le Fessant and Luc Maranget. "Optimizing Pattern Matching". In: *SIGPLAN Not.* 36.10 (Oct. 2001), 26–37. ISSN: 0362-1340. DOI: 10.1145/507669.507641.

[10] Luc Maranget. "Compiling Pattern Matching to Good Decision Trees". In: *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*. ML '08. Victoria, BC, Canada: Association for Computing Machinery, 2008, 35–46. ISBN: 9781605580623. DOI: 10.1145/1411304.1411311.

[11] Andreas Rossberg. WebAssembly Specification. URL: https://webassembly.github.io/spec/core/ (visited on 2020-10-18).

[12] Lin Clark. Creating a WebAssembly module instance with JavaScript. URL: https://hacks.mozilla.org/2017/07/creating-a-webassembly-module-instance-with-javascript/ (visited on 2021-01-23).

[13] C++ exceptions support. URL: https://emscripten.org/docs/porting/exceptions.html (visited on 2021-04-10).

[14] Nathan Burow, Xinping Zhang, and Mathias Payer. "SoK: Shining Light on Shadow Stacks". In: *2019 IEEE Symposium on Security and Privacy (SP)* (2019). DOI: 10.1109/sp.2019.00076. URL: http://dx.doi.org/10.1109/SP.2019.00076.

[15] David J Anderson. *Kanban : successful evolutionary change in your technology business*. Blue Hole Press, 2010.

[16] Jira Software. URL: https://www.atlassian.com/software/jira (visited on 2021-04-10).
[17] OUnit - unit testing framework for OCaml. URL: https://github.com/gildor478/ounit (visited on 2020-11-07).
[18] The core OCaml system: compilers, runtime system, base libraries. URL: https://github.com/ocaml/ocaml (visited on 2021-03-24).
[19] Christophe Troestler. ocaml-benchmark. URL: https://github.com/Chris00/ocaml-benchmark (visited on 2021-02-15).
[20] The Computer Language Benchmarks Game. URL: https://salsa.debian.org/benchmarksgame-team/benchmarksgame/ (visited on 2021-02-15).
[21] Dan Bonachea. Method Inlining in the Titanium Compiler. Sept. 2001. URL: http://titanium.cs.berkeley.edu/papers/bonachea-method-inlining.pdf (visited on 2021-01-09).
[22] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. 2nd. Prentice Hall Professional Technical Reference, 1988. ISBN: 0131103709.
[23] Ilya Grigorik. High Resolution Time. URL: https://www.w3.org/TR/hr-time-3/ (visited on 2021-02-15).
[24] 99 Problems in OCaml. URL: https://ocaml.org/learn/tutorials/99problems.html (visited on 2020-11-07).

2361C

Computer Science Tripos Part II Project Proposal

Optimising Compiler from OCaml to WebAssembly

October 21, 2020

Project Originator: Dr Tim Jones

Project Supervisor: Dr Tim Jones

Signature:

Director of Studies: Dr John Fawcett

Signature:

Overseers: Dr Sean Holden and Dr Neel Krishnaswami

Signatures:

Introduction and Description of the Work

JavaScript is a high-level language that has become a core part of the World Wide Web, providing interactive web pages through the use of JavaScript engines on all major web browsers. A result of this is that there are compilers from many languages to JavaScript, allowing developers to write code for the web in their preferred language.
JavaScript is dynamically typed so is not a convenient target for statically typed languages such as OCaml or C. These languages perform compile-time type checking so additional checks at runtime are often unnecessary and inefficient.

WebAssembly is a low-level language with a binary instruction format supported by major browsers such as Chrome, Firefox and Safari. Having a binary format reduces code size and improves load times compared to human-readable JavaScript files. WebAssembly is also based on a stack machine and its low-level design makes it a more efficient compilation target for statically typed languages. Compilers to WebAssembly therefore allow programmers to use a language of their choice and still have code run on the web without having to sacrifice performance.

This project will implement a compiler to WebAssembly for the statically typed functional language OCaml. Several optimisations will be implemented and evaluated in terms of their effect on output code size, execution time and memory usage. These results will be compared with the performance of similar existing solutions, such as compilers from OCaml to JavaScript or from other functional languages to WebAssembly.

Starting Point

My experience with OCaml is limited to the IB Compilers course and studying the OCaml compiler, looking at the data structures used and the ASTs generated for some short programs. I have read parts of the WebAssembly specification to understand the range of instructions available and compiled a short C program to WebAssembly using Emscripten.

Many languages can now be compiled to WebAssembly, although lots of tools are still works in progress. WebAssembly is supported as a backend of LLVM, which is the primary way to compile C/C++ to WebAssembly. There is also Asterius for compiling Haskell, and a new functional language, Grain, which compiles to WebAssembly. Another approach to running OCaml on the web is Js_of_ocaml, a compiler to JavaScript. These tools should provide an interesting comparison for the performance of my compiler and studying them will help to inform decisions about the intermediate representation I use.

The only example I could find of a compiler from OCaml to WebAssembly was another Part II project from 2020, which worked from the parsed AST produced by the OCaml compiler. By instead working from the typed AST of the compiler, I should be able to include a greater subset of OCaml.

Substance and Structure of the Project

I will use the parser and type-checker of the official OCaml compiler as there is little benefit in reproducing these parts. Only a subset of OCaml will be implemented to allow focusing on implementing optimisations later in the project. I will exclude the class and module language features, as they are a significant extension to the core functional language, and exceptions due to the poor support for them in WebAssembly. Of OCaml's many standard library operations, I will initially only implement the comparison, integer arithmetic, boolean and list operators, which are enough to be able to compile a diverse set of test programs.

I will design an intermediate representation and implement translation from the OCaml compiler's typed AST to this representation. This will require studying both the Lambda intermediate language used by the OCaml compiler as well as the intermediate representations used by other compilers to WebAssembly, in order to produce a data structure that is suitable both for performing optimisations on and for translating between the typed AST and WebAssembly.

The last essential part of the compiler is the back end. This will consist of WebAssembly code generation from the intermediate representation as well as a runtime system. The runtime system will provide functions for memory allocation, creating closures, and OCaml primitives such as comparisons. This will probably be written in WebAssembly's text format, although parts that turn out to be more complex than expected can be written in C and compiled to WebAssembly instead.

Garbage collection will be ignored initially as it is not needed for testing small programs. WebAssembly lacks reference types and the stack is managed implicitly rather than being part of a program's linear memory, so a solution must maintain a shadow stack of any references in use. This can significantly increase code size and memory usage for programs not requiring garbage collection, so it will be left as an extension task.

After building a working compiler, I will add optimisations to both the intermediate code and WebAssembly generation stages. The start of my project will involve researching compiler optimisations to implement in more detail but possible analysis techniques and optimisations include:

- Live variable analysis
- Dead code and common subexpression elimination
- Peephole optimisations
- Constant propagation and folding
- Inline expansion
- Tail call optimisation

Performance Evaluation

My project will include a set of benchmark programs, testing both the validity and performance of a range of the implemented language features. As well as the OCaml programs, I will need to create a testing framework to execute the produced WebAssembly in a browser, requiring a small amount of HTML and JavaScript. The testing framework will also record the code size, execution time and memory usage of each of the programs.

I will collect this data with different sets of optimisations enabled to determine how each optimisation affects these metrics and how multiple optimisation passes interact with each other. This data will be compared with the performance of alternative solutions to running functional code in browsers. One alternative is the JavaScript produced by the Js_of_ocaml tool, another is to write the programs in Grain.

Success Criteria

The following should be achieved:

- Implement a compiler for the specified subset of OCaml to WebAssembly.
- Produce a set of test programs to verify that the output of the WebAssembly code matches the original OCaml.
- Implement a testing framework to run JavaScript or WebAssembly code and measure its execution speed, memory usage and code size.
- Compare the performance of my compiler with an existing solution to running functional code in browsers.

Possible Extensions

- Many optimisations, for example function inlining, make a trade-off between execution speed and code size or other pairs of metrics. It would be interesting to research heuristics for deciding when to make these optimisations to improve the balance between each of the metrics. Similarly, I could research and implement additional optimisations to target specific problems.
- As explained earlier, there are several challenges with implementing garbage collection in WebAssembly. Researching existing solutions in other compilers for WebAssembly would help me to build my own implementation. This would allow running longer and more complex test programs.

- The subset of OCaml supported could be extended. I only intend to implement a small subset of the primitive operations initially so extending this would allow for a greater variety of test programs. Classes and objects would also allow testing more object-oriented style programs, and module support would create the possibility of compiling multiple interacting files.

Timetable and Milestones

26^{th} Oct - 8^{th} Nov

Set up the tools needed to work on the project. This includes creating a repository on GitHub to backup my code and dissertation, and creating a HTML/JavaScript framework to run WebAssembly code, which I will test with some programs compiled from C using Emscripten.
I will also research potential optimisations and heuristics to implement, prioritising them by their expected complexity to implement and impact on the compiler.
Milestone: Be able to run WebAssembly programs. Have an ordering for optimisations to implement later in the project.

9^{th} Nov - 22^{nd} Nov

Study the intermediate data structures used by the OCaml compiler when compiling sample programs with different language features. These programs will be helpful later to check that output WebAssembly correctly handles a range of language features. Also research the intermediate representation used by other compilers to WebAssembly, such as Asterius for compiling Haskell and the standard compiler for Grain.
Use this information to design an intermediate representation for translating the OCaml compiler's typed AST.
Milestone: Produce an intermediate representation datatype.

23^{rd} Nov - 6^{th} Dec

Implement translation to the intermediate representation. This work package is lighter due to also having a unit of assessment exam to complete during this fortnight.
Milestone: Be able to translate the sample programs to my intermediate representation.

7th Dec - 20th Dec

Implement WebAssembly code generation from the intermediate representation. This will require a runtime system to provide operations such as memory allocation and some of the OCaml primitives such as testing equality.
Milestone: Compile the sample OCaml programs and run them as WebAssembly.

21st Dec - 3rd Jan

I will take time off from the project for Christmas.

4th Jan - 17th Jan

Create a set of benchmark programs to test the compiler's performance, using existing test suites if they appear suitable. Build a testing framework to compile these benchmarks and collect data on the output code size, execution time and memory usage. The testing framework must also be able to evaluate WebAssembly produced by other means such as compiling Grain, and equivalent JavaScript programs.
Translate the benchmarks into Grain and compile them to WebAssembly. Repeat data collection for this code, and again for the JavaScript output by the Js_of_ocaml tool.
Milestone: Collect and plot data for both my compiler and the alternatives.

18th Jan - 31st Jan

Collect data for any remaining benchmark programs. At this point I will have met my success criteria and will begin implementing optimisations based on their prioritisation at the start of the project. Each optimisation will require collecting additional data to show its impact on the compiler.
Milestone: Repeat data collection with the optimisations enabled. Success criteria met.

1st Feb - 14th Feb

Prepare the progress report and presentation. Once these are ready, I will continue to implement optimisations. At this point I should have data on the impact of several optimisations so may want to look at adding heuristics to balance execution speed, code size and memory usage.
Milestone: Submit the progress report. Rehearse the presentation and have delivered it depending on the date assigned for presentations.

15th Feb - 28th Feb

Research how existing compilers from garbage collected languages to WebAssembly perform garbage collection. If this looks feasible in the time remaining, begin implementing a garbage collector, otherwise focus on the other extensions.
Milestone: Decide whether to implement garbage collection and, if so, which garbage collection technique to use.

1st March - 14th March

Continue implementing extensions. Improving the optimisations implemented, the subset of OCaml covered, or adding garbage collection should all allow me to collect some new data demonstrating the area extended.
Milestone: Collect data for either new test programs, or showing the impact of the optimisations added.

15th Match - 28th March

Write the introduction and preparation chapters of the dissertation. Finish making additions to the compiler.
Milestone: Submit draft chapters to my DoS and supervisor.

29th Match - 11th April

Write the implementation and evaluation chapters. Make changes to the previous chapters based on the feedback received.
Milestone: Submit the next two draft chapters to my DoS and supervisor.

12th April - 25th April

Write the conclusion of the dissertation and ensure that the bibliography and any appendices are complete. Continue making changes based on the feedback for previous chapters.
Milestone: Submit draft of whole dissertation to my DoS and supervisor.

26th April - 9th May

Make any final checks and modifications to my dissertation.
Milestone: Submit dissertation.

Resources Required

I intend to use my own Windows laptop (8GB RAM, 2.5GHz CPU, 1TB HDD, 256GB SSD) to work on the project. I accept full responsibility for this machine, and I have made contingency plans to protect myself against hardware and/or software failure. My code and dissertation will be backed up regularly to GitHub and OneDrive. In the event of my laptop being damaged or lost, I will use the MCS machines as a backup.
I will be using the WebAssembly Binary Toolkit to manipulate text and binary WebAssembly code, and opam (the OCaml package manager) to run building and testing packages and the Js_of_ocaml tool. I have checked that I can build and use both on the MCS machines, which already have IDEs and browsers installed. My project will contain parts of the front-end of the OCaml compiler, which is available on GitHub.