1 Conclusions

Word budget: ~500–600 words

Likely short, may well refer back to the introduction. Reflection on lessons learnt, anything I'd have done differently if starting again with what I know now.
First paragraph should reiterate what the project was about.
Summarise how my evaluation answered the questions this project was asking
Can briefly outline any ideas for further work

In this chapter I summarise what was achieved in the project and reflect on the lessons learnt. I also offer suggestions of how the work may be taken forwards, if I were to continue the project.

1.1 Project Summary

The project was a success; I met all my success criteria and an extension. I completed an entire compiler pipeline that transforms a subset of the C language into a WebAssembly binary; the front end, which parses the source code into an abstract syntax tree (AST); the middle end, which transforms the AST into my custom intermediate representation (IR); and the back end, which generates a WebAssembly module from the IR. I provided a Node.js runtime environment that runs the WebAssembly binary and interfaces to the system and standard library.

Additionally, I implemented several optimisations to the program in the middle and back ends of the compiler. Unreachable procedure elimination reduced the size of generated binaries by pruning unused functions from the call graph. I used tail-call optimisation to drastically reduce the amount of memory used by recursive functions, to the extent of being able to run programs when GCC couldn't. I created a more optimal stack allocation policy, experimentally tweaking my heuristic to produce significant improvements to programs' memory use.

My compiler produced correct binaries that maintained the semantics of the source program, as demonstrated by comparing my test programs against the programs compiled with GCC. My evaluation showed that my optimisations were successful in improving the performance of programs. Tail-call optimisation reduced $\mathcal{O}(n)$ memory use to $\mathcal{O}(1)$ for recursive functions, and my stack allocation policy significantly decreased memory usage.

I gained experience using Rust, including learning idiomatic ways of structuring my code and how to work with the borrow checker to produce memory-safe code. Throughout the project, I had opportunities to apply theory from the Tripos; ranging from Algorithms course Software and Secu-

rity Engineering, as well as putting into practice the recent II Optimising Compilers and Advanced Computer Architecture courses. This allowed my to consolidate and expand my skill set.

The project progressed in line with the timetable set out in the proposal; a lot of the time, I was slightly ahead of schedule, allowing me to be thorough in my testing. This left me enough time to successfully implement the stack allocation policy optimisation as an extension.

1.2 Further Work

The most obvious continuation of this project would be to expand the scope to support a larger subset of C. For example, I could add support for function pointers or linking.

I only implemented a small subset of the standard library. If I were taking this project further, I would implement more of the standard library, for example all the string manipulation functions. This would provide support for many more source programs. Notably I would also implement `malloc()` and `free()`, to provide support for heap memory allocation.

Furthermore, additional optimisations could be added to the compiler, increasing performance of compiled programs in terms of memory usage, execution speed, and binary code size. These can be as sophisticated as desired; there are many analyses that can be done at compile time, including dataflow analysis, constraint-bases analysis, effect systems, and so on.