Martin Walls
mrw64@cam.ac.uk

Progress Report
Part II Project

Project title: C to WebAssembly Compiler

Supervisor: Timothy M. Jones

Director of Studies: Dr John K. Fawcett

Overseers: Sean Holden, Neel Krishnaswami

My project is on schedule; I have finished implementing the compiler pipeline.

I wrote the inputs to a parser generator to transform the input source code into an abstract syntax tree. This took some careful attention to make sure the grammar was context-free and unambiguous, and correctly parsed the C language. I had to write a custom lexer to handle `typedef` definitions, because this introduces context-sensitivity into the grammar (there is ambiguity between typedef names and identifiers).

I implemented the module to transform the AST into a three-address code intermediate representation. I had to deal with some complexity for some of the AST nodes, for example switch statements took a little while to get right because there are lots of different possible structure variations, all of which had to be accounted for.

I successfully implemented the Relooper algorithm, which turns unstructured control flow into structured control flow. The main challenge of this stage was to figure out the intricacies of the algorithm. The description of the algorithm in the Emscripten paper left out some of the finer details of actual implementation, but I was able to implement a working solution with careful thought.

I implemented the target code generation module, to take the output from the Relooper algorithm and generate a WebAssembly binary. This included managing the frame and stack pointers, as well as defining caller/callee conventions, and writing functions to push and pop a stack frame to the stack, with previous frame pointer, return value, and parameters in the correct locations.

I have implemented a NodeJS runtime environment to allow me to run the generated WebAssembly files. This includes instantiating the WebAssembly file and passing it the necessary function and memory imports. It instantiates the WebAssembly memory and stores the program arguments in it, following a memory layout convention I established. I have implemented some of the C standard library in JavaScript, for example `printf` because this allows programs to interact with the NodeJS console. These functions are imported into the WebAssembly module to allow them to be called.

I have written a test framework in Python to allow me to test the correctness of my compiler easily. For each C source file I specify, the test script compiles it with my compiler and also with GCC, and checks that the outputs and exit codes match.

I have implemented tail-call optimisation and unreachable procedure elimination. For tail-call optimisation, I found all recursive calls in a function, and replaced them with code that sets the parameter variables to the new values and loops back to the entry point. This is effective at preventing out-of-memory and stack overflow errors. For unreachable procedure elimination, I first generated the call graph for the program. Then I walked over the graph, marked all functions we can reach, then deleted all unmarked functions.