Introduction and Description of the Work

This project involves the implementation of a compiler from OCaml to C in such a way that will allow the resulting executable to be compatible with DWARF-based debugging tools, such as `gdb`. This project will use the existing front-end of the OCaml compiler, but implement a very different back-end which outputs valid C code that can be compiled using a tool such as `gcc` into a debuggable executable. The focus of the compiler is to optimise for debuggability, or "observability", by taking advantage of the numerous debugging tools available of the C compilation toolchain.

Since the implementation of the full OCaml language is likely too ambitious for one project, the core project will instead revolve around the implementation of an interesting subset of OCaml which still provides technically interesting challenges in its implementation, while extension tasks may extend the implementation to cover a wider subset of the entire language. An informal description of said subset will be given in the Substance and Structure section.

The "observability" of the compiler means that where possible the debugger should behave *as if* it was debugging the OCaml code. This means that the debugger should consider sections of the compiled machine code to be mapped to the appropriate lines of code in OCaml. Furthermore, locally bound variables should be visible from the debugger when execution has reached the relevant point, meaning that local variables in the C code should mirror the variables in the OCaml code.

In addition, features such as closures and parametric polymorphism must be implemented carefully since these features don't have any similar structures in C for which they could've been mapped to, but also to maintain "observability" these must be "transparent" to the debugger (for example, a polymorphic function of the type `'a -> 'a` that has been instantiated as the type `int -> int` should have this quality be visible from the debugger).

The resulting compiler can be evaluated in mainly two contexts. One of these is a straightforward performance comparison, where a testbed of OCaml programs are benchmarked across the OCaml native compiler, the OCaml bytecode compiler, and this project. The other context is the "observability" of the compiler; where programs are stopped at a certain point of their execution and inspected, to see how much of the internal state of the stack is recoverable. This can be made in comparison with OCaml's bytecode debugger, `ocamldebug`.

Resources Required

A copy of the source of the OCaml compiler is needed, as well as C compilation tools such as `gcc`, and debuggers such as `gdb`. Fortunately, these tools are maintained via open source and are readily available online at `github.com/ocaml/ocaml`, `gcc.gnu.org`, and

`www.gnu.org/software/gdb` respectively.

Later in the project, implementing certain more advanced features may require the uses of libraries such as `libffi` (`sourceware.org/libffi`), `liballocs` (`github.com/stephenrkell/liballocs`), or the Boehm garbage collector (`http://www.hboehm.info/gc/`); these libraries are all open source and readily available.

Starting Point

The project, at least initially, will be a fork of the OCaml compiler obtainable at `github.com/ocaml/ocaml` which will take the same front-end as the existing compiler, but will implement a new back-end for compilation into C. Apart from other resources indicated in the previous section, no other relevant significant bodies of code have been determined for use at this point.

Substance and Structure of the Project

The core of the project will be devoted into the implementation of three incrementally expanding subsets of OCaml. These subsets have been identified such that each is sufficient to write nontrivial programs in, and features have been grouped by relation to each other. In this manner work can be easily split into 3 discrete blocks, each of which will support a new collection of programs by their completion.

Subset 1

Subset 1 will be a very simple language with only a limited number of types and language constructs. The subset will contain only the basic boolean, integer, floating point and `unit` types, basic string support (for input/output), top level function declarations with `let` and `let rec`, which only accept one argument (so as to sidestep the problem of partial application and closures), `if`, `for`, `while`, `ref`, as well as basic arithmetic and boolean operations.

These features were chosen because because they very easily make for a small minimalist language, and all constructs more or less have direct analogues in C meaning translation from AST to C code should be fairly easy. Nonetheless, this small subset is sufficient to write simple programs.

Subset 2

Subset 2 will focus on the implementation of types and polymorphism, and will support tuples, lists, parametric polymorphism, algebraic data types, record types, and `match` expressions.

The implementation of this subset will likely require the design of an object representation for OCaml values. Tuples and record types map easily to structs in C, while a tagged union can be used for OCaml's "sum of products" approach to algebraic data types. On the first pass, it is quite likely that a naive "tagged pointer" type will be used for support for polymorphism, but extension tasks may experiment with eliminating this need via some method.

The addition of algebraic data types greatly increases the scope of supported programs, and allows for functions taking multiple arguments (via tuples), list processing functions, and more complex data structures such as binary trees.

Subset 3

Subset 3 will finally add treatment of functions as first-class values. This will include lambdas, lexical closures, and partial application of functions.

The implementation of this subset will require some representation of closures. While C does support function pointers, it has no lexical closures and as such a representation where closures are function pointers with an extra `void *` pointer that refers to data in the closure. As an extension task, more performant ways of implementing closures may be investigated, such as potentially dynamically generating entry points that push the required arguments before jumping to the body of the function.

The addition of these features will make higher-order functions representable, and make available functions such as `map`, `filter`, function composition etc. At this point subset 3 can be considered a fairly minimal but full-featured functional language.

Evaluation

Evaluation will require the creation of a small library of test programs. These programs should be novel and exhibit the full range of language features supported by the subsets.

Two main methods will then be used to evaluate the compiler. One is benchmarking performance of the test programs with the existing OCaml native compiler and the OCaml bytecode compiler. While it is not expected that the C compiler will match the performance of OCaml official compilers, the goal is to be within a reasonable margin so that the performance of compiled C code is comparable.

The other method will be to compare the debug outputs of the C code and the OCaml

code. The native OCaml compiler does not allow observation of locally bound values at all, so our C compiler should beat this easily; although a more interesting comparison would be with the OCaml bytecode debugger. Breakpoints could be set at randomly chosen call sites, and the stack traces, visible locals etc. could be compared between `ocamldebug` and the debug output of the C code.

Extension Tasks

This project leaves a lot of room for extension tasks. An obvious extension is to extend the subset to be a more complete subset of OCaml. The next steps taken would likely be module and exception support, although both are far more complex features to attempt for implementation in C.

More realistic extension tasks would be changes to the existing code in order to make it more performant. This would involve improvements to the implementation of polymorphic functions and closures, as described in their respective sections prior.

Another plausible extension task is to improve the debug output of OCaml values. Since C does not have an exact analogue for all OCaml values the debug output from `gdb` for example will not look exactly like their OCaml representation. However, `gdb` supports the use of custom formatters written in Python for types and a task to implement a way to generate formatters to print OCaml values correctly is a suitable extension task.

Success Criteria

The following criteria should be achieved at the completion of the project:

- A working compiler, that is able to compile the language as described in Subset 3 into C code correctly;
- Comparable performance between the output of the compiler and the OCaml native and bytecode compilers;
- "Observability" in the compiled executable achieved, with tools such as `gdb` providing a coherent OCaml view of the execution of the program.

Timetable and Milestones

20 October — 3 November

Initial experiments and reading. Familiarise with the OCaml compiler source code and obtain typed AST. Set up development environment. Write testbed of evaluation programs so that features can be tested and evaluated immediately after their implementation.

Milestones: A completed small library of OCaml programs, plus a way of obtaining the typed AST from the OCaml compiler.

4 November — 17 November

Implementation of Subset 1. Evaluation of Subset 1 performance with respect to the OCaml native and bytecode compilers.

Milestones: A collection of programs now compileable by the Subset 1 compiler, along with their running times collected in comparison with the OCaml compilers.

18 November — 15 December

Implementation of Subset 2. This is expected to take longer than the implementation of Subset 1 to take into account that special care needs to be taken when designing object representations of OCaml values and implementation of parametric polymorphism. Evaluation of Subset 2 performance with respect to OCaml compilers.

Milestones: A larger collection of programs now compileable by the Subset 2 compiler, along with their performance results in comparison with the OCaml compilers.

16 December — 29 December

(Two week break for Christmas)

30 December — 12 January

Implementation of Subset 3. Evaluation of Subset 3 performance, as well as comparison between the debug output of the compiled code with `ocamldebug`.

Milestones: Completion of core project with the implementation of Subset 3. Some evaluation results (although perhaps not complete) for the core project.

13 January — 26 January

Write up of the Progress Report. Perform any new evaluation tasks that were missed but become apparent in the write up of the Progress Report. Review the remainder of the project in context with the existing work and determine which extension tasks are best to pursue.

Milestones: Completed core project with evaluation, with a completed Progress Report ready for submission. Entire project reviewed with supervisor and overseers.

27 January — 9 February

Submission of Progress Report. Revise evaluation tasks with feedback from Progress Report. Prepare Progress Report presentation.

10 February — 23 February

Start work on dissertation, as well as the implementation of extension tasks, which will likely be ways of making the compiled code more performant. Investigation and use of perhaps `libffi` and `liballocs` to improve performance and observability.

Milestones: The introduction and preparation sections of the dissertation complete. Improvements on the compiler made, with evaluation results to show improvements in performance of the compiled output.

24 February — 9 March

Start writing the implementation section of dissertation. Implementation of further extension tasks, likely custom formatters to improve the debug output.

Milestones: Implementation section of the dissertation at least half complete. Improvements to debug output of debuggers run on the compiled executables.

10 March — 22 March

Finish writing implementation section of dissertation. Start evaluation section. Pick up any further evaluation tasks at this point that become apparent.

Milestones: Finished implementation section, evaluation section at least half complete. Revised evaluation tasks in accordance with the write up of the evaluation section.

23 March — 5 April

Finish evaluation and conclusion sections of the dissertation. Deliver draft dissertation to supervisor and director of studies for feedback. Spend time working on whatever is in greatest need of attention.

Milestones: Submit a complete draft of the dissertation for review to supervisor and director of studies.

6 April — 19 April

Revise dissertation based on feedback received. Finish any last tasks that require performing, which are hopefully minor at this point in time. Send revised drafts for further rounds of review.

20 April —

Final adjustments and revisions to dissertation and the body of code. Prepare for submission of dissertation.

Milestone: Submission of dissertation.