Computer Science Tripos – Part II – Project Proposal

An observable OCaml, via C and `liballocs`

Cheng Sun, Churchill College

Originator: Stephen Kell

April 28, 2017

Project Supervisor: Stephen Kell

Director of Studies: John Fawcett

Project Overseers: Timothy Griffin & Pietro Lio

Introduction

OCaml is one of the most commonly used members of the ML family of functional languages. It is popular for its expressivity, type system and performance. However, there is not as of yet a good story for debugging OCaml programs, and observing their behaviour at runtime.

The OCaml bytecode debugger, `ocamldebug`, forms part of the core OCaml toolchain. One major problem is that `ocamldebug` is unable to "see through" polymorphism. For instance, suppose we would like to debug the following polymorphic list-reverse function, which has type `'a list -> 'a list`:

```
let my_rev lst =
  match lst with
  | [] -> []
  | x::xs -> List.append xs [x]

let result = my_rev [1; 2; 3]
```

Now let's try to debug the function with `ocamldebug`.

```
$ ocamlc -g -o my_rev my_rev.ml
$ ocamldebug my_rev
        OCaml Debugger version 4.02.3

(ocd) break @ My_rev 1
```

```
Loading program... done.
Breakpoint 1 at 21600: file my_rev.ml, line 2,
   characters 3-62
(ocd) run
Time: 12 - pc: 21600 - module My_rev
Breakpoint: 1
2   <|b|>match lst with
(ocd) print lst
lst: 'a list = [<poly>; <poly>; <poly>]
```

Note that `ocamldebug` is unable to display the contents of the input list, as it does not know the concrete type that the type variable `'a` is instantiated with for this invocation.

There are many other deficiencies with the OCaml debugger, mostly stemming from its immaturity and lack of features. These issues mean that when debugging OCaml code, one often has to resort to "printf debugging" instead.

The aim of this project is to investigate whether the experience of debugging and observing the runtime behaviour of OCaml programs could be improved by utilising the mature C toolchain. My goal is to write a translator that compiles OCaml code into equivalent C code, whilst maintaining a well-defined mapping between the two (in terms of variable names, types, and so on). A user will then be able to debug their OCaml program by using (perhaps an augmented) `gdb` on the generated C code.

In order to solve the problem of "seeing through" polymorphism, I will use a library written by my supervisor, `liballocs`, which keeps track of runtime allocation metadata (including their types) with low overhead. This will allow the debugger to inspect the allocation corresponding to the input to `my_rev`, for instance, and conclude that it is operating on lists with elements of type `int`.

Starting point

The project will be focused on creating a new "backend" for the OCaml compiler, so I will build on the existing code of the frontend. This includes reusing code for lexing, parsing and typing of OCaml programs, as well as the transformation from the typed AST to the Intermediate Representation that we will be using as input.

In order to provide a standard library to compile programs against, I will try to use as much of the existing OCaml `stdlib` as possible. If this turns out not to be feasible then I will resort to writing a subset of the standard library by hand.

I will use the open-source library `liballocs`, written by Stephen Kell. This library provides routines to tag regions in virtual memory ("allocations") with run-time type information.

I may also make use of some further open-source libraries such as:

- `libffi` – a library providing a portable way to perform a call to a function with a foreign function interface (such as the closures that I will dynamically create at runtime);
- `cil` – a library written in OCaml providing a framework for the manipulation of C programs.

There exists similar prior work [1] that shows that the overall project concept is feasible. However, I will be making different design choices to the compiler presented in the paper, due to differing requirements. For instance, the presented compiler generates code that uses continuation-passing style, which I would like to avoid doing, as it harms observability – backtraces would no longer be directly meaningful.

Resources required

The project will primarily be developed on my personal computer for convenience. The computer has a 4-core 3.30GHz Intel Xeon CPU, a 250GB SSD and 8GB of RAM. I am using Arch Linux. No other non-standard equipment is anticipated to be required, and I can easily continue development on MCS machines if required.

The project will be synced to a git server (likely GitHub), and backed up regularly to my MCS network drive.

The project will utilise various standard open-source software packages, including the usual C toolchain (e.g. `gcc`, `gdb`, `make`) along with the OCaml compiler. (As mentioned previously, `liballocs` requires some patches to `binutils`, but these can easily be built.)

Work to be done

The project can be split up into the following tasks:

1. Performing a study of the starting point (OCaml compiler frontend and `liballocs`). This includes an investigation into the forms of the various Intermediate Representations of the OCaml compiler, and selection of the best IR to use as input for our project.

2. Choosing a suitable object representation for boxed objects in memory. There are a variety of alternatives to OCaml's own tagged-pointer representation (which leads to awkward 63-bit integers). This choice requires consideration of the garbage collection strategy, even if a GC is not within the core scope of this project.
3. Implementing the translation from OCaml IR to C (with `liballocs`), for a variety of language features:
   (a) Fundamental types: `bool`, `int`, `float`, `list`, `ref`, ...;
   (b) Tuples and records, which can be implemented using C `structs`;
   (c) (Non-polymorphic) variants, which can be implemented using tagged C `unions`;
   (d) Polymorphic types, which can be represented safely using `void *` opaque pointers, as downcasts are never required in ML (although `liballocs` could provide the necessary information);
   (e) Parametrically polymorphic functions;
   (f) First class functions and closures, which require lambda lifting, or more generally something like the technique of Breuel [2] (dynamically creating an executable stub associated with each closure at runtime).
4. Integrating a subset of the OCaml `stdlib`, or writing my own as necessary.
5. Writing a runtime library to support dynamic closure creation.
6. Creating a corpus of test and benchmark programs.
7. Evaluation, as described in the *Success criteria* section below.
8. Extensions as time permits, as described in the *Possible extensions* section below.

Success criteria

My project will have been successful if I have managed to create a OCaml-to-C translator that works on a commonly used subset of the OCaml language (including closures, parametrically polymorphic functions). Furthermore the polymorphism must be able to be seen through (solving the problem highlighted in the *Introduction* section, for instance).

The project will be evaluated in two ways. Both will require a corpus of OCaml tests and micro-benchmarks.

Firstly, I will measure the "observability" of the compiled C programs compared with their native and bytecode OCaml counterparts. For instance, one metric would be obtained by interrupting these programs at pre-determined points in the code, and counting the number of local variables that could be recovered from the stack frames. This is an accurate metric for observability, as the utility of debugging is determined almost completely by the amount of state that is visible at a breakpoint.

Secondly, I will measure the performance of the resultant C executables, by timing benchmark programs as compiled by both compilers. As the primary objective of the project is debuggability rather than performance, I do not expect the performance to be anywhere near as good as code generated by the OCaml native-code compiler: I will be satisfied as long as the performance is within a reasonable factor for the majority of use cases. It is still necessary to evaluate performance, because it affects the debuggability of CPU-intensive programs.

There are many other qualitative evaluations that I will also consider, such as the ease of interoperability with C libraries using this approach, the ability to insert instrumentation into the code, and the overall debugging experience when using `gdb` on the resultant executables.

Possible extensions

- Improving the runtime, such as adding a garbage collector;
- Adding further language and standard library support, to increase the range of programs supported;
- Optimising the performance of the generated program;
- Improving the debugging experience (such as augmenting `gdb`).

Timetable

1. **Michaelmas weeks 2–3:** Dig into the OCaml compiler frontend; study its Intermediate Representations. Learn to use the `liballocs` library.
2. **Michaelmas weeks 4–6:** Begin work on translator, initially targeting basic features (fundamental types, monomorphic functions). Design an initial representation for boxed objects.
3. **Michaelmas weeks 7–8:** Add support for tuples, variants, records. Test against very simple programs.

4. **Michaelmas vacation weeks 1–2:** Add support for polymorphic functions and types.
5. **Michaelmas vacation weeks 3–4:** Get a small, commonly-used subset of the standard library working. Start creating corpus of test programs, and begin testing.
6. **Michaelmas vacation weeks 5–6:** Reserved for holidays/revision.
7. **Lent term weeks 0–1:** Add initial support for dynamic closure creation. Complete corpus of test OCaml programs; begin evaluation. Start progress report.
8. **Lent term week 2–3:** Milestone: compiler has working support for most test programs (demoable). Submit progress report. Prepare and give presentation.
9. **Lent term weeks 4–6:** Testing; fix bugs. Improve closure creation, and evaluate. Investigate "source maps" to map C code to corresponding lines in the OCaml source.
10. **Lent term weeks 7–8:** Further evaluation, and work on improving evaluated metrics. Start on main chapters of dissertation.
11. **Easter vacation weeks 1–3:** Milestone: compiler works for all intended features. Wrap up evaluation. Continued work on dissertation.
12. **Easter vacation weeks 4–5:** Reserved for holidays/revision. Extension work if time permits.
13. **Easter term weeks 0–2:** Form conclusion and complete dissertation. Time may be limited due to revision.
14. **Easter term week 3:** Proof reading and submission of final dissertation.

References

[1] D. Tarditi, A. Acharya, P. Lee, *No Assembly Required: Compiling Standard ML to C*, November 1990.
`http://repository.cmu.edu/cgi/viewcontent.cgi?article=3011&context=compsci`

[2] Thomas Breuel, *Lexical Closures for C++*, In Proc. USENIX C++ Conf., pages 293-304, Denver, CO, October 1988.
`http://www.cl.cam.ac.uk/~srk31/teaching/redist/breuel88lexical.pdf`