Towards A Debugger for Native-Code OCaml Applications

Fabrice Le Fessant
INRIA & OCamlPro
fabrice.le_fessant@inria.fr

Pierre Chambart
OCamlPro
pierre.chambart@ocamlpro.com

Abstract

In this talk, we will present a starting project at OCamlPro, the development of a debugging framework for OCaml native-code applications, based on the LLDB Debugger, a debugger built on top of the LLVM framework. We implemented a complete binding of LLDB C++ API for OCaml, and then used it to build several tools, one "generic" debugger, and two small utilities to monitor the memory behavior of OCaml applications.

1 Introduction

The OCaml distribution provides a debugger, ocamldebug, but only for bytecode applications. This limitation is unfortunate, as most users, today, only work with native-code applications. Moreover, the bytecode debugger is limited to OCaml code, and cannot debug interactions between OCaml and C.

Recent work has been done by Mark Shinwell at OCamllabs[4] to extend gdb to debug OCaml native applications. It is a complex task, as not only the OCaml compiler itself must be modified to emit DWARF information[2], but gdb itself must be extended to deal with OCaml specifics. Unfortunately, extending gdb is not a simple task, it must be done in a combination of C and Python, and requires an expertise of gdb internals that are not very well documented. Finally, the system integration of gdb is not very good on Windows and Mac OS X platforms, two major targets for OCaml applications in the future.

Recently, a new tool appeared in the LLVM project, a framework to build compilers and associated tools: the new tool, LLDB, uses LLVM capabilities to assemble/disassemble machine-code and object files, and extends LLVM with debugging facilities (managing code and data breakpoints, etc.). LLDB is now the standard tool within Apple's Xcode to debug Mac OS X applications, and is now being ported under Windows. Moreover, LLDB has been designed from the beginning to be more a framework than a tool, providing a clean C++ API to build advanced debugging tools and scripts.

Thus, we decided to explore the capabilities to use LLDB API to build a debugger for native-code OCaml applications. We started by creating an OCaml binding for LLDB C++ API, that allowed us to program our young debugger completely in OCaml. We then tried several approaches: first, we built some little tools using the new OCaml API to monitor the memory behavior of some OCaml applications, using breakpoints and memory introspection; then, we develop a standalone debugger for TypeRex, that takes benefit from some extensions available in our version of OCaml for memory-profiling. We think the result is promising.

2 LLDB-OCaml Binding

LLDB comes both with a clean C++ API and a Python binding, generated in SWIG[1]. After a short look at the C++ API, we decided it was clean enough to automate the generation of a complete OCaml binding: it tooks us a few minutes to manually write working stubs for a few methods, then a few hours to write a stub generator taking the C++ header files as input, and then a few days to test and polish the binding. The stub generator is less than a thousand lines of OCaml, mostly a simple parser for C++ header files (based on top of Genlex...), and two pretty-printers for C++ and OCaml parts of the binding. Of course, we used our experience of developing such binding generators, in particular for the wxWidgets library[3].

The binding itself covers 97% of LLDB C++ API (1065 methods are available in OCaml over 1090 in the complete API), and it works now both with the version of LLDB available in Debian and with the trunk version. We used a purely functional syntax for the binding, i.e. C++ methods are OCaml functions, taking the object as first argument.

3 Applications

We first tried the new born OCaml binding to build a few small tools, to check how it behaved for OCaml applications:

First, we developed gc_stats, a tool that sets a breakpoint on caml_sys_exit on the target application, and then explores the memory of the target application to recover and print garbage collection counters. With only 150 lines of OCaml, we were able to monitor the final memory usage of any OCaml application, without instrumenting its code.

Then, we developed alloc_prof, a tool that sets a breakpoint on caml_garbage_collection, and then outputs the complete stack (including C functions) at every such garbage collection. With only 130 lines of code, this is close to what our previous allocation profiler would do, but again, without using a specific compiler or runtime.

Finally, we started to develop ocp-lldb, an interactive debugger for OCaml native applications. For that, we used the interactive debugger already available in LLDB C++ API, but we customized it be adding a new ocaml sub-command with facilities to debug OCaml specific features:

ocaml modules	prints the list of OCaml modules in the executable
ocaml target MODULE	prints function symbols in module MODULE
ocaml break MOD.FUN	set a breakpoint on the corresponding function
ocaml args NARGS	print NARGS registers, corresponding to the arguments in the current OCaml calling ABI
ocaml global MOD.VAL	print the value of a fully-qualified global value
ocaml heap	print heap information (size of heap, heap segments and minor heap, gc stats)
ocaml print ADDR	print the value at address ADDR

Currently, when debugging a standard OCaml application, we print OCaml values as simple blocks, using only header information (size) and our capacity to know if a value is in the heap. However, when debugging a TypeRex application (i.e. compiled using ocp-memprof OCaml compiler, we are able to use the additionnal typing information present in the header of every OCaml value to correctly display most values depending on their types, as in the OCaml toplevel.

4 Discussion

In the future, we plan to extend this work in several directions:

- Add a graphical interface on top of the debugger to offer visual debugging with variable monitoring;
- Modify the compiler to allow breakpoints to be set in any OCaml expression (currently, breakpoints can only be set on functions), and to provide information on local variables (using a specific format, while waiting for DWARF support to be available);

References

[1] D. M. Beazley. Swig: an easy to use tool for integrating scripting languages with c and c++. In *Proceedings of the 4th conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4*, TCLTK'96, pages 15–15, Berkeley, CA, USA, 1996. USENIX Association.

[2] M. J. Eager and E. Consulting. Introduction to the DWARF Debugging Format. *Group*, 2007.

[3] F. Le Fessant. The design of the wxocaml library. OCaml Users and Developers, 2013.

[4] M. Shinwell. Emission of dwarf debugging information, 2015. http://www.cl.cam.ac.uk/projects/ocamllabs/tasks/compiler.html#Emission of DWARF debugging information.