An LLVM Backend for OCaml

Colin Benner

June 7, 2012

1 Introduction

As part of my bachelor thesis I have implemented a new backend for the OCaml native-code compiler *ocamlopt* for the AMD64 architecture. It uses the Low Level Virtual Machine framework (LLVM, an optimising compiler framework) to generate machine specific assembly code. The goal was to find out, how well LLVM is suited for implementing a backend for OCaml.

2 Implementation

The way OCaml's optimising compiler *ocamlopt* translates OCaml code to assembly is the following. First the source code is scanned, parsed and types are inferred. Then pattern matching, module and class constructs of OCaml are translated yielding the *Lambda* intermediate representation. Another intermediate representation called *Cmm*, a simple imperative language based on C--, is produced from *Lambda* by doing closure conversion, function inlining, uncurrying. In *Cmm* the data is structured in the way used by the OCaml runtime. The *Cmm* code is then compiled to assembly code by the native code backend for the target architecture.

This is where the new backend differs from OCaml's native code backend. Assembly is not created by the OCaml compiler itself. It just creates code in LLVM intermediate representation (IR) which is then translated to assembly using LLVM.

Some things had to be changed with respect to the native code backend. Binaries created using the native code backend are completely incompatible with binaries created via LLVM. The changes responsible for the incompatibility are the use of the calling convention specified in the System V ABI and setjmp/longjmp for exception handling.

3 Results

The new backend is a lot less complex. In particular, there is almost no hardware specific code. Thus porting the new backend to any architecture supported by LLVM should require very little work.

There are a couple of problems when using LLVM to translate *Cmm* to assembly. While neither OCaml's usual calling convention (some registers were treated as callee save when there were no callee save registers) nor exception handling mechanism could be used, the main problem was the garbage collection support LLVM provides. OCaml's way of collecting garbage implies that every pointer used across a function call has to be stored before calling that function and reloaded afterwards. LLVM does not care whether a function call is in a pointer's live range, it stores *all* pointers used in the current function to the stack.

Furthermore, due to a bug in LLVM prevents pointers are not always reloaded after function calls when optimisations are enabled. Hence, the code can not be optimised, making the resulting binary unnecessarily slow. Adding to that, the backend probably produces non-optimal LLVM IR but the most important aspect seems to be that basically every pointer is written to memory.

The results have to be taken with a grain of salt as it was not possible to benchmark the backend with really interesting programs due to certain garbage collection bugs.

The problem with exception handling is mainly the memory footprint of exception handler. While the native code backend produces code that needs just 16 bytes on the stack for every exception handler, the new backend uses 200 bytes.

4 Conclusion

In principle, LLVM is suitable to implement a new backend for OCaml. To do this, however, one has to give up on binary compatibility to the old backend.

The generated assembly could be a lot nicer and probably faster, if LLVM had better support for garbage collection in functional languages. In particular, it would be very helpful, if it was possible to use LLVM's usual single static assignment form when handling pointers instead of dealing with stack slots. The stack slots would still be necessary because OCaml's garbage collector relies on them, but generating good code would be a lot easier if LLVM took care of when to store which register in which stack frame. The same problem has also been encountered by the LLVM backend of the Open Dylan compiler.

What might be surprising is, that the LLVM backend for the Glasgow Haskell Compiler (GHC) has none of these problems, although there is a certain similarity between OCaml and Haskell. However, when using GHC the code is already in continuation passing style when generating LLVM IR. Therefore, no function call ever returns, so callee save registers can not be a problem. In addition, this means that GHC does not have to rely on LLVM for garbage collection, because it manages its own stack and therefore has complete control over where pointers are located.