Compiling with Continuations and LLVM

Kavon Farvardin
University of Chicago
kavon@cs.uchicago.edu

John Reppy
University of Chicago
jhr@cs.uchicago.edu

1. Introduction

Maintaining a native code generator that targets multiple architectures is a hassle for compiler writers that requires expert knowledge of each new processor's quirks. Some functional language compilers avoid this problem by targeting C as a "portable assembly language" [9, 15], but this approach has significant drawbacks in both compile-time and runtime performance. More recently, LLVM [10], which provides a low-level SSA-based representation with many available optimization passes, has emerged as a popular backend for compiler writers. Although designed with imperative and object-oriented languages as its expected clients, LLVM has been used to build backends for functional languages, such as Standard ML [12], Haskell [16], and Erlang [14]. While LLVM addresses the problem of maintaining native code generators and is a better portable assembly language than C code, it still suffers from a bias toward C runtime conventions, which makes it a less than ideal target for a functional-language compiler. Functional language implementations often use specialized register and calling conventions,[1] and require guaranteed tail-call optimization, mechanisms to communicate with the garbage collector, and efficient support for features like first-class continuations.

In this paper, we present our approach to solving the problems of using LLVM as a backend for functional language implementations. In particular, we show how to use LLVM to support the heap-allocated first-class continuation runtime model [4] used by the SML/NJ system and by the Manticore system [7]. We have integrated our approach into the *Parallel ML* (PML) compiler that is part of the Manticore project [7]. Initial observations suggest that the LLVM backend produces more efficient code relative to the previous MLRisc [8] backend.

The LLVM backends for Haskell (GHC) and Erlang (ErLLVM) use special language-specific calling conventions added to LLVM that support TCO. The LLVM backend for the MLton SML compiler uses trampolining to avoid the issues with TCO. As far as we know, no one has successfully implemented first-class continuations with LLVM. In the remainder of the paper, we describe a new calling convention that we have added to LLVM, how we handle the GC interface, and how we support capturing continuations to support preemption. Many details are omitted because of space limits, but can be found in an upcoming technical report [5].

2. Proper Tail-Call Optimization

Most functional languages use tail recursion to express loops and, thus, require that tail calls do not grow the stack. In limited circumstances, LLVM can meet this requirement, but even when LLVM is able to apply TCO to a call, there is extra stack-management overhead on function entry and exit [1–3]. This extra overhead is bad enough for implementations that rely on traditional stacks, but it is prohibitively expensive for implementations that use CPS and heap-allocated continuations [4].

A standard technique to avoid this overhead is to merge mutually tail-recursive functions into a single LLVM function and then use internal jumps instead of tail calls. Unfortunately, this approach does not work for tail calls to unknown functions and incurs substantial compile-time cost. The MLton SML compiler partitions the generated code into multiple large functions (called *chunks*) and uses a trampoline to transfer control between chunks [12]. This approach solves the compile-time issue, but places extra strain on the register allocator because of the nature of the control-flow graphs in the chunks.

Our solution to this problem is to add a new calling convention, which we call Jump-With-Arguments (JWA), to LLVM. This calling convention has the property that it uses all of the available hardware registers and that no registers are preserved by either the caller or callee. Furthermore, the argument registers are exactly the live registers on return (*i.e.*, the call and return have identical register use conventions).[2] Second, we mark every function with the `naked` attribute, which tells the code generator to completely skip the emission of a function prologue and epilogue. This attribute *must* be used with care; it was originally designed to support functions consisting entirely of inline assembly that manages the stack explicitly (*e.g.*, interrupt service routines). Using the `naked` attribute means that the generated code is responsible for ensuring that there is sufficient stack space for register spills. We use an assembly language shim for switching between the runtime system code (written in C) and the code generated by the PML compiler. This shim code allocates a frame that is large enough to handle the maximum number of register spills.[3]

3. Allocation and Garbage Collection

Our JWA calling convention maps function arguments to hardware registers based on the position of the argument. By using standard positions for special runtime registers, we can effectively pin them to hardware registers (*e.g.*, we always pass the allocation pointer as the first argument). Object allocation then defines new instances of the allocation pointer, which thread the current state of the pointer through the code (recall that LLVM code is in SSA form).

One of the advantages of CPS with heap-allocated continuations is that the interface to garbage collection is very simple. The runtime does not need to scan a stack (since there is no stack) or understand any other properties of the code generator. We did not have to make any modifications to our existing collector to support our LLVM backend.

[1] For example, many implementations dedicate a specific register as an allocation pointer to support efficient open-coding of heap allocation.

[2] These properties are why we need to create a new convention, instead of using an existing convention.

[3] This technique is borrowed from the SML/NJ system; the spill limit is enforced by the compiler limiting the number of live variables at any control point.

```
1  declare jwa {i64*, i64*} @invoke-gc(i64*, i64*)
2
3  define jwa void @foo(i64 allocPtr_0, ... ) naked {
4  ...
5    if enoughSpace, label continue, label doGC
6
7  doGC:
8    roots_0 = allocPtr_0
9    ; ... save live vals in roots_0 ...
10   allocPtr_1 = getelementptr allocPtr_0, 5 ; bump
11   retV = call jwa {i64*, i64*}
12            @invoke-gc(allocPtr_1, roots_0)
13   allocPtr_2 = extractvalue retV, 0
14   roots_1 = extractvalue retV, 1
15   ; ... restore live vals ...
16   goto label continue
17
18 continue:
19   allocPtr_3 = phi i64* [ allocPtr_0, allocPtr_2 ]
20   liveVal_1 = phi i64* [ ... ]
21 ...
```

Figure 1. An example of a compiler generated safe point for garbage collection.

The compiler is responsible for generating code to check for heap exhaustion and code to invoke the GC when necessary. In Figure 1, we list simplified LLVM code for the heap-limit check (Lines 4–5) and GC invocation. To invoke the GC, we first save the live variables into a heap object (Lines 8–10) and then do a non-tail JWA call to `@invoke-gc`. When this function returns, we restore the allocation pointer and live variables (Lines 13–15).

We use a non-tail call to `@invoke-gc` for reasons described in below in Section 4. We ensure that LLVM does not try to preserve values across the `@invoke-gc` call by taking advantage of the rules about aliasing. Once the pointer reaching all live values is passed to `@invoke-gc`, which is an external function not visible to LLVM, LLVM must assume that all values may have changed and must use the updated versions from the pointer returned.

4. Preemption and Multithreading

The main motivation for supporting heap-allocated first-class continuations is to enable the efficient implementation of the concurrency mechanisms in the Manticore runtime system [6, 11]. While the mechanisms described in Section 2 are sufficient to support the explicit management of continuations, preemptive scheduling requires capturing continuations that are not explicit in the intermediate code. We use the technique developed for supporting asynchronous signals in SML/NJ [13], which limits preemption to *safe points* where all live values are known. Specifically, those places in the code where we perform a heap-limit check serve as safe points.[4] We store the heap limit pointer in memory, which means that we can set it to zero when we need to force a heap-limit check to fail. The runtime system then constructs a continuation closure, which is passed to the preemption handler where it can be put on scheduling queue *etc.*

This mechanism introduces an additional challenge for our LLVM backend, because the implicit continuations that are captured during preemption do not correspond to LLVM functions and are invoked from unknown locations. For example, consider the heap-limit test in Figure 1. If it is invoked to force a preemption, then a continuation will be created by the runtime system that has Line 13 as its code address (*i.e.*, the return address of the call to `@invoke-gc`). Since Line 13 is not a function entry, there is no way in LLVM to specify a calling convention.

We are saved, however, by the fact that we can specify our own return convention for structs in LLVM. Normal conventions return a struct using a mix of registers or stack space that does not match up with the way arguments are passed to functions. We setup our JWA convention so that struct field i is returned in the same register that argument i would be passed in during a call. This way, return addresses generated through a non-tail JWA call can be jumped to safely using a standard JWA tail call (which is how we throw to a continuation!).

[4] The code generator ensures that even non-allocating loops contain a heap-limit check.

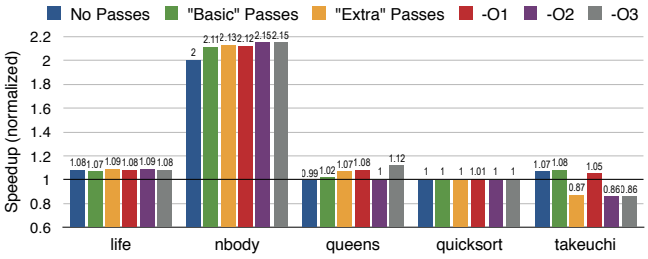

Figure 2. Execution time speedups when compiled with LLVM, normalized to MLRisc. Each bar represents a different set of additional optimizations applied when compiling with LLVM.

5. Evaluation

We measured the difference in application performance between our two backends to get a sense of how well LLVM can handle the unusual code we generate. Figure 2 summarizes our experiment conducted on a MacBook Pro with an i7-4850HQ processor. Each benchmark was compiled with different sets of additional LLVM optimization passes applied before generating assembly:

Basic: Hand-picked optimizations that contract functions into a form the LLVM code generator might normally expect.

Extra: Basic optimizations + memory-operation vectorization and movement.

-Ox: These are the default optimization levels built into LLVM.

LLVM's code generator outputs much better assembly than MLRisc for the floating-point-heavy nbody benchmark. While the takeuchi benchmark only tests the overhead of recursion, its performance takes a hit with LLVM because of the `slp-vectorizer` pass. After the pass is applied, the hot path is smaller because of the use of vector instructions to initialize a closure, but the execution cost of such instructions outweighed the size benefit.

6. Conclusion and Future Work

We have outlined how to extend LLVM to support the heap-allocated first-class continuation runtime model. We are in the process of replacing the MLRisc backend with LLVM using the approach described in this paper. Initial observations suggest that this new LLVM backend produces smaller and more efficient code. We are also hoping to apply these techniques to the SML/NJ system in the future.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant CCF-1010568. The views and conclusions contained herein are those of the authors and should not

be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

References

[1] LLVM Bug 13826 - unreachable prevents tail calls. `https://llvm.org/bugs/show_bug.cgi?id=13826`, 2012.

[2] LLVM Bug 23470 - Inefficient lowering of 'musttail' call . `https://llvm.org/bugs/show_bug.cgi?id=23470`, 2015.

[3] LLVM Bug 23766 - musttail calls are not allowed to precede unreachable. `https://llvm.org/bugs/show_bug.cgi?id=23766`, 2015.

[4] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[5] K. Farvardin and J. Reppy. An LLVM backend for Manticore. Technical Report *Number Pending*, Dept. of C.S., U. of Chicago, Chicago, IL, 2016.

[6] M. Fluet, M. Rainey, and J. Reppy. A scheduling framework for general-purpose parallel languages. In *ICFP '08*, pages 241–252, New York, NY, Sept. 2008. ACM.

[7] M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Implicitly-threaded parallelism in Manticore. *JFP*, 20(5–6):537–576, 2011.

[8] L. George, F. Guillame, and J. Reppy. A portable and optimizing back end for the SML/NJ compiler. In *CC '94*, number 786 in LNCS, pages 83–97, New York, NY, Apr. 1994. Springer-Verlag.

[9] S. P. Jones, N. Ramsey, and F. Reig. C - -: A Portable Assembly Language that Supports Garbage Collection. In *PPDP '99*, pages 1–28. Springer-Verlag, New York, NY, 1999.

[10] C. A. Lattner. LLVM: An infrastructure for multi-stage optimization. Master's thesis, University of Illinois at Urbana-Champaign, 2002.

[11] M. Le and M. Fluet. Partial aborts for transactions via first-class continuations. In *ICFP '15*, pages 230–242, New York, NY, Sept. 2015. ACM.

[12] B. A. Leibig. An LLVM Back-end for MLton. Master's thesis, Rochester Institute of Technology, 2013.

[13] J. H. Reppy. Asynchronous signals in Standard ML. Technical Report TR 90-1144, Dept. of CS, Cornell University, Ithaca, NY, Aug. 1990.

[14] K. Sagonas, C. Stavrakakis, and Y. Tsiouris. ErLLVM: An LLVM backend for Erlang. In *ERLANG '12*, pages 21–32, New York, NY, 2012. ACM.

[15] D. Tarditi, P. Lee, and A. Acharya. No Assembly Required: Compiling Standard ML to C. *ACM LOPLAS*, 1(2):161–177, June 1992.

[16] D. A. Terei and M. M. Chakravarty. An LLVM Backend for GHC. In *HASKELL '10*, New York, NY, Sept. 2010. ACM.