Compiling SML# with LLVM: a Challenge of Implementing ML on a Common Compiler Infrastructure *

Katsuhiro Ueno Atsushi Ohori
Research Institute of Electrical Communication
Tohoku University
{katsu, ohori}@riec.tohoku.ac.jp

Abstract

We report on an LLVM backend of SML#. This development provides detailed accounts of implementing functional language functionalities in a common compiler infrastructure developed mainly for imperative languages. We also describe techniques to compile SML#'s elaborated features including separate compilation with polymorphism and SML#'s unboxed data representation.

1. Introduction

A state-of-the-art programming language must have a state-of-the-art native code generator. Since computer architectures and operating systems are evolving rapidly, a compiler would become quickly obsolete if its backend is not maintained and improved accordingly. Such development requires huge amount of efforts and expertise in the latest platforms and code generation techniques. Due to these costs, implementing a custom code generator is now unrealistic.

An ideal solution to this problem would be to use some compiler infrastructure which is maintained actively and independently of each compiler implementation by experts of code generation and optimization. Several compiler infrastructures have been developed in decades. Examples include SUIF [5] and COINS [1]. Recently, LLVM [2] is attracting attention as a promising candidate for a compiler infrastructure to implement variety of imperative programming languages, such as C and its variants, Java, Ruby, Python, Lua, and many others. When LLVM or something like it would mature, it would free compiler writers from developing a custom code generator.

Unfortunately, this story is not true for functional languages. Most of functional language developers implement and maintain their own custom code generators from scratch. This would be due to the fact that a compiler of a functional language requires its backend to provide some specific features, such as proper tail call optimization, root set management for garbage collection, lightweight unwind jumps, and so on, most of which are not adequately supported in infrastructures mainly for imperative languages.

* This work was supported by JSPS KAKENHI Grant Numbers 24700021 and 25280019.

For the glorious future of functional languages, it is desired that a common compiler infrastructure could be used for functional languages as well as imperative languages. Our general motivation is to establish techniques of implementing functional languages on top of a compiler infrastructure designed for varieties of programming languages.

To achieve this end, we have considered LLVM as a possible candidate and have developed a new compiler backend of SML#, a variant of Standard ML, with LLVM. In this paper, we report details of the development. During the development, we have overcome the following obstacles in implementing Standard ML functionalities: compiling tail calls to jumps as many as possible, determining safe points and explicit root set management for accurate garbage collection, and mapping user-level exceptions to Itanium ABI. In addition to the standard functionalities, SML# has extensions for practical use, such as natural ("unboxed") data representation, true separate compilation with linking, and seamless interoperability with C language. To implement these features in LLVM, we had to meet an additional challenge: the treatment of the polymorphism and separate compilation with unboxed data representations. We have overcome these problems and successfully completed our development of the brand-new SML# compiler working with LLVM, which is available as SML# version 2.

This attempt is not new. Some pioneering works include a LLVM backend of Glasgow Haskell Compiler [4] and Erlang [3]. We believe these and ours would contribute towards establishing techniques of implementing functional languages in a common compiler infrastructure.

The rest of this paper is organized as follows. Section 2, 3 and 4 overview notable issues on implementing ML and SML# functionalities in LLVM. Section 2 discusses the issues related on tail calls. Section 3 describes our strategy to implement an accurate garbage collection with LLVM. Section 4 summarizes our challenge of compiling polymorphic functions. Section 5 shows the resulting structure of the LLVM-based SML# compiler backend. Sections 6 concludes the paper.

2. Calling conventions and tail calls

We use `fastcc` calling convention for ML function applications rather than implementing our own custom calling convention by extending LLVM. `fastcc` is satisfactory for SML# since it is one of the fastest calling convention that is available in any platform supported by LLVM. It also enables LLVM's tail call optimization. However, LLVM's tail call optimization itself is not appropriate for our purpose; this optimization aims only to reduce the size of system stack usage so tail calls are always paired with stack frame deallocation. Implementing all ML tail calls to LLVM's tail calls disables any loop optimizations which LLVM embodies.

To generate optimal native code through LLVM, we need to compile ML function calls to explicit jump instructions as many as possible. In ML programs, which any loops are usually implemented by function applications, this requires some inter-procedural analysis in some lambda terms. So we solve this obstacle in compiler frontend phases by adding the following terms to intermediate representations derived from lambda calculus.

$$e \quad ::= \quad \cdots \mid \texttt{code}\ k\ x\ \texttt{=}\ e\ \texttt{in}\ e \mid \texttt{goto}\ k\ e$$

where x and k range over the set of variables and code labels, respectively. `code` k x `=` e_1 `in` e_2 binds k to a kind of continuation to e_1 with a formal parameter x and evaluates e_2. This k is visible in both e_1 and e_2 but invisible in nested `fn` terms. `goto` k e invokes continuation k with an argument e. These terms are direct abstractions of unconditional jumps that can be naturally translated into code labels and jump instructions.

Tail calls that are not replaced with `goto`s are treated by the LLVM's tail call optimization. We use it only as a workaround to reduce the call depth. In practice, it is effective in most cases, including large set of mutual tail recursive functions that implements a state machine, such as lexical analyzer. A radical way to reduce the call depth in any case is to add a new calling convention for functional languages to LLVM, similar to the previous works [3, 4].

3. Support for Garbage collection

To implement an accurate garbage collection, the root set should be allocated in each stack frame and traversed by scanning the system stack. To do this scan, each stack frame should be carefully designed so that the collector can determine the root set data structure in each stack frame. LLVM provides some intrinsics and a plug-in mechanism to implement some data structure for garbage collection on each stack frame through "stack map" based approach.

We have implemented our GC support as two components; one is a GC plug-in for LLVM that implements our own root set data structures in each stack frame, and another is a compile phase that computes optimal size of the root set data structure for each function and inserts `alloc`, `store` and `load` instructions that manages the root set data. This compile phase briefly performs the following two steps: firstly it computes the live ranges of all of the local variables for each function, and secondly it inserts `alloc` and `store` instruction for each local variable whose live range is lying across a GC unsafe point.

4. Separate compilation of polymorphic functions

To implement SML#'s separate compilation, we need to compile each polymorphic function as individual low-level functions that embodies their polymorphic behavior. This can be straightforward if the source language adopts the conventional "boxed semantics," whose values of any type has uniform representation. A trivial way to implement a polymorphic function with the unboxed semantics is to duplicate the function for each type instance, but this is not reasonable for SML# due to its variety of basic types. As a consequence, we need to compile each polymorphic function to single low-level function that receives and returns values of arbitrary runtime types.

Our strategy to meet this challenge is to perform boxing and unboxing on demand at runtime. A brief summary of our strategy is as follows. Each function has two code entries in low-level representation; one is the entry of original function body with an unboxed calling convention, and another is the entry of wrapper code whose arguments and return values are all boxed form. The wrapper code unboxes of all arguments, calls the original entry, boxes the result of the call, and returns the boxed result. A function closure is now 4-tuple consisting of the two code entries, a pointer to a closure environment, and a *rigidity flag* indicating whether the original entry can be called at any call site or not. The rigidity flag is true if either the function is a monomorphic function or both of the argument type and the return type of the function are not type variables. Each application term is compiled into a branch on the rigid flag in the closure. If the rigidity flag is true, then the original entry is called with unboxed arguments in a calling convention calculated from the type annotation of the application term. Otherwise, the wrapper entry is called with boxing/unboxing all of the arguments and the return value. The branch code can be eliminated by an optimization based on a static analysis.

We develop a formal scheme that performs this compilation over typed lambda terms by extending the type judgment of the form $\Gamma \vdash e : \tau$ to that of the form $\Gamma \vdash e : \tau, b$ where b is the runtime type of e. We also prove its type preservation and its soundness.

5. Overall structure of SML# backend

As a result of the above development, the LLVM backend of SML# compiler consists of the following compilation phases.

ClosureConversion It performs conventional closure conversion but the data structure of function closures are still abstracted.

CallingConventionCompile It determines the runtime type of each sub-expression and performs the compilation mentioned in Section 4. It also decides concrete data structure of closures.

ANormalize It performs A-normalization according to ML's call-by-value semantics.

MachineCodeGen It translates an A-normal form into a sequence of abstract machine code. The destination representation is similar to LLVM IR except that object allocations and manipulations are still abstracted and it keeps the nested structure of `code` and `handle` expressions.

StackAllocation It inserts explicit GC root set management code according to the storategy mentioned in Section 3.

LLVMGen It translates the abstract machine code into LLVM IR data structure.

LLVMEmit It emits LLVM IR by calling LLVM APIs imperatively.

6. Conclusions

We briefly summarized the development of our LLVM-based SML# compiler. This development provides detailed accounts of implementing Standard ML functionalities and SML#'s elaborated features in a common compiler infrastructure. We believe this work would shed some light on implementing functional languages with such an infrastructure.

References

[1] COINS compiler infrastructure. http://coins-compiler.sourceforge.jp/.

[2] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. the 2004 International Symposium on Code Generation and Optimization*, pp. 75–88, 2004.

[3] K. Sagonas, C. Stavrakakis, and Y. Tsiouris. ErLLVM: An LLVM backend for Erlang. In *Proc. Eleventh ACM SIGPLAN Workshop on Erlang Workshop*, pp. 21–32, 2012.

[4] D. A. Terei and M. M.T. Chakravarty. An LLVM backend for GHC. In *Proc. Third ACM Haskell Symposium on Haskell*, pp. 109–120, 2010.

[5] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S. -W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices*, 29(12):31–37, 1994.