A LLVM Back-end for MLton

by

Rahul Dhawan
rdhawan201455@gmail.com

Report submitted to
IIT Palakkad

in partial fulfilment for the award of the degree of

BACHELOR OF TECHNOLOGY
in Computer Science

SUPERVISED BY
Dr. Piyush P. Kurur

Department of CSE
Ahalia Integrated Campus,
Palakkad Dist., Kozhippara, Kerala 678557, INDIA

April 2019

Abstract

An LLVM Back-end for MLton

Rahul Dhawan

Supervising Professor: Dr. Piyush P. Kurur

This report presents the design and implementation of a new LLVM back-end for the ML-ton Standard ML compiler. The motivation of this project is to utilize the features that an LLVM back-end can provide to a compiler, and compare its implementation to the existing back-ends that ML-ton has for C and native assembly (x86 and amd64). The LLVM back-end was found to offer a greatly simpler implementation compared to the existing back-ends, along with comparable compile times and performance of generated executable, with the LLVM-compiled version performing the best on many of the benchmarks. The LLVM IR can used for optimization of source code and JIT(Just In Time).

Contents

Chapter 1

Introduction

Today's world of computing is undergoing a constant, yet silent revolution. Both software and hardware systems - from embedded micro-controllers through to massively parallel high-performance computers - are experiencing an ever-increasing degree of sophistication and complexity. Although it never actually comes to the fore, compiler technology plays a central role in this revolution as the junction between software and hardware. Traditionally, a compiler has to fulfill three main requirements:

- It has to generate efficient code for the target platform.
- It should consume only a reasonable amount of time and memory.
- It must be reliable.

Compilers are generally architected in a three-phase design: The front-end, optimizer and back-end, shown in figure 1.1. The front-end is responsible for the lexing, parsing, and type checking of the source code, transforming it into an abstract syntax tree (AST), which acts as an intermediate representation (IR) in the compiler. The optimizer improves the efficiency and performance of this intermediate representation by transforming the code to simpler yet semantically equivalent versions, possibly using different representations if it is useful to do so. In the back-end, the code is emitted to an executable

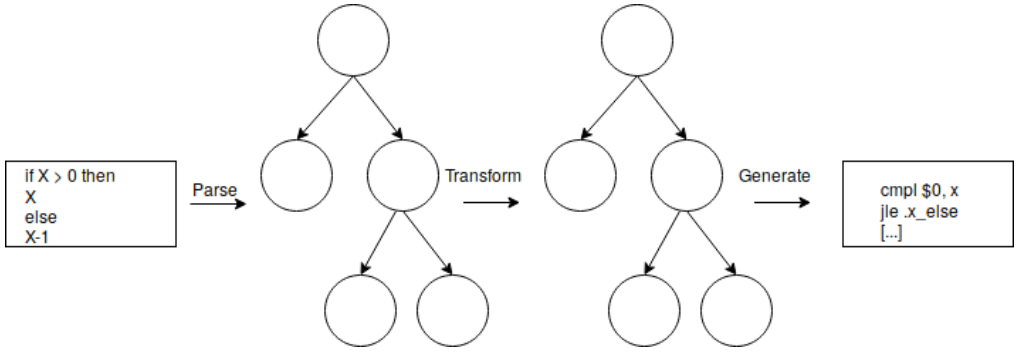

Figure 1.1: General compiler pipeline

form, usually as either machine code that can be directly run on hardware, or byte code that can be run on a virtual machine.

Because of the split between the different components of a compiler, it is possible for multiple compilers for different languages to share the same back-end system. One of the long-standing issues in compiler design is how to best solve the challenges in utilizing a common back-end technology. Compiler developers want to ensure the compiled programs perform the best that they can, but they also want to leverage language-agnostic tools and libraries that free them from being concerned with the low level details required to make the compiler generate high-performance executables. One such project that solves this issue effectively is LLVM, which defines a high-level, target independent assembly language that can be aggressively optimized and compiled to several different architectures

This report examines an LLVM back-end for MLton and conversion of LLVM AST into the LLVM IR source code. MLton5 is written primarily in SML with a runtime system written in C, and is able to self-host. MLton's features include support for a large variety of platforms and architectures, the ability to handle large, resource intensive programs, and aggressive compiler optimizations that lead to efficient programs with fast running times. The MLton compiler currently has three back-ends: C,

x86, and amd64. The C back-end emits the compiled program as C code, and uses an external C compiler to compile to native code. The x86 and amd64 back- ends, known together as the native back-ends, emit assembly language directly which is then assembled by the system assembler into native code. The native back-ends offer better performance and compile times, but have a limited set of supported platforms.

In this report, we will be looking at the design and implementation of the new back- end for the MLton Standard ML compiler, using the LLVM IR as a target and Design and implementation of an algorithm to convert the LLVM ast into LLVM IR.

The rest of the report is organized as follows. Chapter 2 goes over the design of MLton and the techniques it uses to compile Standard ML programs. It also goes over the design of LLVM, and how it can be used as a compiler back-end. Chapter 3 describes the design and implementation of the LLVM back-end, going over the design choices made and the strategy used for translating to LLVM. Chapter 4 goes over the design and implementation of the algorithm to convert the LLVM AST into LLVM IR. Chapter 5 gives concluding remarks for this project and ideas for further work

Chapter 2

Background

2.1 Back-end Targets

A big challenge in the design of compilers for high-level languages is finding the best way to implement the compiler's back-end. Compiler developers want to use a technique that allows the generated executable to run as efficiently as possible, as that is an important factor in judging the quality of the compiler. However, they also want to minimize the effort in implementing the back-end, ideally by sharing infrastructure with other compilers. Due to many compiler writers prioritizing the former, this challenge has led to a situation where the popular implementations of languages like C, Java, Python, and Haskell share little or nothing in common.

A big challenge in the design of compilers for high-level languages is finding the best way to implement the compilerâĂŹs back-end. Compiler developers want to use a technique that allows the generated executables to run as efficiently as possible, as that is an important factor in judging the quality of the compiler. However, they also want to minimize the effort in implementing the back-end, ideally by sharing infrastructure with other compilers. Due to many compiler writers prioritizing the former, this challenge has led to a situation where the popular implementations of languages like C, Java, Python, and Haskell share little or nothing in common.

Major factor in the design of back-end of any Language is the kind of language the back-end will be going to target. achievable targets can be categorized into four categories:

- **Naive Assembly** : This is one of the genuine preferred choice, as it offers the compiler writer the most control over how the compiled executable is written, and it control the dependence on external tools as all you need is the system assembler. However, this way takes the most amount of effort by the compiler writers to implement and maintain. Assembly languages are complex and disclose many low level details, so it takes a considerable amount of effort to write an effective implementation. Also, an assembly back-end is target specific, so adding support for a new architecture in a compiler requires a lot of effort. .
- **High Level Language** : Compilers can produce to a different high-level language, using compilers for that language as foreign tools to complete the compilation process. Most often the language is C , because it is low level enough to not interfere too much with the semantics of the source language or final IR of the compiler, and because the language has attractive performance and limited runtime overhead. Also, this way allocate flexibility for free due to the presence of C compilers across most computing platforms. This approach still has its bugs, due to lack of fine control of code generation details such as tail calls, and longer compilation times due to having to parse and compile source again as part of the back-end stage.
- **Managed virtual environments** : A popular choice for programming languages in the past couple of decades is to compile to a high-level and portable bytecode format, and at run-time have it execute on a virtual machine. This has been the choice of execution model for modern compiled languages like Java and C sharp, and for scripting languages

like Python and Lua. To help overcome the performance penalty of executing on an interpreter, the virtual machines often use just-in-time (JIT) compilation which compiles the executing code to native machine code as it gets executed. The benefits of this technique include portability on all platforms the virtual machine runs on, and allowing code to take advantage of existing libraries and rich runtime features provided by the platform such as garbage collection and excep- tion handling. However, this approach generally suffers from worse performance compared to compiling to native assembly, and can raise issues when the runtime features of the platform do not match up nicely to the runtime features needed by the language. For example, the garbage collection techniques may not work well based on the style and frequency in which the language allocates objects, or the exception handling system on the virtual machine may not match the semantics of exceptions in the language.

- **High Level Assembly** : The final alternative is High level assembly. High level assembly languages are low-level enough to not interfere with the abstractions in high level programming languages, but also high-level enough to abstract away the very low level details of assembly language such as register allocation and instruction scheduling. They also have the ability to be optimized in both target-independent and target dependent ways, producing high-performance executables. One such language that implements all of these features is the LLVM IR.