A LLVM Support 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 support 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. The LLVM back-end was found to offer a greatly simpler implementation compared to the existing back-ends. The LLVM IR can used for optimization of source code and JIT(Just In Time). So LLVM can be used as optimized code.

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 form, usually as either machine code that can be directly run

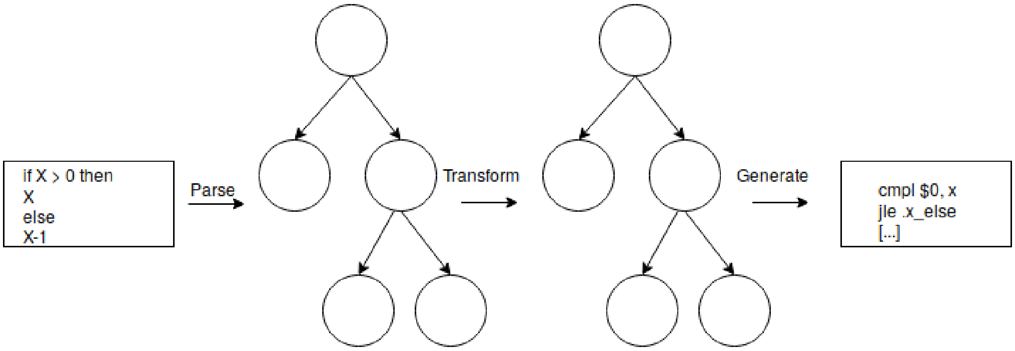

Figure 1.1: General compiler pipeline

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 a conversion of LLVM ast into LLVM binary executable and conversion of LLVM AST into the LLVM IR source code. MLton 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 Background and Related work . Chapter 3 will cover the capturing of LLVM AST. Chapter 4 describes the design and implementation of the algorithm used for the conversion of LLVM AST into LLVM Binary executable . Chapter 5 goes over the design and implementation of the algorithm to convert the LLVM AST into LLVM IR. Chapter 6 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 compiler's 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:

- **Simple 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.

- **Virtual environments** : A popular choice for programming languages in the past couple of decades is to compile to a high-level and portable byte code 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 run-time features provided by the platform such as garbage collection and exception handling. However, this approach generally suffers from worse performance compared to compiling to native assembly, and can raise issues when the run-time features of the platform do not match up nicely to the run-time 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 executable. One such language that implements all of these features is the LLVM IR.

This project adds to LLVM back-end for MLton with the following supports:

- Converting the LLVM AST into LLVM IR which can be used for many useful purposes and that will help in optimizing the actual source code.
- Converting the LLVM AST into LLVM binary executable which

can also can be optimized and directly run with efficiency than running directly.

2.2 What is LLVM ?

LLVM (which originally stood for Low Level Virtual Machine) is a compiler infrastructure project that defines an intermediate representation(IR) that compilers can use to produce high-performance native code for a target platform. The IR is designed to be both language-agnostic and target-independent, which allows compilers for different languages to take advantage of sharing common functionality for back-end work such as optimizing and target-specific code generation. The LLVM project was started in 2000 by Chris Lattner as his master's thesis. In 2003 it was made open source, and has been continually developed with support from companies like Apple, Google, Intel, Adobe, and Qualcomm. LLVM was the recipient of the 2012 ACM Software System Award because of its success and influence, and its high quality design and implementation. One of the first practical applications of the LLVM was the llvm-gcc project, which retrofitted a LLVM back-end on to GCC. (The llvm-gcc compiler is now deprecated, but the idea behind the project lives on in its successor project called Dragonegg, which uses the plugin system found in the newer versions of GCC).

A major design decision for LLVM that contributes greatly to its flexibility and power is its library-oriented design. LLVMâĂŹs design philosophy is to design a multitude of independent libraries that serve a specific purpose and are loosely coupled from each other, 20 allowing clients to easily choose just the parts they need for the features they want implemented. This also makes it easy to make a toolchain of command-line programs (shown in figure 2.1) that serve as driver programs for certain parts of the LLVM system. The LLVM back-end for MLton

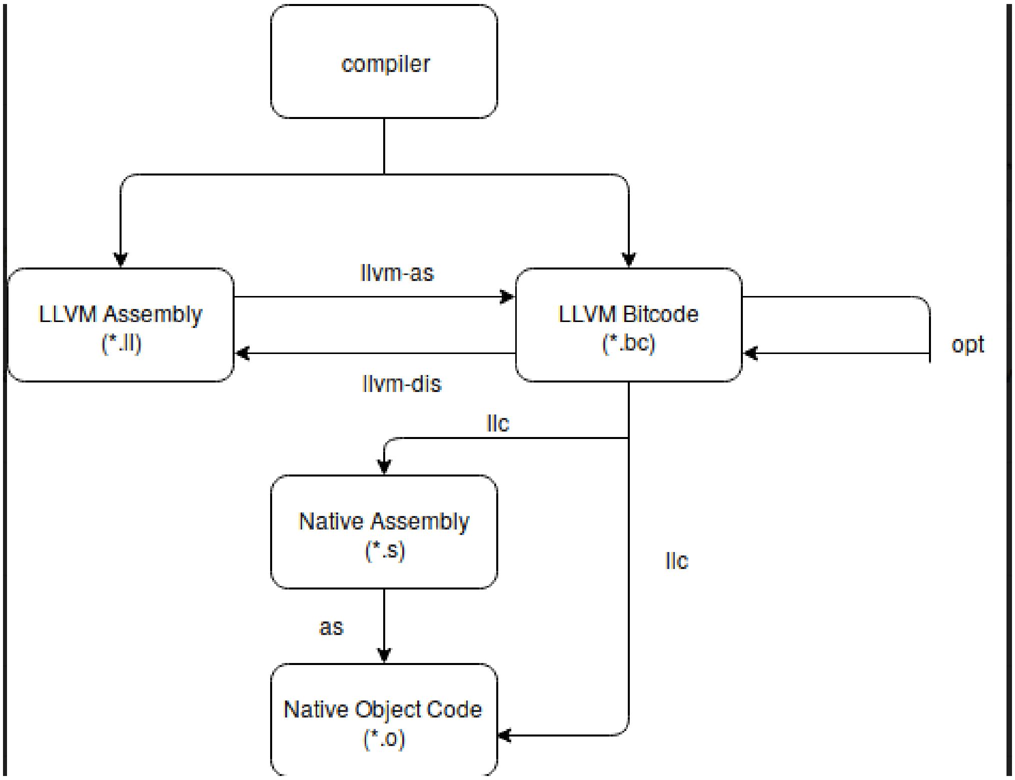

Figure 2.1: LLVM toolchain and compilation process

uses opt, the LLVM optimizer, and llc, the LLVM static compiler, to optimize and compile the LLVM IR code to either native assembly or an object file.

2.2.1 Representation

Another key feature of LLVM is that the IR has three first-class representations that can easily be converted to other forms without any loss of information. These forms are:

- A textual, human-readable assembly language. These are usually kept on disk in a file with suffix .ll.
- A binary format which is also usually kept on disk but is much more compact, in files with the .bc suffix. LLVM provides the command line tools llvm-as which assembles LLVM assembly to LLVM bit code, and llvm-dis which disassembles

LLVM bit code into LLVM assembly.

- An in-memory symbolic representation of the IR that the LLVM libraries use for analysis, optimization, and native code generation. The libraries include functions for reading in from LLVM assembly or bitcode, and also include functionality for creating and manipulating LLVM IR directly, for use by back-ends that use the LLVM libraries directly to generate code.

2.2.2 SSA - Static Single Assignment

LLVM can be described as a mid-level IR, as it offers a higher level of abstraction than typical assembly languages.An important property of LLVM's design is that its instruction set is in static single assignment (SSA) form. In SSA, instructions that produce values assign to a register that can only be written to once. SSA form helps simplify data-flow analysis, which makes it easier to implement more complex and powerful optimizations based on data-flow analysis.

The primary usefulness of SSA comes from how it simultaneously simplifies and improves the results of a variety of compiler optimizations, by simplifying the properties of variables. For example, consider this piece of code:

```
y =: 1
y =: 2
x =: y
```

Humans can see that the first assignment is not necessary, and that the value of y being used in the third line comes from the second assignment of y. A program would have to perform reaching definition analysis to determine this. But if the program is in SSA form, both of these are immediate:

```
y1 =: 1
y2 =: 2
x1 =: y2
```

2.2.3 Types

LLVM has a type system that allows many optimizations to be performed more easily on the IR without any extra analyses.This type system by and large is based off of C's type system.There are two categories of types, primitive and derived, described in table 2.1.

Type	Syntax	Description
Integer	i1,i2, ...,i32, ...	Integer type of arbitrary bit width
Floating Point	float ,double,...	Floating point types of different standards
Void	void	Void type for functions that do not return a value
Label	label	Represents code labels for basic blocks
Function	i32 (i8*, ...)	Block which take arg and output the result
Pointer	[4 x i32]*	Used to specify memory locations

Table 2.1: Description of LLVM types

2.2.4 Instructions

LLVM's instruction set is RISC-like, with each instruction performing relatively simple operations on its operands. This section will give the complete information about the Instruction set and the type of instruction in the instruction set, which will help in making the design of the algorithm to convert the LLVM

AST into the LLVM IR and LLVM AST into the LLVM binary executable. For a complete reference go to the LLVM Language Reference Manual.

Terminator Instruction

Every basic block in a program ends with a "Terminator" instruction, which indicates which block should be executed after the current block is finished. These terminator instructions typically yield a ’void’ value: they produce control flow, not values (the one exception being the ’invoke’ instruction).

The terminator instructions are: ‘ret’, ‘br’, ‘switch’, ‘indirectbr’, ‘invoke’, ‘callbr’, ‘resume’, ‘catchswitch’, ‘catchret’, ‘cleanupret’, and ‘unreachable’.

- **ret :** The ’ret’ instruction is used to return control flow (and optionally a value) from a function back to the caller.
- **br :** The ‘br’ instruction is used to cause control flow to transfer to a different basic block in the current function.
- **switch :** The ‘switch’ instruction is used to transfer control flow to one of several different places. It is a generalization of the ‘br’ instruction, allowing a branch to occur to one of many possible destinations.

Binary Operations

Binary operators are used to do most of the computation in a program. They require two operands of the same type, execute an operation on them, and produce a single value. The operands might represent multiple data, as is the case with the vector data type. The result value has the same type as its operands.

There are several different binary operators:

- **add, sub, mul, udiv, sdiv, urem, srem :** Perform addition, subtraction, multiplication, and division on two integers. The sdiv and udiv instructions compute the unsigned or signed quotient, and urem and srem computes the unsigned or signed remainder of their operands. These instructions can also handle vectors of integers.
- **fadd, fsub, fmul, fdiv, frem :** Similar to the above operations, but for floating-point types or vectors of floating-point values.

Bitwise Binary Operations

Bitwise binary operators are used to do various forms of bit-twiddling in a program. They are generally very efficient instructions and can commonly be strength reduced from other instructions. They require two operands of the same type, execute an operation on them, and produce a single value. The resulting value is the same type as its operands.

- **shl, lshr, ashr :** Perform a left-shift, logical right-shift, or arithmetic right-shift operation respectively on the first operand by shifting the number of bits specified by the second operand.
- **and, or, xor :** Perform bitwise "AND", "OR", or "XOR" operations respectively on the two operands.

Aggregate Operations

LLVM supports several instructions for working with aggregate values.These operations are for manipulating aggregate values such as arrays and structs directly. These are generally rarely used as aggregate values tend to exist in memory rather than in registers, and are thus manipulated by memory access instructions.

- **extractvalue :** Extracts a value in an aggregate value specified by an index.
- **insertvalue :** Inserts a value into a location in an aggregate value, specified by an index.

Memory Access and Addressing Operations

A key design point of an SSA-based representation is how it represents memory. In LLVM, no memory locations are in SSA form, which makes things very simple. This section describes how to read, write, and allocate memory in LLVM.These operations are for manipulating memory. Recall that in LLVM, only registers are in SSA form, so while the pointers contained in registers are immutable, all locations in memory are mutable.

- **alloca :** The 'alloca' instruction allocates memory on the stack frame of the currently executing function, to be automatically released when this function returns to its caller. The object is always allocated in the address space for allocas indicated in the datalayout.
- **load :** The 'load' instruction is used to read from memory.
- **store :**The 'store' instruction is used to write to memory.
- **getelementptr :** Gets the address of a sub-element of an aggregate value in memory offsetting a given pointer operand with one or more indices. This operation per- forms address calculation only and does not access memory. The first operand is the pointer to be indexed. The second operand is an index for the pointer. Optional additional indices may follow to do further indexing if the pointer operand points to an aggregate value. This operation is generally used for getting the pointer to a member of an aggregate value to be used by a later load or store instruction, but this can be used for basic pointer arithmetic as well.

Conversion Operations

These operations convert values of one type to another, either by reinterpreting the value to be a different type (thus performing a no-op cast), or by casting it to a different type while keeping the value relatively the same by rounding.

- **truc, zext, sext :** Converts an integer to a different bit-width by truncating, zero-extending or sign-extending
- **fptruc, fpext :** Converts a floating-point value to a different floating-point type by truncating or extending.
- **fptoui, fptosi :** Converts a floating-point value to its unsigned or a signed integer equivalent.
- **uitofp, sitofp :** Converts an unsigned or signed integer to its floating-point equivalent.
- **ptrtoint, inttoptr :** Converts values between integer and pointer types. If the integer is not the same bit-width as a pointer for the target platform, it will be truncated or zero-extended.
- **bitcast :** Converts a value from one type to another without changing any bits, meaning this is always a no-op cast. Both types must have the same bit-width. If the source type is a pointer, the destination type must also be a pointer.

Other Operations

These are operations that do not fit in any of the categories above.

- **call :** Calls a function defined or declared elsewhere in the current module. Any necessary arguments depend on the function being called, and this may assign to a register if the function returns a value.

- **icmp, fcmp :** Compares two integer or floating-point values of the same type. This operation takes an additional argument indicating what kind of comparison to do, e.g. eq for equality, lt for less-than. The result is a value with type i1, which can be used in the br instruction.

2.3 Optimization of LLVM IR

The dominant aspect of LLVM is its ability to be optimized, both with target-independent optimizations where the IR is transformed to a extra efficient but semantically comparable version, and target-dependent optimizations which occur when translating the IR to assembly language or machine code. Optimizations are managed by having them be written as a Pass, which is code that performs analysis and transformations on an LLVM Module and some or all of its substructures. Passes are completely modular; they are designed to perform one type of analysis or transformation, and multiple passes can easily be applied to the same module in any order. This flexibility allows for passes to be easily added, removed, and replaced in an optimization pipeline.

LLVM passes can apply to module in its in-memory form by using the Pass-Manager API, or be applied to LLVM assembly or bit code files using the command-line opt tool. With opt, optimizations can select from a large collection that is included with LLVM by using a command-line flag. For example, applying the "simplifycfg" pass on prog.ll, opt would be invoked as opt -simplifycfg prog.ll -o prog.ll. Passes can also come come from a dynamic library that implements a pass by using the -load option and the library file. LLVM includes standard optimization sequences with the -ON flag, where N is a number between 1 and 3 indicating the optimization level. A higher number enables more aggressive optimizations but may take longer to run.

LLVM also supports link-time optimization (LTO), which allows for additional optimization opportunities for programs that span multiple modules. LTO works by having a LTO-aware linker use LLVM bit code files instead of normal object code files for linking.

2.4 Related Work

The LLVM project has been a focus of active research and development in the compiler community, and because of its modular and reusable design, it has been easy for external projects to take advantage of the features that the project provides. Some of these projects retrofit an LLVM back-end on an existing functional language compiler similar to the goal of this project.

- LLVM bindings for Haskell.
- LLVM bindings for OCaml.

Chapter 3

Capturing LLVM AST