LLVM Bindings for SML

A B. Tech Project Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Bachelor of Technology

by

Rahul Dhawan
(111501023)

under the guidance of

Dr.Piyush P. Kurur

IIT PALAKKAD

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY PALAKKAD

CERTIFICATE

This is to certify that the work contained in this thesis entitled LLVM Bindings for MLton is a bonafide work of ***Rahul Dhawan (111501023)****, carried out in the Department of Computer Science and Engineering, Indian Institute of Technology Palakkad under my supervision and that it has not been submitted elsewhere for a degree.*

Dr. Piyush P. Kurur
Department of Computer Science & Engineering
Indian Institute of Technology Palakkad

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.

Acknowledgment

The present work is an effort to throw some light on "LLVM Back-end for SML". The work would not have been possible to come to the present shape without the able guidance, supervision and help to me by Dr. Piyush P. Kurur. With the deep sense of gratitude I acknowledge the encouragement and guidance by my organizational guide. I convey my heartful affection to all those people who helped and supported me during the project, for completion of my project report.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As the optimization of the code is a hot topic in the world, everyone is try to optimize the source code so that it will run with less memory and in the less time in comparison to the memory and the time consumed by the original source code. For the above problem people started to convert the source language code into some other Intermediate Representation, that can be used be for optimization because some IR(Intermediate Representation) can be optimized more than the original once or they can be optimized more efficiently than the original source code. All the above process is going to be done by the compiler and it is only going to depends on the language in which the compiler is going to be written. Because if that language in which the compiler is written is going to have the support for the conversion of the Source code to the some IR language. Some of the language is having support for converting the source language into some IR language.

For the conversion we need to use FFI calls, FFI stands for the foreign function interface which means that we can call for function written in some other language from some other language to use the options provided in the other language because that may not be provided in the language so better to call the other language to use property. In SML there is not interface for LLVM so it is not that easy to direct convert the LLVM AST into the

LLVM binary executables for that we are using the C interface of LLVM to convert the AST because SML has FFI support for C so it is easy to convert.

One of the IR(intermediate representation) known is LLVM IR which is used by many languages to convert the LLVM AST into LLVM binary executables or LLVM IR which can be used to optimize. The example of these languages are :

- **C** ———- Imperative Programming Language
- **C++** ———- OOP (Object Oriented Programming) Language
- **Haskell** ———- Functional Programming Language
- **OCaml** ———- Functional Programming Language

In the both functional programming language they are calling the FFI either to C or C++ because both of the language C and C++ have the LLVM interface.

Here to be noticed that many of the functional programming languages have support of LLVM IR to convert like haskell or OCaml. But SML(Standard Meta Language) do not have this support of LLVM IR till now. In the project we are trying to make the components of the Bindings for the SML bindings for LLVM. Mainly the bindings has main part in the Conversion of the captured IR AST corresponding to the original source code into IR binary executables or it has other option which is not use much that is to convert the captured IR AST corresponding to the original source code into IR readable which is also called pretty printing.

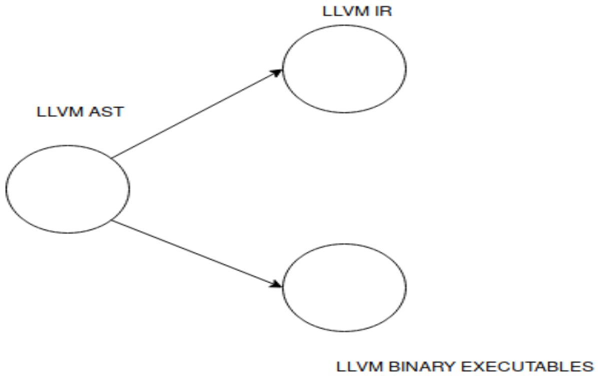

Figure 1.1: Conversion

For doing all of these we have just want to do the below listed things :

- Capture the LLVM AST.
- Converting the LLVM AST into LLVM binary executables.
- Converting the LLVM AST into LLVM IR.

In the first part the capturing the LLVM AST is not so hard we just have to capture the types that is needed or support in the LLVM , and in the third part itâĂŹs just the thing we have to match the LLVM AST type capture by us to the LLVM IR syntax that is known by us. But the main things that we have to worry or to concentrate on it is using FFI(Foreign Function Interface) calls because we can not convert the LLVM AST into the LLVM binary executable so easy, for that we have to call c interface of LLVM to convert the LLVM AST into the corresponding LLVM binary executables.

All these LLVM IR or LLVM binary executables can be used for optimization.The conversion shown in the Figure 1.1.

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 compilerâĂŹs backend. 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. ThatâĂŹs the main point that all the compilers are mainly using that to convert into the other IR (Intermediate Representation) Language , which will execute in less time and uses less memory in comparison of the originally source code memory and time taken. But the main question is in which format does we convert into it ? because there are two types that are mainly common that are :

- **High Level Language :** This option is not used very often because to convert from one source code to another source code and then used other compiler to do things again and again will take more time but it will consume less memory. As a fact that it will consume more time people not often used it. It will take time less in execution but the time consider here is to do whole things at one.

- **High Level Assembly :** This is the option that all the developers or others used more often because here the two facts that is time and the memory will be less in comparison to the original source code time and memory. As a fact that it will use less memory and time people use often.

As of need we are implementing the both the option in the project. The one thing should be known that these two options are convertible from each other by using the LLVM tools.This project adds to LLVM back-end for MLton with the following supports:

- LLVM AST -> LLVM IR
- LLVM AST -> LLVM Binary Executables

2.2 What is FFI Calls?

Now as we are converting the LLVM AST into LLVM Binary executables we need to to do FFI calls as discussed in the introduction section.Basically what is FFI calls ?

A **foreign function interface (FFI)** is a mechanism by which a program written in one programming language can call routines or make use of services written in another.The term comes from the language common lisp which means that to make the inter-language calls , this is also used in the Haskell, Ocaml and in others.

The primary function of the FFI is to use the functions and the support provided in the language by the other language which will define the FFI using the convention and the semantics of the other language. Here we are using FFI calls to C support of LLVM from the SML to convert.

For more information visit http://mlton.org/ForeignFunctionInterface

2.3 What is LLVM ?

As we are capturing the AST for conversion and all. So one should know about the LLVM to capture it.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 most amazing property about the LLVM AST is that it is SSA (Stactic Single Assignment) which means that the variable can be used for reading as many times as you can but it can only be assign once in a period or in a code.

The more about LLVM and LLVM AST can be found on their official website.

2.4 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.

LLVM also supports link-time optimization (LTO), which allows for additional optimization opportunities for programs that span multiple modules.

2.5 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

An AST(Abstract Syntax Tree) is the tree representation of the abstract syntax structure of source code written in any programming language. Each node of the tree represent the part of the source code which is going to construct the instruction. The syntax in the tree is abstract it does not contains all the parts of the source code instead of that it only contains the useful part which is sufficient to represent the source code. For e.g.

```
If x > 0
Then y
Else
Then Z
```

So the capturing of the AST is the first step for making the

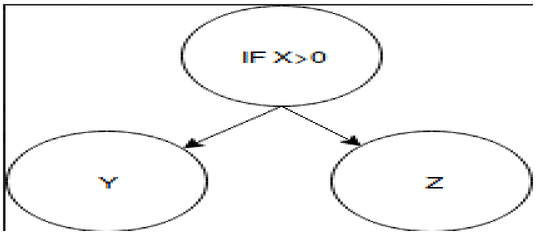

Figure 3.1: Example

backend for LLVM. LLVM AST is the different from the other AST like C, python and all other language. In the LLVM AST the things one have to know before capturing it are :

- **Module :** LLVM programs are basically made up of module which are generally contained the functions, global variables, and symbol table entries. There can be more than one module and the two module can be linked by the LLVM Linker, which merges the component of the two module like function, global definitions and all the variables used in those modules. In general, a module is made up of a list of global values (where both functions and global variables are global values).

- **Basic Block :** Each Module has more than one basic block. Basic block is a package or in a simple language it consist of local variables and a instruction. Every Basic Block has a Entry point and a exit point which is known as terminators. Basic Block is a type of a function which has a entry and exit.

- **Global Variable :** Global Variable in LLVM IR is represented by "@" followed by the name of the global variable name.It can be accessed from anywhere in the context.

- **Metadata :** LLVM IR allows metadata to be attached to instructions in the program that will give the extra valuable information about the code to the optimizers and code generator. All metadata are identified in syntax by a exclamation point ('!').

  A metadata string is only a string covered by double quotes. For example: "!"test/00"", here 00 is hexadecimal number and it can be any.

  Metadata nodes can have any values as their operand. For example:

```
!{ !"test\00", i32 10}
```

- **Types :** The LLVM type system is one of the most important features of the intermediate representation. As a typed enables a number of optimizations to be performed on the intermediate representation directly.Types in the LLVM IR is same as expressed in the types in the LLVM website. For e.g. Int, Float, Double, void etc.
- **Operand :** An Operand is roughly that which is an argument to an Instruction or we can say to a function. Operand can be of three types basically in the LLVM IR which are :
  1. Local Reference
  2. Constant
  3. Metadata

The above mentions are the things that we need to capture in order to capture the AST of the LLVM. After capturing the above things one can represent any LLVM IR code by the LLVM AST capture by us, it will then help us in optimizing the source code further if we want because optimization is the one thing that one can do in LLVM IR.

Till now, first part of the project was to understand the LLVM and LLVM IR language and it's implementation. The next step was to understand how the bindings are implemented for other language like Haskell, OCaml.Then the next step was to capture the LLVM AST to represent the LLVM IR and to use it further. Now our next step is to design and implement the code/algorithm to convert the LLVM AST into the LLVM AST into LLVM binary executable.

Chapter 4

LLVM AST Into LLVM Binary Executables

The LLVM bindings for other functional programming language like Haskell, Ocaml etc have something in common that every language bindings used to call the C API of LLVM or C++ API of LLVM by FFI(Foreign Function Interface). The call by FFI to C API or C++ API is not directly just by the things capture by us in the AST like module, Basic Block etc because the FFI is just only calls by strings , int , and other types so we have to pass the all the things like module , basic block and all other types as the FFI support and then again made it on the other side form the supported type to LLVM Module , LLVM Basic Block etc then call the LLVM C API or LLVM C++ API to convert the AST that we made into the LLVM Binary executable.

For the LLVM bindings for SML we have to use LLVM C API because SML has the FFI With the C. The FFI supports only the passage of the following type Int, Float, String or Pointer so in SML we have to convert the capture AST into the types that is allowable to pass through FFI calls. For that we are passing the following things:

- Module Name -> string
- Array of the Instruction -> Array of string

  Global Variables

  Function

  Instructions of many type like arithmetic, logical, comparitive, calls,load , alloca, store etc.
- Length of the array -> int
- -1 -> string

4.1 Why this??

Why we are using this approach because we can not send directly the modules and the basic block directly through FFI calls so the approach left is to send the module Name separate because to create module at starting before doing anything because we need module in LLVM C API to do anything and then array of instruction in which every things have given ID's to be identified on the other side and then after identifying the type of instruction we have to build the instruction from the string back to AST by using LLVM C API Instruction builder available.

4.2 Work to do

As we have to pass only the parameters available in FFI calls so we have to do some changes in both side that is in sml and in c.

4.2.1 In SML

In sml side we have to give ID's and the parameters to build the same AST on the c side by using these parameters :

- **ID 1 :-> GLOBAL VARIABLE**
  - Followed by the name of the variable and then by the type of the variable
    - **INT**
    - **FLOAT**
- **ID 2 :-> FUNCTION**
  - Followed by following things:
    - **Name of the function**
    - **Return type of the Function**
      INT
      FLOAT
      VOID
    - **Number of parameters**
    - **Parameters type**
      INT
      FLOAT
    - **Parameters Name**
    - **Number of Basic Block**
    - **BB Name**
    - **Instruction inside the Basic Block**
      - **NAME of the BB**
      - **Number of Instruction Inside the BB**
      - **Instructions**
        ADD : (ID = 3) followed by operands
        SUB : (ID = 4) followed by operands
        MUL : (ID = 6) followed by operands
        DIV : (ID = 5) followed by operands
        ICMP : (ID = 7) followed by operands
        AND : (ID = 8) followed by operands

OR : (ID = 9) followed by operands
XOR : (ID = 0) followed by operands
ALLOCA : (ID = 1) followed by operands
STORE : (ID = 2) followed by operands

- **Terminator**

  RET : (ID = 1) followed by Return value

  BR : (ID = 2) followed by Name of the destination (BASIC BLOCK)

  COND : (ID = 3) followed by condition on which we have to check , name of the true destination basic block, name of the false destination basic block

4.2.2 In C

In the c side we have to just make it back to AST by using LLVM C API tools and then call the LLVM tool to convert the whole module into the binary files.

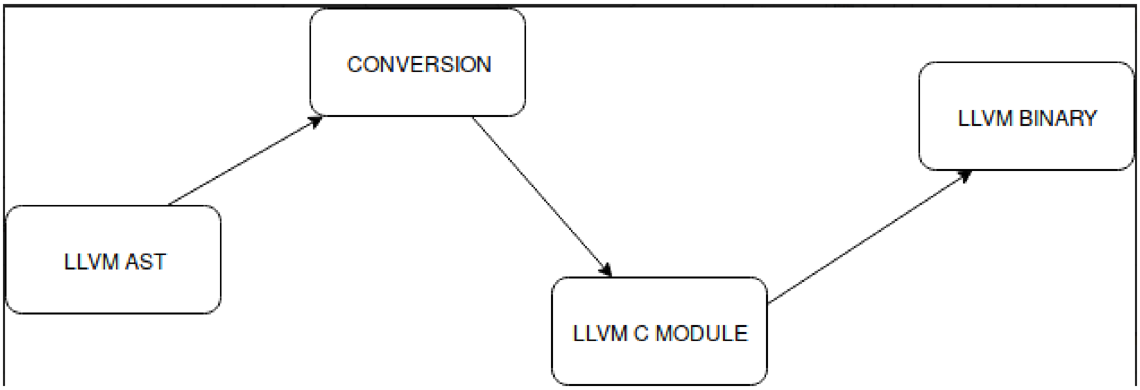

Figure 4.1: Process of Conversion

4.3 Commands to Run

The below command is to make the call and make the file executable:

```
./make.sh
```

After this it will convert the AST into Binary Executable.

4.4 Verification

To verify that we have done the right conversion we have to make back the LLVM IR from the LLVM binary executables by using LLVM tools called llvm-dis and then we can verify then it's correct by comparing the two.

4.4.1 Commands to verify

```
llvm-dis file_name
```

Now we will have the LLVM IR code then we can verify it by comparing the two.

Chapter 5

LLVM AST INTO LLVM IR

5.1 Introduction

As in the previous chapter we build the tool that will take LLVM AST of a LLVM IR code and output the LLVM Binary executable corresponding to the LLVM IR that we input. As also mention in the chapter 2 that the target available for the back-end for any language are of four types only two are efficient from the fours, that are converting any AST into the any HIGH LEVEL Language or into the any HIGH LEVEL Assembly. So we did the second part as explain in the previous chapter and we can get the first part also done just by using LLVM tools but we have to do it by pure SML, in simple language all things should be done by the SML, so for that we have to design and implement the tool written in SML that will take the any LLVM AST and then convert the input into the LLVM IR code corresponding to the input LLVM AST.

The conversion that we are doing is also known as pretty printing. Pretty printing is just the reverse process of generating any AST of any language from the an source code of any language. So it can be view as in fig 5.1.

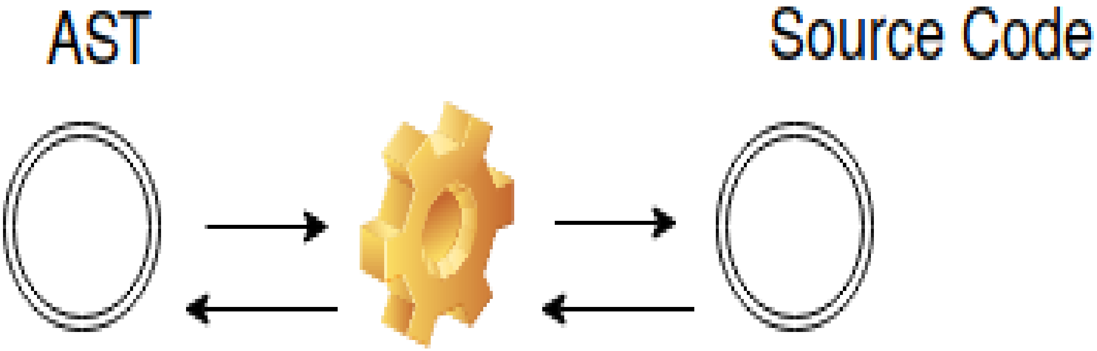

Figure 5.1: Process of Conversion

5.2 Tasks to do

As we are writing in pure SML so we have to do some tasks to make it done. The main task is in this process is to decode all type of instruction that is coming from the LLVM AST. Those things are the following:

- **Module :** As discussed in the chapter 3 the LLVM programs are composed of Module's, each of which is a translation unit of the input programs. Each module consists of functions, global variables, and symbol table entries. Modules may be combined together with the LLVM linker, which merges function (and global variable) definitions, resolves forward declarations, and merges symbol table entries.

- **Global Variable :** In the Global Variable we have to decode the two things that are:

  Type : Type can be of any supported type.

  Initializer : Initializer is the initial value of the global variable.

- **Function :** In the function we have to decode the following listed things:
  - **Return Type :** Return type can be decoded as of two types
    * Int
    * Float
  - **Visiblity :** Visibility of the function can be of two types local or global
    * Global (@) : It define the scope of the function to be globally, it can be access from everywhere where we want to access.
    * Local ("%") It define the scope of the function to be globally, it can be access from the context in which it define.
  - **Name :** Name is the name of the function which have to decode to get the name of the function.
  - **Parameters** : Parameters are neede to decode to get the parameters of the function
    * Type : Type can be decoded as of two types

      Int

      Float
    * Name : Name is the name of the Parameters which have to decode to get the name of the Parameters.
  - **Basic Block** : Basic block should be decode to get the instructions inside the function.It is described below.
- **Basic Block :** Basic block can be decode as follows :
  - **Entry :** Decoding entry will give us the name of the entry to the basic block.
  - **Instruction :** Decoding instruction will give us the type of instruction inside the basic block:

    Add : Arithmetic operation.

Sub : Arithmetic operation.
Mul : Arithmetic operation.
Div : Arithmetic operation.
Eq : Logical operation.
NE : Logical operation.
UGE : Logical operation.
UGT : Logical operation.
ULT : Logical operation.
ULE : Logical operation.
SGT : Logical operation.
SGE : Logical operation.
SLT : Logical operation.
SLE : Logical operation.

- **Terminator :** Decoding terminator will give us the terminator of the basic block and it can be of following types :

  Br : Break.
  Ret : Return.

And one thing to be noticed that by using these all we can compose the if-else or for loop with them.

5.3 Commands To Run

To run the tool that we just made we have to execute the following commands :

```
sml file_name.sml
```

After running this command we will get the .ll file corresponding to the input AST file.

5.4 Verification

For verifying We can just compare the two files ast and .ll that we got.

Chapter 6

Conclusion

This report described the motivations, design, and implementation of a new LLVM bindings end for the MLton SML compiler. The primary goal of the project was to make the both of the two option that is available to us for conversion that is high level assembly or high level language.

Background information was presented on the broad design of MLton, its compilation pipeline, and execution model, to give understanding as to where LLVM would fit in the picture and what kind of work would have to be done. The design and implementation of the LLVM bindings was then presented, showing all of the challenges that were faced in its implementation and how they were overcome.

Chapter 7

Future Work

Although the project met many of the goals that it was set to accomplish, there is still plenty of room for future improvement, as noticed in the report it is not done for all the types and not for all instructions type.

The future work for this will have to add fully all the instruction types and all the types and also to convert any AST into the LLVM AST.

Chapter 8

References

- **http://llvm.org/** -> LLVM Official website
- **https://llvm.org/docs/BitCodeFormat.html** -> LLVM binary code format
- **http://hackage.haskell.org/package/llvm-hs-pure-8.0.0/docs/LLVM AST.html** -> LLVM Haskell AST
- **http://llvm.org/doxygen/groupLLVMC.html** -> LLVM C API
- **https://pauladamsmith.com/blog/2015/01/how-to-get-started-with-llvm-c-api.html** -> More on LLVM C