Extending Simplic with an LLVM backend

Project in Computer Science – EDAN70
January 26, 2016

Johan Forss Lasson
D11, LTH, Sweden
dat11jfo@student.lu.se

Alexander Åhlander
D11, LTH, Sweden
dat11aa1@student.lu.se

Abstract

In this brief article we detail our method for designing an LLVM back end for an ad hoc C-like language known as SimpliC. We outline the features of the language and the extensions to it which we have made. We also give some background on LLVM and the closely related topic of LLVM-IR. The main focus of the article is on how we have solved the different challenges which we came across while implementing the back end, mainly related to allocating memory for structs, using SSA-form variables in an efficient way, and using C-functions in LLVM-IR code. We also compare the performance of our implementation to that of others.

Keywords LLVM, LLVM IR, jastadd, SimpliC, C, Compiler Construction

1. Introduction

This project was undertaken as part of a project course, which aimed to give a deeper understanding of compiler construction. To facilitate this, we were given a number of project descriptions more or less closely related to compilers, and had to select a project which interested us.

We chose to implement a Low Level Virtual Machine (LLVM) back-end for a compiler created in a previous course as well as making some improvements to the front end of the said compiler. Very briefly put, LLVM is an infrastructure which uses LLVM Intermediate Representation (LLVM-IR), whose level of abstraction lies somewhere in between assembler instructions and C-code, to enable very high levels of optimization while still retaining reasonable compilation times. The seminal work in the field is "LLVM: An infrastructure for multi-stage optimization" published by C.A Lattner in 2002 [6]. LLVM is described in depth under section 2.1.

We chose this particular project as LLVM seemed to be an interesting concept at the time, an we were keen to see how it compared to pure assembler code generation. There was also the matter of part of the group having some experience with optimizing compilers, SSA (Static Single Assignment) form variables[4] and other related concepts, which further pushed us towards selecting this particular project.

Starting out, we had a more or less working SimpliC to Assembler compiler, which we created in a previous course. We added front end support for new constructs such as floating point variables, C-like structs and global variables (section 4). We also wrote a new back end from scratch which was capable of SimpliC to LLVM-IR compilation (section 5). To evaluate our work, we compared the performance of our product to that of other groups carrying out identical or similar tasks (section 6).

This article might be an interesting read for someone with basic knowledge of compiler construction, who is interested in creating a compiler for their own language. It might also be of use to those interested in comparing the performance of LLVM and the JVM.

2. LLVM

Under this section we describe the two main concept concerning LLVM, the LLVM itself and the LLVM-IR, which is the code written for usage with LLVM.

2.1 LLVM

LLVM is a rather large framework for compiling and optimizing software. Essentially, it combines beneficial features of many other frameworks of similar purpose to attempt to combine their benefits, thereby creating a superior final product.

On the compilation side of things, a main beneficial factor is that LLVM does not enforce any particular object model, which makes it possible to compile essentially any programming language to LLVM. This in turn means that a large application that may contain code written in several different programming languages may be compiled to a single language, that being LLVM-IR.

Concerning optimization, LLVM is capable of performing profile driven optimization by collecting data related to the execution during run time. Through saving the LLVM-IR instructions alongside the native code it is also able to perform high level optimizations over time [5].

2.2 LLVM-IR

Low Level Virtual Machine - Intermediate Representation (LLVM-IR) is the instruction set to which LLVM compiles high level languages. It is a strongly typed RISC-like instruction set with variables written in Static Single Assignment (SSA) form. It possible to call C functions using LLVM-IR code. It also has support for structs.

Typing

LLVM-IR being strongly typed means that all variables must be given a clear type, such as integer, floating point number, character etc. Each type has several variations, such as *i32* and *i64* being 32 and 64-bit integers respectively and *i1* being a one bit integer, suitable for storing boolean variables.

SSA form

Code is said to be in SSA form if all variables are immutable, that is they may only be assigned once. This makes certain optimizations possible or easier to perform, as the optimizer can assume that no reassignment will take place. This is usually done by creating a new version of a variable each time it is reassigned [4].

Phi functions

A Phi functions is an instruction which keeps track of which path a program took during execution. It uses this information to know which version of a variable should be used after a conditional branch.

Functions

LLVM-IR makes creating functions quite convenient. It uses a C-like structure in which a return type, a name and parameters with types which are explicitly declared. It is also possible to access functions from the C programming language in LLVM-IR code. They are included in the executable at link time.

LLVM-IR example

The following example illustrates a program which assigns 4 and 5 to the variables a and b respectively, and then multiplies these variables, storing the result in c.

```
%a = add i32 4, 0
%b = add i32 5, 0
%c = mul i32 %a, %b
```

3. The SimpliC language

During the course EDAN65-Compilers we implemented a compiler for SimpliC, a simpler subset of C. This compiler generated assembler code. This project aims to extend the SimpliC language with some new constructs, and to create a compiler which generates LLVM code.

The starting state of the compiler

The compiler that we used as a starting point was written in JastAdd, which is a java based language used for building abstract syntax trees[2]. The parser part of the compiler was constructed using the Beaver parser generator[1], and the scanner was written in JFlex [3].

From EDAN65 we already had a somewhat working front end implementation for SimpliC, and an assembler back end for said implementation.

Types

The initial front end implementation only had support for declaring and modifying integers. It was however able to return boolean values from comparative expressions for internal use.

Arithmetic operations

The initial version supported addition, subtraction, multiplication, integer division as well as modulo division. All these were of course only supported for integer operands, as this was the only type fully implemented (see above).

Comparative expressions

We initially supported most comparative expressions, these being equal, not equal, larger than, larger than or equal, smaller than as well as smaller than or equal. These were also, for obvious reasons, only supported integer operands. We did not, and still do not support the booleans operations such as and, or, xor etc, and as such we do not support ”chaining” of comparative expressions e.g

```
(i < 5) || (i == 25)
```

This shortcoming is however quite easy to work around using nestled conditional statements.

Conditional statements

We had support for the conditional statements if else and while. This implementation has not really been changed during the project, as we have extended the comparative expressions to use non integer numbers, which means that they may be used in conditional statements as well.

Functions

Support for functions was present with integers as parameters and return type and could be called as both an expression and a statement. The return value from a function is decided by a simple return statement, but the use of a return statement was not enforced by the compiler.

Comments

There was some support for comments, using c-like block comment syntax, i.e

```
/*comment*/
```

There was no support for line comments.

4. Language extensions

To ensure that the project workload was significant enough to motivate the time and academic credits allocated to it, we extended the SimpliC front end with some new constructs, some quite simple such booleans and floating point variables and some with a more significant effort required, i.e c-like structs and global variables.

4.1 Global variables

Global variables are variables that are not contained in a function, and are accessible from anywhere in the program. Support for these kinds of variables has been implemented. This means that the following programs are legal:

```
int a = 2;
int main(){
    a = a + 2;
    a = b + a;
}
int b = 4;
```

4.2 Booleans

A boolean variable is a type of variable which can only hold one of two values; true or false. This is useful when making and storing results of comparisons. Support for boolean variables has been added. Booleans may be declared and used as in the following program:

```
int main(){
    bool a = true;
    bool b = 5!=3;
    bool c = false
    bool d = a==b;
}
```

4.3 Floating point variables

A floating point variable is a variable which can hold a numeral value containing a decimal point, e.g. 6.3. Previously the language could only handle whole numbers. We will not implement comparison between int and float. Floats may be declared and used as in the following program:

```
int main(){
    float a = 1.5;
    float b = 1.5 + 1.3;
    float c = a + b;
}
```

4.4 Structs

A struct is a structure which can hold a predefined collection of variables. These variables can then be accessed and used by the program. Structs are declared globally. Structs should be able to contain other struct, and as such needs to use reference semantics. Structs and their variables may be declared and used as in the following program:

```
struct astruct{
    int b;
    bool c;
    float d;
}
int main(){
    astruct a;
    a.b = 2;
    a.c = true;
    a.d = 5.5;
    int e = a.b + 3;
    bool f = a.c == false;
    float g = a.d + 1.7;
}
```

5. Implementation of back end

We implemented a new back end for LLVM-IR code generation. In doing this a number of challenges appeared, to which solutions were to be found. Below is a brief description of each of these and our solution for them.

5.1 Variable representation

LLVM-IR is strongly typed, with quite a lot of possible formats for each kind of data. This means that it is a good idea to think quite carefully about how to optimally represent data.

Integer variables

Integer variables are represented using the LLVM type i32, which represents the number as 32 bits. Our integers are signed so that they are able to represent negative as well as positive numbers.

Boolean variables

Boolean variables are represented using the LLVM type i1, which represents the boolean value as a single bit, 0 for false and 1 for true.

Floating point variables

Floating point variables are represented using the LLVM type double, which represents the number using 64 bits. It would probably be possible to use the float type instead, but we had problems getting that to work, so we switched to double, which was more intuitive to use.

5.2 Variable declaration

A variable can be initialized to a starting value when declared in SimpliC, with an exception of structs. The initialization works like an assignment, explained in the next section. Since LLVM does not require a variable to be declared, the most of the information required by the assignment and uses of the variable in question is already calculated in the front end.The exception is the SSA form which requires every variable to be added to a map once before use, which is most easily done in the declaration.

5.3 Variable assignment

In SimpliC it is possible to assign a value to a declared variable as follows:

```
a = 5;
```

It is also possible to assign a value to a variable as it is being declared, as below:

```
int a = 5;
```

However it is not possible to simply assign a value to a variable in LLVM. To work around this we performed the add which was applicable for 0 or 0.0 respectively. This means that in the case of integers and booleans, we used add and 0. In the case of floating point number we used fadd (float add) and 0.0, since LLVM does not perform type promotion of the argument automatically.

When assigning a new value to a variable the version number corresponding to this variable is increased to ensure that future uses of this variable refer to the correct value. The reference variable version is also set to this new version, which ensures that any code pertaining to this variable is using the correct version of the variable.

In LLVM, global variables, that is variables that are not enclosed by a function, can only be accessed using a pointer. This pointer is used when assigning a value to these variables, by using the instruction *store*.

5.4 Conditional statements

We support the conditional statements if and while. Both of these evaluate a boolean and act accordingly. To be able to perform branching we use the label concept in LLVM, which works by creating labels which the execution may jump to given certain circumstances.

5.5 Structs

Structs are declared using only a list of types in LLVM. The elements are then accessed by with the *getelemptr* instruction, using the index instead of the name. As the elements are stored in a pointer, they do not need to have a new version when reassigned. This is however not true for the containing struct. Since the structs use reference semantics, they need to be allocated on the heap, which is done using the *malloc* function from C.

5.6 Static Single Assignment form

To simplify optimization, all variables in LLVM are in Static Single Assignment (SSA) form, which means that they are immutable. While this may be very good for optimization, it does not make for easy code generation. We worked around this using a version system in which we put the a version number after each variable to get around the need to overwrite the variable, which is not possible. To keep track of which version of a variable was the most recent one, we used a map which was passed around among the functions which required it. For an example of this mechanism see listing 1.

5.7 Phi functions

The content of this section is related to SSA (see above).
When executing code containing conditional statements, there will be some code paths that may or may not be run. This means that certain variables may have been modified in different ways on different runs.

When trying to use a variable that may have been modified in this way it is very useful to know where it has been modified,

and therefore what its value should be. To accomplish this one can divide the code into blocks, for example an if statement may be a block. A Phi function can use information about what variable may have been modified and in which blocks this may have happened to resolve which version of the variable should be used.

Listing 1. Example of code converted to SSA form with phi function

```
/* before conversion to SSA form */
int x = 5;
if(/*cond*/){
        x = 10;
}
print(x);

/* after conversion to SSA form */
int x0 = 5;
if(/*cond*/){
        int x1 = 10;
}
int x2 = phi(x0, x1);
print(x2);
```

5.8 Code generation

Code generation is performed simply by traversing the abstract syntax tree (AST) and generating LLVM-IR code for each node. To simplify the code generation, we stored the results of each expression in a predefined variable which is then used by the parent statement or expression. These variables need to be unique since LLVM-IR is in SSA form, which is done by adding the ancestry in the AST-tree to the variable name.

5.9 Example SimpliC program and corresponding LLVM-IR code

The following is a very simple, but valid SimpliC program:

```
int main(){
  int a = 5 + 3;
}
```

It compiles to the following LLVM-IR program (header code redacted, variable names truncated for brevity):

```
define i32 @main() nounwind {
 main_entry:
%d_a_v0_e_l = add i32 5, 0
%d_a_v0_e_r = add i32 3, 0
%d_a_v0_e = add i32 %d_a_v0_e_l, %d_a_v0_e_r
%a_v0 = add i32 %d_a_v0_e, 0
ret i32 0
}
```

6. Evaluation

6.1 Method of evaluation

As there was one other team doing the exact same project, with identical language extensions and very similar SimpliC syntax rules, and another team using the Java Virtual Machine (JVM) as back end for SimpliC, it seemed reasonable to perform a comparative evaluation of the performance of the tools created by each group.

6.2 Test suites

Two test suites were used to determine performance. Test suite one, hereafter referred to as T1 consisted of creating a tree using a loop and then finding the smallest value in this tree and setting all nodes to this value using a recursive technique. Test suite two (T2) consisted of using a very large source code file generated using python to statically create a tree. T1 is focused on testing the performance of the generated back end code, while T2 focuses on testing the speed of code generation. For both test suites, runs were carried out using a number of different tree sizes, the number of nodes specified in the accompanying graphs.

6.3 Results of test suite T1

The results of T1, illustrated in figure 1, are that our compiler and the other LLVM group have quite similar execution speed, with the JVM group being significantly slower at lower node counts, due to the long start up time of the JVM itself. As can be seen, the other LLVM group failed to execute the program for larger trees, running into segmentation fault issues. This discrepancy in largest executable tree size between two groups using the same back end is probably due to a difference in the manner in which the variables are stored.

When measuring the execution time for 8000000 nodes, we noticed that the execution time decreased substantially, from approximately 8000 ms during the first run, to stabilizing at approximately 4000 ms after five runs. We suspect that this might the result of LLVM performing run time optimizations.

Execution times were measured using a very simple shell script. The measurement for each number of nodes were repeated at least three times.

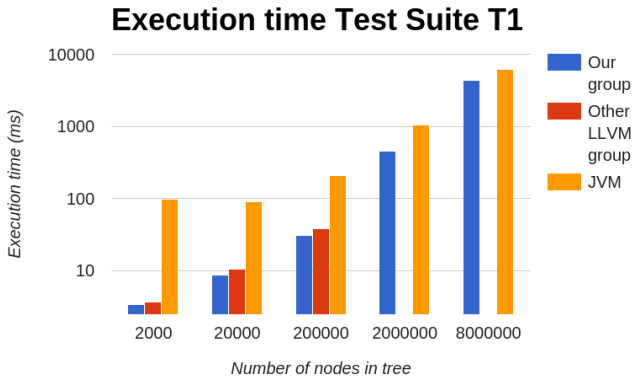

Figure 1. *Execution time of test suite T1.* Total execution time with varying tree size for each group when running T1

6.4 Results of test suite T2

Looking at the results of T2, illustrated in figure 2, show a clear difference in compilation time for programs containing a large amount of code, with our implementation being at a distinct disadvantage. We are not certain as to the cause of this, but we suspect that we may have quite a lot of redundant JastAdd code pertaining to type and name analysis.

Compilation times were established by running an ant script which compiled the source file, and noting the execution time of this ant script.

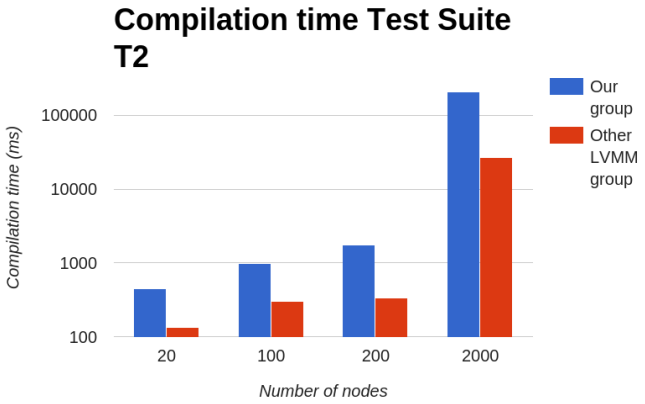

Figure 2. *Compilation time of test suite T2.* Total compilation time with varying tree size for the LLVM groups when compiling T2. JVM group did not record compilation time for T2.

7. Related work

LLVM is, in comparison to other subject such as Unix or C, quite a novel concept. SimpliC is more or less a fictive programming language. These factors combines to make it slightly hard to find relevant related work. Despite this, we have found some works that might be interesting.

LLVM: An infrastructure for multi-stage optimization. C.A Lattner [6]

This PhD thesis is essentially the seminal work in the field of LLVM. It discusses what the developer of LLVM originally had in mind as important properties of the system, as well as measures taken to establish said properties.

JastAdd - an aspect-oriented compiler construction system. G Hedin [2]

This paper explains the design behind JastAdd, why Java and object orientation in general makes sense when creating abstract syntax trees.

8. Concluding discussion

We have implemented an LLVM based back end for the SimpliC language. We have also made some extension to said language, and created front end support for these. The results were that LLVM slightly outperforms the JVM, but that the difference in performance decreases as program complexity increases.

This makes sense as LLVM in turn compiles to assembler, which should be able to utilize the resources of a given platform more effectively than the JVM, since the JVM is a virtual machine. There is also the matter of JVM needing a rather long time to start. These results could be useful when deciding whether to use Java or something that can compile to LLVM for a given task.

Expanding on this subject it would be interesting to implement a similar solution for an actual language such as standard C. This would also allow more relevant comparison between the LLVM $\Rightarrow$ Assembler pathway as opposed to compiling directly to Assembler.

Acknowledgments

The authors would like to extend their appreciation to the members of the other groups mentioned in this paper; Johan Henriksson (the other LLVM group), Elliot Jalgard and Philip Mrtensson (the JVM group), all of LTH, for useful suggestions when troubleshooting as well as creation of bash script and generally being easy to work with. We would also like to thank Mr. Christoff Bürger of LTH for his supervision of this project.

References

[1] Beaver grammar specification. *Available on-line at http://beaver.sourceforge.net/spec.html. Accessed January*, 2016.

[2] G. Hedin and E. Magnusson. Jastadd an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003.

[3] G. Klein. Jflex users manual. *Available on-line at www. jflex. de. Accessed December*, 2015.

[4] C. Lattner and V. Adve. The llvm instruction set and compilation strategy. *CS Dept., Univ. of Illinois at Urbana-Champaign, Tech. Report UIUCDCS*, 2002.

[5] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[6] C. A. Lattner. *LLVM: An infrastructure for multi-stage optimization*. PhD thesis, University of Illinois at Urbana-Champaign, 2002.