LLVM Code Generation for SimpliC

EDAN70 Project Report

Johan Henriksson
D11, Lunds Tekniska Högskola
dat11jhe@student.lu.se

Abstract

This paper briefly describes the process of extending a simple compiler implemented in a previous course with new language constructs and modifying it to output intermediate code for the LLVM compiler infrastructure instead of x86 assembly. This allows us to take advantage of features like built-in machine code optimizations and the ability to generate assembly code for multiple platforms. Advantages and disadvantages of using LLVM as a compiler framework will also be discussed.

1. Introduction

The Low Level Virtual Machine, or LLVM, is a large compiler framework consisting of (among other things) the LLVM Intermediate Representation (IR) and the compiler back-end. LLVM IR is an abstract RISC-like instruction set, but includes key higher-level information such as type information, control flow graphs and data flow representations that allows for more effective analysis of the program[7].

This allows anyone to write a compiler front-end for their language and output LLVM IR instead of compiling directly assembly code. The intermediate representation is then compiled by the LLVM compiler to the target platform assembly language. There are several advantages to this approach, the two biggest ones are getting access to all of the compiler optimizations built in to the LLVM compiler, and the ability to output machine code for virtually any platform.

The objective of this project was to extend the language developed in the compiler course, SimpliC, and modify it to output intermediate code for the LLVM compiler instead of the naive x86 implementation we developed in the previous lab sessions. The language extensions include two new data types for floating-point numbers and boolean values, global variables and support for custom structures with manual memory management.

We will also take a look at the process of generating LLVM IR code, a few common pitfalls, how LLVM IR differs from x86 assembly, and finally compare their advantages and disadvantages.

2. SimpliC Extensions

In order to make code generation more interesting, several new features has been added to SimpliC.

2.1 Original SimpliC

The original version of the SimpliC language is very basic. It has C-like syntax, a single integer data type, support for functions with arguments and return values, local variables and basic arithmetic as well as logical comparison operators.

A program can interact with the user by using the built-in *read* and *print* functions.

2.2 Data Types

Two new data types have been added in order to be able to represent floating-point numbers and boolean values. They are appropriately named *float* and *bool* respectively, much like in C. However, one key difference between my SimpliC implementation and normal C is that both integers and floating-point numbers are always represented using 64 bits. To make the boolean data type more useful, the standard logical operators - and, or, not - have also been added.

2.3 Global Variables

Global variable support has been also been added and they work much like they would in any C-like language. As you might expect, global variables can be declared in the global scope and may then be accessed or modified from anywhere in the program.

2.4 Structures

The extended version of SimpliC also supports custom composite data types or *structures*. Structures are automatically allocated on the heap upon declaration without an assignment. In order to avoid memory leaks, all allocated structs must be manually destroyed with the *delete* keyword. Since SimpliC does not have a notion of pointers, structures are handled using references. References are essentially pointers in the sense that they refer to memory allocated somewhere else, but there is no support for casting or pointer arithmetic. This gives SimpliC some type safety but there are still the problems of null pointers and accidentally accessing freed memory. The syntax is straight-forward and allows structures to be declared as follows:

```
struct int_list {
    int_list next;
    bool last;
    int value;
}
```

Figure 1. Integer list declaration example

The int list structure represents a simple list node in a list of integers. This implementation has no concept of *null* values and so we need an extra boolean value to keep track of whether there is a next list node or not. It is possible to nest the structure within itself due to reference semantics. Since references to allocated memory can be returned without copying and we can decide when to allocate, the example below would work without structure copies or memory leaks.

```
int_list prepend_int(int_list list, int v) {
    int_list node; // allocate
    node.last = false;
    node.value = v;
    node.next = list;
    return node; // return reference copy
}
```

Figure 2. List prepend function example

Structures can only be declared in the global scope - they may not appear inside a function. This was a deliberate choice to make name and type analysis easier.

The abstract syntax tree grammar for structures is defined as follows:

```
Struct ::= "struct" IdDecl "{" StructDecl* "}"
StructDecl ::= TypeDecl IdDecl ";"

IdUse ::= ID | StructIdUse
StructIdUse :: IdUse "." ID
```

Grammatically, the structure definition itself is fairly simple. It is similar to a function block except that only declarations are valid statements inside a structure block.

Identifiers referring to structure fields are a bit more tricky. Normal SimpliC identifier references are represented by an IdUse node, consisting of an ID token - a simple terminal containing upper- and lowercase characters between A and Z. I extended the original IdUse so that it can either be an ID token or a StructIdUse. StructIdUse, in turn, is defined recursively as an IdUse, the dot token, and an ID terminal. This parses structure field identifiers into a tree with the outermost field as the root node.

```
a.b.c = true;
```

Figure 3. Structure field access

This tree structure makes it easier to generate code for calculating the field address later on. The example above would be parsed into the following abstract syntax tree:

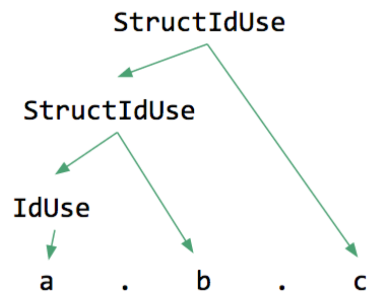

Figure 4. StructId AST Example

3. Code Generation

The goal is to output LLVM IR code from the compiler front end instead of going directly to x86 assembler. LLVM IR is a virtual instruction set architecture, essentially a high-level assembly language, that is annotated with types and other high level information to support multiple target platforms and more effective compiler optimizations [5].

The structure of the actual code generation has not changed much from the lab code from the compiler course. Each AST node has a *codeGeneration* method that takes a *PrintStream* to output the code to. This section will describe how to generate LLVM IR for the non-trivial language features available in SimpliC.

3.1 Functions

Generating code for functions is a simple task since LLVM IR has full support for functions, arguments and returning values, and the syntax is similar to what one would expect from a higher-level language. Because all stack management is handled automatically, this step is almost trivial in comparison with the work involved to do it in x86 assembly directly.

```
define i64 @f(i64 %a) {
    ret i64 %a
}
```

Figure 5. Example function in LLVM IR

3.2 Local Variables & Static Single Assignment Form

Static Single Assignment form (often abbreviated as SSA form) is a property of an intermediate representation (IR) language, which requires that each variable is defined before it is used, and assigned to exactly once [6]. LLVM makes use of this property in order to provide easier and faster compiler optimizations, but it also sets some constraints on code generation.

If registers were mutable, the obvious way to represent local variables in LLVM IR would have been to simply use a register variable for every local variable. One solution to this problem is variable versioning, which means that an index or version number is appended to each variable and incremented each time it is changed. However this has the drawback of having to keep track of the most current version of the variable. The proper workaround is to explicitly allocate local variables on the stack using the *alloca* instruction, and then store the resulting stack address in an immutable register variable.

When accessing or modifying a local variable, it has to be explicitly loaded or stored to its address. Because the address of the variable never changes, there is no longer a problem with mutability[8] and no need to keep track of variable versions.

```
%i = alloca i64
store i64* %i, i64 5
%1 = load i64* %i
```

Figure 6. Example of how to read and write a local variable. Note that %i is immutable.

Function arguments remain immutable in this implementation of SimpliC. This is a deliberate design choice since the LLVM code becomes simpler and it is usually considered bad practice to modify function argument variables anyway.

3.3 Control Statements

SSA form poses another problem when it comes to implementing control structures that rely on branching. If variable versioning is used to deal with SSA, the compiler has to keep track of which variable version is the current one after each branch, e.g. in a situation where a variable is modified in both cases of an *if* statement. To do anything useful with the variable after the if statement, the compiler would need to know which version of the register is the current one. A *phi* instruction is provided for just this purpose, but it's possible to circumvent the problem entirely by implementing local variables using explicit allocation of variables on the stack, as explained in the previous section [8].

Using this method, there's no longer any need to keep track of variable versions or which branch was taken, and so implementing the if- and while statements becomes easy. LLVM provides branching and comparison instructions that are very similar to those available in x86 assembly, which means that the process of implementing control structures is also similar.

3.4 Expressions

Arithmetic and logical expressions are also similar to their x86 counterparts, except that the naive one-register solution for all intermediate values can no longer be used due to the nature of LLVM IR and SSA form. Instead the compiler has to generate a new variable name for each intermediate result, and then pass it on to the parent AST node that needs to use it.

A simple implementation is letting the *codeGeneration* method of each expression node output its code to the *PrintStream* as usual, but instead of returning void it returns a string with the name of the register containing the result of the operation. Intermediate registers are numbered, starting from 1 at the beginning of each function.

3.5 Structures

LLVM has built-in support for custom structure types similar to C, however, structure fields are referred to by a zero-based index instead of a name. Despite the built-in support, there's still a few tricky parts when it comes to generating code for structures.

The first problem is how to access fields. LLVM provides the *getelementptr* instruction to calculate the offset of a given structure field. In most cases, this instruction will be optimized into a LEA instruction (or similar) and thus cause no additional overhead [8]. In order to solve the problem of nested structure references like the one described in the language extensions section, the address is resolved recursively from the bottom up. To compute the address of a field, only the structure base address and the field index is required. This means that as long as we start at the bottom of the tree, each field address can be computed using the "parent field" address and the field index. The bottom node will always end up being a variable or something else with a known address.

The second problem is memory allocation. LLVM exposes *malloc* through a special instruction, but due to the lack of a *sizeof* macro, we need a way to calculate the size of structures. One neat solution is to use the *getelementptr* instruction to calculate the offset of the second element in an array of structures starting at address 0. This operation is optimized away by LLVM and we're left with a constant in the resulting assembly code [8].

The final problem is how to free memory used by the allocated structures. I chose to expose a wrapped version of the C standard library's *free* function, so when code is generated for the *delete* statement, it's turned into a call to *free*.

3.6 Linking with C

Implementing the library functions was perhaps the most interesting part of the project. While they could theoretically be implemented in LLVM IR directly, I wanted to try to link the output from the LLVM compiler with a library implemented in C. It's not only much easier to implement the library in C, it also opens up the possibility of exposing any C library to SimpliC, so that it could potentially do something more than just toy examples.

The process turned out to be surprisingly easy since LLVM IR is compatible with the C calling convention by default. Instead of letting LLVM compile the IR directly to an executable file, it's set it up so that it outputs an object file. This output object file is then linked with a precompiled C library (another object file) using the GNU linker (ld) to produce the final executable.

4. Evaluation

I will try to evaluate this project based on the performance tests we did among myself and the other groups, as well as the advantages of using LLVM for your compiler project based on my own experience working with it.

4.1 Performance

Two tests were used to compare performance to the other groups working on similar projects. Two other groups had compatible SimpliC extensions; one did another LLVM back-end implementation and the other generated bytecode for the Java Virtual Machine (JVM), so we tried to measure performance and compare between these groups.

The tests were simple programs that first generates a large tree structures and then locates a specific node using two different methods for traversing the tree. Test A uses an iterative algorithm while Test B uses a more traditional recursive solution. Compilation time is not included in tests A and B.

4.1.1 Test A

The other LLVM group had trouble running this test, so it only serves as an execution time comparison between the my LLVM optimized x86 code and unoptimized JVM bytecode.

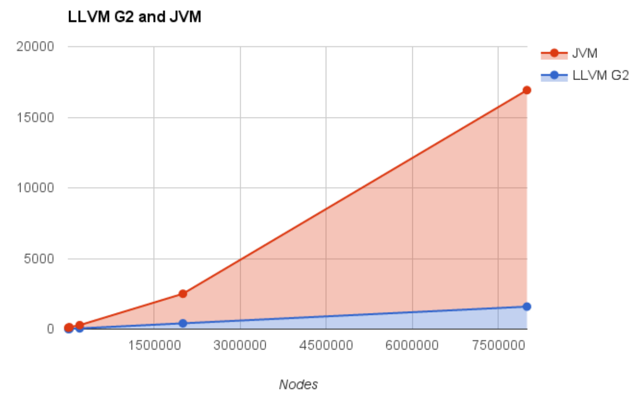

Figure 7. Performance Test A - Iterative Algorithm. Execution time as a function of input size for my LLVM implementation and the JVM implementation. *Vertical axis:* Execution time in milliseconds

The LLVM implementation was around 10x faster than the JVM in all of the measurements. It's not a very interesting comparison since LLVM produces native code while the JVM is executing the program in a virtual machine, but at least it seems to indicate that the machine code generated by the LLVM compiler is fairly efficient and well optimized. Note that the JVM byte code is not optimized except for any adaptive optimization the JVM may have applied during runtime/just-in-time compilation.

4.1.2 Test B

The next test is slightly more interesting because we have a few data points from all the groups. However, the other LLVM group didn't manage to run the larger example (orange bars).

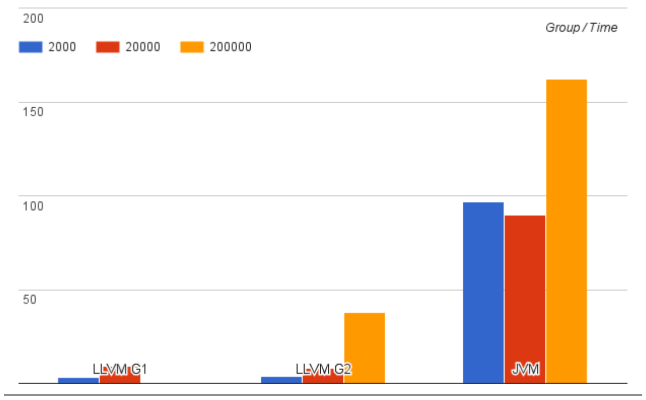

Figure 8. Performance Test B - Recursive Algorithm. Execution time for different input sizes. *Vertical axis:* Execution time in milliseconds. Color indicates the number of nodes.

We can see that for the small examples, execution time is similar between the two LLVM implementations and the JVM is way behind. It looks like the JVM takes almost 100 ms just to start up, while the LLVM executables finish running in just a few milliseconds.

4.2 Larger Programs

For the last test case we used a small python script to output SimpliC code for building large tree structures statically. We measured the compilation time to get an idea of how our compilers coped with larger programs. For every additional node, the source file grows by approximately 6 lines of code. I've only included measurements of my own implementation here due to inconsistent measurements between the groups.

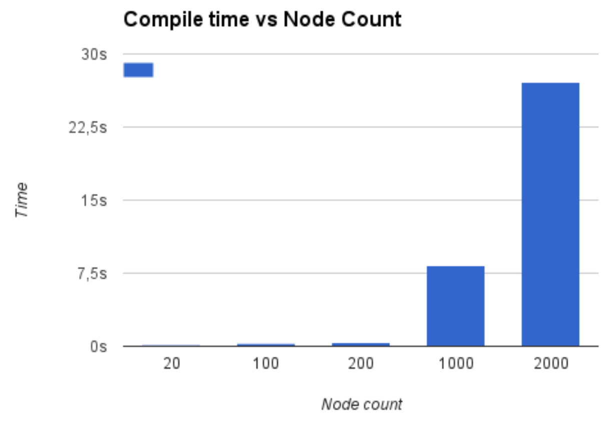

Figure 9. Large Programs test: Compile time as a function of program size. Nodes are created statically and each node is approximately 6 lines of code.

As you can tell from the graph, compile time grows dramatically once the number of nodes becomes larger than a few hundred. This is because the algorithms used in the name and type analysis are very naive and not very efficient. Since every node becomes a local variable, name and type look-ups will be more and more costly as the number of nodes increase.

In its current state, my compiler is unsuited for any program larger than a few thousand lines - note that the source code file for the 2000 node test is around 200 kB.

4.3 Compiler Implementation

As for the implementation itself, it was significantly easier to output LLVM IR than it was to work with x86 assembly directly. This is mostly due to the fact that LLVM IR is a higher level language and so it's much easier to read and understand the output code itself, and there is no need to worry about details like pushing/popping register values or correctly incrementing or decrementing the stack pointer. There's also support for data types and structs which came in really handy for this project. Lastly, since the output of your compiler is compiled again by LLVM, there's another layer of error checking involved which makes it easier to catch errors in the code generation step.

5. Conclusion

Using LLVM as a compiler back-end has lots of advantages compared to implementing a custom code generation back-end. It is much easier to both write, read and understand LLVM IR, which means that less time will be spent dealing with both code generation and debugging assembly code. The compiler can also instantly take advantage of all the optimizations built-in to LLVM without having to reinvent them. Finally it is possible to compile to a multitude of different platforms without any extra work, and even in the rare cases where compilation to some obscure platform is necessary, you also have the option to compile to C.

As for disadvantages, I haven't been able to come up with many. One possible disadvantage would be in a situation where you need greater control over the compilation process, perhaps to add a custom language-specific optimization. But since LLVM is open source and very extensible, customization is still possible if required [4].

Compiling to C is actually an alternative to LLVM - it's low level, fast, portable and there are optimizing compilers available for virtually any platform. However, since C wasn't designed for this purpose, this method solution with its own set of disadvantages. One problem is that advanced language constructs expressed in a few lines in the source language may compile into thousands of lines of C code, which was the reasoning behind starting the Haskell LLVM compiler project [9]. Other common problems are very little control over how the C code itself is compiled, and the fact that compiling C is usually a slow process, especially for larger programs.

The original compiler project was developed using three main libraries. The scanner and parser used are *jFlex*, a fast and flexible scanner generator[3], and the *Beaver* LALR parser generator [1], both written in Java. Finally, the *JastAdd* framework provides reference attribute grammars and aspect-oriented programming for implementing clean computations and operations on the abstract syntax tree [2]. I personally think that LLVM was a very good fit together with *JastAdd*, *beaver* and *jFlex*. It makes it reasonably easy to go from a simple compiler with naive x86 assembly output to something much more useful - optimized and C-compatible machine code for multiple different hardware platforms.

References

[1] Beaver, a lalr parser generator. URL `http://beaver.sourceforge.net/`.

[2] Jastadd. URL `http://jastadd.org/`.

[3] Jflex, a fast scanner generator for java. URL `http://jflex.de`.

[4] Writing an llvm pass. URL `http://llvm.org/docs/WritingAnLLVMPass.html`.

[5] V. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke. LLVA: A Low-level Virtual Instruction Set Architecture. In *Proceedings of the 36th annual ACM/IEEE international symposium on Microarchitecture (MICRO-36)*, San Diego, California, Dec 2003.

[6] M. M. K. M. Jianzhou Zhao, Santosh Nagarakatte and S. Zdancewic. Formal verification of ssa optimizations for llvm. In *Proceedings of the Thirty Fourth ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013)*.

[7] C. Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[8] M. Lyngvig. Mapping high-level constructs to llvm ir, 2015. URL `http://llvm.lyngvig.org/Articles/Mapping-High-Level-Constructs-to-LLVM-IR`.

[9] J. van Schie. Compiling Haskell To LLVM, Thesis Defense, Utrecht University, Netherlands, June 2008. URL `http://llvm.org/pubs/2008-06-CompilingHaskelltoLLVM.pdf`.