LLVM as optimizer and compiler backend

Project in Computer Science – EDAN70

Valdemar Roxling
D11, Lund Institure of Technology, Sweden
dat11vro@student.lu.se

January 19, 2016

Abstract

New programming languages often bring new ideas and tools to the programmer to make certain tasks easier to implement and speed up development. The one thing that many new languages fail to improve is performance, but with some relatively simple backend changes the compiler can use well known and good algorithms to achieve a lot in terms of code optimizations and speed-up.

In this article I have evaluated a backend exchange of an existing small language called `SimpliC` to generate to the intermediate language `LLVM-IR` [1]. This makes `SimpliC` work on any platform and operating system supporting LLVM, and I can make use of already existing code optimizers. In addition to performance measurements I have also extended my language with an additional high level construct, nested functions, not natively supported by LLVM.

1. Introduction

The `SimpliC` programming language is a small subset of the programming language `C`, created mostly as a source of learning. The existing compiler is generating `Intelx86` code for Linux without any considerable optimizations and for large programs it is very slow, mostly because it has no register allocation, and pushes all data to the stack.

The `LLVM-IR` language[1] is an intermediate language, built in a way that makes it easy to optimize in many different ways and at different times. It currently supports compile-time, link-time, run-time and idle-time optimizations [2]. LLVM is already used as backend in a several different compilers of which Clang for `C` and `C++` is the most noticeable, and it can also be found in compilers for both old languages like `Fortran, Ada & Lisp`, and for modern languages like `Python, Scala & C#`. LLVM is also a cross-platform language and assemblers can be found on almost any common architecture and operating system. So instead of having a different compiler backend for every supported platform, you just need to create and maintain one.

All these features makes it a very suitable backend for almost any new language and compiler that wants to increase the execution speed of the compiled programs, but another just as important aspect is how easy high level language constructs can be implemented in this relatively low level language and still maintain speed and flexibility.

Generating `LLVM-IR` can be quite challenging, as the `LLVM` compiler is very strict on how the code should look like, everything is highly annotated and temporal variables can only be assigned once, everything to allow for better optimizations. To build high level constructs it is mostly a challenge of transforming them in to low level constructs supported by LLVM, where the biggest challenge is to find the best transformation.

In this article I will first describe some of the tools I have used, then I will discuss some of the challenges when exchanging a compiler backend and how to implement high level constructs in `LLVM`. I will also evaluate the results by comparing with the previous compiler and a well known existing compiler. At last I will discuss some related and future work.

2. Tools

In order to fully understand the backend exchange with related problem and solution discussions some technical information about `SimpliC`, the existing compiler and LLVM is required.

2.1 SimpliC

The syntax of `SimpliC` is very similar to the one of `C`, and a program consists of one or more functions, of which one is called `main`. Each function has a return type, a name, and zero or more arguments. The function body consists of one or more statements; variable declarations, if-else, while, assignment, function calls, nested functions and return. Variables can be of integer and boolean type, functions can also have the void type. Math operations on variable consists of addition, subtraction, multiplication, division, modulus, equality, non-equality, lesser or greater. Variables and functions do not share the same namespace, and nested functions and variables can be shadowed. Recursion is also supported. The ”standard library” consists of two functions, to read and write data to or from the current standard input and output.

2.2 Compiler

The `SimpliC` compiler is using Beaver and Jflex for scanning and parsing, and JastAdd[3] to create the abstract syntax tree, to perform type analysis, name analysis, error checks and to generate backend code.

2.3 LLVM

The LLVM framework, consisting of the `LLVM-IR` specification together with associated compilers and libraries, was initially released in 2003, and is still under heavy development, with the most recent stable release in September 2015. It was originally designed to be easy and fast to optimize, in many different stages of compilation. The standard package for linux comes with a static compiler, `llc` that compiles and optimized the llvm file format `.ll` to object code, `.o`. It also comes with an JIT[2]-compiler/interpreter that optimizes the code during runtime.

[1] Low Level Virtual Machine intermediate representation

[2] Just in time

The basics of the LLVM-IR language consists of global/local variables of many different types (integer or floats of arbitrary bit length, and pointers to those), functions and function calls, all the basic mathematics operations and branches. Most variables in the language are of temporary kind, and can only be assigned once, but read many times. When variables are used their type must be explicitly told, and type casts are available. Everything in LLVM-IR can be annotated with, to tell the compiler about which optimizations it is or isn't allowed to do.

LLVM-IR can be seen as a language somewhere between C and an assembly instruction set.

3. Adapting to LLVM

The original compiler for SimpliC, created in 2014, produced a naive stack-based approach for the Linux Intelx86 instruction set, without any performance considerations. Exchanging this simple approach with LLVM, will produce something that does not only work, but is fast, flexible, multi platform, and uses some of the latest code optimization algorithms.

3.1 Backend exchange

To just change the backend of a compiler may sound easy, but requires a deep understanding of the language you are going to use. LLVM has a lot of documentation, but not much of practical use, and I found out that using Clang to create examples for my self was a great source of information. The LLVM-IR compiler is very strict and will fail to compile with vague error messages at best if you get any of the syntax slightly wrong. And since the language is very expressive it is very likely to get it wrong at one point or another.
A simple C function call:

```
int num = 5;
printf("%d\n",num)
```

Can look like this in LLVM:

```
@fmt = internal constant [4 x i8] c"%d\0A\00"
%num = alloca i32
store i32 5, i32 *%num

call i32 (i8*, ...)* @printf(i8* getelementptr
                     inbounds ([4 x i8]* @fmt,
                     i32 0, i32 0), i32 %num)
```

Another thing worth noticing is the fact that LLVM uses temporal variables for everything, and these variables can only be assigned once, and this was a common source of problems for me.

3.2 Nested functions

Nested functions, or local functions, are functions inside of other functions that are only visible to the local scope, and can access objects and variables in any of its enclosing functions. A simple example illustrating nested functions can be seen in figure 1. LLVM-IR does not support nested functions natively, so nested functions, along with any other high level construct you wish to implement has to be transformed in to something supported by LLVM-IR without affecting its functionality[4].

Quite naturally a nested function can be transformed into a normal function, but with a unique name only known to its enclosing functions. After that the only problem that remains is how the function should be able access objects from all of its enclosing functions. To access those objects you can choose one of the following:

- Send pointers to accessed objects as additional implicit function parameters.
- Encapsulate pointers in a C-like struct and send struct as an additional implicit parameter.
- Pass a pointer to frame of the caller as a implicit parameter, just like a static link in a runtime system.

I choose to go with the first approach as it seemed easier to me, but any of the others would work just as well, and might even be seen as a more elegant solution, but may require more work.

What is left now is to determine which variables that needs to be passed as arguments, and how many levels of nested functions the variables has to be passed through. This is done by first looking at the enclosing function. If it's declaration is not found there, then a pointer to it should be passed as an argument and you go up one scope to the next enclosing function and repeat, until you reach the "global scope".
An example of two SimpliC nested functions may look like this:

```
int main(){
        int a;
        int c = 7;
        void f1(){
                int b = 2;
                void f2(int num){
                        a = 2 * b + num;
                }
                f2(3);
        }
        f1();
        print(a+c);
}
```

Figure 1. A simple SimpliC nested functions example.

And when transformed it would look like this (transformed to C instead of LLVM-IR for clarity and higher readability) :

```
void main_f1_f2(int num, int *hidd_a, int *hidd_b){
        *hidd_a = *hidd_b * 2 + num;
}

void main_f1(int *hidd_a){
        int b = 2;
        main_f1_f2(3, hidd_a, &b);
}

int main(){
        int a;
        int c = 7;
        main_f1(&a);
        printf("%d\n",a+c);
        return 0;
}
```

Note how the function names have changed to something only known to the main function, and that a pointer to the variable a is passed to first f1 and then f2 even when it is not accessed by the first function. And a pointer to the variable b is only passed to f2 since it's declaration was found in f1. And the printed result of this program is 14 $(2 * 2 + 3 + 7)$.

4. Evaluation

In order to evaluate my implementation I have measured the performance improvements of exchanging the compiler backend.

4.1 SimliC, LLVM & Intelx86 performance

The performance boost in terms of program execution time of using LLVM-IR as a backend opposed naive Intelx86 can be measured in many different ways. I choose to execute an $O(n^2)$ algorithm for

calculating the Fibonacci sequence recursively, from the definition seen below.

$$fib(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fib(n-1) + fib(n-2) & \text{if } n > 1 \end{cases}$$

This test case is created without nested functions and the evaluation is between the naive `Intelx86` compiler backend and the `LLVM-IR` compiler backend. The complete test code can be found in appendix A.1, and the result can be seen in figure 2. The speed-up factor seems to be constant around 140% faster for this particular test program, with `LLVM-IR` as backend. This result was rather expected since the first approach of generating `Intelx86` naively used no optimizations, and `LLVM` uses almost all of the common ones, including register allocation and fast calls. Another perk of using `LLVM` is that the generated assembly file is of around half the size of the unoptimized one generated with the previous backend.

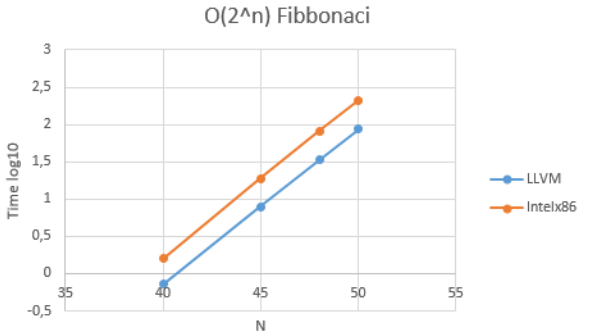

Figure 2. Performance plot of recursive Fibonacci calculations. Note the log_{10} Y-scale. Each sample is the mean of 5 measurements, and the maximal 95% confidence interval is the value ± 0.280s, for $n = 50$. No confidence intervals are overlapping hence the measurements are statistically significant.

4.2 Nested functions, SimpliC & GNU C performance

In `GNU C`, a non `Iso C` extension, nested functions are allowed with basically the same syntax as `SimpliC`. So comparing my solution of nested function with another, professional implementation, is very interesting. The test I performed is basically the same recursive Fibonacci series calculations, with the big difference that the recursive function is now nested, and in each call a variable in the enclosing function is incremented, calculating how many recursive calls that has been made. The result can be seen in figure 3, and surprisingly, yet hard to see with the logarithmic scale, my `LLVM-IR` implementation is around 12% faster than the `GNU C` GCC implementation. The complete test code for both `SimpliC` & `GNU C` can be found in appendix A.2 & A.3 respectively.

4.3 Known limitations

Almost no programs are entirely correct and without bugs or limitations, and neither is this compiler. The compiler itself terminates with a `STACK OVERFLOW` and fails the compilation of programs exceeding around 1500 lines of code (LOC) due to ineffective analysis of the abstract syntax tree. But due to limitations in the language no "real" programs are likely to reach even close to the size and complexity that causes the compiler to crash.

Another limitation is that the stack is ***only*** collapsed after the return of a function call, due to the nature of `LLVM-IR` containing no scopes inside a function. In the following simple example the stack does not collapse after each iteration of the `while`-statement, and

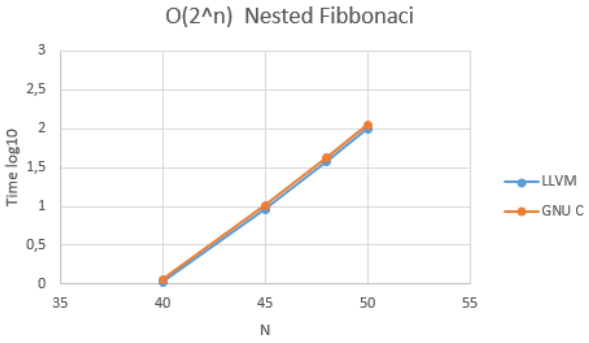

Figure 3. Performance plot of recursive Fibonacci calculations with nested functions. Note the log_{10} Y-scale. Each sample is the mean of 5 measurements, and the maximal 95% confidence interval is the value ± 0.181s, for $n = 50$. No confidence intervals are overlapping hence the measurements are statistically significant.

the variable declaration `int j = i` will quickly fill up the stack, causing a `SEGMENTATION FAULT` exception.

```
int main(){
        int i = 0;
        int MAX_STACK_SIZE = ...
        int calc = 0;
        while (i < MAX_STACK_SIZE){
                i = i + 1;
                int j = i;
                calc = calc + j;
        }
        print(sum);
}
```

Each iteration of the loop will allocate some memory for the variable `j`, and this memory has to be deallocated, either by collapsing the stack by hand, or allocate the variable once in the function scope instead. This has not been implemented due to time limitations in the scope of this project.

5. Related work

Almost every programming language out there has a compiler using LLVM as backend, ranging all the way from `Erlang`[5] to `Scala`. This is very interesting, especially for new small languages, as they can link and use the libraries of the larger languages, providing they use the same LLVM data layout (memory model), allowing these languages to more quickly become usable, and developers does not have to "re-invent the wheel". The most commonly known compiler is, as previously mentioned, Clang, and I use this compiler, to implement `SimpliC`'s predefined functions `read() & print()` using `ISO C`'s `printf() & scanf()`.

Another very interesting compiler is GHC[3] for `Haskell` because they encounter a similar problem of nested function as I have solved, since it is a functional language. GHC uses two-stage backend, with a the first stage using a language called `Cmm`[6], and the second stage using `C`, but can switch to LLVM if requested. This option was added as the `Haskell` language is very complex and had high level constructs not easily represented with `C`, resulting in slow program execution. One attempt to solve this was to instead use `GNU C` as intermediate language, and use its non `ISO C` extensions to solve the problem. This was later cancelled due to not gaining the desired performance increase and `GNU C` not being compatible with every desired platform, so `LLVM` was approached.

[3] The Glasgow Haskell Compiler

LLVM solved the initial performance problem, but not without its own set of drawbacks, hence LLVM is still just optional [7].

6. Conclusions

Exchanging the backend to LLVM turns out to be well invested time, providing my language with the features of using some of the best optimization algorithms there are, resulting in very fast program executions, both with a static compiler, or a JIT. It also provides portability of the language to almost any platform and operating system, as well as library compatibility with all the other language compilers out there with LLVM as backend. But generating LLVM-IR is not the easiest of tasks, and there are many, well camouflaged pitfalls causing undesired results, and generating limitations instead of removing them.

It also seems like most high level language constructs can quite "easily" be transformed into LLVM-IR code since my example of nested functions turned out to be relatively simple rewrite, and many other, very complex, languages and compilers uses LLVM as main or optional backend, like Clang for C and GHC for Haskell.

6.1 Future work

There are many suitable extensions to this project. The most obvious are to correct the limitations mentioned in section 4.3. Furthermore extending the language with more types, allowing arrays and structures, implementing more high-level constructs like pointers, classes and λ-functions, just to mention a few will greatly increase the usability of the language.

A. Performance tests

A.1 Naive Fibonacci series calculation in SimpliC.

```
int fib(int num){
        if(num < 2){
                return num;
        }
        return fib(num-1)+fib(num-2);
}

int main(){
        print(fib(read()));
}
```

A.2 Naive Fibonacci series calculation as a nested function with a call counter in SimpliC.

```
int main(){
        int numcalls = 0;
        int fib(int num){
                numcalls = numcalls + 1;
                if(num < 2){
                        return num;
                }
                return fib(num-1)+fib(num-2);
        }
        print(fib(read()));
        print(numcalls);
}
```

A.3 Naive Fibonacci series calculation as a nested function with a call counter in GNU C.

```
#include "stdio.h"

int main(){
        int numcalls = 0;
        int fib(int num){
                numcalls++;
                if(num < 2){
                        return num;
                }
                return fib(num-1)+fib(num-2);
        }
        int n;
        scanf("%d",&n);
        printf("%d␣\n%d␣\n", fib(n), numcalls);
}
```

Acknowledgments

I would like to thank:

- Anton Klarén for assisting me in developing the "original" SimpliC languange with compiler in the fall of 2014.

References

[1] *LLVM-IR language reference.*, 2015-12-05 http://llvm.org/docs/LangRef.html.

[2] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 2004, pp. 75–88. [Online]. Available: http://dx.doi.org/10.1109/CGO.2004.1281665

[3] T. Ekman and G. Hedin, "The jastadd system - modular extensible compiler construction," *Sci. Comput. Program.*, vol. 69, no. 1-3, pp. 14–26, 2007. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2007.02.003

[4] *Common high level language constructs transformed to LLVM-IR.*, 2015-12-06 http://llvm.lyngvig.org/Articles/Mapping-High-Level-Constructs-to-LLVM-IR.

[5] K. F. Sagonas, C. Stavrakakis, and Y. Tsiouris, "Erllvm: an LLVM backend for erlang," in *Proceedings of the Eleventh ACM SIGPLAN Erlang Workshop, Copenhagen, Denmark, September 14, 2012*, T. Hoffman and J. Hughes, Eds. ACM, 2012, pp. 21–32. [Online]. Available: http://doi.acm.org/10.1145/2364489.2364494

[6] *Supported backends for GHC.*, 2015-12-06 https://downloads.haskell.org/ghc/latest/docs/html/users_guide/code-generators.html.

[7] D. A. Terei and M. M. T. Chakravarty, "An llvm backend for GHC," in *Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell, Haskell 2010, Baltimore, MD, USA, 30 September 2010*, J. Gibbons, Ed. ACM, 2010, pp. 109–120. [Online]. Available: http://doi.acm.org/10.1145/1863523.1863538