LLVM code generation and implementation of nested functions for the SimpliC language

Oscar Legetth
Lunds University
dat12ole@student.lth.se

Gustav Svensson
Lunds University
dat12gs1@student.lth.se

Abstract

This report will cover our work on the extension of the SimpliC language(a simple C-like language) and its compiler. We will describe SimpliC's features and the extensions we have made to it. The compiler has been extended with support for LLVM code generation. LLVM code lets us use the different optimization tools that LLVM has. We will then compare the optimized LLVM code with the assembly code that our compiler generated before we started this project in terms of execution time. We will also describe how we implemented the extensions and the LLVM code generation, and the different problems that came along with it.

1. Introduction

The SimpliC language is a very simple C-like language that features variables, functions, conditional statements and loops. We will use this language to construct a compiler that produces both x86 assembly code and LLVM intermediate code. These two results will be compared in terms of performance to evaluate and discuss the compiler optimization. This work started with a SimpliC compiler with only an x86 assembly back-end. We will extend the SimpliC language with more types, such as floating point numbers and booleans, as well as nested functions. We will also add a LLVM back-end to our SimpliC compiler.

Since the SimpliC language is a language only used in LTH's compiler and project courses, not much background research or work exists, except the similar projects to this from earlier years of this course. It is however (as the name suggests) a simple C-like language. The compiler is built using JastAdd(Ekman and Hedin 2007).

LLVM(low level virtual machine) is a low level platform independent compiler framework used to optimize code (Lattner and Adve 2004). The LLVM intermediate code representation contains both high and low-level information about the program such as type information and Control Flow Graphs. This information is vital to the various optimization steps LLVM goes through.

We will extend the SimpliC compiler with an LLVM intermediate code back end and see if performance can be improved compared to the x86 assembly back end by doing some performance tests. The results of our performance tests will then be evaluated.

2. SimpliC

The SimpliC language is a very simple C-like language that features variables, types, functions, arithmetics, conditional statements and loops. In this section we will describe these features in more detail and describe the extensions with nested functions and more types.

2.1 First version of SimpliC

The first version of our SimpliC language was basic and had support for integers, arithmetics, functions, variables, conditional statements and while-loops. Figure 1 shows an example program in SimpliC that calculates a fibonacci number.

2.1.1 Functions

Functions works as they do in C with support for parameters and obligatory return statements. Since the first version of SimpliC only supported the type `int`, all functions had `int` as both parameter and return types. Functions can be called and used in arithmetic expressions. The language has two built-in functions: `print` and `read` which could be used for printing to the standard output, or reading from standard input. Every program has to include a function labeled `int main` where program execution will begin.

2.1.2 Variables

Variables can only be declared inside functions, i.e, there are no global variables, and variables can only be of the type `int` in the first version.

2.1.3 Arithmetic operations

The arithmetic operations possible are addition, subtraction, multiplication, division and modulo. It is possible to use both integer literals and variables in the arithmetic operations. The arithmetic operations can also be chained, i.e `x + 1 + 1 + 1`.

2.1.4 Comparative expressions

The comparative expressions that SimpliC supports are less than($<$), greater than($>$), less than or equal($\leq$), larger than or equal($\geq$), equal($=$) and not equal($\neq$). They could be used to compare integers with integers, variables with variables, or variables with integers in the first version. SimpliC does not support chaining of comparative expressions, i.e `1 < 2 < 3`.

2.1.5 Conditional statements and loops

SimpliC supports if, if-else statements and while-loops. They use comparative expressions to evaluate if the condition is `true` or `false`.

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

int main() {
    print(fib(40));
    return 0;
}
```

Figure 1. Fibonacci program in SimpliC

2.2 Extensions to SimpliC

We extended the SimpliC language in this project with nested functions and more types.

2.2.1 Nested functions

A nested functions is a function which is defined inside another function. A nested function is invisible outside of its enclosing function, but it can access everything ”above” itself, i.e, everything that is declared in its enclosing function(s). In Figure 2 below you can see an example of how nested functions work. As in Figure 1 this program calculates the n^{th} fibonacci number, but it is implemented using a nested function unlike Figure 1. As one can see the `fib` function is now declared inside the `main` function. It would also be possible for the `fib` function to access variables that are declared inside the `main` function”

```
int main() {
   print(fib(40));
   return 0;


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

Figure 2. Fibonacci program in SimpliC with nested function

2.2.2 More types

The types we added to SimpliC are `float`, `boolean` and `void`. Floats are numbers which has a decimal point. Our floating points in SimpliC works as they do in C and Java. In SimpliC they can be declared as follows, `float f = 1.5`. Like with integers, it is possible to compare floats with floats, but it is not possible to compare integers with floats. Floats can only be used with floats in arithmetic operations.

Booleans can be assigned `true`, `false`, or a comparative expression. For example `boolean b = 1 != 2`. Booleans can be used as the condition for while-loops and if-statements.

The `void` type lets us declare functions with this type, which is useful since then it's not obligatory for functions to return a value.

3. LLVM

LLVM (Lattner and Adve 2004) is a compiler framework with a source-language-independent code representation. Figure 3 shows some of the front and back ends of LLVM. In this project the x86 backend was used for testing and evaluation. LLVM can represent programs of an arbitrary language and be optimized in several ways using the LLVM optimizer(LLV 2016). LLVM does not make use of physical registers directly, but instead allows the user to use an infinite amount of virtual registers. All the virtual registers in LLVM are on single static assignment form(Cytron et al. 1991). This means that a register can only be declared a value once and can never be given a different value. If a different value is needed, a new virtual register must be declared. Static single assignment form makes it easy for the LLVM optimizer to optimize different programs, because it does not need to keep track of different versions of a variable, since they can only be declared once. Data can also be stored in memory. LLVM uses a load/store architecture to transfer values between memory and registers. Loading and storing is done in one line of code each, and must be done before the data can be used.

LLVM has types with predefined sizes to ensure platform-independence, it does however allow non-portable types to be expressed. There are only four derived types: arrays, pointers, structures and functions. Any other type in any other language is believed to be broken down into some combination of these components.

The LLVM instruction set consists of 31 operations. Some operations are overloaded and can take operands of different types, which has reduced the number of operations. For example, the add instruction can add integers or floating point numbers.

LLVM comes with a textual code representation (LLVM IR code) that can be read by humans and is what our compiler will produce.

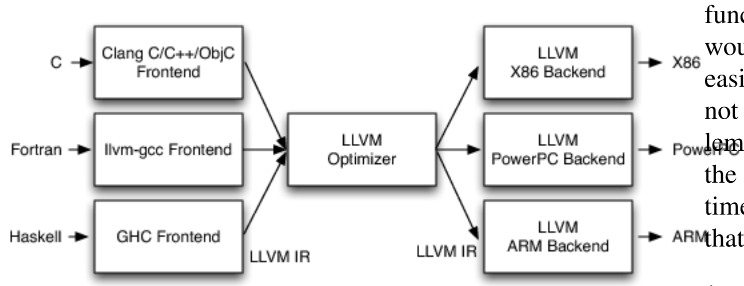


Figure 3. How LLVM is used to optimize code (Lattner 2016)

```
int add1(int a, int b) {
  return a+b;
}

define i32 @add1(i32 %a, i32 %b) {
entry:
  %tmp1 = add i32 %a, %b
  ret i32 %tmp1
}
```

Figure 4. Example of how a simple function looks like in LLVM

Figure 4 shows a simple function that adds two integers in SimpliC and the same function in LLVM IR code. The `%tmp1` variable is a virtual register. The generated LLVM IR code is quite similar to the original SimpliC code. A basic block in LLVM can begin with a label, like the `entry` label in Figure 4 and must end with a terminator, like the `ret` instruction, which returns a value to the calling function. The keyword `i32` is a 32 bit integer type.

4. Implementation

Our work started with extending the compiler with an LLVM intermediate code back end with the existing features of SimpliC. This included integers, arithmetic expressions, comparisons between integers, loops and (non-nested) functions. We then extended the language with nested functions and more types, and implemented the extensions both in the LLVM backend and the x86 backend.

4.1 Nested functions

We will now describe the implementation of the nested functions both in the LLVM backend and x86 backend.

4.1.1 LLVM

The first version of the LLVM implementation of nested functions was to simply put them in the function they were defined in as a ”loop” that would be run once when we wanted to call the nested function by jumping to the functions label. If the nested function’s basic block was encountered in the enclosing function (not called by the enclosing function, just encountered because it was defined there) it would just be skipped. We thought this solution would be easily readable and debuggable. However, this solution did not fully support recursion, and we quickly ran into the problem of a nested function allocating local variable. In Figure 5 the local variable `a` in the function `f` needs to be allocated 6 times, and each new allocation needs to be a new variable that does not interfere with the previous ones.

```
int main() {
    f(5);
    return 0;

    int f(int n) {
        int a = 0;  // allocate a variable
        if (n == 0) {
            return 0;
        }
        f(n - 1);
        return 0;
   }
}
```

Figure 5. A nested function that allocates a local variable

Local variables were allocated using the `alloca` instruction in LLVM which allocates memory on the current stack frame and returns a pointer to the allocated memory. The current stack frame is based on which function we are in, and `f` was not made a function, just a ”loop”. The 6 pointers that must be allocated must then have unique names (since they were defined in the same namespace). The same line of code was handing out the same name 6 times, which was a problem.

We solved this problem by moving all of the nested functions out of their enclosing functions and used LLVM’s `call` instruction to call them when needed, instead of jumping to a label. Now that the nested functions were considered functions in LLVM, the pointer names returned by the `alloca` instruction would not collide, as they were all local to their respective functions. To illustrate this a bit clearer, consider the C code in Figure 6 roughly equal our first LLVM implementation of nested functions. The `while` loop is meant to illustrate the function `f`.

```
int main() {
    int n = 5;
    while (n != 0) {
        int a = 0;
        n--;
    }
    return 0;
}
```

Figure 6. A C program

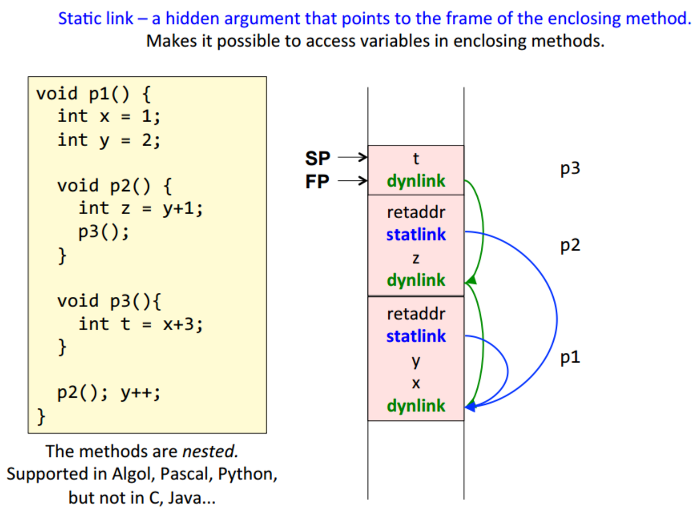
This program would not create 6 different variables named `a`, but would just redefine the same variable 6 times, much like our first LLVM back end (except redefining variables in LLVM is not allowed). To achieve the intended behavior of having 6 different variables in C, the `while` loop can be turned into a function and called recursively.

However, this implementation lead to another problem, local variables in the enclosing function could now not be referenced, since they were local to another function. This was solved by adding the needed local variables as extra implicit parameters to inner functions when they were called. All local variables were implemented using pointers, so a nested function changing the value of an enclosing function's local variable would work as intended. Only the needed variables would be passed to a nested function to avoid unnecessary amounts of extra arguments.

4.1.2 x86

As in the LLVM backend, the backend for x86 moves nested functions out of their enclosing functions. Nested functions were moved out of their enclosing functions. The problem of accessing enclosing function's variables was solved using static links(Hedin 2011). Every time a function was called (nested or not) a dynamic link would be put on the stack. The dynamic link always points to the *calling* function's stack frame. Then a static link would be put on the stack that pointed to the *enclosing* function's stack frame. These pointers can differ because a nested function can call it self recursively, like in Figure 5. These pointers will all be unique. In Figure 5 the static links should always point to the stack frame of the function `main`, since `main` is the enclosing function of `f`. These pointers should all be the same for all calls to the function `f`. The static links allowed a nested function to always find the stack frame of an enclosing function and from that point find local variables belonging to the enclosing function, without caring about where it was called from.

Figure 7 illustrates how static links work, where the functions p2 and p3 are called. They both have `p1` as their enclosing function so their static links will point to `p1`'s stack frame.



Static link – a hidden argument that points to the frame of the enclosing method.
Makes it possible to access variables in enclosing methods.

```
void p1() {
  int x = 1;
  int y = 2;

  void p2() {
    int z = y+1;
    p3();
  }

  void p3(){
    int t = x+3;
  }

  p2(); y++;
}
```

The methods are *nested*.
Supported in Algol, Pascal, Python,
but not in C, Java...

Figure 7. How static links were implemented (Hedin 2011)

4.2 More types

The LLVM back end was extended with support for floating point numbers, booleans and void functions. LLVM already supported all of these types, so implementing them was not a lot of work. We decided to use the 64 bit `double` type for floating point numbers because of LLVM's inability to properly handle 32 bit floating point constants. The extended type system was never added to the x86 back end and no casting or conversion between types was implemented due to the project deadline.

5. Evaluation

All testing was done on the same computer in one of the computers on LTH running a unix environment. All test programs were run shortly after each other with no other programs running to minimize noise in the data. The LLVM IR code was optimized with the LLVM optimizer[1], compiled into x86 assembly with the LLVM static compiler[2], assembled using `as`[3] and lastly linked using `gcc`[4]. Optimizing and linking was done with the respective `-O3` flags.

[1] The LLVM optimizer: `http://llvm.org/docs/CommandGuide/opt.html`

[2] LLVM static compiler: `http://llvm.org/docs/CommandGuide/llc.html`

[3] The as command: `https://sourceware.org/binutils/docs/as/`

[4] GCC: `https://gcc.gnu.org/`

```
int main() {
  int a = 0;
  print(fib(39 , 0));
  return 0;


  int fib(int n) {
    a = a + 1;
    if (n < 2) {
      return 1;
    }
    return fib(n-1)+fib(n-2);
  }
}
```

Figure 8. The program that was used for evaluation

We evaluated the LLVM backend and x86 backend by doing performance tests on a program with a nested function, see Figure 8. We varied n from 35 to 40 and for every `n` we did 5 tests where we increased the number of variables. In the first n was set to 35 and the only variable was `int a`. In the second test for `n = 35` we had another variable, `int b` (See Figure 11 in Appendix). All these variables were incremented in the function once per call. Every combination of `n` and number of extra variables was run 5 times.

For every `n` we measured the execution time and then computed the confidence interval for the measured values that we received from the 5 tests for this specific `n`. We computed the 95% confidence interval, which means that the mean of our measured values have a 95% chance to lie between the interval. In Figure 10 we have plotted the confidence intervals for the measured execution times in user time over the number of variables. For every variable the upper and lower confidence bound(the interval) were plotted. As expected the optimized LLVM code was faster than the unoptimized x86 code. The confidence interval was calculated the following way in MATLAB:

```
Define the measurements as a vector
x = [1.1, 2.2, 3.3]
We will now compute the confidence interval
with the 95% confidence level.

To get the number of measurements

length(x)

if n < 30, use Students t-distribution:
t = tinv([0.025  0.975], length(x)-1);
c = mean(x) + t*std(x)/sqrt(length(x))

The variable c then contains the confidence interval and
c-mean(x) is the distance from the mean value.
```

Figure 9. How we calculated the confidence interval in MATLAB.

GCC has support for nested functions as well. We wrote the same program from Figure 8 in C and then compiled with GCC, where we set the optimization flag to `-O3`.

5.1 Results

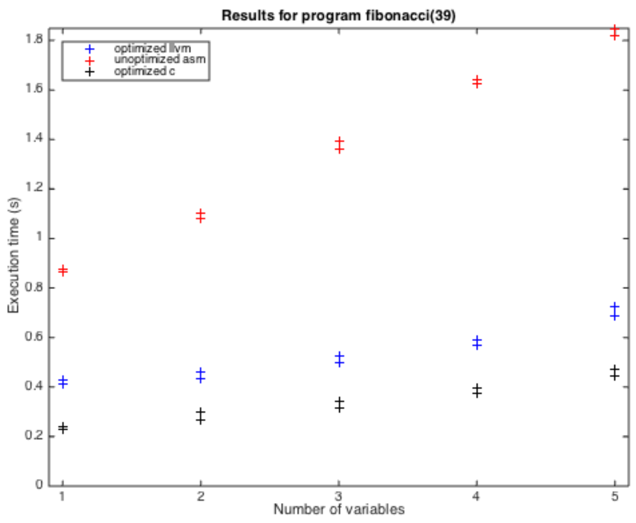


Figure 10. Comparison of execution times between optimized llvm, unoptimized assembler(x86) and optimized C code.

6. Conclusion

In this project we have extended the SimpliC language with nested functions and more types, and added a LLVM-backend to the SimpliC compiler. Implementing an LLVM-backend was easier than implementing the original compiler, because it is easier to understand LLVM IR since it has some features that higher-level languages have.

The optimized LLVM code was much faster than unoptimized x86 code and the execution time increased when we increased the number of variables. This came as no surprise, since no optimization effort was made when generating x86. The focus was always to just get it to run.

The C code was significantly faster than the optimized LLVM code which we did not expect. This could be because C is a lower level fast language and that GCC has support for optimizing programs. It would be interesting to compile a C program with Clang to LLVM and then compare the same program written in SimpliC compiled to LLVM, and see if the the result would be different from when we compiled the C program with GCC.

Further work on this project could of course be extending SimpliC with more features to make it a more complete language, such as, logical bitwise operators(and, or, and xor), structs/classes, arrays or strings.

No measurements on compile times was measured, none of the programs took any significant time to compile. All test programs were very small however, so a larger program would probably be needed to achieve any interesting results.

7. Appendix

```
int main() {
  int a = 0;
  int b = 0;
  print(fib(39 , 0));
  return 0;


  int fib(int n) {
    a = a + 1;
    b = b + 1;
    if (n < 2) {
      return 1;
    }
    return fib(n-1)+fib(n-2);
  }
}
```

Figure 11. The program that was used for evaluation with two variables.

References

R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 1991. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.100.6361&rep=rep1&type=pdf`.

T. Ekman and G. Hedin. The jastadd system - modular extensible compiler construction. *Science of Computer Programming*, 2007. URL `http://www.sciencedirect.com/science/article/pii/S0167642307001591`.

G. Hedin. Lecture notes, compilers course, lund university implementation. 2011. URL `http://fileadmin.cs.lth.se/cs/Education/EDAN65/2016/lectures/L10.pdf`.

C. Lattner. *The Architecture of Open Source Applications*. lulu.com, June 2016. URL `http://www.aosabook.org/en/llvm.html`.

C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, March 2004.

LLVM: LLVM Language Reference Manual. LLVM project, November 2016. URL `http://llvm.org/docs/LangRef.html`.