Jason Fong

Brent Lee

CSC 431

Aaron Keen

Mini Compiler

(Design of the world’s worst compiler)

Overview

The compiler is built by running “make” using the given makefile. This creates a “MiniCompiler” executable which can compile .mini programs. The chosen language for implementing the compiler was Java for two reasons: familiarity from all parties, and easy use of the given classes from the provided parser.

Parsing

After compiling, the user can provide two optional flags. The –llvm flag compiles the given mini program into an equivalent LLVM program. Without the flag, the default behavior is to compile the mini program into ARM assembly. The –stack flag will indicate to the compiler that a stack based approach should be used for variable allocation, as opposed to the register based allocation that also takes advantage of static single assignment.

The first step of the compilation process is to tokenize the program into a list of strings using the given lexer. Following the lexer, the parser analyzes each token and translates each string into a tree of statements and expressions. This tree of abstract objects and syntax representation is interestingly called an abstract syntax tree (AST).

Following the creation of the AST, a symbol table and struct table are created for static type checking as well as generating the control flow graph. These two tables are implemented as hash tables mapping variable/struct names to custom objects containing information on the declared struct/variable. The symbol table maps to a class which contains fields such as type, parameters (if the identifier maps to a function), and whether the variable is global. The struct table maps to a class that contains the name and a type map for the struct’s fields.

Static Semantics

After table creation, we are all set to begin type checking. Each statement and expression implements a type check method which passes along the tables and return type of the function, and also returns the type that the expression/statement results in (e.g. calling type check on a == 0 will return a BooleanType object). The statements and expressions recursively call the type checking methods, and if there is a mismatch in expected type and the returned type, the system prints and error and aborts the program. The typeCheck functions also check that each path of the function eventually returns and that it returns the correct type of value. A block statement returns if any of the contained statements is a return statement. A conditional statement returns if both paths contain return statements. All functions either require a return statement unless the declared return type is void. Once type checking is complete, the core process of compilation begins.

Intermediate Representation

An intermediate representation (IR) is the compiler’s internal representation of source code. It is important and useful for two main reasons: it makes optimization of the code easier and allows you to create assembly for multiple machines without starting from scratch. This makes the source language much more portable across different machines with different instruction set architectures.

Control Flow Graph

The “backbone” of any compiled program is the control flow graph (CFG). It is an intermediate representation of the conditional logic, broken up into “blocks” of code that split whenever a decision needs to be made (i.e. if statements, while statements, and return statements).

In this compiler, LLVM instructions were added to the CFG as the graph was being constructed. Each statement implements a createCFG method which adds the LLVM instructions of all expressions contained in the statement. If the statement contains further statements (e.g. BlockStatements), the createCFG method is called recursively. Expressions add LLVM instructions that are equivalent to the AST representation and returns the pseudoregister which contains the result of said instruction.

While statements and if statements are noteworthy because of their alteration of the program’s control flow. New blocks are created in their createCFG methods because the program can take different paths at this point based on whether the guard expression is true or not. The new blocks are linked to the parent block by a list of predecessors and successors found in every block. These lists are also necessary in the implementation of phi functions. Phi functions are used to get the value of a variable that could be assigned differently based on the flow of the program. Phi functions requires the program to conform to static single assignment form(SSA). SSA requires that values are assigned to only once in the program. Once a value is assigned, it cannot be changed. Instead, a new value is assigned and the change is represented in a map. Fetching the values of variables and assignment of variables are all done through the map. SSA makes many compiler optimizations (constant propagation, dead code elimination, register allocation) simpler and more effective by simplifying the properties of variables.

Optimizations

Sadly, no optimizations were implemented for the compiler.

Code Generation and Register Allocation

The translation of LLVM to ARM code is a relatively mechanical and simple process. Many LLVM instructions either have a 1 to 1 translation into ARM, or are expanded into a few instructions. This portion of the project was easier on the brain, but harder on the fingers. There were a lot of classes to create and methods to implement.

The ARM code created in milestone 4 is mostly there, but cannot be run because the values are all stored in pseudoregisters with names from the LLVM code. These pseudoregisters must be translated into actual ARM register names (r4-r10 in the case of this compiler) for the program to be runnable. The allocation happens in 4 steps:

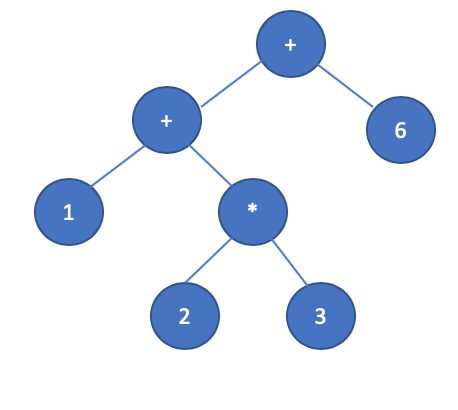
1. Generate gen and kill sets
   1. For each instruction in each block, add each source to the gen set if it is not in the kill set
   2. Add target to the kill set
2. Generate live out sets
   1. Starting at the bottom block, union the gen sets of its successors with the difference of the live out set and kill set.
   2. Do the same process for every block moving up
   3. Repeat process for whole graph until no more changes occur
3. Create interference graph
   1. For each block, iterate through its instructions from bottom to top.
   2. Remove the instruction’s target from the live set.
   3. Add an edge in the interference graph from the removed target to every element in the live set.
   4. Add each source in the instruction to the live set.
4. Color interference graph
   1. Remove a node and its edges from interference graph, place it into stack.
   2. Reconstruct the graph by popping from the stack and assigning a color that is different from all its neighbors.
   3. If no colors remain, we must add spill code to shorten the live range of a variable and recompute everything.

Other

Data Structures

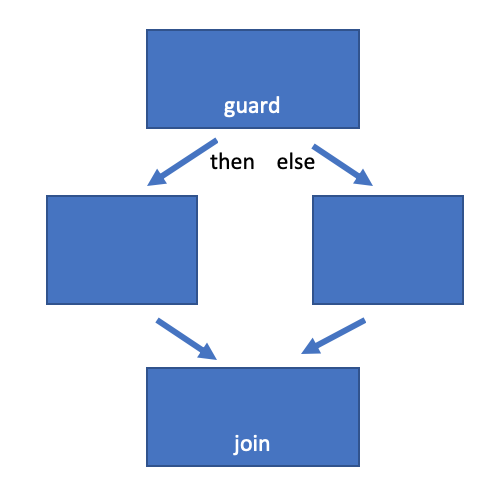
The first and biggest data structure used is a tree. We used a tree to represent the Abstract Syntax Tree and the order of which the code was parsed. The tree is read in so that it does depth-first search and is left associative so that binary operators can behave like they are supposed to. Here is an example

1 + 2 \* 3 + 6

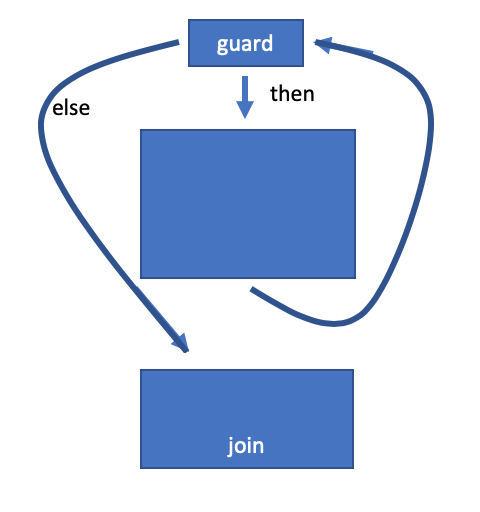


Another data structure used is a graph for the intermediate representation in LLVM. The graph, specifically the Control Flow Graph, is used to represent the flow of the graph. There are different ways to represent a loop, but we used one where the conditional guard is in a separate node, and the body is another node, and the exit is another node, as well. Here is an example of a conditional and a while loop as used in our compiler.

Conditional CFG

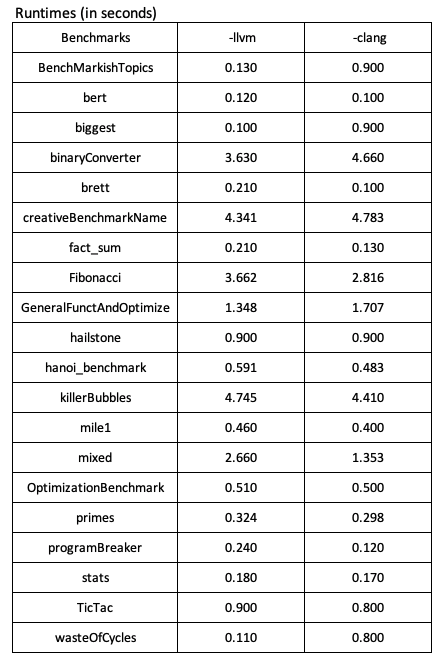


While CFG



Another use of graphs was in the interference graph for register allocation. The interference graph is an undirected graph where each node represents a pseudoregister, each edge represents interference of the two pseudoregisters, and the color of each node represents an actual ARM register.

Performance



After the processes of generating ARM code and allocating registers, only two benchmarks were able to run on the given long inputs and produce correct outputs. Some programs wouldn’t compile, some would produce incorrect output, and some would result in segmentation faults. For the two that worked, my compiler could hold its own compared to GCC without optimizations. Between the two benchmarks, there was only about a 10% decrease in runtime. However, using GCC to its full potential with optimization blows the world’s worst compiler out of the water. It ran 30 seconds faster on the mixed benchmark and 47 seconds faster on the Fibonacci benchmark, averaging an 87% decrease in runtime.

The stack-based LLVM-derived executables produced correct ouput for all the given inputs. Therefore, this configuration is used for comparison against clang. The experiments were run on the school Linux servers. The average run times for some benchmarks were faster with my produced LLVM code, but that could be due to the scheduling of the OS or load on the server at the time. Overall, the executables produced by clang were faster.

Analysis

The generated ARM code is about as inefficient as you can get. There is no branch folding, no constant propagation, no anything. Every r4-r10 register were pushed and popped for the sake of simplicity, but this has a cost on performance.

The compiler is a little buggy for complex programs. For milestones 2 and 3, however, it works on all benchmarks. Past that, not all of the benchmarks pass or even compile correctly.

Calling Convention

Other than the fact that we pushed the r4-r10 registers onto the stack instead of choosing which ones are used, the caller saved registers are required to be saved and restored by the caller. The r4-10 registers, since they must be pushed onto the stack first and then popped off at the end of the function, require 14 memory operators per function call, which is not efficient at all.

Conclusion

In conclusion, compilers are complicated pieces of software. Making the compiler initially was not the terrible part. Optimizing the compiler and using registers instead of the stack are the hard parts. Our compiler, overall, works as a compiler is expected to work; however, the thing is that it is not fast at all. This class, in general, was very useful because knowing how something works on a lower level allows us to become better programmers as a whole even on the higher level end.