Project 3 Design Document: Code Generation

Maddie Mott (mmott@mit.edu), DeJuan Anderson (dejuan@mit.edu), Youyang Gu (yygu@mit.edu), Desai Chen (desaic@csail.mit.edu)

- Division of labor
    - Maddie: statement and control flow handling, testing, cleanup, bugfixing
    - DeJuan - all expressions involving operations, conditionals, and comparisons. testing, cleanup, bugfixing
    - Desai: - cleanup, compacting and streamlining code, Context creation, testing, bugfixing, all declaration types
    - Youyang: all constant expressions and ones involving arrays, testing

Overview

For this phase of the project, we decided to take the second option of the ones presented in class for code generation. We decided to forego directly generating a low-level IR, and instead went from our AST, directly to assembly code by tree traversal in conjunction with a new Context class and interpreting code to convert each node in the tree into assembly code. We divided up the work by person and into several stages, and mainly worked independently.
We agreed to use the stack machine model for this phase of the project.

Design Decisions

Low Level IR

We initially planned to convert the high-level IR to low-level IR before finally generating assembly code. Upon discussion, we realized that the process to generate the low-level is not trivial, as we need to be able to flatten the entire tree of nodes while creating new variables for every new assignment. In addition, we must *still* generate assembly code from the low-level IR. With the difficulty in scheduling a time for all 4 members of our group to meet, and the conflicts with other classes for time, we decided bypass the low-level IR and instead focus on generating our assembly code. We recognize that this will make optimization and control flow analysis much more difficult. But at this stage, we want to prioritize having a working decaf compiler over optimization. This allows us to quickly get started on the important part of this phase of the project; we saved a lot of time by not writing a second IR and instead going straight to assembly. Doing this also gives us more time to understand how code generation works, allowing us to be able to do a better job of building the low-level IR for the next stage of the project.
(We currently represent assembly instructions as a triplet(mnemonic, arg1, arg2). To generate a low level IR instead of assembly code, we can simply replace the arguments with their symbolic representations.)

Choosing a Code Generation Model

As described during the project information session, we have 3 options for doing our code generation: single static assignment, infinite register machine, and stack machine. We all agreed that single static assignment is the most ideal direction for code generation, as it gives us a control flow diagram and allows us to do program flow analysis relatively easily. However, implementing the phi function is daunting, and none of us knows how to do it. Between infinite register machine and stack machine, we decided not to go with infinite register, for the simple reason that we do not have infinite registers to work with. Plus, we do not want to deal with register allocation at this stage. The stack machine model seems fairly straightforward: push everything to the stack. Whenever the program needs something, it simply pops them off the stack. The code is relatively straightforward and simple, so that is the model we decided to pursue.

Design Details

Overview

We have a method to handle the code generation for each of our high-level IR node. Most of the methods are implemented recursively, and we used the stack machine model to great effect; we pushed and popped variables and resolutions of instructions to and from the stack between invocations of our code generation structures. The key invariant we are relying on is: *everything gets pushed to the top of the stack*. The caller is then responsible for popping the return value off the stack, and handling the return value appropriately.

Each method takes in a high-level IR node and outputs a list of assembly instructions that corresponds to the instructions in that node. By traversing through the IR tree, we will be able to generate a set of instructions for the whole program.

Instruction

For every IR node, we return a list of Instruction objects. Instruction is a class that represents a single line of assembly (which can be generated via “toString”).

Codegen

This is the main function for all of our code generation, and holds all the methods to generate assembly code based on the IR node.

CodegenContext

Contains the symbol table and accumulates sizes of local variables. Also provides several helper methods such as generating unique jump labels. It can also be used to generate unique temporary variables for generating low-level IR.

LocationMem

This is the superclass that contains information about the memory locations of everything we are tracking. It has a type (stack, label, literal, array, register) and a “toString” method that outputs the assembly representation of the memory location.

Array Location

1) Get the label/address of ID (pointer to the beginning of the array).
2) Get the index. If it’s an expr, push it to the stack. If it’s a literal, we can use it directly.
3) Two cases:
    a) Global variable: If index is on the stack, pop it into a register and directly append the register to the array address. If the index is a literal, multiply by 8 and add it to address of the beginning of array loc.
    b) Local variable: If index is on the stack, pop it into a register and add its address to %rbp. If index is a literal, multiply it by 8 and add it to the beginning of the array loc and %rbp.

Field Declarations

- Global variables and arrays are stored in the .bss section of the header. String constants are stored in the .rodata section.
- Local variables are allocated on the stack at the beginning of a method declarations. The maximum size of all local variables that can potentially be used in any execution path is computed in order to allocate the right amount of stack. This is the same behavior as unoptimized c code.
- These locations (usually as a LocStack) are an attribute of the variable descriptors which are accessible in the context symbol table

Method Declarations

- Callouts are simply added into the symbol table.
- Decaf methods
    - Function parameters that live in registers are immediately pushed on to the stack
    - Then more stack is allocated for all potential local variables usage
    - At the end of the function, a runtime check is generated to check if control falls out of the function.

Literals

Whenever we encounter a literal IR node, we make a LocationMem (LocLiteral) object with the value of the literal and push it onto the stack, although some additional handling is needed for integers requiring more than 32 bits.

Expressions

Expressions are fairly complex; they may be one of many different things, including nested expressions. For this reason, in the main generateExpr method, whenever we encounter an expression IR node, we check the type of expression. Based on the type, we convert the generic expression node into a more specialized form approrpriate for its type and dispatch it to a helper method. For example, if we encounter an Expresssion node that, on examination, turns out to be a ternary expression, we would then cast it into an IR_Ternary node and dispatch it to a method that specifically handles assembly generation for ternary expressions. Because of the potentially nested nature of expressions, many of the helper methods have recursive calls to the main dispatcher.

This allows us to solve the problem of nested expressions by writing a structure that handles basic cases and calls itself on larger problems made up of multiple basic cases, to break the problems down and analyze each part. Once we solve the subproblems, we can solve the main problem.

So to continue our ternary example, after the IR_Ternary is dispatched to the helper, the helper calls generateExpr to resolve the expression that we will be using to resolve the tenary operator. The helper also calls generate Expr to evaluate both the “true” and “false” branches, and generates assembly for each, then jumps to the proper instruction set.

The accepted convention for generateExpr is that when the returned instructions are executed, there will be one additional value on the top of the stack, which is the value of the expression.

If

Generating if and ternary statements work like this: we recursively generate the set of instructions for the true and false blocks. We push the result of the boolean statement onto the stack, and do a “cmp”. We then jump to the correct set of instructions accordingly.

For and While

To produce for and while statements, at the top of the loop, we check that the conditions to proceed (haven’t exceeded max while loops/while loop condition still true or for end value not yet exceeded) are valid. If they are, we fall into the block. At the end of the block, the loop variable is updated (or for while loops with a maximum number of loops, the loop counter is incremented).

Testing

In addition to the provided tests, we wrote several other tests to verify that the correct behavior is exhibited in certain cases, for instance, we wrote a test to ensure that if control fall sout of a non-void function without returning a value, a runtime error occurs.
We also wrote several decaf programs like binary search and find-longest-palindrome to check that code to perform some sequence of operations is likely to be correctly generated.

Known Bugs/Issues:

- Our compiler will currently begin to exhibit undefined behavior if the number of branches requiring labels exceeds 2^32.
- In fact, if there are more than 2^32 statements, our compiler will most likely exhibit a segmentation fault; however, this is more of a limit of stack memory than a fault of our compiler.

Next Steps

Our current scheme of going directly from high level IR to assembly makes it hard to generate more optimized code, since it would be difficult to run program analysis on this design. Therefore, we will mostly like need to design a low-level IR in order to run code optimization passes. That will be our first focus of the data-flow analysis project: design a low-level IR. This involves understanding the control flow, having a flattened structure, assigning temporary variables, etc. Having a low-level IR will enable us to perform various optimization techniques such as common subexpression elimination, copy propagation, dead code elimination, and algebraic simplification. Currently, we are unable to do those things because we do not have a flat control flow graph on which we can run the program analysis: a high-level IR only contains information about each node, and not about the specific statements or blocks themselves. Once we generate a low-level IR, we will be able to write code that will do various optimization passes and modify the structure of the IR each time. We will then generate assembly code from the final IR structure.