6.035 Project 2 Write-Up

Nicolaas Kaashoek, Endrias Kahssay, Tony Wang

1 Design

1.1 Control Flow Graph

We transform the IR outputted by project 2 into a control flow graph that allows us to plot the execution path of a program. To start with, we turn each field declaration at the top level of a program into a global variable. We represent these variables as `VRegisters` (virtual registers), which each have a unique identifier (UUID), and store this information in a `VRegScopStack` to keep track of them. After handling the global variables, we handle the functions in a program.

We begin by pushing a new scope onto the scope stack, turn the parameters into `VRegisters` and add them to the scope. From there, we generate the sub-section of the control flow graph corresponding to the current function. We take the `BlockNode` that represents the function's code, and begin by adding another scope to the stack for that block. From there, we add the field declarations to the scope as `VRegisters`, and proceed to the statements. We have two kinds of blocks in the control flow: `BasicBlocks` are used for general instructions, while `BranchBlocks` mark a split in the control flow.

`BasicBlocks` consist of a set of instructions and an exit point. We have a number of instruction classes that are closer to assembly instructions than the statements created by project 2. Converting our program into statements such as this makes it *much* easier to actually generate the assembly later. The format of these instructions was heavily inspired by the intermediate instructions created by LLVM (`https://llvm.org/docs/LangRef.html`).

We expand each statement into a sub-section of the control flow and get the start and end point of that sub-section. We chose this approach as a single statement may expand into many instructions. For example, the statement `a = (b + c) * (d + e)` needs to evaluate `b + c`, and `d + e` and then multiply these results together and store them in a `VRegister`. Once we have the start and end points of the statements, we are able to join all the statements together by assigning the end of the previous statement to the start of the next statement. In addition to returning the start and end of the statement, we also return the `VRegister` containing the final result of the expressions. This ensures we can use the value later in the program (as an assignment for instance). This allows us to generate the control flow in a program.

It needs to be noted that the above process does not generate *maximal* blocks in the control flow, and maintains a large number of nops as well. We handle this by making a second pass over the control flow to create a maximal graph.

1.1.1 Handling Branches and Short-circuiting

The way we handle statements also deals with statements that cause branches (if, for, while). When a comparison expression is encountered in the code, we first look at the left side of the comparison and evaluate it. We then connect this result to a `BranchBlock` which takes a different path based on the result of the left side. We follow the short-circuiting paradigms discussed in lecture, with false in an `and` and a true in an `or` allowing us to skip the right side of the comparison. As with other arithmetic expressions, we store the final result of the comparison in a `VRegsiter`.

Once we have the `VRegister` containing the final result of the comparison, we can hand this to another branch block which controls the direction a loop takes. For example, in the statement `if (a>b){...}`, the final result of `a>b` is stored in a `VRegister`, and then a `BranchBlock` is used to take the corresponding path based on the blocks the if statement points to. The same logic is applied to for and while loops.

In the case of for loops, we simply attach an extra instruction to the end of the loop body which executes the loop increment expression. For while loops, nothing extra needs to be done.

1.2 Maximizing the Control Flow Graph

Once we have the control flow graph as described in the previous section, we need to make it maximal to avoid unnecessary `jmp` instructions in the assembly. To do this, we pass over the graph and collapse any `BasicBlock` to `BasicBlock` transitions. The only exception to this is at the end of a comparison branch. The last instruction in each comparison moves either true or false into a VRegister. Both of these `BasicBlocks` then exit to the same point. We avoid collapsing these blocks and their exit points to more clearly mark the end of a comparison statement. It also simplifies assembly generation.

1.3 Conversion to Assembly

Once the control flow graph has been maximized, we can pass over it an generate assembly. Each kind of instruction in a `BasicBlock` is turned into assembly using a simple templating method. We maintain a map from `VRegisters` to locations on the stack which allows us to easily look at where a given variable or expression result was stored. Using the stack for everything is obviously less than optimal, but works well for this project. This will be a large focus of our optimizations in the future.

Each block in the control flow graph is also given a label in the assembly that allows the exits of blocks and branch blocks to be easily converted to `jmp` instructions to those labels.

Global variables are simply stored at the top of the file using the `.comm` directive, and imports are automatically handled by the linker.

Because we primarily use the stack as the location for variables and temps, we use only use `r10, r11, and rax` as our intermediate registers. This maintains the callee-caller conventions. Arguments are handled by placing them first into the six standard argument registers, and then onto the stack. This follows standard calling conventions as well. Finally, the return value is put into `rax` as expected by the program. We use `leave` and `ret` to take care of the function epilogue, but manually use set the stack and base pointers in the prologue to a function. Currently the offset of the stack pointer from the base pointer isn't as close as it could be, but again, this project doesn't focus on optimization.

2 Extras

2.1 Assumptions Made

The only assumption we made was the addressing of the return type of an argument as a runtime check. In our semantic checker project, we checked the return types of functions and checked for return statements as well, so we kept this strategy as opposed to failing at runtime. The code returns the expected error code for these problems as well.

This does impact one of the hidden test cases, causing it to fail as it expects assembly to be generated. To address this, we generate some dummy assembly on such an exit code that just exits. This is done purely to pass the test cases and is completely unnecessary otherwise.

2.2 Graphviz

In order to make debugging easier, we chose to export our CFG to the dot language used by GraphViz when the debug flag is supplied. The dot files are generated in the cfgGraph folder, and can be compiled with the command `dot -Tps FILENAME.dot -o OUTPUT.ps`. An example of the output of a cfg is included in that folder by default.

2.3 Unit Testing

As mentioned the previous project, we have provided our own unit testing scripts as well, implemented with scalatest to improve the speed of as well as general usability of the testing environment. These scripts as

well as our own test cases can be found in the test directory. The testing script also runs the tests provided by course staff.