Random Compilers

Design

The “cfg” folder of our project contains all the classes needed to make a CFG representation of the code. Each basic element of the program has an associated CFGLine, and these are clumped together into CFGBlocks which have lists of CFGLines within them. (CFGBlock is technically a subclass of CFGLine with the extra field for a list of lines, so that they can take advantage of common functionality.) CFGLines know their true and false children CFGLines (which are the same for non-branching statements) and how many parents they have (to determine whether they are a merge of control from multiple previous statements). The CFGCreator file goes through the IR in a single pass to make the CFGLines for each object and add pointers connecting them to each other, then makes another pass to condense these into blocks, starting a new block whenever it sees a branch or merge. Meanwhile each CFGLine is assigned a block, so it can point to the block it is in.

There are various types of CFGLines: CFGExpression, CFGDecl, CFGMethodDecl, CFGStatement, and CFGNoOp. Each stores the relevant IR object. As CFGCreator runs through the program, it keeps a stack of loops that it is currently in (one for each for/while statement it has entered) and where they start and end, so that when it encounters break or continue statements it can cause that node of the graph to point to the correct next line. Expressions are short-circuited as described in lecture notes.

The actual code is built in two files. There’s Assembler, which takes in methods and an outputstream, and is in charge of making sure the entire assembly is generated and output to the stream. It does this by adding a few global things and then creating a different (misnamed) BlockAssembler for each individual method’s code generation. The BlockAssembler stores a number of important things about the method: how many variables it has allocated, a table for them, how many blocks or string literals it’s written so far (so it knows how to number the next one), what type it’s supposed to return, etc. It starts with the first block of the method, generates code for it, then does a DFS on the block’s children generating code for anything that’s not already in the map from labels to code, and placing jumps to the label if it is already in the map. Within blocks it does an ugly mess of switch statements and helper functions to figure out what sort of thing it is supposed to be handling and handle it.

The code currently only uses registers %r10 and %r11 and very rarely %r9; it’s assumed that calls to helper functions will generate code that puts the desired temporary value into %r10. Other registers are basically only used for passing arguments. Unary expressions currently handle “not” by subtracting the boolean 0/1 value from 1 and “negate” by subtracting from 0.

One design choice we partially embraced was using the visitor pattern. Our code works similarly, but involves a lot of unchecked type casting, which added not only to the complexity of the code, but also introduced bugs which we then had to track down. Going forward, we would like to clean some of this code up, as this will likely help eliminate some of the difficulties we have faced and described below.

Another design choice we used was having many of the same types present in the IR, symbol tables, and control flow graphs. While this cut down on the code we needed to write, it has introduced a lot of interdependence between the different parts of the code. While this code eliminated some storage overhead, it has definitely made the logic more complicated, possibly

adding more work than the overhead would have been. This is also another likely candidate for refactoring.

We also neatened up the semantic checker. We made sure it passed all public/hidden tests. We also added exceptions to avoid accidentally passing around null pointers, as in the case of getting the expression associated with a possibly-void return statement. We also refactored how we parse a ConcreteTree to the IR, by creating static methods in ASTCreator as opposed to most IRNode subclasses having a constructor with a ConcreteTree input and passing between constructors. In addition to this allowing us to have all of the code in one place, it allowed us to do things like parse “-1” as the integer literal with value -1 as opposed to the unary expression consisting of the tokens “-” and “1”.

Extras

We have added the possibility for unit testing to our project, which involved adding the JUnit libraries as well as modifying the build file to add the target ‘ant unit-tests’, which runs all of the available tests directly through the command line.

Difficulties

Having code generation spread across so many different functions adding strings together seems inelegant and error-prone; is there a better way to structure this? (Frankly, everything feels inelegant, and we’ve been staring at it a long time.) We’ve also only partially dealt with possible issues arising from variables of the same name in different scopes, though we have made sure that all temporary variables created will have unique names.

At this point, we’ve eliminated many of the bugs in the program, though not all programs we have written work fully. In particular, horrible.decaf in tests/moar_codegen compiles, but segfaults on execution. Additionally, it seems that Athena does not like Arkadiy running some of the tests cases, as test case 10 for codegen refuses to work there, despite working for Maya and Jackie.

Contribution

Arkadiy - Refactored earlier code to be more elegant. Broke down compound expressions into smaller steps with temporary variables, as well as fixing the general structure of how expressions are converted into assembly. Fixed input parameter register/stack handling for calling methods. Debugged various problems associated with expression breakdown.

Jackie - Wrote the CFGCreator to destruct and shortcircuit IR nodes into CFG lines, and the function to condense those lines into blocks. Wrote the preliminary version of codegen to get everything at least producing code. Fixed variable lookup, argument passing, for/while break/continues, array indexing. Implemented runtime out-of-bounds checking. Wrote up this document.

Maya - Fixed remaining problems with the semantic checker. Updated variable tables to include stack offsets, and handled global variables. Worked on issues with array vs not-array and global

vs local accessing. Tracked down bugs. Added a VariableDescriptor class, to pass around stack offsets in addition to name/type.