6.035 Phase 2 & 3 Documentation

Decaf Optimizing Compiler

An Bo Chen, Alex Dang, Youran (Yoland) Gao
{anbochen, alexdang, youran}@mit.edu

Design

Intermediate Representation

Using the abstract syntax tree (AST) generated by the ANTLR parser generator, we processed the AST to generate our own intermediate representation (IR) in `DecafIrVisitor` following a visitor design pattern. First, we defined `IrTypes` to declare various abstract and case classes for each of the nodes in our IR. The hierarchy of the `IrTypes` classes resembles the Decaf grammar where each class contains fields of relevant information. As we visit each node in the AST, we return a node of the corresponding `IrType` and build symbol tables for each method and block. These symbol tables map identifiers to `IrMemberDec`l types representing a declaration of a variable with a type.

During our IR tree construction, we perform some initial semantic checks such as checking for duplicate identifiers within a given scope, array declarations having either a declared length or an initializer list but not both, and void parameters. These errors cause an IR that cannot be represented without error, so the checks were done here instead of the semantic checker and documented at the top of the file. In the visitor, we also converted the cases of `++` and `--` to instances of `+= 1` and `-= 1` to reduce the amount of case classes we have to deal with.

Special Case: Ternaries

We decided to remove ternaries at the intermediate representation level. This is possible because for most statements in the intermediate representation, a ternary expression is semantically identical to a if-else statement evaluated directly prior to the statement.

```
// original statement
a = i < 5 ? 1 : 0;
// is semantically equivalent to the following:

// 1. Evaluate the ternary
.tern_result = 0;
if (i < 5){ .tern_result = 1; } else { .tern_result = 0; }

// 2. Ternary turns into a variable read
a = .tern_result;
```

This is true for all but two cases: `while` and `for` loops. If the ternary expression occurs in a while statement's `conditional`, or in a for statement's `conditional` or `forUpdate`, they will need to be evaluated after every iteration.

In the case of a `while` loop with a ternary condition, we made the generated semantically identical IR by doing the following:

```
// original statement
while (i < 5 ? true : false) {
      if (i == 2) {continue;}
      printf("Hello World");
}
// is semantically equivalent to the following:

// 1. Evaluate the ternary for the first time
.tern_temp = false;
if (i < 5) { .tern_temp = true; } else { .tern_temp = false; }

// 2. Turn the ternary to a var read
while(.tern_temp){
      if (i == 2) {
            // 3. For continue statements, evaluate the ternary again
            if (i < 5) { .tern_temp = true; } else { .tern_temp = false; }
            continue;
      }

      printf("Hello World");

      // 4. At the end of the loop, we evaluate the ternary again
      if (i < 5) { .tern_temp = true; } else { .tern_temp = false; }
}
```

For `for` loops, we were able to follow similar logic for both the `conditional` and `forUpdate` blocks. To keep things concise for the semantic checker and the control flow graph generator, if a ternary appears in either the `conditional` or `forUpdate` portion of the for loop, it was converted to a semantically equivalent `while` loop in the IR.

Semantic Checker

`DecafSemanticChecker` performs all semantic checks by traversing the IR and accessing the symbol tables. Semantic checking is done top-down starting from the root `IrProgram`. `DecafSemanticChecker` utilizes a `scopeStack` such that each time we enter a new scope, we push its symbol table onto the stack and each time we exit from the scope, we pop its

symbol table off the stack. We also noted all of the semantic rules on top of `DecafSemanticChecker` for reference.

Note that in `DecafIrVisitor` the following rules are initially checked: rule 1: “No identifier is declared twice in the same scope”, rule 5: “Array initializers have either a declared length or an initializer list, but not both”, rule 6: “If present, the `⟨int literal⟩` in an array declaration must be greater than 0”, and rule 25: “All integer literals must be in the range `−9223372036854775808` ≤ x ≤ `9223372036854775807` (64 bits)”.

Main()

Using the `checkMainMethod` function, we check for rule 3: “The program contains a definition of main with type void, and no parameters”. We search through the global symbol table to find the identifier ‘main’ which maps to a `IrMethodDecl` type, checks its type is void, and its list of parameters is empty.

ImportDecls

When we visit `ImportDecls`, we assume them to be semantically correct as duplicate imports should already handled in the `DecafIrVisitor`

FieldDecls

For `FieldDecls`, we check whether the type of the declared variable matches the type of its initializer. An instance Rule 4: “All types of initializers must match the type of the variable being initialized is being checked here”. As well, we check that the contents of the initializer for array literals are the same and match the corresponding type of the variable.

MethodCalls

Using the `checkMethodCall` helper function, we check rule 7: ”number and types of parameters must be the same as the number and types of the declared parameters for the method” and rule 9: “String literals and array variables may not be used as parameters to non-import methods”. This helper gives the return type of the method, which helps check for rule 8: If a method call is used as an expression, the method must return a result.

Blocks

In `IrBlocks`, we first check the semantics of its `FieldDecls` as described above. Then we check the semantics of its `IrStatements`.

Statements

- `IrAssignStatements`: We check its `assignLocation`, check its `assignExpr`, and check for rule 4 and rule 23 as the type of assigned location and its expression must match and if the the assignOp is `+=/-=`, then the type of assigned location and its expression must be both `int`.

- `IrMethodStatements`: We check for rule 13: “The `<id>` in a method statement must be a declared method or import”.
- `IrIfStatement`: We check for rule 16 on the conditional expression, recursively check its then nlock, and if it's else block exists, recursively checks as well
- `IrForStatement`: We check the identifier, check for rule 16 on the conditional expression, and check rule 23 on the for update, then recursively check its block
- `IrWhileStatement`: We check for rule 16 on the conditional expression and recursively check its block
- `IrReturnStatement`: We check for rule 10: “A `return` statement must not have a return value unless it appears in the body of a method that is declared to return a value” and rule 11: “The expression in a `return` statement must have the same type as the declared result type of the enclosing method definition”.
- `IrReturnStatement` & `IrContinueStatement`: We check for rule 24: “All `break` and `continue` statements must be contained within the body of a `for` or a `while` statement.” We search from the top of the `scopeStack` to check for at least one symbol table that corresponds to either while or for loop.

Identifiers

Using `checkId` helper function, we check for rule 2: “No identifier is used before it is declared” by searching from the top of the `scopeStack` until it finds the first matching identifier or assert that the identifier is used before it is declared.

Locations

Using the `checkLocation` helper function, we check for rule 12: “an `<id>` used as a `<location>` must name a declared local/global variable or parameter: and rule 14: “For all locations of the form `<id>[<expr>]`, `<id>` must be an array variable, and the type of `<expr>` must be `int`”.

Expressions

Using the `checkExpr` helper function, we recursively traverse the nested `IrExpression` and evaluate any subexpressions first, then check the semantics of the current expression, and return the results to the parent, if any. Here, we check for rule 15: “The argument of the `len` operator must be an array variable” to rule 23: “The `<location>` and the `<expr>` in an incrementing/decrementing assignment, `<location> += <expr>` and `<location> -= <expr>`, must be of type `int`”, except rule 16.

CFG

The Control Flow Graph stage walks through a semantically checked, high level IR using another visitor pattern and produces a low level IR suitable for assembly code generation.

In this step, control flow structures such as loops and if statements are replaced with basic block and control flow graphs, conditions are short-circuited, nested expressions are flattened, and all immediate values are converted to `Long` values. Other convenient information such as global string constants and array declarations are collected during the pass and made available to Codegen.

Control Flow Graph Generation

Basic blocks are either `CfgRegularBlock` or `CfgConditionalBlock`, which are self-explanatory.

A `CfgScope` represents a destructed scope, containing a series of basic blocks and their connections, where every basic block is uniquely mapped to by a `String` label. The crucial design choice is that `CfgScope`- not the basic blocks- contain information about the connections. The connections are represented in `CfgScope` by 3 `(String -> String)` maps, which contain the regular, true, and false edges of the CFG respectively. `CfgScope` also contains an `entry` and `exit`, which are the label of the entry and exit block of the scope. The `exit` block is guaranteed to be a regular block.

With the guarantee that block names are unique (ensured by a counter of block names), the design of `CfgScope` makes many operations relatively easy. To merge two scopes sequentially, one would quite literally add all the map fields together in Scala, set the entry and exit appropriately, and add a single edge `(before.exit -> after.entry)`.

As for the actual functions that construct scopes,

- `shortCircuit` is implemented similarly to what was discussed in lecture.
- `visitBlock` visits a possibly nested scope from the high level IR, constructing a `CfgScope` sequentially for each statement as discussed above, and is first called on the top level scope within a function.
    - `visitLoopBlock` is a variant of `visitBlock`, but in the context of either a loop structure, to support `continue` and `break` statements. It accepts the names of the blocks to continue/ break to as additional parameters, and maintains this information to any recursive calls.
- `visitIf` accepts a function parameter (which is either `visitBlock` or some variant of `visitLoopBlock`) for visiting the nested scopes contained in if statements. This is necessary because if statements may contain continue/ break statements when inside a loop. Otherwise, the implementation is as discussed in lecture.
- `visitWhile` calls `visitLoopBlock` on its nested scope.
- `visitFor` is very similar to `visitWhile`, but constructs a new block for the increment expression, since continue statements will jump to this statement before the loop condition.

- `expandExpression` is used to flatten a possibly nested expression. It assumes that ternaries are handled in `DecafIrVisitor`.

Finally, `simplifyScope` takes a `CfgScope` and continually collapses edges until its basic blocks are maximal. Though this is an optimization, this turned out to be necessary to make the generated assembly somewhat readable.

Codegen

Overall Design

The design of `CfgCodegen` follows a fairly similar visitor pattern over the control flow graph generated in the previous step as `CfgVisitor`, `DecafIrVisitor`, and `DecafSemanticChecker`.

The `visitProgram` method is the entry point to the code generation. It takes a `CfgProgram` which represents the control flow graph generated by the previous step, as well as flag `isMac`, which indicates whether the assembly generation should generate code that can be linked on macOS.

For the linker on macOS to be able to find the entry point of the program (`main`), as well as link to external calls to the C Standard Library (e.g. `printf`), there must be underscore before the method name. This is not the case on Linux, so that is why a special flag was introduced to make the appropriate changes in the assembly.

Throughout the code, there will be many labels and temporaries that start with "`_.`", which is because we want all programmatically generated labels and variables not to be a possible Decaf identifier name to avoid conflicts.

Setting up the Assembly Program

At the top of the program, we generate globals for all strings and arrays we encounter. We also generate global variables that are referenced relative to `%rip`. For strings, the `.string` assembler directive is used. All other variables and arrays are declared using `.quad`. The first element in each array is its declared length.

We then visit each function (covered in the next section). Finally, we emit the panic code (labeled `_.falloff_panic` and `_.bounds_panic`) that is used for bounds checking after all of the program code.

Generating Functions

To generate functions, we first need to move the parameters to the appropriate locations on the stack. In this phase, all variables are stored on the stack and pulled from the stack when needed as we have not implemented register allocation.

If there are more than six variables, the additional variables are popped off the stack first and then stored in a different location in the stack that is at a negative offset relative to `%rsp`. This is then followed by the more common case of fewer than six variables. We follow the x86 calling conventions for the order of the registers.

In each function, there is a scope that contains all of the basic blocks. We will visit that next. The scope and basic block visiting steps will generate all of the appropriate `ret` instructions. Therefore, if we have reached the end of a function without hitting an appropriate `ret`, that means the execution has fallen off the end of the function. For this case, we emit a `jmp` to `_.falloff_panic` at the end of each function.

While allocating space on the stack, we allocated the number of variables used in the function, and then round up to the nearest number of bytes divisible by 16. This way, we can maintain 16 alignment for function calls.

Generating Scopes/Blocks

We generate all basic blocks without regard to order. A future optimizing step will be to reorder the basic block generation to yield the fewest jump instructions.

All basic blocks have a list of statements, which are visited in the next section. After all the statements have emitted their code, we must figure out how this block connects to the next. In the case of a regular basic block, it simply looks at `CfgScope.regCfg` to figure out where it needs to jump next. In the case of a conditional basic block, it evaluates the conditional and emits instructions to conditionally jump to either the block defined in `CfgScope.trueCfg` or `CfgScope.falseCfg`.

Generating Statements

By the time we have reached the control flow graph, there are very few types of statements. All higher-level statements such as loops and control flow statements have been destructured into basic blocks. The three that remain that are used to generate assembly are `CfgAssignStatement`, `CfgMethodCallStatement`, and `CfgReturnStatement`. Each of these may contain `CfgExpression` which must also be evaluated in assembly. This process is described in the next section.

For `CfgAssignStatement`, we must handle the possibility of reading and writing from arrays. This is where the bounds checking is done at runtime. The index is moved to a register, and two compares and conditional jumps are used to determine whether the index of the array access is too large or too small.

```
movq -16(%rbp), %rdx // %rdx contains index
cmpq $0, %rdx //compare index to 0
```

```
jl _.bounds_panic
cmpq $10,%rdx //compare index to length of array (10 in this case)
jge _.bounds_panic
```

Otherwise, for assigns, they involve simply shuffling around values from the stack and registers to move data to their appropriate locations. For simplicity, the assembly generation always uses `%rcx`, which was chosen arbitrarily from the set of caller-save registers.

`CfgMethodCallStatement` emit call assembly instructions after moving call parameters to the appropriate registers. For strings and arrays, we load the effective address into the register using `leaq`. For all others, we use `movq`. For functions with more than six parameters, we push the extra values onto the stack. We throw away any return values by not moving anything from `%rax`. If we load an odd number of arguments onto the stack, we push an additional `$0` to the stack to maintain 16 alignment. We assume that everything is 16 aligned at the top of every function.

`CfgReturnStatement` emit the appropriate `leave` and `ret` instructions. If there's a return value, it is moved into `%rax` in accordance with the x86 calling conventions.

Generating Expressions

At the control flow graph level, there are only expressions that are directly translatable to assembly. There are also no nested expressions (whereas IrExpression can have other IrExpressions nested within them).

The expressions are `CfgBinOpExpr`, `CfgUnaryOpExpr`, `CfgValueExpr`, `CfgLenExpr`, and `CfgMethodCallExpr`.

`CfgBinOpExpr` represents binary operations such as basic math operations, modulo, and most logical operations. `CfgUnaryOpExpr` represents the negation of both integers (`-`) and booleans (`!`). `CfgValueExpr` represents constants and variable reads, including array reads. `CfgLenExpr` represents obtaining the length of an array. `CfgMethodCallExpr` represents a call to a function with a return value. This is generated with similar logic to `CfgMethodCallStatement` except that the return value is preserved.

Miscellaneous

Yoland worked to build an integration test suite using Scalatest directly that emulates the behavior of "`test.py codegen`". This was because our compiler is written directly in Scala, and it is easier for us to call the compiler multiple times rather than having the `test.py` script repeatedly starting the Java Virtual Machine for each test file.

The execution time of the integration test suite in Scala is about 4 times faster than the provided Python suite. In addition, code coverage tools can be run for Scalatest much more easily, which aided in writing tests for the compiler.

Difficulties with Phase 3

Known Problems

- We are not passing 5 of the private test cases for codegen on the autograder
- Declarations in `while` and `for` loops are not being reinitalized on each iteration. We know how to fix the issue, but simply ran out of time to fix it before the deadline

Phase 2 Fixes

Phase 2 Private Tests Fixes

1. We failed to check for usages of methods defined in the same scope before assignment
    - In the `DecafIrVisitor`, we number the order in which the methods are defined. Then in assignment, when we call a method, throw an error if the order number of method being called is greater than the order number of the current method
    - This means that we throw an error if we attempt to call a method that is defined after the current method

2. We had bad scope checking that did not stop at the first found identifier (it matched all possible identifiers up to the global scope)
    - In `DecafSemanticChecker` , we have a done flag which is set to true on the first identifier found and stops searching through the stack of symbol tables

3. We had poor integer overflow checking (it wasn't taking care of edge cases and negative hex literals correctly)
    - Implemented better Integer overflow checking logic in `DecafIrVisitor`

4. We had ternaries to be left associative, so we changed the grammar to be right associative

5. We did not check method bodies and method parameters having duplicate identifiers (i.e. int `a` in the method body and int `a` as a parameter to the method)

- In `DecafIrVisitor`, we check whether the identifiers in method body's symbol table contains duplicate identifiers that matches identifiers in the method parameters's symbol table

6. We had method declarations being allowed in assign statements (i.e. you can't assign a method declaration to a value)
   - In `DecafSemanticChecker` , when we check assign statements, we have an isMethod flag to check whether or not the identifier of the assigned location is a MethodDecl or ImportDecl and throw an error if true.

Highlight for Feedback

- *Design of temp variables*: temp variables are introduced during the CFG step while flattening expressions. When they are created, they are not distinguished from normal variables. For optimization steps, would it be better to include information that distinguishes temp variables? I.e. do the special properties of temp variables (only used within a block, only used once) lend themselves to make some optimizations easier, or should all variables be treated the same?

Contributions

Phase 2

For Phase 2, we decided to divide the work into two subteams. Alex and Yoland worked on implementing `DecafIrVisitor` to transform the abstract syntax tree generated by the ANTLR parser generator into our own intermediate representation, using a visitor design pattern to process the AST.

An Bo and Sophie (before she left) worked on implementing `DecafSemanticChecker` to semantically check the generated IR tree to ensure the program is well-formed. Sophie wrote the documentation in collaboration with the rest of the team.

Phase 3

In Phase 3, Alex worked on designing and implementing `CfgVisitor` from our semantically-correct IR tree generated from `DecafIrVisitor`. An Bo and Sophie (before she left) implemented the `expandExpression` helper function to flatten nested expressions. An Bo and Alex also wrote concise unit tests for various functions used within `CfgVisitor` such as `simplifyScope`. Yoland wrote the assembly code generation in `CfgCodegen` from the generated CFG. He also created the special case generator for ternaries in `DecafIrVisitor`, most of the tests for the codegen phase, as well as a new Scala integration test suite to check code coverage. We all collaborated in writing the documentation.