Project 3 Report: Unoptimized Code Generation

Allen Mi, Hanxiang Ren

November 2, 2018

1 Design

The unoptimized code generation component of the the Scalars Decaf Compiler is built via the following procedure:

1. Convert high-level IR tree into low-level IR by flattening expression trees.
2. Generate a control-flow graph based on the low level IR tree.
3. Generate `x86-64` assembly from the CFG.

1.1 Expression Tree Flattening

For each maximal `Expression` tree `t` rooted at an `Operation`, we need to flatten `t` such that `t` can be represented by a series of 3-address operations involving temporary variables. The flattening process is described by the following:

For each `Operation` `t`, we augment `t` with two fields, `.eval` and `.block`, where

1. `.eval` is a `Location` instance that tracks the temporary variable containing the result of the `Operation`, and
2. `.block` is a `Block` instance that contains a list of temporary variable declarations in `.declarations` field, and a list of 3-address operations for the `Expression` tree rooted at `t` in `.statements` field.

When we process each `Operation` `t`, `.eval` is initialized as an empty location, and`.block` is initialized as an empty block. We then recursively apply flattening to the children of `t` (i.e. `child` for unary operators, `lhs` and `rhs` for binary operators,and `condition`, `ifTrue`, `ifFalse` for the ternary operator). We then populate `t.block` by combining the `.block` fields of `t`'s children. We do so for both the `.declarations` and the `.statements` fields of the block.

After populating `t.block`, we declare a new temporary variable `tmp` and add its declaration to `t.block.declarations`. We set `t.eval` as a `Location` instance of `tmp`. Finally, we add an assignment operation to `t.block.statements` for calculating `tmp` from the `.eval` fields of `t`'s children.

Throughout the recursive flattening process, we keep a global iterator that increments whenever we initialize a new temporary variable. We use this technique to generate a unique name for each temporary variable in the program.

1.2 Control-Flow Graph Generation

A control-flow graph is generated from the low-level IR. The processes is separated into 3 stages.

1. Destructuring logical constructs.
2. Short-circuiting logical expressions.
3. Peephole optimization.

1.2.1 Destructuring

Logical constructs are destructured as outlined in the lecture notes. We scan for adjacent blocks of declarations and 3-address statements, and join them into a single node in the CFG. After this process, every node in the CFG is represented by one of the following 6 categories:

1. `VirtualCFG`: Start and end nodes that do not contain statements.
2. `CFGBlock`: Basic block in CFG that does not contain conditional statements.
3. `CFGConditional`: Basic block in CFG that represents a single conditional statement.
4. `CFGMethod`: Basic block in CFG that represents a method declaration.
5. `CFGMethodCall`: Basic block in CFG that represents a method call.
6. `CFGProgram`: Basic block in CFG that represents a program.

1.2.2 Short-Circuiting

Given a logical expression `t` that has a depth $d \geq 2$, the short-circuiting of `t` is done as follows:

1. Examine the type (i.e. unary, binary or ternary) and the operator (i.e. `&&`, `||`, etc.) of the root of `t`. For each operator, generate a CFG snippet from a pre-specified template.
2. Recurse into each conditional in the template and repeat the same operation.
3. If the expression for a conditional is atomic (i.e. not a compound logical expression), replace the conditional in the template with the actual expression.
4. Once all conditionals are filled in a CFG snippet `s`, add `s` to its parent snippet by replacing `s`'s corresponding conditional in the parent snippet with `s`. Connect the inbound/outbound edges of `s` with its parent's edges as necessary.

1.2.3 Peephole Optimization

Peephole optimization is performed to eliminate `nop` nodes in the CFG.

1.3 `x86-64` Assembly Generation

The CFG is linearized, and from which `x86-64` assembly is generated. When generating code for method calls, the Linux/GCC `x86-64` calling convention is used. We start with generating code for the CFG structure. Upon encountering each IR element, we recurse and generate code for the IR element.

2 Extras

2.1 Producing Debug Information

When using the `--target=assembly` target, the user may use the `--debug` switch to pretty-print the low-level intermediate representation tree. For each node of the IR, its corresponding IR class, row/column and other relevant info are displayed. Additionally, for each expression tree, a list of flattened code, as well as the declarations of any associated temporary variables are displayed.

2.2 Build/Run Scripts

We re-implemented the build and run scripts. The scripts and their functionalities are specified as follows:

1. `build.sh`: Attaches `scala` if the current machine's FQDN ends with `.mit.edu`. Invokes `setenv.sh`. Passes all additional parameters to `ant`.

2. `run.sh`: Attaches `scala` if the current machine's FQDN ends with `.mit.edu`. Invokes `setenv .sh`. Sets `JAVA_OPTS` appropriately. Invokes program and passes all additional parameters to program.

3. `setenv.sh`: Checks if the current machine's FQDN ends with `.mit.edu`. If so, populate the current shell with the environment variables specified in `athena.environment`; otherwise, populate the current shell with the environment variables specified in `local.environment`.

Each `.environment` file follows a one-line-per-variable format. Variable expansions (e.g. `$var` are allowed, as long as the variable is defined in the current shell or in previous lines of the environment file.

The `athena.environment` is attached as follows:

```
SCALA_HOME=/mit/scala/scala/scala-2.11.2/
```

3 Difficulties

1. It is challenging to design and implement an effective expression tree flattening scheme. An important aspect of expression tree flattening is to maintain information necessary in generating the CFG, especially for short-circuiting and destructuring. In our final solution, each node `t` of an expression tree contains all 3-address code and temporary variable declarations needed to compute the subtree rooted at `t`. This design choice borrows from the idea of persistent data structures, and it allows us to augment the IR with flattened code, rather than replacing it. Hence we are able to retain the original expression tree structure for short-circuiting and destructuring.

2. Performing short-circuiting in CFG generation.

3. Array allocation and array indexing.

4 Contribution

4.1 Allen

1. Designed and implemented expression tree flattening and part of code generation.
2. Created and maintained debugging/execution infrastructure.
3. Completed documentation.
4. Debugging.

4.2 Hanxiang

1. Re-designed and implemented CFG generation and the majority of code generation.
2. Debugging.