Scalars

Project 2 Report: IR/Semantic Checker

Jack Choi, Allen Mi, Hanxiang Ren

October 14, 2018

1 Design

The intermediate representation of the Scalars Decaf Compiler is built via the following procedure:

1. Implement ANTLR structural annotations in the parser grammar for generating an abstract syntax tree.
2. Design a hierarchy of IR nodes using traits and case classes.
3. Build the IR structure and perform semantic checking via 3 preorder traversals:
   (a) Traverse the abstract syntax tree; convert each AST node into a corresponding IR node.
   (b) Traverse the IR structure; maintain and utilize an environment stack during the traversal to fill in the type information for each node. Perform scope-related check during the traversal.
   (c) Traverse the IR structure; perform type-checking during the traversal.
4. Perform global/miscellaneous checks.

1.1 Generating the Abstract Syntax Tree

The AST is generated by ANTLR via structural annotations in the parser grammar. For each node of the AST, the following information is maintained:

1. ANTLR token type
2. ANTLR token name for virtual tokens
3. Token text for non-virtual tokens
4. Row/column info for non-virtual tokens

1.2 Traversal 1: AST to IR Conversion

We implement the hierarchy of IR nodes via traits and case classes. An advantage about the this approach (as suggested in the handout) is streamlined type checking and code generation. For example, we only need to a single checking/codegen/optimization definition for the trait of all binary operators, rather than a separate one for each of them.

The process for converting AST to IR is detailed below:

1. Process `import`, create a method descriptor for each ID, mark them as external functions, and add them to global symbol table.
2. Repeat the same process for each `field_decl`.
3. For each `method_decl`, do the following:

(a) Create a method descriptor by its `return_type`, `name`, `param_list`; add the descriptor to the environment stack.

(b) Process the block of the method declaration. Store the result in the `.code` field of the method descriptor.

4. Process `if`, `while` and `for` constructs similarly.

5. Process `continue` and `break`. We need to associate them with their closest loop. This could be solved by keeping track of their closest loop throughout the traversal.

6. Associate `return` with its corresponding function.

7. Create an empty `.type` field for each `expr` descriptor. The `.type` field will be filled in the second traversal.

1.3 Traversal 2: Type Identification and Type Checking

1.3.1 Building the Environment Stack

When traversing the IR structure, we maintain en environment stack. Each element of the stack contains a pointer to a symbol table. The stack supports the following operations:

1. `push(t)`: Create an element containing a pointer to symbol table `t`, and push the element onto the stack.

2. `pop()`: Pop the topmost element off the stack.

3. `top()`: Return the symbol table pointer of the topmost element.

Additionally, each element of the stack supports a `next()` method that returns the next (`+1` depth) element of the stack.

1.3.2 Building Symbol Tables

Symbol tables are built as we perform the traversal:

1. Whenever we enter a scope (`method_decl`, `for`, `if`, `while`, etc.), we initialize an empty symbol table, and push it onto the environment stack.

2. Whenever we exit a scope, we call `pop()` on the environment stack.

3. Whenever we encounter a variable/method declaration, we add its corresponding descriptor to `env_stack.top()`, where `env_stack` denotes the environment stack.

1.3.3 An Integrated API for Symbol Table Building

Combining the previous two sections, we can see that it is convenient to extend the environment stack API to support the following:

1. `add(desc)`: Add the descriptor `desc` to `this.top()`.

2. `lookup(name)`: Return the descriptor corresponding to string `name`, return `none` if found nothing. The lookup procedure is as follows: Call `this.top().lookup(name)`. If the symbol table returns a `desc`, then return `desc`. Otherwise (if the return is `none`), enter `this.top().next()` and call `lookup(name)`. Recurse until reaching the bottom of the stack.

1.3.4 Performing Scope-Related Checks

The scope–related checks are performed at the same time when we build the environment stack. Whenever we see a usage for `name`, we perform `env_stack.lookup(name)` and outputs a violation if the return is `none`. Conversely, whenever we see a declaration for `name`, we perform a lookup and outputs a violation if the return is not `none`; otherwise, we register a valid declaration by setting the `.type` field of the corresponding descriptor to the type specified in the declaration.

1.4 Traversal 3: Type-Checking

For type-checking, we associate a check with the case class or trait it belongs to. As we traverse the IR structure, we perform the checks associated with each node. A typical check examines the types of the local (limited depth) subtree rooted at the node of interest. If the types of some nodes on the subtree do not match those specified in the check, we output a violation.

2 Extras

2.1 Producing Debug Information

When using the `--target=inter` target, the user may use the `--debug` switch to pretty-print the abstract syntax tree. For each node of the AST, its corresponding token type, token name/text and row/column info are displayed.

2.2 Cross-Verification of AST

We cross–verified our AST conversion against Hanxiang's previous work.

2.3 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:

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

3 Difficulties

1. Initially, we intended to keep the scope hierarchy within a static symbol table (instead of building an environment stack), since Decaf does not support nested method declaration. However, we quickly noticed that nested scopes can still be present since each block defines a local scope. Hence we designed and implemented an environment stack to keep track of the scope hierarchy.
2. Differentiating the unary negation `-` and decrement operator `--`, especially in the case where there exists multiple contiguous unary negations.
3. Checking `int` literal overflow.

4 Contribution

4.1 Jack

1. Conversion from flat CommonAST to a tree whose depth corresponds to hierarchy or tokens.
2. Above tree to ScalarAST, which is an AST with a more intuitive API and line/column information of tokens.
3. ScalarAST to IR (traversal 1), which is our intermediate representation whose semantics and types have not been checked yet.

4.2 Allen

1. Designed the general 3-traversal structure; designed traversal 2.
2. Participated in implementing ScalarAST.
3. Added command line interface for `--target=inter` switch and its `--debug` variant. Created and maintained run and build scripts.
4. Completed documentation.

4.3 Hanxiang

1. Modified Jack's IR design and ScalarAST to IR.
2. Designed and implemented traversal 3.
3. Implemented semantic checker according to our discussion.