Semantic Analysis : CS3423

Code Documentation

Saksham Mittal CS16BTECH110032
Shubham Kumar ES16BTECH11028

October 10, 2018

Contents

1 Code Overview

The code can be divided to perform these major steps :

- The program traverses the **AST** to prepare the Inheritance graph of all classes. Then, we check for inheritance cycles, which can also detect multiple cycles. In the case of inheritance graph errors, we do not recover, and exits after giving suitable error messages.

- Then we again perform a **BFS** traversal on the inheritance graph, to store the class information in a HashMap. In this, each class name is mapped to an object which contains all the information of the class like parent's name, attribute list, and method list.

- Then we build out Scope Table, for which we traverse over all the **AST** nodes(We use Visitor mechanism for that) and add the corresponding attributes. Appropriate checks are in place to ensure that all types conform, and variables are in scope.

- Finally, we check if the **Main** class is present or not, and raise an error if it is not present. A similar check is performed for the **main** method in the **Main** class.

1.1 Building Inheritance graph

We maintain three Hashmap to map class names to their index, **AST** classes to their class name, and indices to their class name.

Object and IO classes are added initially to the inheritance graph and class size is incremented by 2. Then, we traverse over all the **AST** classes and add them to the hashmaps, provided they are not redefinitions or inheriting from String, Int, or Bool class. In that case, we raise appropriate errors.

Then, we do another pass on the **AST** and fill our graph after checking for undefined parent behavior. This means cases where a class is inheriting from an undefined class are caught.

After that, comes the code for detecting loops. We traverse our graph in a BFS manner maintaining a visited array. If a cycle is detected, we report it and continue finding cycles. Finally, we perform **System.exit(1)** if cycle(s) are detected.

The psuedo code for detecting cycles is straightforward:

```
q <- new queue
append the start node of n in q
while q is not empty do
    n <- remove the first element of q
    if n is visited
        output 'Cycle found!'
    mark n as visited
    for each edge (n, m) in E do
        append m to q
```

1.2 Filling class information

We again traverse our Inheritance graph in a BFS manner, adding the classes in the clsInfo object. For base/default classes(Object, IO, Int, String, Bool) we add them to the 'cls' Hashmap in the constructor itself.

For other classes, we call the **insertClass()** method. This method instantiates a new BasicClassBlock object and checks for multiple attributes and method defintions in the class. Then, a check for redefinitions of inherited attributes and method definitions are checked. If a class passes all these checks then it is added to the 'cls' mapping.

Let's see the method **checkAttrMthd()** which checks for multiple attributes and method definitions in the class. We traverse over the features of the given class. If the feature is of AST.attr class, then we check if the attr list already contains this attribute, if so, we raise an error. Else we add it to the attribute list. The same way we check for methods.

Next method is **checkAttrMthdInherited()** which method checks for redefinitions of inherited attributes and methods in the class. First, we traverse over the attribute list to check if the parent class contains the attribute of the present class. If so, we report the error.

Then, we do so for the method list. But here the different thing is that methods can be different in 3 ways:

- Different number of formals
- Different return types
- Different parameter types

Respective errors are reported in the appropriate places.

1.3 Scoping and Visitor Mechanism

The scope checking and type checking is implemented through a visitor mechanism which visits the nodes of AST in a recursive manner.

The scope table stores the attributes of the current scope, which includes :

- Class attributes
- let-declarations
- method-parameters
- assign-variables

We have a function called **fillTable()** of class BuildScopetable, Which loops over all the classes in the AST. A new scope is entered for each class. **Self** is inserted in the scope table when it is present in any of the classes. Then we insert all the attributes of ancestor classes of the current class from the inheritance graph hierarchy into the scope of the current class. This is done using **insertAll()** function of ScopeTable class. Then, **VisitNode()** function of class VisitorMechanism is dispatched for the current class.

For each class, we visit the nodes of all its features(attributes and methods).

Whenever an attribute is visited we check if its type class is a defined class by referring to the cls HashMap in ClassInfo class. It returns **null** if that class is not defined. The suitable error message is generated. If the attribute has a value expression in its definition, we visit that expression and check if the types of this attribute and expression conform. The suitable error message is generated in case of failure.

Whenever a method is visited we enter a new scope. We iterate over the formal list of the method and add the formals to the current scope. Redefinition of formals is also checked. Then we visit method.body which is of type expression. The type of method.body should conform to the return type of the method. In case of failure suitable error is given.

The major part of the visitor mechanism is to visit all the expressions in a class. The operation that occurs when any of the expressions are visited through the visitor pattern is given below corresponding to the class of the expression.

- **AST.assign:** Checking if the variable to be assigned is present in the scope table through **lookUpGlobal()** function of ScopeTable class. Visiting the value expression. Checking if the types of variable and expression conform. Errors are generated in a case of failure at any step.

- **AST.dipatch:** Visiting the caller expression. Visiting all the actuals(expressions) in a loop. Checking if the type class of caller expression is defined. Checking if the number of actuals and formals are same. Checking if the types of all the actuals conform to the types of corresponding formals. Checking if the method is defined or not. Errors are generated in a case of failure at any step.

- **AST.static_dispatch:** Same as **AST.dispatch** except that we also check if the static dispatch type conforms to the caller expression type.

- **AST.cond (if-else):** Visiting the predicate expression. Checking if the type of predicate conforms to type "Bool". Visiting the expressions inside of the if and else bodies. The type of the if-else expression is assigned using the **commonAncestor()** function of ClassInfo class. Errors are generated in a case of failure at any step.

- **AST.loop (while):** Visiting the predicate expression. Checking if the type of predicate conforms to type "Bool". Visiting the body expression (loop.body) of the loop. Assigning the type of loop expression as "Object".

- **AST.block:** Visiting all the expressions inside the block in a loop. Assigning the type of block expression as the type of the last expression in the list.

- **AST.let:** Checking if the type of the value expression conforms to the type of the variable defined. Entering a new scope. Inserting the variables defined in the current scope using insert() function of the ScopeTable class. Visiting the body expression. Assigning the type of let expression as the type of body expression. Exiting from the current scope. Errors are generated in a case of failure at any step.

- **AST.typcase:** Visiting the predicate expression. Iterating over all the branches of typecase expression. Entering a new scope for each branch. Checking if the type class of the branch is undefined. Inserting the branch attribute into the current scope. Visiting the value expression. Exiting scope. Checking if there are more than branches with the same type using a HashMap. Setting the type of typecase expression as the common ancestor types of all the branches. Errors are generated in a case of failure at any step.

- **AST.new_ :** Checking if the class with new is undefined by creating an object of BasicClassBlock class and comparing it with "null". The error is generated in a case of failure.

- **AST.isvoid:** Setting the type of isvoid expression as "Object".

- **Arithmetic Expressions:** In case of any arithmetic operation expression (+, -, *, /) visiting both, the left-hand expression and the right-hand expression. Checking if the type of both the expression is "Int". Setting

the type of this expression as "int". Errors are generated in a case of failure at any step.

- **Comparison Expressions:** In case of any comparison operation expressions (<, <=, =) visiting both, the left-hand expression and the right-hand expression. For "<" and "<=" checking if the types of both the expressions are "Int". For "=" expression checking if the types of both the expressions conform if their type is one of "String", "Int" or "Bool". Setting the type of this expression as "Bool". Errors are generated in a case of failure at any step.
- **Unary operation expressions:** Visiting the value expression. Checking if the type of value expression is "Int" in case of compliment( ) expression and "Bool" in case of not expression. Errors are generated in a case of failure at any step.
- **AST.Object:** Searching for the Object (attribute) in the scope table using **lookUpGlobal()** function of ScopeTable class. Errors are generated in a case of failure at any step.
- **Constant expressions:** Setting the type of the expression as the type of corresponding constant.

2 Code structure

The code is organized into following files:

2.1 AST.java

This file contains information of each node in the AST. We have not changed this file.

2.2 BasicClassBlock.java

This class contains the basic information of the classes in the AST. The constructor takes the following parameters and initializes them.

- Class Name
- Class Parent Name
- Class Attribute List
- Class Method List

2.3 BuildInheritanceGraph.java

This class has the following functionality :

- It initializes and builds the inheritance graph
- It traverses the class list from AST.class_, and adds them to the graph
- It checks if the class to be added is already among Object, IO, String, Integer, Bool
- It checks if the class is inheriting from Integer, String, Bool
- It checks if the parent class is already defined or not

2.4 BuildScopeTable.java

This class builds the scope table as given in the **ScopeTable.java** by iterating over all the classes of the program. Then we visit the nodes of the AST by the Visitor Mechanism. We added a new method called **insertAllAttrs()** which inserts all attributes of its parent class and ancestors, and its other attributes (current class). It directly adds a Hashmap to the current scope.

2.5 ClassInfo.java

We store the class information in this class in a Hashmap which maps the class name to it's BasicClassBlock object. There is a method called **insertClass()** which :

- Inserts all the attributes and methods of the parent class
- Checks for multiple methods attribute definitions
- Checks for method overrides and attributes override

2.6 ReportError.java

This is helper class which is used for reporting errors. This is inheriting from Semantic class, so that we can use the provided method **reportError()**.

2.7 Semantic.java

This is the main class, which manages all the operations. In it's constructor, we build an inheritance graph first, then fill the class information, fill the scope table and finally check for **Main** class and **main** method.

2.8 VisitorMechanism.java

This class is responsible for visiting over all the nodes of the AST. It contains overloaded method **VisitNode()** whose parameter defines the type of node we are visiting. This is responsible for type checking and conformance. It uses the scope table for performing these checks. Suitable errors are raised in a case of failure.

The file hierarchy is as follows:

```
semantic/
|___src/
|    |___java/
|    |    |___cool/
|    |         |___AST.java
|    |         |___BasicClassBlock.java
|    |         |___BuildInheritanceGraph.java
|    |         |___BuildScopeTable.java
|    |         |___ClassInfo.java
|    |         |___ReportError.java
|    |         |___ScopeTabble.java
|    |         |___Semantic.java
|    |         |___VisitorMechanism.java
|    |
|    |___test_cases/
|         |___Error/
|              |___test1.cl
|              ..
|              |___test26.cl
|
|         |___NonError/
|              |___test1.cl
|              ..
|              |___test12.cl
|
|___Readme.pdf
```

3 Test Cases

We have divided the test cases into two categories :

- **Error:** These are semantically incorrect COOL programs. This is to check if a particular error is caught and reported. We have tried to handle most of the errors.
- **Non-Error:** These are semantically correct COOL programs. This is to cover as many semantic rules as possible.

3.1 Error

The test cases are :

- **test1.cl:** Cycle formation in the inheritance graph (handles detection of multiple cycles).
- **test2.cl:** Redefinition of class not allowed.
- **test3.cl:** Redefinition of default classes not allowed.
- **test4.cl:** Classes cannot inherit from String, Int, and Bool class.
- **test5.cl:** Parent class should be defined.
- **test6.cl:** Multiple definitions of an attribute in the same class
- **test7.cl:** Multiple definitions of a method in the same class.
- **test8.cl:** Redefinition of an attribute in some class with the attribute already defined in its parent class.
- **test9.cl:** Redefinition of a method in a child class does not have same Number of formals as parent.
- **test10.cl:** Redefinition of a method in a child class does not have same return type as parent.
- **test11.cl:** Redefinition of a method in a child class does not have same type of formals as parent.
- **test12.cl:** Defined type of an attribute is different than that of the expression assigned to it.
- **test13.cl:** Redefinition of formal.
- **test14.cl:** Return type of method does not conform to body return type.
- **test15.cl:** Undeclared variable while assignment.
- **test16.cl:** Type of expression doesn't match during assignment.
- **test17.cl:** Errors while dispatching a method.
- **test18.cl:** Condition of a loop is not of type Bool
- **test19.cl:** The type of let expression does not conform to declared type.
- **test20.cl:** Errors in case expression.
- **test21.cl:** Errors While creating a new object of a class.
- **test22.cl:** Errors in Arithmetic operations.

- **test23.cl:** Errors in Conditional operations.
- **test24.cl:** Undeclared identifier.
- **test25.cl:** **Main** class not defined.
- **test26.cl:** **main()** method not defined.

These test cases generates corresponding errors as per semantic specifications of COOL manual.

3.2 Non-Error

The test cases are :

- **test1.cl:** Testing inheritance of classes.
- **test2.cl:** Initialization of attributes and creating new attributes.
- **test3.cl:** Dispatching of methods.
- **test4.cl:** Testing rules of method definitions (Same function in inherited classes).
- **test5.cl:** Testing rules of assign.
- **test6.cl:** testing rules for if-else
- **test7.cl:** testing rules for while.
- **test8.cl:** Testing rules for block expressions.
- **test9.cl:** Testing rules for let expression.
- **test10.cl:** testing rules for arithmetic operations.
- **test11.cl:** testing rules for unary oparations.
- **test12.cl:** Non Trivial test case.

These test cases generates correct AST as per semantic specifications of COOL manual.