Semantic Analysis

Overview

Performing the semantic checking involves the following steps:

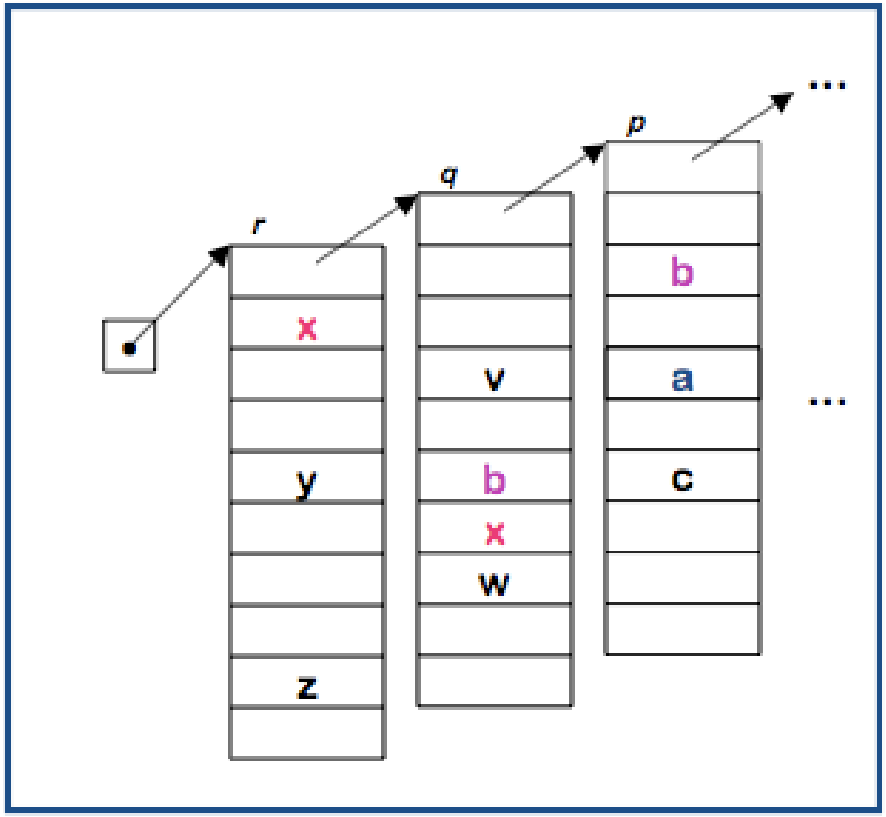
- Build inheritance graph & check to see there are no cycles.
- Build Symbol tables for each class.
- Perform Type checking based on the inheritance tree and the symbol tables.

You may find it easier to perform these steps in 3 different passes of the AST tree or just one.

SymbolTable class

SymbolTable can be used to perform the following:

- Adding Attributes
- Managing and checking scope
- Type Checking




```
B0: procedure b {
        int a, b, c
B1:     {
            int v, b, x, w

B2:         {
                int x, y, z
                ....
            }

B3:         {
                int x, a, v
                ...
            }
            ...
        }
    ...
}
```

semanticAnalyzer.Semant

- The main method that calls the semantic analyzer for the Program.
- The semant() method of the Program class calls

  ClassTable classTable = new ClassTable(classes);
- This installs the basic classes (Object, IO, Int, Bool and Str) look at *semanticAnalyzer.ClassTable* for more information

treeNodes.Program

- Once the basic classes are installed, walk the AST for all the classes, and call semant on those classes

```
for (Class_ c : classes) {
c.semant(new SymbolTable<Info>(), classTable, c);
}
```

- This creates a new scope for each class.

Semant() for a class

The following steps need to be done when the semant method of a class is called:

- Check if the class is present in the inheritance graph.
- Start a new scope for the symbol table " st.enterScope()"
- Add all the variables into the symbol table.
- Call semant for each method in that class.
- Exit the scope "st.exitScope()"

Variables and functions

- Perform similar passes to build the symbol table for functions.
- During this phase you will have to implement the semant method for all the treeNodes.* classes

Finally

- Section 12 of the cool manual would provide you with all the details of type checking for the 3rd pass of the AST tree.

Phase4: Semantic Analysis

Cont.

```
semant(//attribute table,  part of O of O,M,C
            SymbolTable<AbstractSymbol> at,
             //method table, M of O,M,C
           SymbolTable<LinkedList<AbstractSymbol>> mt,
             //class table, type hierarchy, part of O of O,M,C
            ClassTable ct,
             // C of O,M,C
             Class_ c)
```

2.Build symbol table for each class(top-down, recursively)
First thing to do is to write a copy funtion for SymbolTable class, this must be a very deep copy because SymbolTable is a data structure including Stack, HashTable, and possibly some other data structure which mainly depends on the generic type T. Then our symbol table building is relatively easy. Before we add local methods and attributes to this table we first make a copy from parent. In the process of adding local features, we do a lot of semantic checking, like, method redefinition, attribute redefintion, no Main class, no main method in Main class, any arguments in main method of class Main, method overriding, etc.
self_type:
at this stage, do not resolve self type, we need self_type to spread from parent to children recusively.

3.do semantic checking for treeNode
first to check method and attribute, then enter a new scope in attribute table, add self and SELF_TYPE entry(resolve self_type to corresponding class), do semantic checking for expressions.
Hard: except dispatch, involving new ids, so new scope is needed
CaseExpression, let, StaticDispatch, Dispatch, Method.
Easy:
Arith, Compare, loop, condition.

Each time encounter a semantic error, fake a perfect legal type instead, thus much more errors will be found.

```
public void semant(SymbolTable<AbstractSymbol> at,
                SymbolTable<LinkedList<AbstractSymbol>> mt, ClassTable ct, Class_ c) {
        // assign.java   semant
        if(RuntimeConfig.IS_DEBUG_MODE){
                System.out.println("-- Assign --\n");
        }

        AbstractSymbol temp = null;
        if(at.probe(this.name) != null) {
                temp = at.probe(this.name);
        } else if(at.lookup(this.name) != null){
                temp = at.lookup(this.name);
        } else {
                new SemanticError("Assignment to undeclared variable " +
                                this.name + ".", c.getFilename(), this);
        }

        if(temp == null) temp = TreeConstants.Object_;
        this.expr.semant(at, mt, ct, c);
        if(!ct.inherits(this.expr.getType(), temp)) {
                new SemanticError();
        }
       // setType&getType in TreeNode.java
        setType(this.expr.getType());
   }
```