Lecture 9: Semantic Analysis II

I. Handout 2 as an example for this lecture

II. Semantic Passes

A. We’ve now covered what is needed for semantic analysis
   1. Strategy for processing symbols
      a. First define all classes
      b. Then define all methods and fields
      c. Then resolve all the symbols
   2. This time we want to see how this is implemented

B. A pass is a walk over the AST
   1. Can be structured (e.g. left-right)
   2. Can be directed arbitrarily w/ multiple passes
      a. Each node can control its own subnodes
   3. Can be semi-structured
      a. Left-to-right, but nodes can say ignore or ignore children

C. A walk is managed by a Visitor (these days, object-oriented)
   1. Instance of the visitor design pattern
      a. Each AST node has an accept(Visitor) method
      b. And a visitChildren(Visitor) method
      c. Visitor class provides separate methods for each AST node type
         i. visitXXX(XXX node)
   2. Visitor also provides local storage for information to be passed up or down the AST
      a. Initialized by its constructor
      b. Maintained throughout the tree visit
      c. Information here is global
         i. Localize by storing in tree nodes
         ii. Localize by using a map from node to value
         iii. Typically not maintaining long-term history
   3. Essentially a nicely structured attribute grammar
      a. Variables that are permanent are stored in the AST
b. Variables that are temporary are stored in the visitor
c. The visitor predefines the computation path
4. You have to think in terms of visitations
D. Default visitation method
  1. AST accept method
     a. Just calls v.visitXXX(this)
  2. Visitor visit<Type>(<Type> t) method
     a. Calls visitChildren(t) at some point
        i. Equivalent to t.visitChildren(v)
     b. Can call super.visit<Type>(t) to do default processing
        i. Note that this does a visitChildren
     c. All visitors can throw SemanticException

II. Semantic Analysis for Java and similar languages (Start here)
A. Goal: resolve everything
  1. For each identifier, find the definition (symbol lookup, definition)
  2. For each expression node, determine the type at that point
     a. Why is this needed
     b. Overloaded operations
     c. Overloaded methods
  3. For each operator node, determine the actual operator used
B. Goal: find semantic (non-syntactic) errors
  1. Also syntactic errors not caught by the grammar
C. Method varies with language
  1. C, for example, can be all done in one pass
  2. Java requires 3+ passes
     a. Check extra syntax and semantic errors
     b. Find all type definitions and create the corresponding types
     c. Find all field/method definitions and define the symbols
        i. This requires type information since you have forward references
        ii. Involves defining variables referring to types
     d. Resolve everything else
        i. At this point, we can lookup names for an expression
        ii. Can handle local definitions (need to be declared before use)
        iii. We have the types for all identifiers, methods
        iv. We can resolve overloading, determine methods
        v. We can assign a type to each expression node

III. Semantics Passes for Decaf (simplified Java)
A. Pass 0: Clean up the AST (blue)
   1. Operations to be performed
      a. Field and variable declarations
         i. Pass down types if necessary
         ii. Check semantic constraints
            ➢ No static fields, no initialized fields
      b. Other semantic checks
         i. Declarations can’t appear in RHS of IF
         ii. Break and continue can only occur inside loops
         iii. super(...) only at start of constructor
         iv. returns only if return is allowed
         v. Constructors can’t be static
      c. Normalization of the AST
         i. Define super class if not explicit
         ii. Insert super calls if they are not explicit
         iii. Add default constructors
   2. Can be combined with first symbol pass
B. Pass 1: defining types (light blue)
   1. Class node:
      a. Define the type in the current scope
      b. Set super type either here or at SuperType node
      c. Add supertype scope to class scope
   2. Scope nodes
      a. MemberList, MethodBody, OuterScope, StmtBlock
      b. Maintain (track) the current scope
   3. If inner types are allowed
      a. Track scopes inside a class as well
      b. Define inner types
C. Pass 2: defining methods and fields (green)
   1. Class node:
      a. Set/reset the current class
   2. Scope nodes
      a. Maintain the current scope
   3. Field node
      a. Define the field in the current scope
      b. This requires types having been passed down
   4. Method node
a. Define the method
b. Gather parameter types either here or during pass over the tree
c. Create a method type requires knowing parameter and return types

D. Pass 3: Resolving symbols (red)
   1. Symbol nodes: Class, Method, Variable
   2. Scope nodes
      a. Maintain the current scope
   3. Local nodes
      a. Define local variables in the current scope
   4. Otherwise
      a. Need to do full expression resolution with typing
      b. Do lookup of names as you get to them
   5. Functions:
      a. Define all local symbols
      b. Resolve expressions
      c. Complete semantic analysis
   6. Defining local symbols is easy
      a. Maintain current scope as before
      b. At local and parameter node, do definition
      c. Could this be done in pass 2?
         i. What about overridden names
   7. Semantic analysis is straightforward
      a. Most of the checks have been done
      b. What is left involves symbol lookup and expressions
      c. We concentrate on expression resolution

IV. Expression Resolution

A. Purpose
   1. To facilitate symbol resolution
      a. To help resolve overloading
      b. To provide a basic scope for symbol lookup of methods and field
   2. To resolve overloaded operators
      a. All languages have overloaded operators (+ for example)
      b. The compiler needs to know the exact operator being used
      c. Need to map source operators to target operators
   3. To resolve overloaded constants
      a. What is the type of ‘0’ in C
      b. What is the type of 55 in Java
4. To handle hidden operators  
   a. Casts, implicit and explicit conversions  
5. To clean up elided information  
   a. *this* argument in non-static methods  
   b. StringBuilder in Java  
   c. Operators that are really calls  
6. To provide appropriate error messages  
7. To simplify later passes  
   a. Some program analysis can be done here  
   b. For example constant propagation  

B. Methodology  
1. Make a bottom-up pass over each expression  
   a. Assume children have been resolved first  
   b. What if desired type came into account?  
      i. In an if_statement, does that fact we need a Boolean affect how we interpret the conditional expression  
2. For each expression node  
   a. Set the type of the expression  
   b. Set the target operator if relevant  
   c. Edit the AST to insert casts, etc.  
   d. Normalize the AST to reflect what is really happening  
      i. Not needed if just doing program analysis?  
      ii. Can be done during code generation as well  
3. For each name node  
   a. Resolve the name, looking it up in the appropriate scope as the appropriate type  
   b. Check access rules as well  
4. For each use of an expression ensure it is correct  
   a. IF, WHILE require Booleans  
   b. Return requires the appropriate type  

C. Expression Resolution  
1. Constants: just set the resultant type  
2. Operators  
   a. Determine the target operator  
   b. Ensure subexpressions have the right types  
   c. Insert casts if necessary (modifying the AST)  
   d. Determine the type of the result
3. NEW <type>
   a. Set result type, ensure it is an object type
   b. Note that this is a call to the constructor of type of new with args
      i. Modify the AST appropriately
4. NEW []
   a. For arrays, ensure base type is a primitive type
   b. For dimensions, ensure expressions are integer types
   c. Set result type base on # of dimensions
   d. Note need for constant propagation in some languages
5. Array reference

D. Name Resolution
1. Figure out what you want the tree to look like
   a. For a simple variable access
   b. For a field access (this.x or just x,...)
   c. For a static method call (type.method or method or expr.method)
   d. For a virtual method call
2. Names are constructed with a flag denoting whether they represent a call or not
3. Simple names (unqualified)
   a. Need to set the type and the symbol it refers to
   b. Special case <class>, <super>
   c. Look up name in the current scope
   d. If name is a field, modify tree to be this.field
   e. Otherwise just set the symbol and type
   f. Set local variable last_class for type names
4. Qualified names
   a. Handle x.length for x for non-calls
   b. Get scope associated with lhs
      i. From type of lhs
      ii. Or from last_class local in visitor
   c. Either lookup method or lookup name in given scope
   d. Set symbol, type, last_class variable
   e. If static call, replace lhs with null
   f. Else if LHS was a type name, replace with ‘this’
5. Calls for DECAF
   a. Ensure lhs type is a method type
   b. If type is non-static and lhs of method name is a type name
i. Ensure this is defined & is the appropriate subtype

ii. Replace lhs with ‘this’

C. Check parameter types match (insert casts if needed)

d. Set return type

6. Calls when there is overloading

a. Defer name lookup until all parameters are resolved

   i. Modify the tree walk or set appropriate flag

b. Build the set of parameter types

c. Then do name lookup using this set and the appropriate rules

d. Then generate appropriate casts and do the call