CS 444, Winter Term 2018

A4 Documentation

By Alexandru Gatea, Carl Zhou

Due on March 16

Output from A1
For each input file, the program creates an ASTTree using the scanner, parser, weeder and ASTbuilder (called ASTVisitor) from A1. All of the compiler phases after A1 are implemented using tree traversals, and each tree traversal is written using the visitor pattern.

Environment Building (A2)
We created an Environment data structure to store the symbol table information that is a "linked tree", in the sense that each Environment has a parent Environment (which is null for the root) and a list of children Environments. Each ASTNode has a reference to the Environment in the Environment tree corresponding to the ASTNode. Also, each Environment stores a list of EnvironmentPairs, where EnvironmentPair is a simple data structure holding a (String name, ASTNode declaration) pair. This list of pairs corresponds to objects (classes, field, variables, methods) declared in the scope of the ASTNode (i.e. the objects "visible" to the ASTNode). In this way, each name is visible to the list of all EnvironmentPairs in its Environment or in a subEnvironment of its Environment. Finally, each Environment stores the package, the imports, and the root of the file where the ASTNode is found in as well as an EnvironmentPair called the scopePair that is the (name, decl) pair in the parent of this Environment that corresponds to the current Environment.

To build the Environment tree, we start with a global Environment which we pass to an EnvironmentBuilder visitor for each ASTTree obtained from A1. This visitor creates an Environment for the file (that is linked to the global Environment), then does the following:
-Stores the root information (the import and package information). If there is no package declaration, then the packageName is "_DEFAULTPACKAGE_". Also, the implicit import-on-demand "import java.lang.*" is stored among the imports.
-When visiting a Class/Interface/Constructor/Field Declaration/Method Header/Block/ForStmt it creates an EnvironmentPair for that entity with its name and its declaration node, it stores this EnvironmentPair in the input Environment, it creates a child in the input Environment with this pair as its scopePair, it passes the child Environment to all subnodes and it stores the child Environment in the node. For a MethodHeader, it also adds EnvironmentPairs for the formal parameters in the child Environment.
-When visiting a BlockNode, it goes through the BlockStatements adding a new child Environment for each local variable found to the current Environment (so that the parent Environment of the 1st local variable declared is the Block's Environment, and the parent Environment of the (n+1)st local variable declared is the nth local variable's Environment).
-When visiting Imports it checks against clashing imports, when visiting a FieldDeclaration it checks against duplicate fields, when visiting a LocalVariableDeclaration it checks against duplicate variable names with overlapping scope.

-When visiting any node, it stores the input Environment in the node and visits all subnodes with this Environment

We went through several Environment structures before the final one; this was one of the biggest challenges with A2-A3. Initially we had a LinkedList structure, but we soon figured out that this wasn't sufficient for our purposes, because each Environment needed to have different branches (e.g. a class needs a different Environment for each methodDeclaration). Also, we needed the feature of being able to go up the Environment tree (via the parent field) because the scope of each node is the list of Environments above it in the tree. Lastly, for the package information we tried to use a separate data structure based on the package hierarchy, but this was error-prone so we decided to give up the efficiency of this separate data structure and just store the package information in the Environment of each class/interface.

Type Linking (A2)
For this phase, we wrote the ClassTypeLinkerVisitor, which is called on each file's ASTTree once the EnvironmentBuilder has been called on all trees (so that the package names are known). The ClassTypeLinkerVisitor first visits the imports, trying to determine and store a reference to their declaration (by using the global Environment); for the on-demand imports, it stores references to all files the import refers to. Then the ClassTypeLinkerVisitor traverses the rest of the tree, and updates the toCheck flag when the node being visited has a possible ClassOrInterface subchild (e.g. in ClassOrInterfaceTypeNode, TypeNode etc.). When it visits NameNode and toCheck is true, it tries to resolve the name by calling findTypeLink on the NameNode. Finally, when visiting a Class/Interface Declaration it checks that its name is not the name of any other class/interface.

The method findTypeLink determines if a name can be type-linked. It first calls PerformNameResolve on the name, and if this returns true then it checks that no strict prefix of name resolves to a type. The actual type-linking occurs in PerformNameResolve. Its body is split into two cases based on whether the name is simple or not (as defined by the JLS):
-If the name is simple, it searches the single-type-imports followed by the files in the current package followed by the on-demand-imports (checking if the name can be resolved uniquely).
-Otherwise, it searches all files and compares the prefix to the package name of those files.

Hierarchy Checker (A2)
This is phase done in 3 parts: computing the DECLARE set, computing the SUPER set, the hierarchy checks (which includes computing the REPLACE set).
1) After the type linking is done, computeDeclare is called on each ASTTree to compute the methods and fields declared. This information is stored via lists of FieldData and MethodData where FieldData, MethodData is a wrapper data structure for the information of a field, method respectively (such as modifiers, type/return type, signature). computeDeclare goes through the list of EnvironmentPairs of the class/interface Environment, creates and stores a FieldData/MethodData for each one, and checks that no two fields have the same name and no two methods/constructors have the same signature. computeDeclare stores these lists along with the modifier information of each class/interface in its ASTTree.
2) Once all of the DECLARE sets are computed, computeSuper is called on each AST-

Tree to compute the list of ASTTrees corresponding to superclasses/superinterfaces. This is done by going through the extends and/or implements list, and calling getSuperTree on each name in the list. The method getSuperTree goes through the list of ASTTrees and finds the one corresponding to the class/interface name. computeSuper also checks against invalid extends/implements (e.g. an interface extending a class, duplicate names in the extends/implements lists, a class extending a final class). NOTE: In the ASTVisitor, we added "java.lang.Object" to the extends list of any class/interface which are base classes.
3) Once all of the SUPER sets are computed, we call hierarchyCheck on the list of ASTTrees. This method calls the following methods:
-cycleCheck: This checks that there are no cycles in the class hierarchy by running a simple depth-first search on the class hierarchy that keeps track of visited classes (by "colouring" classes with 0 when not visited, 1 for currently visiting and 2 for already visited).
-computeInheritFields: This computes and stores the list of inherited fields for each ASTTree by going through all of the superclasses/interfaces' fieldsDeclared and fieldsInherited lists and comparing field names (if the field name of a field in a superclass/interface occurs in the current fieldDeclared list then the declared field shadows the inherited field(s), so it is not added to the fieldsInherited list). If any new field is added to any ASTTree's fieldsInherited list then computeInheritFields returns 1. While computeInheritFields returns 1, computeInheritFields is called again; this made sure that the discovery of an inherited field in any class is propagated properly to all of its subclasses.
-computeInheritMethods: This is very similar to computeInheritFields, except that it now computes and stores the list of inherited methods for each ASTTree by comparing signatures of declared methods with methods declared/inherited in any superclass/interface. Shadowed methods are not added to the inherited list.
-computeReplaceMethods: This computes the REPLACE set for methods, which is a list of (MethodData m, MethodData m2) pairs in ASTTree where "m replaces m2" (as defined in class). This method first calls performReplaceMethod on all pairs (MethodData m, MethodData m2) where m is a declared method in some ASTTree t and m2 is a declared/inherited method in a superclass/interface of t; this checks that if m,m2 have the same signature then the necessary conditions hold (e.g. same return types, m2 is not final etc.). Then computeReplaceMethods calls performReplaceMethod on all pairs (MethodData m, MethodData m2) where m,m2 are declared/inherited methods in superclasses/interfaces of some ASTTree t and m is not abstract and no method with the signature of m is declared in t; if m is static, then it throws an exception (no static method can implement an abstract method).
-checkAbstractModifier: This traverses the list of methods declared/inherited of each ASTTree (i.e. the CONTAINS set) checking that no abstract method is in a non-abstract class.
-checkAbstractInheritance: This goes through all pairs (MethodData m, MethodData m2) of inherited methods in each ASTTree, where m, m2 are abstract and checks that if they have the same signature then they have the same return type.

This phase was more algorithmic than others, but the rules for the DECLARE, INHERITED, REPLACE sets from class made it a straightforward task to compute these sets and check that they are well-formed.

Type Checking and Name Disambiguation (A3)

This is done in one pass, with the TypeChecker visitor. The method resolveAllNames is called on the list of ASTTrees, and this method resets some internal fields and calls the visitor on each ASTTree. These internal fields include the global Environment, a ClassTypeLinkerVisitor (to be used to resolve type names), the root of the current ASTTree being traversed (called _root), the name of the current "enclosing entity" (the field, constructor or method that the current ASTNode is in the declaration of), three nameNodes for Object,Cloneable,Serializable (linked to their classes) which are used to determine if a particular type is one of these types (this is needed to check type assignability for arrays), as well as some flags indicating if the current ASTNode is in the LHS of an expression (isLHS), is an expression name (isExprName), occurs in a variable initializer (occursInVarInit), occurs in the declaration of a static field/method (occursInStatic).

In order to deal with types, we added a type field to each ASTNode. For any expression node, the visit method performs its checks/computations and stores a type for that expression. The type assignability is determined by the method isTypeAssignable using the JLS type assignability rules in JLS 5.2. This method takes in two TypeNodes (left and right) and returns a boolean value indicating if the right TypeNode is assignable to the left TypeNode.

Most of the visit methods just follow the simple type rules (e.g. the expression in an if stmt must be of boolean type). Also, the flag variables in the TypeChecker are updated as needed by the visit methods (e.g. isExpr name is set to true when visiting the Name in a PostfixExprNode, ArrayAccessNode or AssignmentNode). Here are some of the more lengthy/complex visit methods:

-returnNode: This method first obtains the return type of the current enclosing method/constructor by going up the Environment tree until the declaration of the scopePair is a method or constructor declaration, and then compares this return type to the type of the expression.

-classInstanceCreationExprNode: This method first checks that the class being created is not abstract. Then it searches the list of constructors declared in the root of the class type for a constructor with matching signature; when a match is found, it checks that this constructor is either public or is in the same package as the current node.

-BinOpExprNode (node for any binary operation or a unaryExprNode): For the INSTANCEOF operation, checks that the Right is not null and is a valid type, and that one of Left, Right is type assignable to the other. For -,*,%, / checks that the Left and Right's types are numeric. For +, checks that either both Left and Right's types are numeric or one is a String and the other anything but void. For $<=, <, >, >=$ checks that the Left and Right's types are numeric. For ==, != checks that one of Left, Right is type assignable to the other. For $\&\&, \&, |, ||$ checks that the Left and Right's types are boolean.

-ClassDeclNode: If the class is not java.lang.Object (this is the only class that does not extend another class), checks that its superclass has a zero argument constructor (by calling resolveImplicitConstructor, which searches the set of declared constructors of its superclass for a constructor with no arguments).

-LiteralNode: Creates and sets the type for the literal by calling a constructor in TypeNode that sets the type based on the literal category.

-PrimaryNoNewArrayNode: Checks that if occursInStatic is true then there is no THIS.

Also, if there is a THIS then creates and stores a type corresponding to the current class.
-NameNode: If isExpr is true, calls resolveExprName.
-FieldAccessNode: If the type of the primary is an array, then checks that field identifier is length and isLHS is false. Otherwise, if the type of the primary is a classOrInterfaceType, tries to resolve the field by calling searchFields on a NameNode for the field identifier and with the root of the primary's type.
-MethodInvocationNode: If the method name is simple, tries to resolve it by calling searchMethods on the method name and the current root (_root). Otherwise, calls resolveAmbiguousName on the prefix of the method name and then tries to resolve the method name by calling searchMethods on the method name and the root of the prefix type (if the prefix is a type name) or the root of the prefix's type (if the prefix is an expression name).

Here are the descriptions of the search methods from used above:
-resolveExprName: If the name is simple, tries to resolve it using local variables (by calling searchLocals) and then using field names by calling searchFields with the current root (_root). Otherwise, it calls resolveAmbiguousName on the prefix and then tries to resolve the field name by calling searchFields with the root of the prefix type (if the prefix is a type name) or with the root of the prefix's type (if the prefix is an expression name).
-resolveAmbiguousName: This method reclassifies an ambiguous name using the rules in JLS 6.5.2 and returns 0 for a package name, 1 for a type name, 2 for an expression name.
-searchLocals: Searches a name in the EnvironmentPairs in each Environment starting with the name's environment up until the class's Environment.
-searchFields: Searches the fieldsDeclared and fieldsInherited of the input root for the field name; if found, it checks that if the field should be static or should be nonstatic (as indicated by the input) then it is. Also, it checks that if the field is a reference by a simple name and occursInStatic is true then the field is static. Moreover, it checks that the field found is accessible by calling isFieldAccessible. Finally if the field is a declared field, the root is _root, the field is not static, occursInStatic is false, isLHS is false, the name is simple, occursInVarInit is true then checks there are no forward references by calling checkDeclOrder.
-checkDeclOrder: Goes through the EnvironmentPairs in _root, checking that the input name occurs before enclosingEntityName.
-isFieldAccessible: If the input fieldData is protected and is declared in a different package than its current usage then it returns false if either the class where it's used is not a subtype of the class where it's declared or if the field is not static, the name is not simple and the prefix root is not a subtype of the root where it's used. Otherwise it returns true.
-searchMethods: This is very similar to searchMethods except that it searches the methodsDeclared, methodsInherited and calls isMethodAccessible to determine accessibility and the forward reference check is not done.
-isMethodAccessible: This is identical to isFieldAccessible except for a MethodData input rather than a FieldData input.

This biggest challenge in this assignment was the large number of rules needed to be checked, along with the time and effort spent debugging and fixing any errors from previous assignments that arise (there were several of these, although all of the A1, A2 tests had passed).

Static Analysis (A4)
This is implemented in 3 phases (i.e. 3 AST visitors).
1) First, constantComputeVisitor visits each ASTTree to compute the values of all constant expressions. This traverses the tree and at any node corresponding to an expression it computes its constant value (if any subexpressions are constant and the operation is not the INSTANCEOF operation). To store these constant values, we created an abstract ExprNode that is implemented by the different expression nodes and that has fields isConstant (indicating if it's a constant), constantValue (that is a LiteralNode) so that we can tag a constant value to any expression node.
2) Secondly, the reachabilityChecker visits each ASTTree to determine the reachability of ASTNode. This is done by keeping a _reachable flag that is updated using the simple reachability rules from JLS 14.20. At each statement, if _reachable is false then it's an error. Also, after visiting the BlockNode of a MethodBodyNode corresponding to a method with non-void return type, it checks that _reachable is false.
3) Lastly, the initializationChecker visits each ASTTree to make sure that no local variable is used before being declared. To do this, it keeps a flag _inInitializer that is set to true when visiting the VariableInitializer in a LocalVariableDeclNode and is (temporarily) set to false when visiting the LHS of an AssignmentNode. When visiting a NameNode, if _inInitializer is true and the name is precisely the simple name of the current local variable being initialized, then it's an error.

Testing
Other than the Marmoset tests, we wrote a couple of basic unit tests as well tests for conditions not checked in the Marmoset tests but that are present either in the JLS or in the Joos page on the CS 444 website. To elaborate on this latter set of tests, we had tests with:
-some input files with only imports and/or a package (no class/interface)
-classes with the name Object, String (but not in the java.lang package)
-duplicates in the implements/extends lists
-two abstract methods with the same signature but different return types in two superinterfaces of an abstract class that does not have a method with that signature declared
-types that could be linked to more than 1 possible class/interface (these tests ensure that the precedence is single-type imports, current package, on-demand imports)
-a simple type name being used to reference a type in a subpackage of the current package (files in a package do not have special visibility regarding subpackages of that package)
-invalid forward references occuring a field access or an array access
-bitwise operations
-general native methods not of the restricted type in Joos
and many others. All of these tests passed. Together with the Marmoset tests, these give a thorough check against all possible compile-time errors in the input program.