CS444/644 Assignment 1 Technical Report

Ho-Yi Fung, Jianchu Li, Zhiyuan Lin

March 17, 2016

Abstract

Contents

1 Introduction

2 Design

2.1 Name resolution

In this stage of the program, it was decided we would follow closely the 7 steps specified in class:

1. Constructing a global environment
2. Resolving uses of syntactically identifiable type names
3. Making sure class hierarchy is correct
4. Resolving ambiguous names
5. Resolving uses of local variables, formal parameters and static fields
6. Type checking
7. Resolving uses of methods and non-static fields.

In name resolution step, we connect names to their declarations.

The first step involves construction of a symbol table and resolution of syntactically identifiable names.

To construct the class environment, we would first need to build the global class environment. The global class environment contains information of all the types (classes and interfaces) defined, including those from the standard library. Once the global environment is in place, we would then construct the scopes within each class. This process needs to go over the declaration of all the types, methods, fields, and variables, and place these declarations in the proper scope.

The next step is to resolve the use of type names. This step deals with the following use of type names.

1. In a single-type-import
2. As super class of a type
3. As interface of a type
4. As a Type node in the AST, including:
   (a) In a field declaration or variable declaration (including the declaration of formals)

(b) As the result type of a method

(c) As the class name used in class instance creation expression, or the element type in an array creation expression

(d) As the type in a cast expression

(e) In the right hand side of the instanceof expression

The above list is derived from the Java Language Specification, which defines syntactically identifiable type names in Section 6.5.1, and modified based on the Joos1W specification.

The resolution of type names would need to deal with both qualified names and simple names.As Java (and thus Joos) allows for use of fully qualified type name without import, and names cannot be partially qualified in Java, the resolution of qualified type name in Java is straightforward. One simply looks out the type from global environment. On the other hand, to resolve simple type names, we would need to look up the name, in the enclosing type declaration (the class or interface in which this type is used), in the single-type-import of the current compilation unit, in the package of the current class, and in the import-on-demand packages, exactly in this order. An error would be raised at this stage if no type under the name could be found. Besides if the name is resolved to more than one type (and thus is ambiguous), error would be thrown too. We also checks for prefix requirement in this process of resolving type name, that is for any qualified type name, its prefix should not be a valid type name in the class environment.

To increase efficiency the group has decided to conduct both environment construction and type linking in one traversal of the AST. This is feasible because if a simple type name is used within a type declaration (e.g. as return type), it should have already been declared, imported, or exist in the same package as the current type.

After type names are resolved, the class hierarchy needs to be built and checked for well-formedness. When building the class hierarchy, all classes that do not inherit from a super class implicitly inherit from the *java.lang.Object* class. Similarly every interface that does not have a super interface would inherit from an abstract version of the Object class. Checking the class hierarchy is straightforward. Besides the constraints given in class, it is also checked at this stage that no cycle exists in the class hierarchy.

The next step is to disambiguate ambiguous names. The Java Language Specification defines an *Ambiguous Name* to be the prefix of an *Expression Name*, a *Method Name*, or another *Ambiguous Name*, where an *Expression Name* could be the reference in an array access expression, a postfix expression, or the left-hand side of an assignment. The disambiguation process follows the simple steps in Algorithm 1. Every use of local variable or a static field is resolved at this stage.

Algorithm 1 Disambiguation of Field Name ($A_1.A_2....A_n$)

Require: Qualified Field Name ($A_1.A_2....A_n$)

1: If lookUp(A_1) is a local variable, then the rest are all non-static fields
2: If lookUp(A_1) is a class field, then the rest are all non-static fields
3: If lookUp(A_1) is a simple type, then A_2 is a static field, A_3 and onwards (if they exist) would be non-static fields.
4: if lookUp($A_1....A_k$) is a qualified type, then A_{k+1} is a static field, and every thing after that if exist would be non-static fields.

The resolution of instance fields and methods are slightly more complicated, because it requires static type information of the prefix which would be provided by the type checking step. The type checking step associates every expression (including name) with a static type, and checks that the program is statically type correct. Being statically type correct means that the program satisfies all the type rules, which were provided in class and also available in JLS.

Once the type information is in place, we only need to look up the field in the field environment of its object's static type to resolve the name. For example, given instance field a.b where the object a has type A, we only need to look for b in the field environment of A. The process for resolving method name is similar. The only difference is that method overloading needs to be taken into consideration, therefore given a method invocation, the static types of its arguments are needed to find out which method is actually being called.

2.2 Static Analysis

The major part of our static analysis is the reachability analysis. In terms of reachability, Java requires that:

1. all statements are reachable (no dead code); and
2. all methods whose return type is not *void* must eventually return a value

These properties are non-trivial and thus undecidable by Rice's theorem. However our program provides a conservative approximation to the problem. This means that the reachability analysis implemented would not detect all occurrences of unreachable statements, but the analysis is always correct when it identifies statements as unreachable.

The reachability analysis follows in general the rules provides in Section 14.20 of JLS. The reachability rules define whether a statement is *reachable* and whether it could *complete normally*. A compile-time error would be raised if any statement is identified as unreachable according the rules.

In Java, a local variable must be assigned before its value could be used. A definite assignment analysis is required to ensure this property. In Joos1W, this analysis is greatly simplified by requiring that all local variables are initialized when they are declared. Besides it is required that a variable cannot be used in its own initializer.

3 implementation

3.1 Name Resolution

The *Environment* class provides an implementation of scopes. The functions for managing scopes are implemented in the *SymbolTable* class. The construction of the global class environment is also implemented in the *SymbolTable* class, as the *buildGlobal()* static method which takes as input all the ASTs (one for each compilation unit). The *Environment* class also provides methods for looking up names, which would be used in later steps.

It was discussed in the last report that, when constructing the abstract syntax tree, a *Visitor* interface was created so that future steps could follow the visitor design pattern. And the construction of symbol table is implemented following this pattern.

The *TopDeclVisitor* constructs the symbol table. This class extends the class *TraversalVisitor* which as its name suggests is a visitor that traverses the AST. This inheritance allows *TopDeclVisitor* to focus on the AST nodes that are relevant to its function while skipping the nodes that are useless to the process. Without the *TraversalVisitor* the *TopDeclVisitor* would have to implement every abstract visit method specified in the *Visitor* interface. The use of *TraversalVisitor* avoids repetition and results in clearer code, therefore some of the other visitors implemented also inherit from *TraversalVisitor*.

The *TopDeclVisitor* builds 4 types of scopes:

1. the *CompilationUnit* scope, which contains information of the imports; and
2. the *Inherit* scope, which contains fields and methods inherited by this class; and
3. the *Type* scope, which contains fields and methods declared in the current type; and

4. the *Block* scope, which is used within and as the method body.

The TopDeclVisitor traverses the AST, creates and fills the scopes when appropriate. To be more specific, the *CompilationUnit* scope is first created and then the *Inherit* and *Type* scope. It was decided that the *Inherit* scope would not be populated until later steps where we build the class hierarchy. The *Type* scope however is immediately populated with fields and methods declared in this type. Then every method body is visited, and a new *Block* is created whenever a new block is opened.

In Java, a local variable must be declared before it is used. To ensure this, we create a new scope whenever a local variable declaration is created.

One challenge here is how to store the scope so that it could be retrieved efficiently in the future. It was decided that the environment created would be attached to the corresponding AST nodes. For example, when we open a new scope for a block, a reference of the scope would be attached to the block.

Another challenge at this stage is how to deal with overloaded methods. The full signature of a method must recorded in the environment so that when a overloaded method is called, it can resolved unambiguously to the correct implementation. To achieve this feature, we have decided to mangle the method name with the fully qualified name of its parameter types. This creates a unique signature for every overloaded function, and allows for very efficient resolution of overloaded methods. The name mangling function is implemented in the *NameHelper.mangle()* method.

It was mentioned in Section 2.1 that environment building and type linking would be done in one traversal of the AST for the sake of efficiency. However, in the implementation we still would like to separate the code of these functions and put them in different classes. Therefore type-linking was implemented in the *TypeLinkingVisitor*, and this class is instantiated and used by the *TopDeclVisitor* when appropriate (when visiting the type of a field declaration for example). With this implementation we achieve both efficiency and modularity of code.

The well-formedness of the class hierarchy is checked in the *Hierarchy* class. As it was discussed earlier in this section, the *Inherit* environment for each type was left empty. The *Hierarchy* class starts by populating the *Inherit* environment for each type declared in our program. This is achieved by one traversal through the hierarchy tree.During this process, the *Hierarchy* also conducts implicit inheritance from *java.lang.Object* as was specified by the Java language. Certain simple checks are carried out at this stage. For example, we check that no cycles exist while traversing the hierarchy tree, if a cycle is detected the traversal would be stopped and an error would be thrown immediately. The detection of cycle is fairly simple because the class hierarchy can be view as a directed graph. We simply check whether there is a back edge in the graph during the traversal. A back edge is defined as an edge from a node to itself or to its ancestors. After the *Inherit* environment was populated the *checkHierarhcy()* method was called to check that the hierarchy is well-formed.

The name disambiguation process is implemented in the *Disambiguation* class, this is again an implementation of the Visitor interface. The *Disambiguation* class inherits from the *EnvTraversalVisitor* class, which is an extension of the *TraversalVisitor* discussed earlier. The only different between *EnvTraversalVisitor* and *TraversalVisitor* is that the *EnvTraversalVisitor* would keep track of the current scope as it moves down the AST, so that we could look up names in the symbol table with ease. This disambiguation and name linking process follows Algorithm 1 in Section 2.1.

3.2 Type Checking

Type checking is implemented in the *TypeCheckingVisitor*. This visitor conducts one traversal of the AST and assigns a type to every expression. It also checks that all the type rules are satisfied. The type proof is constructed bottom-up. This means that when checking an expression, its subexpressions would be evaluated first and assigned types, and the expression would itself be evaluated only if all its subexpression satisfy the corresponding type rules.

More name linking

4 Testing

5 Conclusion