CS444/644 Assignment 1 Technical Report

Ho-Yi Fung, Jianchu Li, Zhiyuan Lin

March 18, 2016

Abstract

In this part of our compiler project, we design and implement the semantic analysis and the static reachability analysis phases of the compiler. The implementation has been completed successfully and works as expected. In this report we discuss the design decisions that we made for our program and the details of the implementation.

Contents

Chapter 1

Introduction

This part of our project has two components. The first component is the semantic analysis, including name resolution and type checking. In the process we would also implement checks for the well-formedness of the class hierarchy. The second component is the static analysis phase of the compiler. In the Joos 1W compiler, this includes two analyses – the reachability analysis and the definite assignment analysis. The latter as we will see is greatly simplified in Joos. All the analyses are performed on the abstract syntax trees (ASTs) constructed from the previous phases. Details of the AST design was discussed in the last report.

Chapter 2 of the report discusses the requirements for the program and design decisions the team made in order to satisfy the requirements, while Chapter 3 explains in detail how the program is actually organized and implemented. The testing strategies and tools we have adopted for our program are discussed in Chapter 4.

Chapter 2

Design

2.1 Name Resolution and Type Checking

The name resolution and type checking phase of our compiler follows 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 well-formed
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 the name resolution phase, we connect names to their declarations. The first step involves construction of a symbol table and resolution of syntactically identifiable names.

The first part of the symbol table that needs to be constructed is the *global* class environment. The global class environment contains information of all the types (classes and interfaces) declared, including those from the standard library. Once the global environment is in place, we would then construct the symbol table for each class. This process needs to search the AST for all the declarations of 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 is also called type linking. It deals with the following use of type names with in the AST:

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 [1], which defines syntactically identifiable type names in Section 6.5.1, and modified based on the Joos1W specification [2].

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 up the type from global environment. On the other hand, to resolve simple type names, we would need to look up the names 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), an error would be thrown too. We also checks for prefix requirement in the 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 we have 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.

Static fields and local variables are immediately linked to their declarations during the lookup in Algorithm 1. The resolution of instance fields and methods could not be done yet, 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 specified in JLS [1].

Once the type information is in place, to resolve a field name, we only need to look up the name in the field environment of its object's static type. For example, given instance field $a.b$ where the object a has static 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 [1]. 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 to 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 Joos 1W, this analysis is simplified because it requires that all local variables are initialized when they are declared. Besides it also needs to be checked that a variable is not used in its own initializer.

Chapter 3

Implementation

3.1 Name Resolution

The *Environment* class provides an implementation of scopes/environments. 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). Besides, the *Environment* class 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 follows this approach. 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. Most of the other analyses we have implemented also inherit from *TraversalVisitor*.

The *TopDeclVisitor* builds 4 types of scopes:

1. the *CompilationUnit* scope, which contains information of the types imported or from the same package; and
2. the *Inherit* scope, which contains fields and methods inherited by this type; 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. In Java, it is possible for a field and a method to have the same name. Therefore we create different namespaces for fields and methods. Afterwards every method body is visited, and a new *Block* environment is created whenever a new block node is reached.

In Java, a local variable must be declared before it is used. To ensure this, we create a new *Block* scope whenever a local variable declaration is found. Take Figure 3.1 for example. We would create a block scope for each variable declaration in 3.1a, as shown in 3.1b. Java also requires that local variable of the same name cannot be declared twice, therefore whenever a local variable declaration is found, the program would look up the name to make sure it does not already exist in the method body.

```
int x = 1;
int y = 1;
int z = 1;
```

(a) original code

```
{int x = 1;
{int y = 1;
{int z = 1;}}}
```

(b) actual scope

Figure 3.1: New Scope for Variable Declarations

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* node in AST.

Another challenge at this stage is method overloading. Because of the existence of overloaded method, it is not enough to store the name of method in the environment. The full signature of a method must be recorded so that when a overloaded method is called, it can be 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, in the order that they appear. For example, the method declaration *void foo(String bar);* would be stored under the name "3foo15java.lang.String" (3 is the length of "foo" and 15 the length of "java.lang.String". The numbers are used to delimit names). This creates a unique name 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*. 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 of its subexpressions satisfy the corresponding type rules.

We have chosen to hard-code the type rules into the visitor. This means that the rules corresponding to each expression are implemented in the visiting methods in *TypeCheckingVisitor* class. For instance, the rules for checking the *assignment* expression is implemented in the *visit(Assignment node)* method. This approach matches the expressions with their corresponding rules efficiently and creates a clear separation of different rules. All checks specified in lecture and the assignment have been successfully implemented. The resolution of instance field and methods is done after type is resolved. We implemented this in the same traversal as type checking so that there is no need to traverse the AST again. The method for resolution was already described in Section 2.1.

3.3 Static Analysis

The major part of static analysis phase is the reachability analysis, this is implemented in *ReachabilityVisitor* in the *static_analysis* package. The *ReachabilityVisitor* traverses the AST and makes sure that all statements are reachable at some point of the program. The *VariableAnalysis* class implements the definite assignment checks.

The major challenge in the reachability analysis is the evaluation of constant expression (as conditions in for-loops or while-loops). The definition of a constant expression is specified in JLS [1] Section 15.27. The actual implementation of this functionality is in the *ConstantExpression* class. The evaluation not only decide whether an expression is constant, but also computes its value. For example, for the purpose of reachability analysis it is not enough to know that the expression $((1 + 2) == 2)$ is constant. The value of the expression needs to be computed, and to do that the subexpression $(1 + 2)$ needs to be evaluated and compared to the right hand side.

Chapter 4

Testing

Following the practice during the implementation of the scanner and parser phases, we adopt the unit testing strategy, and utilizes the JUnit 4 testing framework to test our program. The unit testing strategy allows us to ensure that each requirement from the assignments is satisfied. For example, in order to test that all accesses of protected class members are legal, we created test cases with different class hierarchy and access scenario. This way we can make sure the protected access checking works as expected. Both positive and negative test cases, derived from the language specification, were used in this process. For instance, each rule that should be checked by the type checker has its own test cases to verify that the implementation is correct.

A major purpose of the semantic analysis is to ensure that the subject program conforms to the Joos 1W specification semantically. In each step, a large number of checks is performed. Therefore there are a lot of different functionalities/requirements to be tested. To allow for more efficient testing, we combined smaller test cases for different scenarios into three test suites, one for each assignment.

Because the development includes occasional modifications and bug fixes to the previous modules, we decided to test the program incrementally. Once a module is integrated into the compiler or a

major change is made, all the previous test suites are run to ensure that no new bugs are introduced to the program, and the changes work as planned.

The test cases from Marmoset were added to our test suites and tested against after each assignment is finished. In each JUnit 4 test suite, the parameterized testing method was used to run the test cases from Marmoset, so that we can make sure the program passes every test case.

The final product has successfully passed all the test cases from our test suites. This includes both the test cases created by the team, and those from Marmoset. This gives the team confidence that the implementation is robust and our design reasonable.

Chapter 5

Conclusion

In this report we discussed the design and implementation of the semantic analysis phase of our compiler. The implementation is complete and successful. All requirements from the assignments are satisfied, and all test cases, both those devised by the team and those obtained from Marmoset, are passed. The analyses are performed in time linear to the size of the ASTs, therefore their running time is acceptable and would work for even very large programs. There are, of course, ways to improve the program. For example, reachability analysis and variable analysis could be combined to save one traversal of the AST. It was a conscious choice to separate the two functions to achieve better modularized code. Besides, this choice does not affect the program's runtime asymptotically.

Bibliography

[1] J. Gosling, *The Java language specification*. Addison-Wesley Professional, 2000.

[2] "The Joos language." `https://www.student.cs.uwaterloo.ca/~cs444/joos.html`. Accessed: 2016-02-03.