1 Introduction

The parse tree will be converted to an AST tree and run through **BuildEnvironment**, **LinkType**, **HierarchyCheck**, **Disambiguate**, **TypeChecking**, **Definite Assignment** and **Reachability Check**

1.1 Name Data Type

In Java, a name can be a simple name or a qualified name. A simple name is a string and a qualified name is defined as a sequence of strings. We've defined 3 classes: Name, SimpleName and QualifiedName.

SimpleName extends Name class and contains a string.
QualifiedName extends Name class and contains a sequence of strings.

1.2 Enum Types

Literal Types, *Modifiers*, *Primitive Types* and *Binary Operators* are defined as sequences of enum array.

2 Abstract Syntax Tree

Our Abstract Syntax Tree has been structured into a tree with different types of nodes. Each type of node contains different information and in general can be classified into the following categories:

Declarations **Expressions** **Statements** **Types** **Others**

All Ast nodes extends to a superclass **AST** which contains 3 elements:

1. **Children**: An AST Node can have any number of children AST Nodes and this is based on our CFG, detail discussions will be made in in the following subsections.
2. **Name**: Each kind of AST Node is given a name and will be identified by its name.
3. **Environment**: Each AST Node may or may not be assigned with an **environment**. Environment represents scopes and is discussed in section 3. Not all nodes are required to contain an environment, therefore, we've used **Option[Environment]** in Scala to handle this situation.

2.1 Constructing the Abstract Syntax Tree

An Abstract Syntax Tree is constructed from a **Parsing Tree**. All different AST nodes are built from recurring the **Parsing Tree**. Along the way, some of the parsing tree nodes are flattened and all other nodes are transferred into different AST Nodes.
For example, for the following 4 CFG:

InterfaceTypeList InterfaceType
InterfaceTypeList InterfaceTypeList COMMA InterfaceType
InterfaceType ClassOrInterfaceType
ClassOrInterfaceType Name

The **InterfaceTypeList** parsing tree node will be ignored. All of **ClassOrInterfaceType**'s Interfaces name (Simple Name) will be collected as a sequence of Simple Names and is assigned to the **Class Declaration AST Node**. As can be seen, all unnecessary information have been abstracted away. Efficiency has been improved by reducing the number of AST nodes in the Abstract Syntax Tree.

2.2 Declaration AST Nodes

Each declaration from the CFG is matched and created as a corresponding declaration node. These nodes collect their own data. We will list some of these nodes as examples.

Import Declarations : An import declaration can contain a **single type import declaration** or an **on demand import declaration**. An import declaration node only contains one **Name** data type. Although the Import Declaration AST Node only contains one **Name** data.

Class Declarations : A class declaration contains a **Class Name (Simple Name)**, a sequence of **Identifiers**, a **Super Class (Simple Name)**, a sequence of implement **interfaces (Simple Name)** and a sequence of children nodes containing any method declarations or field declarations.

There is a special case for constructing **Class Declaration AST Nodes**. When a class does not have any parents, we assign to it an *Object* class as its parent.

2.3 Expressions, Statements and Types AST Nodes

All of **Expressions, Statements** and **Types** AST Nodes do not have children, but they have AST Nodes as parameters. This is due to the fact that their children can all be resolved into either **Expressions, Statements** or **Types** AST Nodes. More importantly, the CFG defined strict rules for what their children should be and the their positions in the corresponding **Parsing Tree**. To illustrate the point, an example of **Cast Expression AST Node is presented**.

2.3.1 Cast Expression

The CFG for **Cast Expression** is:

CastExpression LEFT_PAREN PrimitiveType Dims RIGHT_PAREN UnaryExpression
CastExpression LEFT_PAREN PrimitiveType RIGHT_PAREN UnaryExpression
CastExpression LEFT_PAREN Expression RIGHT_PAREN UnaryExpressionNotPlusMinus
CastExpression LEFT_PAREN Name Dims RIGHT_PAREN UnaryExpressionNotPlusMinus

Our **Cast Expression AST Node** contains a **Type AST Node** and an **Expression AST Node** as its parameters.

- The *Primitive Type Dims* will be resolved to an **Array Type**.
- The *PrimitiveType* will be resolved to a **Primitive Type**.
- The *Expression* will be first resolved into an **Expression AST Node**, a check will be made to see if it is a **Variable Expression**. If it is not a **Variable Expression**, an error will occur because it is not possible to cast to a type that is not an interface nor a class. If it is a **Variable Expression**, we will build a **ClassOrInterface Type** with that **Name** in the **Variable Expression**.

  A **Variable Expression** is an **Expression AST Node** that only contains a **Name**, which can be either a **Simple Name** or a **Qualified Name**. We convert names into **Variable Expressions** when an **AST Node** expects an **Expression** and a **Name** is present. In later **Name Disambiguation Stage**, we only need to disambiguate the **Name** contained in the **Variable Expression**.

  A **ClassOrInterface Type** might be a **Class Type** or an **Interface Type**. However, when building the AST, the information of whether the **Name** represents a class name or an interface name is not known. This type will be updated to a **Class Type** or an **Interface Type** in the type linking stage where the **Type** is known.
- The *Name* will be resolved into a **ClassOrInterface Type**

Next, the *UnaryExpression* or *UnaryExpressionNotPlusMinus* will be resolved into an **Expression AST Node** and it completes the construction of the **Cast Expression AST Node**.

2.3.2 Type Nodes

7 different **Type Nodes** are defined:

VoidType, PrimitiveType, ReferenceType, ArrayType, ClassOrInterfaceType, ClassType, InterfaceType, NullType.

ClassOrInterfaceType, ClassType and InterfaceType extends to ReferenceType.
ArrayType is special. It contains a **Type** as its parameter. For example, in *int[]*, the arrayType will contain a **PrimitiveType** as its type parameter.

All **Type** in our design must be one of the above type. This makes the later process of **Type Checking** to be trivial.

2.4 Helper Functions

We've defined 3 functions, *getType*, *getExpressionAst* and *getStatementAst* to return the corresponding AST Node, they take in input of a **Parsing Tree Node** and will process its children. They are recursive functions, and will recursively build AST Nodes for the **Type, Expression** and **Statement AST Node** of AST Nodes.

3 Name Resolution

There are two purposes for **Name Resolution**.

1. Assign **Environment** or scope to types in the AST Tree
2. Throw exception if a Name cannot be resolved

Environment is an abstract class which is used to represent the scope.
There are 3 environments in this design to represents 3 scopes:

1. **Root Environment** is the global scope which contains all **Type Scope Environment** for interfaces and classes.
2. **Type Scope** is the class or interface scope.
3. **Statement Scope** is the scope for all blocks, methods, constructors and local variables. Whenever a local variable or a parameter is declared, a new statement scope is constructed and the variable or parameter name is added into this statement scope.

Name resolution is resolved by traversing all ASTs 5 times. Note that an AST tree is constructed for each java file. These 5 traversals are defined in 5 different functions, in the order of:

1. **BuildEnvironment**
2. **LinkType**
3. **HierarchyCheck**
4. **Disambiguate**
5. **TypeChecking**

After analyzing these 5 traversals, it is possible to use 3 traversals to perform the name resolution. Specifically, **HierarchyCheck**, **Disambiguate** and **TypeChecking** can all be performed in one traversal. However, for simplicity and bug detection, this optimization is not performed at the moment.

In the following sections, the above 5 traversals will be discussed in detail.

3.1 Build Environment

Environment building is responsible to collect information and to build scope structures. A **RootEnvironment** (global scope) is constructed before the AST traversal because there will only be one global scope.

All **TypeScope** contains the **RootEnvironment** as their parent. All **StatementScope** contains either another **StatementScope** or **TypeScope** as their parent.

The **BuildEnvironment** function is using recursion and AST Node matching to handle different node types. During each recursion, It will update its current scope. Initially, the scope is a **RootEnvironment**. After class or interface declaration, the scope is updated to the newly constructed **TypeScope**. After any blocks, variable declarations, method declarations, field declarations or constructor declarations, the current scope will be updated to a **StatementScope**.

Whenever an interface declaration AST node is encountered, we construct a **TypeScope**. All information of class modifiers, extends, implements, full class name or full interface name will be stored in the **TypeScope**. The **TypeScope** will then be stored into the **RootEnvironment**.

The **RootEnvironment** contains 4 map structures to contain **TypeScope** information:

```
packageToClassMap: Map[QualifiedName, Set[SimpleName]]
packageToInterfaceMap: Map[QualifiedName, Set[SimpleName]]
canonicalNameToClassMap: Map[QualifiedName, TypeScope]
```

```
canonicalNameToInterfaceMap: Map[QualifiedName, TypeScope]
```

Both **packageToClassMap** and **packageToInterfaceMap** will take the package name as a key and map it to the set of simple class or interface name.

The full name of any interfaces or classes is known as a **Canonical Name** which is the type of a **Qualified Name**. Both **canonicalNameToClassMap** and **canonicalNameToInterfaceMap** maps such a full name to a **TypeScope**.

When traversing field declarations, method declarations and constructor declarations. These information will be stored into the following 3 maps of the **TypeScope**:

```
fieldMap: Map[SimpleName, (Type, Seq[Modifier.Value])]
constructorMap: Map[Seq[Type], Seq[Modifier.Value]]
methodMap: Map[(SimpleName, Seq[Type]), (Type, Seq[Modifier.Value])]
```

When traversing local variable declarations. The variable information is stored into a **StatementScope**. It contains one field.

```
variableMap: Map[SimpleName, Type]
```

Each scope will only contain its parent scope and data for its own scope. In later type resolution and name disambiguation step, it is possible for scopes to move to upper scopes to request for information.

3.2 Link Type

Link Type will check validity of any type names, It will handle import declarations and It will build a graph for **Topological Sort**.

All type names should be resolved to some class or interface name. Simple Type Names and Qualified Type Names are handled differently according the JLS. If the type name is valid and should be some class or interface, It will update the type name with a **Canonical Name** (full type name). One condition is that such class or interface type can be called in the current class or interface environment. It must be imported, or is in a same package or is part of the *java.lang* package.

For class declarations, edges from the current class to its extended class and edges from the current class to its implemented interfaces will be added to the cycle checking graph. For interface declarations, edges from the current interface to any of its extended interfaces will be added to the cycle checking graph.

There is one exception. If the current class is *java.lang.Object*, no edges will be added.

3.2.1 Cycle Check

After all java files have been processed by the **Link Type**, we check cycles by using the **Topological Sort** algorithm. It is guaranteed that no same type will occur as two or more different nodes in a type checking graph, because all nodes use full type name, and full type names should be unique.

3.3 Hierarchy Check

Hierarchy check is used to check inheritance relations and validity of method overriding. We've defined a method *gatherMethods* in our **typeScope**. All method override checkings are included in this method. *gatherMethods* will trigger *gatherMethods* from the current class's or interface's parent classes or parent interfaces. Its main structure is as following:

Checks such as *A class that contains (declares or inherits) any abstract methods must be abstract.* is performed after *gatherMethods*.

In addition, to the above *gatherMethods* pseudocode, all public *java.lang.Object* public methods are implemented by any interfaces that does not extend to other interfaces.

```
1: procedure GatherMethods
2:     result = empty method map
3:     p = all class or interface parents
4:     for type ← p do
5:         m = type.GatherMethods
6:         test inheritance between m and result
7:         merge result and m
8:     test inheritance between result and current scope's methodmap
9:     merge result and current scope's methodmap
```

3.4 Disambiguate Name

Disambiguate Name is used to disambiguate any non-static field access and method invocations. As discussed in section 2, **Variable Expression** is the only ambiguous name that needs to be disambiguate. There are two extra fields created in the **Variable Expression AST Node** named as **BaseType** and **AccessName**. Any resolved type will be placed into **BaseType** and the remaining **Qualified Name** will be placed in the **AccessName**

For example, if in the **Name Expression** *a.b.c.d*, if *a.b.c* is a valid type, then *a.b.c* will be placed into **BaseType** and *d* will be placed into **AccessName**.

In the disambiguate process, the parent scope of the **Variable Expression** is checked:

1. If the parent scope is a **TypeScope**, check if this **VariableExpression** is referring to a field name.
2. If the parent scope is a **StatementScope**, check if this **VariableExpression** is referring to a local variable or a field name.
3. If non of the above matches, check if this **VariableExpression** is referring to a static field name. 4. Throw an exception if the ambiguous name cannot be disambiguate.

3.5 Type Checking

Type Checking make full use of our **Type AST Node**. In general, type checking, assignability, narrowing, etc are followed from the JLS and is performed by the function *checkTypeAssignmentConvertible*, it takes in two types as input and return a boolean value, where **True** represents a successful check.

From **Disambiguate Name**, the **AccessName** of a **VariableExpression** is evaluated in **Type Checking**.

```
1: currentType = BaseType
2: for identifier ← accessName do
3:     if currentType == ArrayType and identifier = length then
4:         currentType = PrimitiveType(Int)
5:     else if currentType == ClassType then
6:         find identifier in currentType's field map and currentType's parent's field map
7:         Error if newField if identifier is not found
8:         Change the currentType to the found identifier's type
```

We do not check for **InterfaceType** in the above pseudocode, because from the CFG, interfaces do not have fields. However, It can be the **Type** for the last identifier in an **AccessName**. For example in the **AccessName** *a.b.c.d*, *d* can have type of an **Interface** and will not throw any exceptions.

We've also handled the **Array**'s *length* field. It is considered as a special case.

4 Static Analysis

Static Analysis is performed in two AST traversals through recursions, and we have implemented **Constant Expression Evaluation**, **Definite Assignment Analysis** and **Reachability Analysis**.

4.1 Constant Expression Evaluation

A field **constEval** is assigned to each **AST Node**. This field can be any of **Const Int**, **Const Boolean**, **Const String**, **Const Char**, **Const Byte** and **None**. Most evaluations are performed for the binary expression where any 2 constants might resolve to a new constant and compile time evaluation is required. The result of such evaluation is assigned to the **constEval** field of the binary expression. Division by 0 is handled during this traversal.

4.2 Definite Assignment Analysis

If any fields or variables are assigned, this information will be added to a set and passed down the tree through recursion. Assignment is only performed in the **Assignment AST Node**. If any of the RHS of **Assignment** is using an uninitialized field or variable, an exception will be thrown.

4.3 Reachability Analysis

For each statement, its children's reach-ability is analyzed in order. If a previous child's **outMayber** is analyzed to false, then an exception will be thrown in **checkBlockNode**.

```
def checkBlock(block: Block): Boolean = {
 var outMaybe = true
 block.children.foreach {
   stmt => outMaybe = checkBlockNode(stmt, outMaybe)
 }
 outMaybe
}
```

Reachability checks are performed on **IfElseStatement**,**ForStatement** and **WhileStatement**. The check will be performed on the **Block Node**, which is a child of these statements.

We've handled two special case for **MethodDeclaration** and **ConstructorDeclaration**. An exception is thrown if the **MethodDeclaration** does not return and the method type is not **VOID**.
ConstructorDeclaration should not contain any return statements, an exception will be thrown if it does contain.

Using recursion and traversal simplifies the check because information regarding return types can be pass through recursion from the parent to the children.

5 Testing

Tests for A2, A3 and A4 are split into 4 categories: **Single File Pass**, **Single File Fail**, **Multi File Pass** and **Multi File Fail**. Our test focuses on **Parenthesized Expression**, **Implicit Type Conversions** and **Cast Expression**.

The main source of testings is base on A2 - A4 marmoset test files. Most cases are covered and does not need custom tests.

It is possible to initialize field or variables in the **Parenthesized Expression**, therefore individual **Single File Pass** and **Single File Reject** tests are performed to check for identifier initialization.

Implicit Type Conversions is used to test the **Type Checking**. Tests like using a char variable as index for **ArrayAccess** should be allowed and is verified during tests.

The purpose of **Cast Expression** is used to check whether multiple casting expressions (e.g *(int)(byte) a*) is valid.