Asmar Khalid, ShengMin Zhang, Freddie Feng
Group 12

CS 444 - Compiler Construction

Assignment Four Design Doc

This design document succeeds the assignment one design document.

AST Building

In the previous assignment, we decided to defer AST building to assignment two. It is modelled from the Eclipse AST for Java. For the purposes of this document, an AST is inferred to be an AST rooted at a CompilationUnit AST Node. A CompilationUnit node represents an optional package declaration, sequence of import statements, and an optional type declaration.

Our AST construction process is heavily leverages Scala's pattern matching mechanism. Our parse tree memoized the production rule which was used to create every node, and therefore we pattern match the production rule and recursively construct a node and its children.

Every AST node is a scala *case class* with immutable pointers to children (or a token if it is a leaf). An example of an AST node (and its factory construction method) follows.

```
case class ParenthesizedExpression(expression: Expression) extends Expression {
  override def toString = s"(${expression})"
}

object ParenthesizedExpression {
  def apply(ptn: ParseTreeNode): ParenthesizedExpression = {
    ptn match {
      case TreeNode(ProductionRule("PrimaryNoNewArray", Seq("(", "Expression", ")")),
_, children) =>
        ParenthesizedExpression(Expression(children(1)))
      case _ => throw new AstConstructionException("No valid production rule to create
ParenthesizedExpression")
    }
  }
}
```

A comprehensive list of all of the defined AST nodes can be seen in the joos.ast package.

AST Traversal

We have made use of the visitor pattern to traverse and update the AST. Additionally, we use pattern matching when appropriate. (ie. look-aheads on the AST).

Semantic Analysis

We realized that in order to conduct semantic analysis, we would have to augment the AST with various environments and links.

Our very first design decision was whether to (1) add mutable fields (and therefore lazy initialization logic) to the AST nodes, or (2) maintain an immutable AST, consistent with a functional style of programming.

```
Design choice 1: For performance reasons, we added mutable environment fields (and
therefore lazy initialization logic) to the AST nodes.

Challenges incurred: Ensure that the AST was initialized (and correct) when
environments were referenced
```

Assignment Two: Name Resolution [git checkout A2]

We followed the suggested ordering given by the assignment specification. That is, we conduct *Environment Building*, followed by *Type Linking,* and finally *Hierarchy Checking.*

Environment Building

In this phase of semantic analysis, we build the scopes that each AST node has a reference to, and then ensure the following:

- No two fields declared in the same class may have the same name.
- No two local variables with overlapping scope have the same name.
- No two classes or interfaces have the same canonical name.

The environment building phase is implemented as a *stateful* visitor over our AST. It can be found in *EnvironmentBuilder.scala*. It is stateful because it keeps track of already-defined field names.

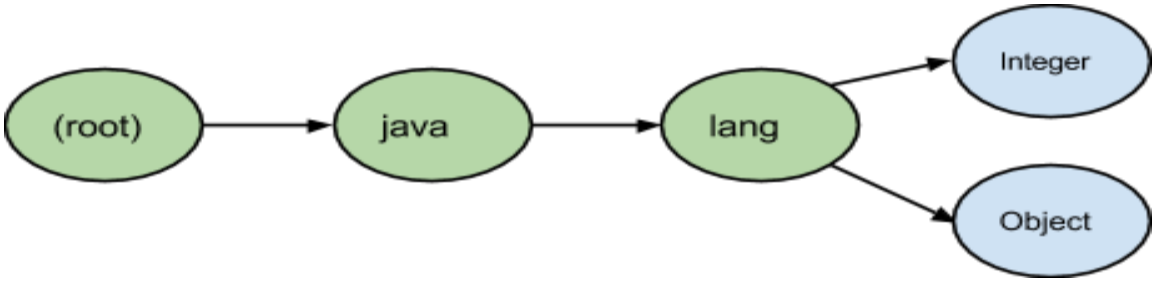
The following is an enumeration over all of our environments. Not all are used in this phase of the compiler, and their descriptions will be revealed later.

- ModuleEnvironment
- CompilationUnitEnvironment (scala trait or mix-in)
- TypeEnvironment (scala trait or mix-in)
- BlockEnvironment (immutable object that supports cons-like operations)

All of these environments can be found in the joos.semantic package.

A *NamespaceTrie* is a mapping from a name to a type declaration. It is implemented as a trie-like data structure where leaves represent a type, tree nodes represents a package segment. Transitions are strings (“java”, “lang”, “Object”), rather than characters. It is also capable of detecting equivalent type declarations, or type declarations with a common prefix.

Below is an illustration of a NamespaceTrie with java.lang.Integer and java.lang.Object added.



A *ModuleEnvironment* is an environment that holds all type declarations. It is a wrapper around a NamespaceTrie. When a type is declared, it is added to the ModuleEnvironment’s NamespaceTrie. This allows us to prevent two types from having the same canonical name.

Duplicate field declarations are checked when traversing ASTs FieldDeclaration nodes.

We created an immutable *BlockEnvironment* structure which is injected into the appropriate AST nodes. The BlockEnvironment holds a mapping for locally defined variables to its type. When adding a new variable, the current BlockEnvironment checks whether or not it is defined already. It is important to note that BlockEnvironments are built-on top of each other in nested scopes to avoid losing previous variable declarations.

Type Linking

A *CompilationUnitEnvironment* is an environment that holds all *visible* type declarations. Each CompilationUnitEnvironment is mixed into a CompilationUnit AST node. Therefore, every AST has precisely one CompilationUnitEnvironment.

The main interface to the CompilationUnitEnvironment is shown below. We will further elaborate on the two different cases of type linking (qualified and unqualified (simple) names).

```
// Gets the type with the {{name}} if it's visible within this compilation unit
def getVisibleType(name: NameExpression): Option[TypeDeclaration] = {
  name match {
    case simpleName: SimpleNameExpression => getUnqualifiedType(simpleName)
    case qualifiedName: QualifiedNameExpression => getQualifiedType(qualifiedName)
  }
}
```

Resolving unqualified (simple) types

A CompilationUnitEnvironment has four instances of a NamespaceTrie. Each trie handles a different namespace of type declarations. These are used when non-qualified types are referenced.

The following code snippet is taken from *CompilationUnitEnvironment.scala* should further clarify the roles of each NamespaceTrie.

```
 trait CompilationUnitEnvironment extends Environment {
  self: CompilationUnit =>

  val enclosingClass = new NamespaceTrie
  val concreteImports = new NamespaceTrie
  val enclosingPackage = new NamespaceTrie
  val onDemandImports = new NamespaceTrie

...

 /**
   * Unqualified names are handled by these rules:
   * 1. try the enclosing class or interface
   * 2. try any single-type-import (A.B.C.D)
   * 3. try the same package
   * 4. try any import-on-demand package (A.B.C.*) including java.lang.*
   */
  private def getUnqualifiedType(name: SimpleNameExpression): Option[TypeDeclaration] =
{
    var typed = enclosingClass.getSimpleType(name)
    if (typed.isEmpty) {
      typed = concreteImports.getSimpleType(name)
    }
    if (typed.isEmpty) {
      typed = enclosingPackage.getSimpleType(name)
    }
    if (typed.isEmpty) {
      typed = onDemandImports.getSimpleType(name)
    }
    typed
  }

...
```

Resolving qualified types

Resolving qualified types are delegated to the ModuleEnvironment's NamespaceTrie.

```
  private def getQualifiedType(name: QualifiedNameExpression) = {
    // Omitted prefix error checking...
    moduleDeclaration.namespace.getQualifiedType(name)
  }
```

The *TypeLinker* is a *stateless* visitor that handles the type linking phase. The following is a comprehensive description of the type linking phase:

1. For each import declaration, it will add the referenced type declaration (if it can be resolved according to the ModuleDeclarations NamespaceTrie) to the appropriate CompilationUnitEnvironment's NamespaceTrie
2. For each referenced type name in the subtree, it will check whether or not the *getVisibleType* method resolves to a *TypeDeclaration* (or throws a name prefix error) or not.

Hierarchy Checking

The Hierarchy checking phase is split into two separate visitors, the *SimpleHierarchyChecker* and the *AdvancedHierarchyChecker*. Both are implemented as *stateless* visitors. The cases that each cover are documented in their respective files in the joos.semantic.names.hierarchy package.

For brevities sake, we do not list all of the checks here. With one caveat (which is discussed further) the checks are relatively simple and require little finesse.

Cyclic Hierarchy Checking

The algorithm we use to check for a cyclic hierarchy structure is relatively simple. We create a topological ordering of the hierarchy graph and ensure that no type occurs twice.

```
private def checkCyclicHierarchy(typeDeclaration: TypeDeclaration) {

  val children = mutable.HashSet.empty[TypeDeclaration]

  def visit(typeDeclaration: TypeDeclaration) = {
    if (!(children add typeDeclaration)) {
      throw new CyclicHierarchyException(typeDeclaration.name)
    }
  }

  def leave(t: TypeDeclaration) = assert(children remove t)

  // Topological sort of dependency graph
  def findCycle(currentType: TypeDeclaration) {
    visit(currentType)
    getSuperType(currentType) ++ getSuperInterfaces(currentType) foreach findCycle
    leave(currentType)
  }

  findCycle(typeDeclaration)
}
```

The caveats here are in the *getSuperType* and *getSuperInterfaces* methods. All objects (but Object) implicitly extend Object, and all interfaces implicitly extend an Object interface. We do

special handling in these two functions to return the appropriate super type (or None if Object is given).

Two solutions that were plausible were to create an Object interface source file, and inject this resource into every compilation phase, or create a fake object interface type declaration.

```
Design choice 2: Instead of creating an ObjectInterface source file, we hack together a
“fake” TypeDeclaration with abstract object methods. This is the easiest and most
immediate way to handle this problem

Challenges incurred: We have to deal with some inconsistencies in our environment. For
example, the ModuleEnvironment has no notion of this Object interface. We must be very
careful not to treat this class like a user-defined type.
```

TypeEnvironment

A *TypeEnvironment* is an environment that is mixed-in to every TypeDeclaration.

It contains information of all of the fields, methods, constructors, and supertypes of a type.This includes local and inherited members, all parent super types, and all ancestral super types. Since the hierarchy checking phase checks this information anyways, it is then stored in the TypeEnvironment for later stages of the compiler to use.

Assignment Three: Type Checking [git checkout A3]

We followed the suggested ordering given by the assignment specification. That is, we conduct *Disambiguation and Linking of Names*, followed by *Type Checking.*

Name Disambiguation

We classify all NameExpressions as either PackageName, TypeName, InstanceFieldName, InstanceMethodName, StaticFieldName, StaticMethodName, or LocalVariableName.
Because the Joos specification is simpler, this process is only partly adopted from the JLS 6.5.2 *Reclassification of Contextually Ambiguous Names.*

NameClassifier is an AST Visitor that traverses the entire AST until it finds a NameExpression. If it is a qualified name it will resolve the qualifier first. Otherwise, it will use the available context (local fields, instance fields, visible types/packages) to try and disambiguate any unqualified (simple) names.

Name Linking

After the NameClassifier has classified all names, the *StaticNameLinker* is kicked off. This visitor will use the classification of each NameExpression to find a linkable declaration. It will also handle detecting forward declarations as per section 8.3.2.3 of the JLS.

Due to the nature of the joos grammar, method names cannot be linked yet. This is because the prefix of a method invocation can be any expression, but we have not lifted the types of each name into any outer expression that uses it yet.

For clarity, in the expression: `(new Object()).hashCode()`, hashCode() is the method invocation and (new Object()) is a parenthesized expression which is the prefix of the method invocation. Clearly, the type of the parenthesized expression is required to link the method invocation to the correct method declaration.

```
Design choice 3: Instead of lifting out the NameExpressions type and declaration into
outer expressions (which is required to link method invocations to the appropriate
method declaration) during this phase, we defer that to the type checking phase. This
reduces the complexity of this phase.

Challenges incurred: Linking for method invocations must also be deferred to the type
checking phase, which adds some complexity there. Fortunately, the type checking phase
is simple enough that the added complexity isn't a huge burden.
```

Type Checking

The type checking phase ensures that all expressions and statements are typed correctly. This is handled by *TypeChecker,* an AST visitor.

```
Design choice 4: Although the implementation for each expressions type check is
relatively simple, the amount of code is large. Therefore we have partially implemented
the TypeChecker across a number of files.

class TypeChecker(implicit val unit: CompilationUnit)
    extends AstEnvironmentVisitor
    with AssignmentExpressionTypeChecker
    with ParenthesizedExpressionTypeChecker
    // Omitted type checker traits
    with InstanceOfExpressionTypeChecker {
```

Since we made the decision to defer the type linking of method invocations, we first propagate (lift) the types outward from each inner expression. Afterwards, we do the obvious type checking for each statement and expression. Finally, we will store the type of the given expression for any encompassing expressions to make use of.

An example of a type checker is provided for clarity:

```
trait ParenthesizedExpressionTypeChecker extends AstVisitor {
  self: TypeChecker =>

  override def apply(parenthesis: ParenthesizedExpression) {
    parenthesis.expression.accept(this)  // Recursively type check inner expression

    // No type checking needs to be done for parenthesized expressions
    // Lift inner expression type into current expression
    parenthesis.expressionType = parenthesis.expression.expressionType
    parenthesis.expression match {
      // Lift inner declaration references into current expression
      case expression: DeclarationReference[_] => parenthesis.declaration =
expression.declaration
      case _ =>
    }
  }
}
```

The only special case is the *MethodInvocationTypeChecker*, as noted before. It will first recursively lift the types out of the prefix expression (if it exists). Then, it employs similar logic as the StaticNameLinker to link the appropriate method declaration to the method invocation.

Assignment Four: Static Analysis [git checkout A4]

This assignment was relatively painless and straightforward. It was only a few hours of implementation work. As a result, there is not a great deal of content discussed here.

Visitor Dispatching

Up until now, we have decided to not allow a visitor to dispatch another visitor on a subtree. Our method of implementation required a strict ordering of visitors during each phase of semantic analysis. For example, for type linking, we could not link the types of a method invocation until the expression types had been lifted. Further, we could not lift the expression types from sub expressions until names had been disambiguated.

We have attempted a different approach for this assignment. We attempt to abstract this ordering by allowing visitors to dispatch secondary visitors on a subtree when a dependency must be fulfilled. This creates an implicit ordering which does not have to be strictly maintained.

```
Design choice 4: Allow visitors to dispatch other visitors when a dependency is
present. This allows all dependencies to be implicitly ordered.

Challenges incurred: As opposed to creating two visitors (one to instantiate the
expected environment, and the other to operate on the environment), we suffer a loss of
performance, as dispatching a secondary visitor may not always be necessary.
```

Reachability Checking

ReachabilityChecker is the AST visitor that is responsible for deciding whether or not all statements are potentially reachable. It will traverse all MethodDeclaration bodies and decide whether or not all the statements within are potentially reachable.

Every statement node in the AST is augmented with a *Reachable* trait which contains *canStart* and *canFinish* fields, which represents whether whether we can start and finish executing this statement. These fields are filled with the correct values (No or Maybe) as the visitor descends into the AST using the rules defined in class and in JLS 14.20. It checks that all statements' *canStart* field is not set to No. If a statement is deemed unreachable, we will throw an *UnreachableException*.

The *ReachabilityChecker* is also capable of dispatching the secondary *ExpressionEvaluator* visitor which evaluates constant expressions and produces their value if it can be determined according to the rules defined in JLS 15.28. This is used to evaluate the condition expressions in *ForStatement* and *WhileStatement* so the correct reachability rules can be used based on whether the condition is evaluated to true or false or unknown

Furthermore, during this process, if a method has a non-void return type, we ensure that a return statement exists as the last statement for every execution path by checking *canFinish* field is not set to Maybe

Variable Initialization Check

VariableInitializerChecker is responsible for checking:

- Each local variable declaration has an initializer
- Inside initializer, it does not refer to anything that's undeclared at that point(including referring to itself)

It is a secondary visitor that is kicked off by the *TypeChecker* when it encounters a *VariableDeclaration*

Testing

Unit Tests

As usual, we write unit tests for most of the isolated classes in our code. For example, the aforementioned NamespaceTrie structure must adhere to the behavior defined in *NamespaceTrieSpec*. Unfortunately, due to time constraints, and tight coupling, unit tests for many classes has been either omitted, or not comprehensive. Instead we rely on integration testing.

Integration Tests

Within integration tests, we have two separate categories, marmoset tests and custom tests.

Marmoset Tests

These tests are directly imported from the assignment test cases and are run to ensure the correct output, or correct exception is thrown every time we build. This greatly helps us reduce the risk of causing regressions. These can be found in *MarmosetAXSpec*, where X is an assignment number.

Custom Tests

Sometimes, we feel that the marmoset tests are not comprehensive for every feature we implement. When we come across a tricky case, or one that is not covered by the marmoset tests, we have the ability to create our own joos source files for testing. This logic can be found in *IntegrationSpec* and our custom tests can be found in the compiler/src/test/resources/integ/{valid,invalid} directories.

Your report is organized and clearly written, and provides more than sufficient detail about the design of your compiler. You have made good design choices, and you have tested your compiler thoroughly. Your compiler appears to be in good shape for A5.

Presentation: 20/20
Content: 10/10
Total: 30/30