CS444 Assignment 2, 3, 4 Design Document

An, Qin Kuan, Wei Heng McIntosh, Daniel David

March 10, 2020

Overview

After all compilation units have been successfully scanned and parsed, they are fed into what is called the semantic analysis stage. The goal of this stage is to prepare the AST for code generation. It is important to reject all programs that do not conform to the JLS, e.g. ill-typed programs so that the code generator can make stronger assumptions for ease of implementation. Once the AST clears this stage, it is assumed to be free from any statically checkable errors, so most of the information used during analysis (e.g. access modifiers) can be ignored during code generation.

(Global) Environment Building

The semantic analysis begins by computing fully qualified names (FQN) of all classes. This is fairly straightforward since the only thing that needs to be done is prepending package name to class name. A global hash map recording all classes is created to enable later stages to perform lookups with fully qualified class names. In addition to that, a trie data structure is also created, with each node denoting a component between consecutive dots in a package name. This allows checking whether the prefix of a package name is some other class name efficiently.

The rest of the environment (the local environment) is built, used, and destroyed during expression resolution, and so will be discussed later.

Type Linking / Type Resolution Pt 1

The next stage is to resolve Names that unambiguously indicate classes. During AST construction, we frequently made a distinction between NameType and other Name AST nodes since the locations in a program can often require a Name to represent a Type. For example, Names inside “extends” or “implements” clauses must be a Type. Since we know unabiguously that the implements and extends clauses must be Types, we can resolve them to their TypeDeclarations. This stage prepares for building class hierarchy by establishing a parent-child relationship. The algorithm for resolving type names is a sequence of lookup attempts in the global FQN->TypeDeclaration hash map, coded according to Java specification regarding resolving type names.

Hierarchy ordering

When all unambiguous type names have been resolved, the semantic analysis is ready to move on to the next stage which is checking class hierarchy. The formal algorithm with contain/declare sets constructs is recursive in nature and there are two possible top-down or bottom-up approaches to it. Since doing top-down algorithm with memoization introduces extra complexities of maintaining results that have been calculated, we used bottom up solutions, as they are more explicit and easier to understand. The prerequisite for this though is a topological sort on the graph of class dependency. This is a typical textbook algorithm in depth first search and, with the previous stages having already set up the parent-child relationship records, is fairly simple to implement. It also comes with the bonus of detecting cycles in class hierarchy so that no other part of the class hierarchy building algorithm needs to worry about this scenario.

Type Linking / Type Resolution Pt 2

After heirarchy checks have been completed, the next step is to link the remaining Types to their declrations. For NameType AST nodes, this generally follows the same pattern as resolving the Types in the "extends" and "implements" clauses. For other Names, we don't know if they or one of their prefixes will disambiguate to a Type yet. However, we'll still try to find the smallest prefix which can be resolved to a type, then build the remaining field access expressions. If there is no such prefix, that's fine, we will handle that during expression resolution. we'll also deal with the case where we should have resolved it to a local variable or a field then.

Hierarchy Checking

We perform hierarchy checking in topological order from the root class to the leaves. Since we never need to query information about subclasses, we are guaranteed to have all the information we need when generating the various set (e.g. super set).

As a general principle, we try not to modify the AST destructively to maintain the integrity of the source program. Adding new fields to AST nodes is acceptable, since these fields can be ignored if we ever needed to process the AST in its original form, e.g. when converting it into printable code. Following this principle, a new field is created to hold all modifiers directly expressed in the source program and implicit ones given by the language rules.

We also implemented JLS 9.2 by creating a source file for an interface with the public method signatures in `Java.Object.Lang`. This `IObject` interface is then generated using the same compilation steps as any other user-provided code. Hence, we were able to get construct the data structures representing the AST without having to implement a bunch of methods to manipulate the AST which are not used elsewhere.

The `IObject` interface is then inherited by interfaces without superinterfaces in this phase, making the inheritance rule in JLS 9.2 explicit.

Name Linking / Expression Resolution

In order to perform Name Linking for expressions(referred to in our code as Expression Resolution) we needed to do 3 things for each AST Expression node:

- Namespace Disambiguation
- Type Deduction for child nodes
- Where appropriate, search the environment for a matching definition

Since Type Deduction requires that expression resolution be performed first, this was a recursive process.

Local Environment Building

In order to perform Namespace Disambiguation, we first need to build up the environment. The local environment is contained in the Scope class. A scope contains a list of all parameters and local variables, the enclosing TypeDeclaration, the enclosing method/constructor if it exists, and if we're resolving the initializer for a local variable, the declaration of the variable being initialized. The enclosing method/constructor is only used to validate return statements, and the current variable declaration is only used to ensure we don't have a self-referencing initialzer. The imporant portion of a scope is the parameters and local variables, and the enclosing TypeDeclaration.

It is worth pointing out here that the scope does not contain any information about the fields or methods of the enclosing class, except indirectly through the enclosing TypeDeclaration. Furthermore, the fields and methods are not accessed through the enclosing class pointer in a Scope. We will discuss how they are accessed later in "Declaration Search". For now we will focus on the parameters and local variables.

In order to simplify the handling of "this" tokens, we add a "this" parameter declared with the type of the enclosing class to all constructors and non-static methods. Then, we treat "this" just like any other local variable. For field initializers we do something similar as well, except instead of modifying the Method/Constructor Declaration, we add "this" directly as a parameter/local variable to the scope.

Since we process statements in a block sequentially, by adding local variables to the scope as we encounter them, we are able to prevent forward references to variables before they're declared. Similarily, since we construct a new scope, and copy over the contents of the parent scope everytime we encounter a block, if, for or while statement, passing that new scope to our child expressions, any declarations the children add to it will not be visible after the end of the block/if/for/while. Unfortunately, the rules for forward references in field initializers are too complex to be handled in the same fashion. To detect forward references in field initializers, we used a weeding stage which ran before expression resolution.

Namespace Disambiguation

The first step of disambiguating a name is determining whether the first identifier is a local variable, a field in the enclosing class, or part of a (possibly package-qualified) TypeName.

Since we have a list of all parameters and local variables visible to an expression in a scope object, resolving to a local variable is simple. A basic lookup in the list of variables/parameters for the first identifier will tell us whether the name should be resolved to a local variable. If that fails, determining whether we should resolve to a field in the enclosing class is equally simple. Since unqualified access to static fields is forbidden in Joos, it could only be a non-static field. In which case, we can make the implicit "this.FIELDNAME" explicit by constructing and resolving a "this" local variable expression, then build a field access expression from that "this" expression and the first identifier in the name. If we fail when we try to resolve either the "this" or the field access expression, we shouldn't resolve it to a field in the enclosing class.

If the first identifier is either a local variable or a field, the rest of the name must be a sequence of field accesses. If it's not a local variable or field, the only remaining option is that the first identifier is part of a TypeName. Since we already built the fully converted Expression/TypeName for this case during the type linking stage, we just use that.

During namespace disambiguation, similar to fields, we also add an explicit "this" to method invocations which don't have any explicit source.

Type Deduction

By the time we have to deduce the type of a node, we will have already linked ourselves to a declration. Since the resulting type of all primary expressions is either a Type in the AST node (e.g. class instance creation expressions result in the type of the class), or they have an associated declaration, we can easily deduce the types of all primary expressions. Since all other expressions are built from primary expressions, we just need to write rules for what the resulting type of the other expressions are using the types of their component expressions.

One of the most complicated cases to handle in all type checking phase is binary expression. This is the place with the most scenarios to consider due to the combinatoric explosions of the type of left/right hand side and operand. With the exception of instanceOf operand, all other operations are commutative. Therefore a helper function is created to only check for half of the cases, and high level type checking code for binary expressions simply calls this helper function twice with left hand side and right hand side switched.

Declaration Search

The last step in Expression resolution is performing a search for the declaration in the environment. For local variables and parameters, this means searching through the scope. For fields and methods, this means searching the type declaration of the source (the thing before the .) for a field/method with a matching id/signature. Recall that during namespace disambiguation, we added an explicit "this" so all field accesses and method invocations have an explicit source - either an expression in the case of an instance field, or a type in the case of a static field. As such, we never use the enclosing class pointer from the scope object to perform the lookup.

Reachability Checks

For reachability checking, our algorithm closely matches what is described in the class. Each AST node type has a different rule for calculating in/out results, and the algorithm performs a top-down recursive traversal of the AST to detect unreachable statements. We simplified the handling of ternary logic with the observation that the boolean relations used in this analysis still hold after replacing "maybe" with "yes".

$$\begin{aligned} \text{maybe} \lor \text{maybe} &= \text{maybe} & \qquad \text{yes} \lor \text{yes} &= \text{yes} \\ \text{maybe} \lor \text{no} &= \text{maybe} & \qquad \text{yes} \lor \text{no} &= \text{yes} \\ \text{no} \lor \text{no} &= \text{no} & \qquad \text{no} \lor \text{no} &= \text{no} \end{aligned}$$

This allowed us to simply use the boolean type with the builtin boolean operators.

Testing

To recap, our compiler contains functionalities that automatically run all test cases for a given assignment. This is one of the most important features in the compiler, as we rely on it to confirm our assumptions and verify implementations of new features.

As the project evolves, so do the complexity of test cases and the time it takes to compile them. Specifically, with the additon of standard library files that are bundled with each test case, the time it takes to run our automated tests increased by over 100% since all standard library files are re-scanned and re-parsed every single time. To reduce the time we have to wait before seeing test results, we implemented caching of parse trees for standard libraries files. We cannot include caching of abstract syntax trees for them, because AST contains some fields referencing other compilation units that may not be part of standard libraries, and having to reset all of those fields is error-prone and cubersome. The performance gain obtained through caching parse tree alone is already sufficient for the purpose.

Additionally, we also implemented parallel test execution with multiple threads, since there are no dependencies at all between different test cases therefore not much sychronization or concurrent programing trickery are required for this additional speedup. In the end, we are able to run all 323 test cases in assignment 3 under one second on linux.student machines. This enabled us to work at a faster pace and get more work done in the same amount of time.

Maybe include a blurb here if there was anything new or different we did for A2-4 Up to you whether any changes warrant discussion in the doc. If not, just remove this section