CS444 Assignment 2, 3, 4 Design Document

An, Qin Kuan, Wei Heng McIntosh, Daniel David

March 10, 2020

Overview

@Daniel and/or @Titus and/or @Wei wei

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

@Daniel link remaining NameType's to declarations predictive linking for ambiguous names

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 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.

Expression Resolution

@Daniel

Namespace Disambiguation

@Daniel Goal is to convert Names to either a FieldAccess, NameType, or LocalVariableExpression Treat qualified and simple names the same - can't always differentiate qualified names from field access For local variables and fields, only need to look at first identifier in a name. Access to fields in the enclosing class via simple names is accomplished by prefixing ambiguous names with a "this" parameter and trying to resolve it

Local Environment Building

@Daniel Scope contains: ... check for forward references in fields during a pre-emptive weeding stage

"this" added as parameter to non-static functions field initializers treated as though they were part of a function with only a this parameter? add local variables to scope as we go to prevent forward references

Type Deduction

@Daniel and/or @Titus, since BinaryExpressions were the biggest part

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

@Daniel

Heirarchy Checking v2?

@Daniel Not sure if this is actually worth talking about

Reachability Checks

@Titus

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".

maybe $\vee$ maybe	= maybe	yes $\vee$ yes	= yes
maybe $\vee$ no	= maybe	yes $\vee$ no	= yes
no $\vee$ no	= no	no $\vee$ no	= no

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

Testing

@Titus

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