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.

Global Environment Building (import processing)

@Titus?

The semantic analysis begins by computing fully qualified names 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 next stage is to resolve references that are unambiguous to classes. During AST construction, we made a distinction between Name and NameExpression AST nodes since they appear at different locations in a program. NameExpression cannot be disambiguated until type checking. Therefore we only need to attempt resolving all AST node of type Name. The algorithm for resolving type names is coded according to Java specification regarding resolving type names. This stage also prepares for building class hierarchy. A parent-child relationship is recorded when resolving type names that appear in "extends" or "implements" clauses.

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 topological sort on the graph of class dependency. This is a typical textbook algorithm in depth first search and, with the previous stages already properly 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

@Daniel

Heirarchy Checking

@Wei wei

Expression Resolution

@Daniel

Namespace Disambiguation

@Daniel

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 number of 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 noticed that since maybe v maybe = maybe, maybe v no = maybe, and no v no = no, the v operator is exactly the same as boolean or, if maybe/no is assigned boolean true/false. Therefore internally the code uses boolean variable to record in and out, and in/out operations become logical operators on booleans. Everything worked naturally.

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