Prof. Jingke Li (FAB120-06, lij@pdx.edu), Tue&Thu 12:00-13:15 @SH 212, Lab: Fri 10:00-11:15/11:30-12:45 @EB 103

Homework Assignment 4: Static Analysis

(Due 12/3/15 @ 11:59pm – Firm Deadline!)

This assignment explores static analysis. Specifically, it asks you to implement analyses that detect unreachable statements and uninitialized variables in miniJava programs. You will implement your analyses by writing code to traverse the miniJava AST trees that you targeted in Homework 3. The assignment assumes familiarity with the OO tree-traversal techniques practiced in Labs 7 and 8, and is intended to be attempted after you have completed those labs.

This assignment carries a total of 10 points.

Preparation

Download the zip file "`hw4.zip`" from the D2L website. After unzipping, you should see a `hw4` directory with the following items:

`hw4.pdf` — this document

`Ast0.java` — the AST definition program file, with a starting version of the two analyses

`AstParser.jj` — a parser for reading back the dumped AST format

`StaticCheck.java` — a top-level test driver

`Makefile` — for building the analyzer and test driver

`run` — a script for running tests

`tst` — a directory containing sample tests

Task 1. Detecting Unreachable Statements (4 points)

As you may have observed, the `javac` compiler sometimes rejects a Java program because it contains an *unreachable statement*. A statement is unreachable if control can *never* pass to the beginning of that statement in any run of the program (no matter what values are given as input, if the program requires any). Since unreachable statements cannot affect the behavior of the program, it is reasonable to treat their presence as indicating a mistake on the programmer's part. (It is not uncommon for compilers for other languages to flag unreachable statements with a warning; Java is somewhat unusual in making them hard errors.)

`javac` performs a static analysis to detect and flag unreachable statements. Like many other static analyses, this one calculates an *approximation* of the program's runtime behavior; that is, its prediction of the runtime behavior may be wrong in some cases. But the analysis only errs in one direction: it might fail to flag some statements that really cannot be reached, but it never flags a statement that might actually be reached during some program run. This means the compiler will never reject a program unless it is *certain* that it contains one or more unreachable statements; this behavior is appropriate because unreachability just indicates a likely mistake, not a fatal problem, so we don't want programs that only *might* contain unreachable statements to be rejected.

For this part of the assignment, you will write an unreachability analysis for the miniJava language. You will implement the analysis by modifying the provided `Ast0.java` file to add a new instance method, `boolean checkReach(boolean reachable)`, to every `Stmt` class. Name your modified program `Ast.java`.

The input parameter `reachable` indicates whether the current statement is reachable; and the return value indicates whether the next statement is reachable. (*E.g.*, the `checkReach()` method for the `Return` class should return a `false`, since a statement following a `return` statement is not reachable.) The top-level entry-point to this analysis is a new method `void checkReach()` in the `Program` class.

When an unreachable statement is detected, your analyzer should throw a `StaticError` exception (which is defined in `Ast0.java`). For example, with the following AST as input (assume it is in `tst/test.ast`):

```
# AST Program
ClassDecl test
  MethodDecl void main ()
   Return ()
   Print "hello"
```

your program should print out an error message:

```
linux> java StaticCheck tst/test.ast
StaticError: Unreachable Statement: Print "hello"
```

To help you get started, a very simple (and highly incorrect!) version of the `checkReach()` method is included in the provided `Ast0.java` file. The implementation of `checkReach()` given here traverses the AST and marks *every* statement as reachable. The point of this is just to illustrate what a simple tree traversal looks like in the AST context. You will need to change the behavior of this traversal to get the desired behavior for your analysis code.

For this analysis, make the simplifying assumption that the values of boolean expressions are always unknown, no matter how trivial they might be. For example, even in a statement like

```
if (true)
  x = 0;
else
  x = 1;
```

your analyzer should not try to figure out which branch of the `if` is taken (and hence will not be able to flag `x = 1` as unreachable). You should also assume that the first statement of every function is reachable. Otherwise, your analysis should be as precise as possible.

Task 2. Detecting Uninitialized Variables (6 points)

Another static analysis `javac` performs is to detect and flag *uninitialized variables*. A variable is uninitialized if it does not have a value at the time of a use. Again, the `javac` compiler computes an *approximation* of the program's runtime behavior. Only this time the analysis errs in the opposite direction: it might flag some variables that really are initialized, but it never misses a variable that is not initialized on some execution path. In other words, the compiler rejects a program unless it is certain that it doesn't contain any uninitialized variable.

In miniJava, variables can be classified into three categories, and their initialization situations are different.

- *Instance variables* — They are always initialized. Per Java's rule, if an instance variable is not explicitly initialized by the programmer, a default value will be provided at the time of its creation.
- *Method's parameters* — They are always initialized. In Java, method's parameters are passed by values. Thus, every parameter is guaranteed a value at the start of a method's invocation.
- *Local variables* — They are initialized either through initialization during declaration or through an assignment statement, both provided by the programmer. This is the only category of variables that uninitialized usage may occur.

For implementing this analysis, you will add a new method, `checkVarInit(VarSet)`, to every `Stmt` class *and* some `Exp` classes. The input parameter to this routine is a set of variables that are confirmed to have

been initialized. For the `Stmt` version, the method also returns a `VarSet` object, representing the updated version of the set (*e.g.* after an assignment, the set may be augmented with a new variable). For the `Exp` version, there is no need to have a return value. (Thus the method has a return type of `void`.) Not all `Exp` classes need to have the `checkVarInit()` method (*e.g.* the classes for the literals).

For this assignment, we assume that the input programs are free of other syntax or semantic errors. In particular, all variables are assumed to have been defined. With this assumption, if a variable is not in the `VarSet`, we can conclude that it is not initialized.

For this analysis, make the same simplifying assumption that the values of boolean expressions are always unknown, no matter how trivial they might be. Therefore, given the following code,

```
if (true)
  x = 0;
```

your analyzer should not conclude that the variable `x` is initialized after the `if` statement.

A class for representing immutable `String` sets is included in the program `Ast0.java`. It is defined as an extension to Java library's mutable `HashSet` class. A small library of three operations, `union`, `intersection`, and `add`, are provided.

Here are some hints for implementing this analysis:

- Establish a traversal framework for the method `checkVarInit()` across the AST node classes. For each node class, decide which components to invoke recursive calls.
- For the `ClassDecl` class, establish a `VarSet` to contain the class' field variables (if any). Get the traversal rolling by calling its `checkVarInit()` method with the `VarSet` as argument.
- For the `MethodDecl` class, add its parameters, as well as all local variables with initial values to the `VarSet`. Then continue the traversal.
- The main check action happens in the `Id` class. For an ID node representing a variable use, check if it is initialized using the `VarSet`. Note that not all ID nodes represent variable uses, *e.g.* in `x = 1`, `x` is not a variable use.

Testing

To test your analysis code, you will need to try it out on the ASTs of various miniJava programs. File `AstParser.jj` provides a parser for the AST dump format (`.ast`) files that you generated in Homework 3. It can be compiled using `javacc` in the usual way. The top-level driver program `StaticCheck.java` operates by reading in the `.ast` file specified on the command line, thus generating an internal `Ast` tree; invoking `checkReach()` and `checkVarInit()` on the root of that tree. If `StaticCheck` does not detect any unreachable statements or uninitialized variables, it ends silently with no output; otherwise, it shows a proper error message.

A small set of test inputs are provided in the `tst` subdirectory, each contains an unreachable statement or an uninitialized variable. Note that these tests cover only a small set of possible cases, especially with respect to the uninitialized variable analysis. You should construct more test inputs of your own, or you'll risk failing our more comprehensive tests during grading.

To obtain additional `.ast` files for testing, you can either run your own parser from Homework 3, or else write programs directly in the `.ast` dump format. The latter is not at all difficult to do, especially if you use an existing `.ast` file as a template; many such files can be found in the `tst` subdirectory of Homework 3. It is permissible to share test files with other students.

The provided `run` script can be used to conduct your test:

```
linux> ./run StaticCheck tst/*.ast
```

Note that if you comment out the following line in `Ast0.java`:

```
abstract VarSet checkVarInit(VarSet initSet) throws Exception;
```

then the program will compile and run. But it won't detect any of the error cases.

Requirements and Grading

In grading this problem, primary emphasis will be placed on behavioral correctness of your code, *i.e.*, how well it does on the tests we devise. However, if you wish to obtain partial credit for less-than-perfect solutions, you should try to make your code as intelligible as possible, and to include comments that explain your overall strategy.

The minimum requirement is that your program runs and detects at least one error case.

What to Turn in

Submit a single file, your version of `Ast.java`, through the "Dropbox" on the D2L class website.