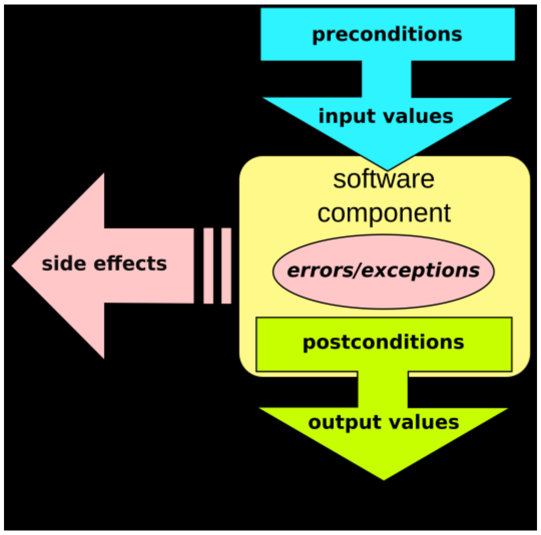
Postcondition Assertion Checker for Java

Proposed By: Hans Jorgensen(thehans) and Jacob Chong(hindelc)
Date: March 28th, 2018

Problem Overview



In procedural programming languages, most procedures have both *preconditions* - requirements of the program state that must be true when the procedure is entered - and *postconditions* - requirements that are guaranteed to be true when the procedure exits. Preconditions usually manifest as conditions on the arguments, and postconditions usually manifest as conditions on the return values or on any exceptions thrown. In object oriented programming languages, most classes also have *class invariants*, or conditions that must be true of the object's state both before and after each of its methods. This notion is known as *design by contract*, and is illustrated by the diagram at right.

The Java programming language, which is both procedural and object oriented, has an assert functionality that can be used to check these conditions, but because it is at the statement level only, it is difficult to check postconditions and class invariants for all possible exit cases. The try-finally statement can be employed for this purpose, but it must be deliberately inserted by the programmer and introduces needless overhead that is still present even if assertions are disabled.

Proposed Project

Our proposed project is to research and implement solutions to provide a better postcondition and class invariant checker in Java. The goal is to provide a feature that:

- Is easy for programmers to use
- Requires minimal additional configuration
- Can be disabled with the assertion system at runtime, with minimal performance impact.
- (Preferably) Maintains JVM compatibility (official Java runtimes can still run the resulting code)

- (Preferably) Maintains source compatibility (old compilers can compile the new source and vice versa)

We plan to modify the Open JDK 10, available via the GPLv2 on http://openjdk.java.net/, to implement our solution. We will explore parsing, bytecode generation, and other parts of the compilation and execution process to implement this assertion checker. We will primarily focus on a postcondition assertion checker, since class invariants can be expressed as postconditions, and will incorporate features specific to class invariant checking as we have time, such as possible dataflow analysis to only check the parts of the class that have changed.

Our hope for this project is that, in providing a better postcondition assertion checker, more programmers will be inclined to include assertions for postconditions and class invariants in their work, and programs will thus be tested more thoroughly during their development cycles.

Risk Analysis

As this is primarily a research project, we cannot predict the solutions that we will eventually discover, if any. It is extremely likely that we will be able to invent a solution that checks postconditions and class invariants at all procedure exit points (since such code can be written in Java source today with try-finally statements), but the goal of the project will be to discover what can be done with the Java ecosystem to implement this solution, and it is not guaranteed that our constraints will allow us to progress very far beyond this point.

Our primarily bottleneck will be working with the OpenJDK codebase, if annotations do not provide a sufficient framework for any reasonable solutions. The OpenJDK is written in C and has been in development for over 10 years, so it is likely that we will be dealing with some legacy code elements, including poor design decisions or bugs in the implementation.

The OpenJDK repository is also maintained with Mercurial, which is a DVCS like Git but has some functional differences that we will have to account for. If we use Git as our primary version control system (such as to interface with Gitlab), it may be more difficult for us to fetch any upstream work done on the OpenJDK itself in order to get new features or bug fixes. If we use Mercurial, we will have to figure out an alternate coordination solution as well as familiarize ourselves with Mercurial if we are not already.

While maintaining source compatibility is a goal of ours, there is a high chance that we will have to break it if our solution requires us to add an element to the language. For instance, if we add a new keyword to the language to implement this change (such as a “safereturn” keyword), any program that uses that keyword as an identifier cannot be used with the new version of the parser that we create. As stated above, we hope to maintain complete compatibility with existing Java code if possible, and it remains very high priority to continue to generate class files that can be run by existing JDKs.