Christopher Addison - cjteam

Conner Knight - connerk

Pluggable Typing In Java For The Masses

Motivation

We hope to apply the same type of pluggable type system used in IDE's like Android Studio and put them into Java to allow constraints like @nonnull to function arguments and return values. This will allow clients to provide more constraints on their input and output, which will be extremely helpful in eliminating programs with bugs relating to not expecting a certain output or input like null, and eliminate the need for the client to have to check for null in every single public function they write. A pluggable type system is a type system that extends or adds to separately an existing type system in a language. Pluggable types are interesting as they expose a metaprogramming interface to an existing programming language allowing for existing ones to gain new functionality. In this case it allows us to increase the effectiveness of static analysis of a given program. Currently, Android Studio supports pluggable typing in the signatures of methods to allow for static and thereby compile-time checks to avoid some costly dynamic checks and to strengthen the contract of the method in question. A useful side effect of these extra types is improved interoperability with functional languages that use the JVM such as Kotlin and Scala which have explicit nullability built into their type systems. By adding this functionality, we can bring pluggable types to the wider Java world. We are solving the problem of Java not having a effective way to allow functions to make guarantees about what they receive and what they return, such as a guarantee a function won't return a null, or guarantees about the range that the return will be in. This seems like it would be a very helpful tool to anyone programming in Java since it would help get rid of pointless checks for corner cases in functions that were not intended to take in a certain input. This would also get rid of buggy programs with functions that don't expect certain values but allow those values to be passed in and cause errors, which would be eliminated with our proposal. Currently the closest solution people have for this problem is Java has a built-in @nullable , but it suffers since it only gives a warning, and still allows the program to compile, which does not allow the programmer to assume if they put on that tag that their program will not be run with null input. It also suffers when it comes to normal functions calling non-nullable functions with null inputs not even showing a warning. We hope to make ours much more complete than this.

Approach

The Checker Framework is a static analysis tool that works for Java. It is built around a concept of compiler plugins that each add some specific static analysis. The tool is open source and there are currently 23 first-party plugins and a larger number of third-party plugins. Java 8 adds pluggable types via tags and leaves hooks open for frameworks to use for static analysis. Checker Framework uses that and extends the capability back to Java 4. We will write a plugin in Java for Checker Framework that handles the same types that Android Studio provides. The

types we intend to add are @NonNull, @Nullable, @IntRange, @FloatRange. When one of the conditions is violated we will log an error, which will prevent the program from finishing to compile. The Java compiler will display the error as a native Java compilation error to the user. The limitations to this approach, if we want to make sure all the guarantees are met, meaning we make our checker complete, means we can't have soundness, so it is possible we will reject perfectly good functions just because in our static analysis we could not be sure that null was not null if we gave that function the @NonNull tag.

Challenges and Risks

There are a few challenges while developing a plugin for the Checker Framework. The biggest one is in our ability to understand and create code that works on top of the framework. As is the nature with large pieces of software, the documentation is far from perfect and will present a continuing challenge throughout the project. This will primarily slow down the process of writing our plugin and will increase the time of debugging. The real risk is in our ability to produce a fully functional product within the quarter and the possibility of feature creep. To minimize this risk, we intend to actively work on the project, explore the workings of the existing first-party plugins, and make use of the mailing list when we encounter an issue we cannot resolve. Another challenge will be to ensure that we cover all possible cases of handling possible ways that a nonnull value or value in a range we don't want to return can occur. We hopefully would want to make this complete, which means we will have to make sure there is no way for the user to possibly get around our @NonNull or @IntRange guarantees.