CSE 331 Software Design and Implementation

Handout T8: *Checker Framework for pluggable type-checking*

Contents:

Introduction

It is better to detect a bug at compile time than at run time. This handout describes the Checker Framework, which creates a type system that is stronger than Java's built-in type system. This helps you to make your code more robust, by preventing even more errors at compile time.

This document augments but does not replace the Checker Framework Manual. You should read chapters 1, 2, and 3. You can skim some parts of chapters 1 and 2 (for example, the installation instructions are only relevant if you want to work on your own computer), but read chapter 3 with more care. Also, skim chapters 18, 19, 20, and 24, and 25, so you know where to look if you have trouble. Do not stint on this step; it will save you time in the long run. Feel free to ask questions if you have any.

Eclipse setup

You may notice that the provided examples generate compiler errors in Eclipse but will compile properly with the Ant build file. This section describes the cause of the errors and how to fix them.

The first error you may notice is that "The import checkers cannot be resolved". This error occurs because Eclipse does not know where to look for the checker package, which is located at `/cse/courses/cse331/checkers/checkers.jar`. To eliminate this error, you should just remove the checker `import` statements from both example files. The Ant build file is configured to automatically perform these imports for you.

Next, you may notice errors that "NonNull cannot be resolved to a type". These errors

occur because the Eclipse compiler is not importing the package `checkers.nullness.quals.*` which contains the annotation definitions. To eliminate the error, you should enclose the nullness annotations in comments -- i.e., `/*@NonNull*/`.

Finally, you may notice that the nullness annotations on generic parameters were enclosed in comments while those on local variable types were not. This is because annotations on functions and variable types have been supported since Java 5, but annotations on generic parameters will not be supported until Java 8. The examples were designed to be backwards compatible.

Dealing with type-checker errors

If the type-checker issues an error, there are a number of possible reasons why.

- Your code has an error. Maybe your code can throw a null pointer exception, or maybe your annotations are wrong (you have too many or too few annotations, either at the location of the error or elsewhere).
- Your code is correct, but for a subtle reason that is beyond the ability of the type-checker to reason about. In this case, you need to suppress the warning.
- Your code is correct, but there is a bug in the type-checker. This is unlikely, but possible. Maybe same library method is incorrectly annotated, or maybe the checker has a different bug. In this case, first construct an argument that your code is correct and the type-checker is wrong. Then, report the problem to the course staff. In the meanwhile, you can suppress the warning (with an appropriate comment) so that you can continue to do work.

Suppressing warnings

Your goal is to obtain a compile-time guarantee of correct use of `null`. Therefore, you are strongly discouraged from suppressing warnings.

If you suppress 5 or more warnings, during the entire quarter, you are doing something seriously wrong and should talk to a staff member. It is perfectly reasonable to complete the term without suppressing any warnings, and that should be your goal.

For each suppressed warning, write a Java comment with a single sentence explaining why the `@SuppressWarnings` annotation or `assert` statement is correct. In other words, explain why adding the annotation/assertion is guaranteed to be correct and does not compromise the guarantee that your program never throws a NullPointerException.

Rather than suppressing a warning, it is usually better to refactor your code so that the checker issues no warnings. That usually improves the design: if the correctness of your code is so obvious that it is apparent to the checker, then it will also be more apparent to other programmers!

If you have to suppress a warning, try to suppress it as early as possible, such as at at earlier line of a method rather than a later one, if you have a choice. Placing the suppression earlier generally leads to clearer code.

The Checker Framework Manual states a number of ways to suppress warnings. You should only use `@SuppressWarnings` annotation or the `assert` statement, preferring to use the annotation when possible. Each such annotation should be on a single variable declaration (not on an entire method or class). For example:

```
// No @SuppressWarnings("nullness") annotation here on the class!
class MyClass {

  // No @SuppressWarnings("nullness") annotation here on the method!
  void myMethod() {
    ...
    // The expression is never null because ...
    @SuppressWarnings("nullness")
    /*@NonNull*/ Object myVar = expression;
    ...
  }

  ...
}
```

Do not circumvent the type-checker

Do not circumvent the type-checker. One example would be replacing code such as

```
  x.method();
```

with

```
  if (x == null)
    throw new Error("x is null");
  x.method();
```

That replacement may quiet the compiler warnings, but it does does no good in terms of guaranteeing that your program does not crash: the original version crashes if `x` is `null`, and so does the modified version. Instead, figure out how to ensure or prove that `x` can never be null.

A reasonable rule of thumb is: don't add a test that cannot fail at run time, just to eliminate type-checker warnings. If a programmer sees a test in your code, the programmer will assume that the test can evaluate to either true or false, so the added code degrades the readability of your code.

When to use dynamic checks and @SuppressWarnings

Adding a dynamic check is OK as a temporary workaround while you are trying to do something more principled.

However, a general rule is: any "if" test that you write in your program should be a test that can return true, and can return false. Anyone reading your code will assume this to be true (why else would you write the test?). If you write a "fake" test that can only return one particular value at run time, then you have made the code more confusing than before. Also, you have no static guarantee that your reasoning is correct and that the test will really always return the value that you think it will. You have lost the static guarantee of correctness, which was the whole point of using the Nullness Checker in the first place!

In summary: writing a test whose outcome is a fixed value is a workaround, it degrades your code, and it fails to give any static guarantee, so the wrong thing might still happen at run time. It is not a good solution.

If you are sure that your containsKey test will return true, then you should think about how and why you know this, and express that as an informal argument. The argument probably takes the form of a description of data flow from one point at which a key is used at a map insertion, until a later point when the key is used for a lookup. Now, convert your informal argument into annotations, such as that a particular variable is a key for a given map.

When you have done this, then in many cases you will find that the Nullness Checker suddenly works, and you have a static compile-time guarantee of all the properties that you have written. In other words, you had written some annotations incorrectly, or you had forgotten some necessary annotation: it was necessary for your proof, but you had not written it in the code.

In some other cases, you may find that your code is complicated enough that the Nullness Checker is unable to prove a property/annotation that you have written. In such a situation, you may write a @SuppressWarnings annotation, along with a clear description of why you know the property to be true. Writing @SuppressWarnings in just the one place that the Nullness Checker is inadequate, rather than just at the point where you finally call get(), ensures that you still get guarantees in all the other parts of the program, and helps other people to understand your code.

Writing @SuppressWarnings is **OK** and **permitted**, so long as the reason for it is that either your property is too complex to be expressed for the checker, or the checker has a limitation/bug that prevents it from verifying the annotations you have written. I've seen student code this quarter that required @SuppressWarnings, and also much student code that did not require @SuppressWarnings. Before you add @SuppressWarnings, first make sure that you aren't just missing a way that the Nullness Checker would have worked. And, any time that your code is so complex that the Nullness Checker cannot figure it out, that's a good sign that maybe your code is too complex. Even if you choose not to change your code, please do think about alternate designs that you could have used instead, that would have been clearer both to the tool and to people. The Nullness Checker and similar tools can lead you to a better design.

Working at home

You may work on an instructional workstation such as attu, or at home.

Working at home takes a little bit more effort with the Checker Framework, because you will need to install the Checker Framework. The installation is a simple 3-step process that is documented in the Checker Framework Manual. After installing the Checker Framework, you will need to set the CHECKERS environment variable; see the above "Setup" instructions.

You may also install the Eclipse plug-in for the Checker Framework.

Additional Setup for Eclipse on Windows

In order to use the build file from Eclipse on Windows, you must set the `CHECKERS` and `PATH` variables within Eclipse:

a. Right click on `hw4/build.xml` and select "Run -> External Tools Configurations" from the context menu
b. Click on the "Environment" tab
c. Add a variable named `CHECKERS` with the location of wherever the Checker Framework checkers are installed on your computer. On the CSE Lab machines this is at `O:\cse\courses\cse331\checkers`.
d. Add a variable named `PATH` with the value `C:\Program Files\Java\jdk1.7.0_21\bin` (or wherever the JDK is installed on your computer)

e. Check that "Append environment to native environment" is selected

Hints

Many more hints appear in the Checker Framework Manual. One example is Section 2.4.4, How to get started annotating legacy code.

If you're still having trouble, the forum is a good place to look for help.

Write type annotations in comments - for example, always use `/*@Nullable*/`, not `@Nullable`. You do not need to, and should not, write declaration annotations, such as `@SuppressWarnings("nullness")`, in comments.

Run the checker frequently. The sooner you discover an error, the easier it is to fix. If you are not using the Eclipse plug-in, then compile by running the ant target from Eclipse. If you just use the built-in Eclipse compiler, you will *not* be notified of nullness errors.

Where to write annotations

You don't have to write very many annotations! The staff solution for HW2 contains only 4 annotations in total, and no warning suppressions. If you are writing many more annotations than that, then ask a staff member for help. Because `/*@NonNull*/` is the default, you will probably never write it. Because of the type-checker's built-in program analysis, you will rarely or never write annotations in method bodies - only on method signatures. One place you do need to write a `/*@Nullable*/` annotation is on the parameter to each `equals(Object)` method, like so: `equals(/*@Nullable*/ Object)`.

A trick is to search your source code for the string "`null`" (in documentation or source code), and to write `/*@Nullable*/` annotations on signatures accordingly. Remember, you usually don't have to write annotations within method bodies.

A good way to proceed is to express your rep invariant as annotations. That will help you ensure that your code does not violate the rep invariant, and a strong rep invariant will make it easier to ensure that no null pointer exception occurs. For example, suppose that you had a Graph whose representation is

```
    // maps from each node to all of its children
    Map<Node, Set<Node>> edges;
```

and your rep invariant is:

- Each key of `edges` is non-null.
- Each value (these are the Sets) of `edges` is non-null.
- Each element of the Sets is is non-null.
- Each element of each set is a key in `edges`.

Because `/*@NonNull*/` is the default, the following annotation can express this entire invariant! Then, the type-checker will warn if your code violates the rep invariant.

```
    Map<Node, Set</*@KeyFor("edges")*/ Node>> edges;
```

Likewise, you should express as many other invariants of your implementations as annotations. Whenever there is an invariant that you cannot express as an annotation, that is a good indication that you will probably need to suppress some false positive warnings.

You don't need to add any `import` statements for classes like `checkers.nullness.quals.Nullable`, because the Ant build file is configured to have the compiler automagically insert `import checkers.nullness.quals.*` for you. (See "Annotations in Comments" in the Checker Framework manual.)

You do **not** have to write annotations on the JDK. If you find that there is a missing or incorrect annotation on the JDK, then just suppress the related warning and inform the course staff.

Q & A

Q: When I write an annotation, Eclipse cannot find my classes, saying "MyClass cannot be resolved as type".

Go to Project -> Clean. In the "Clean" dialog make sure that "Start a build immediately" is checked and click "OK". If the "Start a build immediately" checkbox in the "Clean" dialog is missing, then uncheck Project -> Build Automatically, and then try again.

Q: I get a warning "No processor claimed any of these annotations"

This warning is innocuous. You can ignore it.

Q: When using ant, I get an error like "[jsr308.javac] javac: invalid flag: -J-Xbootclasspath/p: ..."

This error happens because the build system does not know which `javac` program to use. To fix the error, edit the `local.properties` file in the `cse331/src/` directory of your working copy. Change the line beginning with `javac.location=/...` to point to the location of your `javac`.

NOTE: this file is your your local machine configuration only. Do not commit `local.properties` to your SVN repository, because the settings are intended to be specific to one working copy.

Who made the Checker Framework?

One member of the Checker Framework team is Stephanie Dietzel (pictured), a former CSE 331 student. In May 2010, she had never heard of pluggable type-checking and began to use it in her assignments. She joined the research team, performed case studies, improved some type rules, and now she is a published author.

Another CSE 331 success story is Laure Thompson. After using Daikon in CSE 331, she joined that research team. She made Daikon perform better comparisons with method call results.

If you are a student who likes thinking about programs - and especially about programs that write programs, or programs that reason about programs - then you should also think about getting involved in undergraduate research. The Checker Framework and Daikon are exciting projects, but there are many other potential research projects led by Michael

Ernst, and even more elsewhere in the CSE department. Once you have completed CSE 331, you will be ready to contribute!

Back to the CSE 331 home page.
Shortcut to Checker Framework homepage.
For problems or questions regarding this page, contact: cse331-staff@cs.washington.edu.