Project Proposal

Team: Nullness_Lite

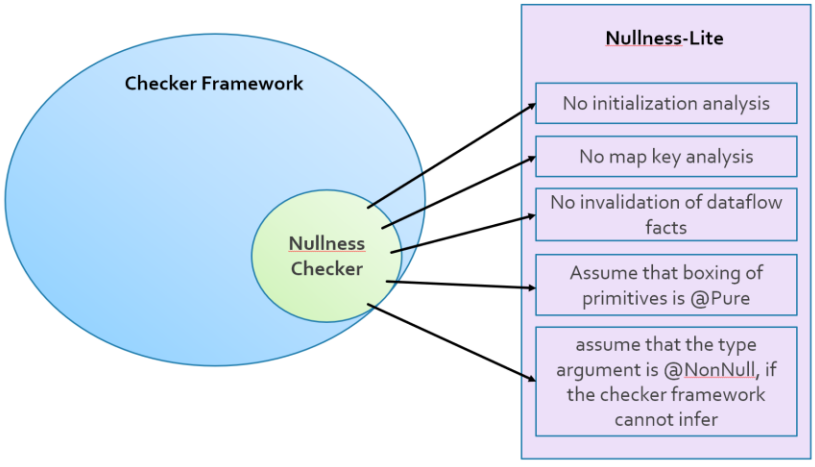
1. Introduction

Nullness is always an unignorable property when developing a software. In small-scale programs, since the number of variables is not large, developers might be able to avoid NullPointerException in most cases. However, as the scale of program increases, it is hard to keep track the possible state of each variable. As a result, NullPointerException continues to exist and often takes so much time for developers to figure out where the exception comes from. Due to fact that Java's type system is not strong enough to find the bugs causing NullPointerExceptions, the specific type checking systems are needed to find nullness bugs at compile time and thus to eliminate NullPointerExceptions at runtime.

2. Recent Solutions & Limitations

There are a few checkers which are built specifically to detect nullness bugs, and some of them are quite useful. For instance, one of the most powerful nullness bug checker, the Nullness Checker of Checker Framework, can detect 9 more true positives of nullness bugs than other general checkers. However, the balance between effective testing and expensive runtime is still hard to find. Nullness Checker contains confusing and expensive features (such as initialization analysis, map key analysis and so on) for full verification, but oftentime full verification is not the best choice for developers who wants to fast partial verification that is easy to start with.

Therefore, it motivates us to create such a faster and easy-to-use bug detector, Nullness_Lite, which is a improved version of Nullness Checker.



Nullness-Lite uses a powerful static analysis tool provided by Nullness Checker to detect nullness bugs and disables the expensive features we mentioned above. Also, Nullness_Lite Checker is a supplement to the original Nullness Checker. To make it work, developers only

have to add a few annotations to their programs in order to check NPE (Null Pointer Exceptions) statically. By introducing these new annotations, they will be able to detect some bugs in compile time without examining some time-consuming features involved in Nullness Checker.

3. Nullness-Lite's Benefits & Impacts

Before the idea of Nullness-Lite Checker, checkers developers can use are either short in function or hard to use. If Nullness-Lite, a pluggable type checker for nullness bugs in Java based on Nullness Checker, becomes realized, not only it enhances Java's built-in type system but also helps developers avoid reasoning out nullness bugs, which is tedious and error-prone especially when the pointer values are created by external components or are part of a chain of object references. Thus, all programmers including professional developers and students like us would benefit from it.

Since Nullness-Lite is based on Nullness Checker (in Checker Framework) which involves a lot of annotations and thus incurs a high cost in learning and building, it is appealing to more programmers due to its user-friendly property. Also, by introducing new annotations such as @Nullable and @NonNull, Nullness_Lite brings a change to Java programmers' coding style.

4. Risks & Challenges

Although Nullness-Lite is good alternative for avoiding NullPointerExceptions, it posts a challenge for us to build. As Checker Framework is a really complicated project with more than 200,000 lines of mere java codes, it would be quite hard for us to read through all documentations and codes to obtain a solid understanding in a short period of time. Therefore good time management to plan ahead plays an important role in our project success. To be specific, we are going to appropriately distributing time for reading the Checker Framework Manual, learning how the original version works, and editing for a light version. It is a good idea to look over other existing Nullness Checkers such as NullAway and FindBugs, for the sake of getting a basic sense of how our Nullness-Lite should look like or where we could improve on before we start to build it. Last but not least, we should work on eliminating as many errors and false warnings as possible.

5. Problems & Approaches

Since the main goal of our project is to build an improved version of Nullness Checker, a crucial part we have to analyze is the balance between features and runtime. For instance, if the scale of the program is not large, then it might be better to use the complete version of Nullness Checker. However, when dealing with large-scale programs, how many annotations do we require clients to add in our checker compared with annotations needed in the complete checker. How fast can our checker achieve and how many potential bugs can we find during that time? Are users willing to use this version instead of the complete version which requires longer annotation and runtime but might be able to find more bugs in one trial? In order to complete this analysis, we need to test our checker in difference scaled programs. We will

manually add annotations to the source code and record the time, and run the checker and record the number of bugs it finds. We will also test on the complete checker and compare the time they take and the bugs they find. For the problem of users, we could send out surveys to other students. The survey will contain the brief description of Nullness Checker, and the above test result on open source programs. We will ask them what version would they choose and why. We can also include the way to set up our checker if they are willing to test the checker on their own programs, and record their feedback to improve our implementation.

We estimate that it takes one week to read and analysis Nullness Checker, including its source code, manual, etc. Also, it takes around three weeks to implement our Nullness-Lite on the basis of Nullness Checker. Finally, it takes around two weeks to test Nullness-Lite and two weeks to systematically compare Nullness-Lite with other checkers, such as Nullness Checker and NullAway, and then evaluate Nullness-Lite and the future impact.

The “midterm” will be finishing up the implementation of Nullness-Lite. And the “final” will be completing a systematic report on the evaluation of Nullness-Lite and the comparison between Nullness-Lite and other checkers. Here is our week-by-week schedule:

Week 2 (4/4 - 4/11)	Set ups - Start reading Checker Framework Manual(Especially Nullness checker chapter) - Finish BUILD checker framework - Finish Eclipse/IntelliJ setup
Week 3 (4/11 - 4/18)	Preparation - Finish reading Checker Framework Manual - Begin understanding Checker Framework source code - Setup roles in group - Start implementing Nullness_Lite
Week 4 (4/18 - 4/25)	Implementation of Nullness_Lite - Begin disabling annotations - Begin making it independent of other type checkers-light
Week 5 (4/25 - 5/2)	Implementation of Nullness_Lite - Finish disabling annotations - Finish making it independent of other type checkers-light
Week 6 (5/2 - 5/9)	Testing - Correctness Test - Code Coverage Test
Week 7 (5/9 - 5/16)	Testing - Improvements according to correctness tests - User tests - User tests report for improvement
Week 8 (5/16 - 5/23)	Evaluation and Comparison - Improvements according to user tests - Write code examples - Evaluate on Nullness_Lite and original Nullness checker
Week 9 (5/23 - 5/30)	Evaluation and Comparison - More code examples

	- More comparison between Nullness_Lite and original Nullness checker(#annotations/100 lines, time to set up, time to check, easiness to use, ...) - Evaluation between Nullness_Lite and other current nullness tools(Nullaway, FindBugs, ...)
Week 10 (5/30 - 6/6)	Evaluation and Comparison - More improvements - Finish final reports

6. Experiments

To achieve our purpose of build a fast and easy-to-use checker that can be an alternative choice for Java developers to detect nullness bugs at compile time. It is crucial to convince them the benefits and drawbacks to use the new checker. Here is the measurement table for evaluating Nullness-Lite:

Checkers Name	Bugs Revealed (true positives/real bugs)	Bugs Not Revealed	Number of Annotation Used	Average Time to Check X Programs (X = 100)
Nullness-Lite				
Checker Framework				
NullAway				
Intellj				
Other Checkers				

According to the measurement table above, we will take three experiments to prove that Nullness-Lite is competitive alternative among checkers to detect nullness bugs.

6.1. Number of Annotations Used in Nullness-Lite

As one feature from Nullness Checker disabled, developers of Nullness-Lite will marked down the annotations used in that feature. The number of annotations used in Nullness-Lite will be the amount of original annotations of Nullness Checker subtracting the amount of abandoned annotations. We consider Nullness-Lite as improvement if it uses annotations as fewer as possible than Nullness Checker.

6.2. Nullness-Lite Is Fast

Since we disable the confusing and expensive features from Nullness Checker, Nullness-Lite is guaranteed to be at least faster than Nullness Checker.We expect it to be also

faster than other checkers. In order to verify that, we will do an experiment on Nullness-Lite and other checkers.

As the measurement table indicates, we will use a set of programs with nullness bugs as inputs to each of the checkers. The bugs will be various, covering all bugs detectable and undetectable by current checkers. The number of programs should be big enough to show the difference, so we take 100 as standard amount of programs.

Run each checker with the set of inputs 20 times (where first 5 times as warm-ups, excluded when compute the average time) and record the average time consumed for it to finish checking. Regardless of the fact that checkers detect different nullness bugs at this moment, we expect the Nullness-Lite to be good if it is not only faster than Nullness Checker and its time consumed is also within a reasonable variation range (5 sec) relative to that of other checkers.

Both 6.1 and 6.2 mainly verify the improvement made in Nullness-Lite relative to Nullness Checker, with the reference of other checkers for comparison. The next part, although illustrating the drawback of using Nullness-Lite, help developers make better decisions together with experiment in 6.1 and 6.2.

6.3. Nullness-Lite Covers Fewer True Positives

Removing features from Nullness Checker surely has an obvious drawback that Nullness-Lite unable to detect some bugs compare to Nullness Checker.

During the process of implementing Nullness-Lite, we will mark each disabled feature with its corresponding bugs. As we mentioned before, Nullness Checker can detect 9 more true positives than most of other checkers, we consider Nullness-Lite better than other checkers if Nullness-Lite is still able to detect more true positives than other checkers.

In conclusion, if the experiments together show that Nullness-Lite maintains a good balance between improvements (time and fewer annotations) and drawbacks (unsound), we can argue that it is a good alternative for users to choose when they require easy-to-use, less time-consuming tool for partial verification before they step into full verification.

Bibliography

- Dobolyi, Kinga. “Changing Java’s Semantics for Handling Null Pointer Exceptions.”
- Maciej Cielecki, Je ̨drzej Fulara, Krzysztof Jakubczyk, and Jancewicz. Propagation of JML non-null annotations in java programs. In *Principles and practice of programming in Java*, pages 135–140, 2006.
- Dietl, Werner, et al. “Building and Using Pluggable Type-Checkers.” Proceeding of the 33rd International Conference on Software Engineering - ICSE '11, 2011, doi:10.1145/1985793.1985889

Total Hours Spent: 12