Nullness_Lite Proposal

Team Members: Mengxing Chen, Anny Kong, Yuqi Huang, Xinrong Zhao

1. Introduction

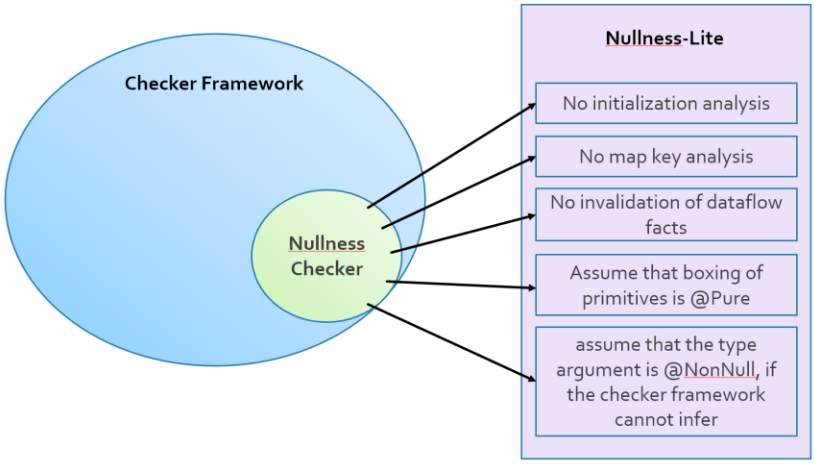
Null is so significant to the software development that almost all the programming languages are bonded with null. But null has been the cause for countless troubles in the history of software development. Tony Hoare, the inventor of null, calls it a synonym with “the billion-dollar mistake” [2]. More than that, since null is the main reason for the infamous null pointer error (called NullPointerException in java), there is hardly a programmer, who is not troubled by null. According to research, the null pointer error is reported as the most frequent bug [2]. And Professor John Sargeant from Manchester school of Computer science puts some remarks like this, “[o]f the things which can go wrong at runtime in Java programs, null pointer exceptions are by far the most common” [3].

Therefore, programmers start to avoid null. They try to use 0 instead of null, or return an empty string instead of null, for instance. As they seek for a world where no NPEs (NullPointerException) are raised, some nullness bug detectors are introduced, such as FindBugs, NullAway, and the Checker Framework [4], as a supplement of weak Java’s type system that does not support compile-time nullness checking. As we realized there are still not as many people around us using these checkers, there comes our idea: make it lighter and easier to use. As an initial step and evaluation of this idea, we propose to build on the Checker Framework, the one we found most powerful, so that a lite version of nullness checker which still does an excellent job in static analysis of nullness errors, could be used in a faster and easier way.

2. Recent Solutions & Limitations

There are several checkers which are built specifically to detect nullness bugs, and one of them, the Checker Framework was found to be especially successful in finding null pointer errors when an experiment of checking the Lookup program was carried out. In comparison, the Nullness Checker of the Checker Framework, detects all 9 true positives of nullness bugs while most other nullness tools missed all [4]. This becomes one of the most significant reason for us to choose it as our base. However, it also raises a problem in finding a balance between effective testing and expensive runtime. Although it guarantees the absence of errors, it as well reports 4 false warnings and requires 35 extra annotations written. As we look into details, Nullness Checker contains some confusing and expensive features (such as initialization analysis, map key analysis and etc.) 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.

Hence, it motivates us to build a lite version of nullness bug detector, Nullness_Lite, which removes some confusing features and will be faster and more usable for programmers.



(figure 1. Nullness_Lite Outline)

Referring to the diagram above, we will mainly disable three confusing analysis and remove two expensive features by assuming by default. As a result, Nullness_Lite Checker will be more practical and usable to programmers. To make it work, developers only have to add a few annotations to their programs in order to check NPEs 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 the Nullness Checker.

3. Nullness_Lite's Benefits & Impacts

Before the idea of Nullness_Lite Checker, checkers that developers can use are either short in function or hard to use. If Nullness_Lite 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 would benefit from it.

Since Nullness_Lite is based on Nullness Checker (in Checker Framework), on one hand, it is a type checker which is usable and accessible since type-checking is familiar to programmers. On the other hand, as the original one involves a lot of annotations and thus incurs a high cost in learning and building, the lite version requires less annotations and learning, will be 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 way for avoiding NPEs, it posts a challenge for us to build. As Checker Framework is a really complicated project with more than 200,000 non-comment, non-blank lines of Java codes as a test result of Cloc, 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 [4], 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, as we are going to disable some features, we should work on avoiding the increase in occurrences of false warnings and try to keep it still at a small percentage.

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, runtime and ease of use. 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, and 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 for better correctness tests
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. Architecture & Implementation Plan

Nullness_Lite, shown in figure 1, has 5 features disabled from Nullness Checker. These features may be related to each other. For example, they may share the same annotations, such as @EnsureNonNullIf. Thus, when we implement Nullness_Lite, we will first analyze how these features are used in Nullness_Lite, by reading the related codes. When disabling these features, we will pay attention to the features that are related.

Since our goal is to make Nullness_Lite an extended option for Nullness Checker of Checker Framework, we will keep Nullness Checker's architecture but add more sub-functions to realize the behaviors of Nullness_Lite. We will also change the control flow of the Nullness Checker. Thus, when Nullness_Lite is enabled, its corresponding behaviors will be invoked. For instance, Nullness Checker has three components: Nullness Checker proper, Initialization Checker and Map Key Checker, which are completely independent. We will add the new behaviors of Initialization and MapKey parts for Nullness_Lite. Then, when clients use Nullness_Lite, the functionality of Initialization Checker and Map Key Checker will be disabled,

because we will invoke the new behaviors instead of these checkers' original behaviors used for Nullness Checker.By adding new behaviors and control flows, we will not change the behaviors of Nullness Checker when Nullness_Lite is not used.

Nullness_Lite has a command line interface. To use Nullness_Lite checker, clients will add an command line argument "-NullnessLite" when using Nullness Checker. Here is an example running Nullness_Lite through javac:

```
D-108-179-139-147:checker-framework Anny$ javac -processor org.checkerframework.
checker.nullness.NullnessChecker -NullnessLite <your test file>
```

(figure 2. Nullness_Lite's Command Line Usage)

"-NullnessLite" option will only work for Nullness Checker. It will not be ignored if clients use other checkers from Checker Framework. Besides, all original commands in Checker Framework still work.

The main technology we will use is Checker Framework due to the fact we are extending its Nullness Checker. Checker Framework also provide us with the manual for developers, dataflow for strong analysis, and other mechanisms we might find useful.

7. Experiments

Our purpose is to building a fast and user-friendly checker that can be an alternative choice for Java developers to detect nullness bugs at compile time. And once we have built it, it is crucial for us to fully analyze our new checker by comparing it with the existing checkers in the market. By showing our analysis, we can attract customers. Our measurement for Nullness_Lite is shown below:

Checkers Name	Bugs Revealed (true positives/real bugs detected)	Bugs Not Revealed	Number of Annotation Used	Average Time to Check X Programs (X = 4)
Nullness_Lite				
Nullness Checker				
NullAway				
IntelliJ				
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.

7.1. Number of Annotations Used in Nullness_Lite

Developers of Nullness_Lite will mark the annotations used in each disabled feature as "abandoned". 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 improved if it uses as fewer annotations as possible than Nullness Checker.

7.2. 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 faster than other checkers, also. 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. We will use 4 programs with 10k+ lines of code as inputs.

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 7.1 and 7.2 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 7.1 and 7.2.

7.3. Nullness_Lite Covers Fewer True Positives

Removing features from Nullness Checker surely has an obvious drawback：

Nullness_Lite will detect less error than 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 (less time and fewer annotations) and drawbacks (unsoundness), we argue it is a good alternative for users to choose when they require easy-to-use, time-saving tool for partial verification before they step into full verification.

Bibliography

1. "Interesting Facts about Null in Java." GeeksforGeeks, 14 Sept. 2017, www.geeksforgeeks.org/interesting-facts-about-null-in-java/.
2. Neumanns, Christian. "Why We Should Love 'Null'." *Why We Should Love 'Null' - CodeProject*, 18 Nov. 2014, www.codeproject.com/Articles/787668/Why-We-Should-Love-null.
3. Sargeant, John. "Null Pointer Exceptions." *Null Pointer Exceptions*, www.cs.man.ac.uk/~johns/npe.html.
4. Dietl, Werner, and Michael Ernst. " Preventing Errors Before They Happen The Checker Framework." *Preventing Errors Before They Happen The Checker Framework*, www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=7&cad=rja&uact=8&ved=0ahUKEwiup6_V5bPaAhXKrFQKHW-gCxYQFghaMAY&url=https%3A%2F%2Fstatic.rainfocus.com%2Foracle%2Foow17%2Fsess%2F1492901668615001brln%2FPF%2F2017-10-02%2520CF%2520%40%2520JavaOne_1507012791774001WJ2t.pdf&usg=AOvVaw3mAtzExTzYm6gr3sCn0cXb.
5. Osman, Haidar, et al. "Tracking Null Checks in Open-Source Java Systems." *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016, doi:10.1109/saner.2016.57.
6. Admin. "Java NullPointerException - Reasons for Exception and How to Fix?" *The Java Programmer*, 5 June 2017, www.thejavaprogrammer.com/java-nullpointerexception/.
7. Dobolyi, Kinga. "Changing Java's Semantics for Handling Null Pointer Exceptions."
8. 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.
9. 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+6