Nullness_Lite Proposal

Team Members:
Mengxing Chen (chenm32)
Anny Kong (yk57)
Yuqi Huang (yh73)
Xinrong Zhao (zhaox29)

1. Introduction

Null is so significant to the software development that almost all the programming languages are bonded with null (allow programmers to write 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” [10]. 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. Professor John Sargeant from Manchester school of Computer science puts some remarks like this, “of 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[11], NullAway[12], 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 Nullness Checker of Checker Framework was found to be especially successful in finding null pointer errors when an experiment of checking the Lookup program was carried out. The Nullness Checker of Checker Framework detects all 9 true positives of nullness bugs while most other nullness tools missed all [4].

As a full verification tool, however, Nullness Checker also has a problem in raising many false positive warnings which can overwhelm the developers. To be specific, although Nullness Checker can guarantee the absence of errors, it as well reports 4 false warnings and requires 35 extra annotations written. Besides, as we look into details, Nullness Checker contains some confusing and expensive features for full verification (such as initialization analysis, map key

analysis, etc.), but oftentime full verification is not the best choice for developers who wants to do fast partial verification that is easy to start with.

Hence, the facts above motivate us to build a lite version of Nullness Checker: Nullness_Lite. Nullness_Lite is a nullness bug detector based on Nullness Checker but it disables a set of features in Nullness Checker. We choose Nullness Checker to be our base, because it has the powerful analysis mentioned above. Our hypothesis is that Nullness_Lite should be faster and more usable for programmers by deliberately giving up soundness to some extent. In other word, with the trade-off, it should produce fewer false positive warnings and have fewer annotations used.

Yet the project still have many questions remain to be answered. For example, to show Nullness_Lite has the desirable traits above, we need to evaluate our Nullness_Lite and many other nullness bug detectors. Besides, the features we choose to remove are not randomly chosen. We also need to prove that these features will make Nullness_Lite more competitive among other nullness bug detectors. We will address these questions with more details in section 5 & 6.

3. Nullness_Lite's Benefits & Impacts

Before the idea of Nullness_Lite, 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 Java programmers including professional developers and students would benefit from it.

Since Nullness_Lite is a lite version of the Nullness Checker of Checker Framework, it requires no learning or behavior changes for users who are already familiar with Checker Framework. For users new to either type checkers or Checker Framework, they will need to learn how to use Nullness_Lite from the user manual and possibly change their implementation behaviors, adding annotations such as @Nullable and @NonNull into their source code.

4. Risks & Challenges

This project posts a challenge for us to understand the Checker Framework. 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 lite version. It is a good idea to look over other existing nullness checkers such as NullAway [12] and FindBugs [11], 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.

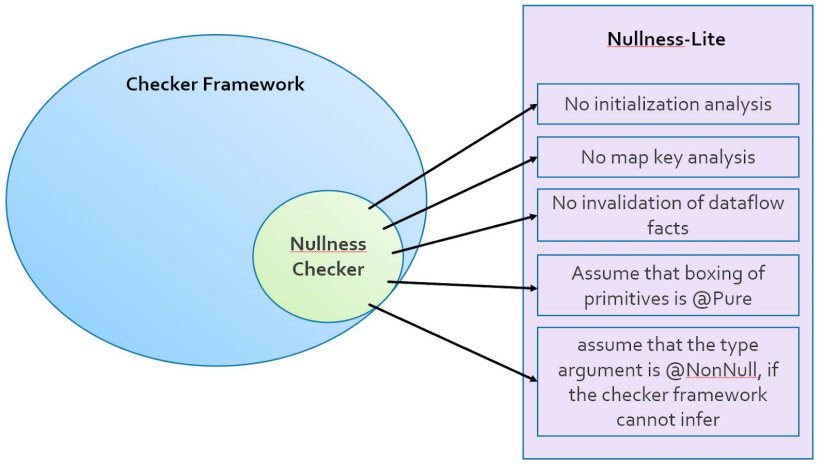
5. Problems & Approaches

5.1 Why Build Upon Existing Nullness Checker?

We choose to build Nullness_Lite on one of the existing nullness checkers, because the existing checkers that can detect nullness bugs such as FindBugs are popular and developers use them for years. Yet the limitations of current tool exist and developers always prefer a more powerful checker that requires less effort to learn. Thus, we realize there is a better design place in modifying an existing checker to reduce limitations and meet developers' needs.

And the reason why we choose Nullness_Lite to be a lite version of Nullness Checker is addressed in section 2.

5.2 Architecture & Implementation Plan



(figure 1. Nullness_Lite Outline)

The diagram above includes five features we choose intuitively to disable. These features are all included in Nullness Checker, but how do we know they are good choices? We cannot answer it without evaluation. Therefore, our plan is to evaluate Nullness Checker with each feature disabled respectively. If the evaluation shows a positive result when we disable some feature, then we will consider including it in Nullness_Lite. We will discuss more about how we determine whether or not an evaluation result is positive in section 6.

The brief implementation plan is to remove functions related to these features in the Nullness Checker under the fork of Checker Framework. Features are disabled separately for evaluation. After evaluation, we choose the features we want to keep and merge the implementations for different features. Then we evaluate Nullness_Lite again with these

features included, to see if Nullness_Lite will meet the expectation to be a competitive one among all other nullness bug detectors.

Then we will make Nullness_Lite an extended option for Nullness Checker. 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 MapKey Checker, which are completely independent. We may add the new behaviors of Initialization and MapKey parts for Nullness_Lite. Then, when Nullness_Lite is used, the functionality of Initialization Checker and Map Key Checker will be turned off, because we will invoke the new behaviors instead of these checkers' original behaviors used for Nullness Checker. By possibly adding new behaviors and control flows, we will not change the behaviors of Nullness Checker when Nullness_Lite is not used.

5.3 The Interface & Technologies

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

```
zhaox29@ubuntu:~/jsr308/checker-framework$ javac -processor nullness -ANullnessLite <MyFile.java>
```

(figure 2. Nullness_Lite's Command Line Usage)

Note that "-NullnessLite" option will only work for Nullness Checker. The behavior is undefined if the option is passed to other checkers. All original commands in Checker Framework will 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.

5.4 Implementation Details

The way we how we disable the features are addressed in section 5.2.

We have finished disabling Initialization Checker by adding a command line argument. Disabling MapKey Checker is almost done by commenting out all related parts. Now we are analyzing the tests results and begin disabling invalidation of dataflow.

6. Evaluation Plan & 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	Feature to Be Tested	Bugs Revealed	Bugs Not Revealed	False Positive Warnings	Number of Annotation Used	Average Time to Check X Programs (X = 4)
Nullness_Lite	Features we choose tested together					
	Default inference is @NonNull					
	No Initialization Checker					
	No Map Key Checker					
	No Invalidation of Dataflow					
	Assume boxing of primitives is @Pure					
NullAway						
FindBugs						
IntelliJ						
Eclipse						
Nullness Checker						
… (if time allows)						

(figure 3. Measurement Table)

6.1 Annotations Evaluation

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.

6.2 Speed Evaluation

As the measurement table indicates, we will use a set of programs with nullness bugs and annotations. The bugs should 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.

6.3 False Positives Evaluation

We will analyze the false positive warnings with the reports generated in 6.2, because we expect more false warnings will expose in larger programs. We will analyze the reports and check whether an error is a real bug. To distinguish the results of different checkers, we will consider manually add false positives to the source code if possible. Then we will record the score of each checkers in the measurement table.

6.4 True Positives Evaluation

We will mainly use the specification tests of Nullness Checker for evaluation in this part, because Nullness Checker is our base and has stronger analysis than other nullness checkers, which means its specification tests will cover more bugs that other nullness checkers cannot detect. We will run them on each checkers and record the number of bugs revealed and unrevealed in the measurement table.

6.5 Determine The Features to Include In Nullness_Lite

This part answer the question in section 6 that how to know a feature is good to be disabled. Remind the goal we mentioned earlier that Nullness_Lite should be faster with fewer false positives warnings and annotations, thus easier for developers to start with. Accordingly, it is considered to be good if disabling a feature results in the desirable traits above. Then, we will also care about the extent of unsoundness after disabling these features.

Since we care if a feature can reduce as many false positive warnings as possible, based on results, we will prioritize the features with an increasing order of false positive warnings and choose preferable features to include in Nullness_Lite. After including this features in Nullness_Lite, we will evaluate Nullness_Lite as a checker by experiments 6.1~6.4.

6.6 Determine The Quality of Nullness_Lite

Trading off soundness for speed and handiness is the intention of Nullness_Lite. Thus, Nullness_Lite can never be perfect. The measurement table will show its pros and cons to users, so they can decide whether or not to use Nullness_Lite.

7. Schedule

The “midterm” will be finishing up the implementation of Nullness_Lite. 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 - Evaluation preparation: familiar with other checkers and find tests
Week 5 (4/25 - 5/2)	Implementation of Nullness_Lite - Finish disabling annotations - Finish making it independent of other type checkers-light - Start evaluation of other current nullness tools(Nullaway, FindBugs, ...)
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 - Finish evaluation of other current nullness tools(Nullaway, FindBugs, ...)
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

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
10. Hoare, Tony. “Null References: The Billion Dollar Mistake.” *InfoQ*, Tony Hoare, 25 Aug. 2009, www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare.
11. Ayewah, Nathaniel, et al. “Using Static Analysis to Find Bugs.” *IEEE Software*, vol. 25, no. 5, 2008, pp. 22–29., doi:10.1109/ms.2008.130.
12. Sridharan, Manu. “Engineering NullAway, Uber's Open Source Tool for Detecting NullPointerExceptions on Android.” *Uber Engineering Blog*, 15 Feb. 2018, eng.uber.com/nullaway/.
13. Hovemeyer, David, et al. “Evaluating and Tuning a Static Analysis to Find Null Pointer Bugs.” *The 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering - PASTE '05*, 2005, doi:10.1145/1108792.1108798.

Total Hours Spent: 12 + 6 + 10