Reducing error prone code by analyzing immutability

Oskar Berg
dat15obe@student.lu.se

Ruolangxing Zhang
zhan0443@e.ntu.edu.sg

Abstract

Immutability is long known for its advantages of automatically being thread-safe and it works without synchronization issues. As we were very interested in this area and intend to reduce errors for programmers. We built a tool based on java compiler Extendj, so when compiling the program, the tool produces warnings for potentially wrong mutable fields, which includes every mutable parameter type as well as every static field that are mutable. To achieve this we implemented functions such as isMutable(), exposeReference() and so on. These are two typical cases that mutability could cause errors. It is a useful tool which can be further developed.

1. Introduction

A common source of bugs and security vulnerabilities is undesired modification in shared resources. In multi-threaded software this risk increases. Without proper synchronization there is high risk of race conditions and it can be difficult to achieve expected behavior. This can actually be reduced by introducing immutability. Immutable object means this field or class cannot be changed once created. For example, a string is naturally immutable(Koved et al. 2005). As when trying to change a string, you actually creates a new one, and the old one remains the same.

The advantage of immutability can be summarized into three parts: Firstly, as Immutable objects have only one state, once constructed properly, it is not possible to get into inconsistent states. Secondly, it allows safe and efficient sharing. Developers are able to share and cache references freely to immutable objects without copying or cloning them. Also their fields or the results of methods can be cached without the worries of values becoming stale or inconsistent with other objects' state. Thirdly, immutability ensures thread-safety. There is no need to synchronize access to immutable classes across threads, as they are thread-safe themselves. Furthermore, when immutability is reached, it introduces the option of using pure methods, which in turn can further improve reliability (Finifter et al. 2008).

Due to the great benefits of immutability, there are many languages which already started supporting partial enforcement of unchangeable structures (Coblenz et al. 2016). However, in many cases, there is no way to mark if a object is intended to be immutable. For existing language features supporting immutability, many developers often tend to misuse a lot of them as a result of complexity and unfamiliarity (Coblenz et al. 2017).

In this paper we investigate techniques of adding analysis to an extending compiler. Our aim is to inform the programmer of the potentially misused mutable data structures when our tool is run on code so that they can make changes in time to prevent data loss or security flaws due to mutability. To achieve this, we first implemented a mutability solver to classify classes as mutable. Then we can use this classification in situations where mutability is risky and supply a warning. The programmer can then improve security and reduce bug prone code by making the classes immutable or change the contexts in which they are used.

2. Motivating Example

In the course EDAN65 which leads up to this project, an instance of this issue presented itself. In that course, the library provided contained its own implementation of a list, separate from the list provided by Java's standard library. Due to not wanting to use two different lists, we decided to use the one provided in the course. This however caused a bug which took several hours to solve. It is because that this list is actually independent from the Java's list, which means they have some minor differences. So it is not intended to be used in the same way as the java's list. The thing is, in this list, When things are added or removed, these elements will be modified by the list. Because it was intended to be used in a node hierarchy, it changes the elements so their parent is the list itself.

The issue can be blamed on many different factors. But since we intend to do a program analyzer, the easiest way to find such issues is probably by analyzing when mutable classes are used. The issue could of course been prevented by providing more appropriate names or other cases of ensuring proper use of the library. But since it is a very complex task to analyze the intentions behind names, we did not find that appropriate.

3. Method

3.1 Choice of language

The language we chose to analyze is Java, which is based on the possibility of proving immutability in different languages. Since for languages such as c++ or python, it is impossible to determine whether variables and data can be changed from outer scopes, they were not contenders for the language to analyze. Java is also a object oriented language where true immutability cannot be achieved in most cases. With the use of the reflection API, it is possible to access data which otherwise would be protected, which violates the immutability property. However, since reflection is not used commonly and once it occurs developers would know the possible unforeseen bugs caused by it, in this project we consider these functions out of the scope of our tool.

In the implementation of this tool we chose to use a tool called JastAdd(Ekman and Hedin 2007), which allows us to use reference attribute grammar where we define properties for syntax tree nodes. These properties are then implemented using functions. However since our tool was built upon an existing Java compiler, ExtendJ(Öqvist 2018), instead of implementing the grammar, we added functionality by declaring aspects in the nodes where features are added. In this tool we heavily depended on transferring information up and down the syntax tree since the context of declarations and statements is used to derive useful information about classes. In Jastadd, we can use inherited properties for nodes and it is easy to derive the status of classes, which is why it is appropriate for this task.

3.2 Implementation of Properties

As our aim is to identify the risky mutable objects, in this tool We created warnings for the following two cases: The first is that there is a warning for every function which has parameter which is mutable. I.e. when a we call a function that might have a ArrayList as a parameter. We identify all the parameters used first then define a property ”isMutable” in order to test into them one by one. Another case is that we warn for static field that are mutable by creating functions isMutable() and isStatic() accordingly.

For function isMutable(), we first set some simple cases. For example, we define ArrayDecl.isMutable()= true. Then for classDecl, if its superclass or its bodydecl is mutable, then we return true.

Speaking of more complicated cases, For FieldDeclarator it's similar but more complicated. It involves enclosing class which has inner class. If a enclosing class it exposes reference, the fieldDeclaractor would be mutable. We implemented an property called ExposesReference, where when a reference is returned or otherwise leaked to an outside scope, exposeReference() will return true.

If inside of enclosing class, variable is changed, this fieldDecl is also mutable. This case is checked by function mutatesVariable(), once called, it searches layer by layer until it encounters the assign operator, it checks if the left part is the field. If it is, which means the object is changed, this class should be mutable.

4. Evaluation

As for evaluation, our main method is by the use of various test cases. Another usually suggested way of evaluating our tool is to run it on existing Java projects. However, as a result of absence of time, this was not really possible for us. So in our case, we did one test case larger than average with code extracted from the motivating example. The motivating example may not be optimal because it is partially generated code. But on the other hand, this code and methods will still be used by developers so the analysis is still relevant and meaningful.

There is another approach which can also be taken. Changing the language rules of java in order to further support the usage of immutable data structures. This is experimented with in (Zibin et al. 2007). However, our approach is favorable since it does not demand the developer to learn additional language features. This is very important when considering the usability of a tool. Hence code that has been produced while using our tool can easily be understood and used by others who don't, which is not the case when changing the language.

4.1 Test cases

The majority of our test cases are designed in order to test whether or not we can infer mutability properly. Our tool is built upon the Extendj compiler which is not written by us. There is a great risk of encountering bugs or unforeseen behavior since our additions may not use the original code as intended. We therefore try to create as many different test cases as possible, where we try to change the structure of the code to test for various situations. For example, we try to ran our tool on code which contains generics, nested classes, casts and many other cases.

Great amount of test cases is highly required since we noticed that even small changes to the sample code can change the outcome of our analysis. One such instance was the use of generic classes. The analyzer worked as intended on a class, but when a generic type was added to the class, the analysis would give a false negative. Even though the crucial parts of the class was unchanged. This is because as mentioned before, we make use of existing code, the analysis we built was analyzing ordinary classes and in the Extendj compiler generic classes creates an object which we had not attached our analyzer upon. Thus proving the importance detailed test cases.

4.2 Implementation

Since our tool was implemented on top of the ExtendJ compiler, it requires a thorough-out understanding of its implementation rules. In this progress, we encountered some issues in getting our tool behaving as intended. One of the main roadblocks was that there are many different but similar types of nodes in their node hierarchy system, so we had to understand fully and make use of the properties correctly, which sometimes could be very complicated. Take generic class and normal classes as an example, as they had separate nodes, when our tool worked already quite well on normal classes, it performed hardly any analysis on generic classes. To solve such problems, we created various testings and identified a number of common problems in running them, generally after solving these small cases one by one, we managed to get decent consistency in the whole project.

4.3 Correctness of tool

As discussed in the previous section, we can not guarantee the function of our tool on cases which are not covered by our tests. However, our tests does cover a wide variety of cases so most the tool should be able to perform correctly.

5. Related Work

Since the benefits of immutability has been known for long, it is no big surprise that there already exists similar tools which also analyze immutability.

One of them is Error prone(Aftandilian et al. 2012), which can find a few bugs related to mutability. This analyzer has the option to do data flow analysis in order to infer mutability. Their focus is however on using an annotation to mark classes that should be immutable and their tool will warn if this contract is not upheld. They do find a few bugs without relying on annotations. One is when using final and static variables, they are preferred to be immutable, a bug which we can also find with a bit of further work. The annotation method of assuring immutability has both upsides and downsides, the contract of immutability is better when in a multi-developer environment since it better communicates the intentions of objects. However, main function of bug detection tools are to warn for unforeseeable problems and requiring developers to use annotations will remove the possibility of warning users who are unfamiliar with the concept of immutability.

There is a well-known built-in analyzer in jet brains IDE called intellij(IntelliJ 2011), which is also able to do some minor analysis related to mutability. It can recommend programmers to use final variables or other minor changes to decrease mutation. However, it does lack the ability to analyze immutability to the same extent as we do.

6. Concluding Discussion

6.1 Usability

The final version of our tool is more a proof of concept than a production ready analyzer. However it does work and useful information can be obtained from it. But in order of it being considered to be used as a tool in real world applications it has to be integrated in the work flow of the production. The easiest way to achieve this is probably to integrate it in a existing analyzer used in production today.

This tool does not propose solutions to the warnings it produces. It relies on the user to have high level understanding of designing data structures. When it creates a warning the user has to understand that it should change it's data structures so they can no longer be mutated. This causes the tool to be ineffective if used by beginners and can become a obstacle since these warnings are not intended to teach the concepts of immutability in a efficient way.

Since errors related to mutability is often encountered when developing in teams that is the most appropriate use case for our tool. Using immutability is a great way to signal other developers what information is allowed to change where, and our tool can warn when developers releases data to be changed by others, so by using our tool they will receive warnings for those cases and can reconsidered if it should actually be restricted.

6.2 Conclusion

After spending hours and hours analyzing mutability we can conclude that it is a really important and potential subject. We are also very confident that it can be used to improve both code quality and reliability of projects. However, the complexity of the project was more than we had foreseen and due to the differences of previous coding experience between our members, a simpler subject as it is now is more suitable and achievable for us. In this whole process, we did some researching and also did a lot hands-on practice to implement the functions we plan to achieve. In the meanwhile, by writing the report along the process, not only our writing skills are practiced, but also we gradually became more and more clear about our aims and figured out possible methods to achieve them. Then After trying out, debugging and retry, we gained a deeper understanding of this area. So we felt that we both have learned much through it and it's great that we managed to produce this tool in the end.

6.3 Further work

As we learned more about immutability we also learned more about the benefits of it and are now more confident that it should be a core focus when designing projects. Thus we think it is a great area to do further work. The areas where more work is suitable are probably in how we can inform or enforce the user to use immutability. We choose to generate warnings for usages where bugs are likely to occur. But if the further development instead focuses on just reporting on what classes are and can be immutable it might open up the area to more interesting use cases. If we instead want to build upon our work by adding more cases where to warn for mutability a great area is the subject of concurrency, as mentioned in (Lea 2000) immutability is preferred in concurrent programs.

Acknowledgments

We thank Grel Hedin for assistance and guidance with our project.

We also acknowledge the developers of the JastAdd library and the ExtendJ library which without this project would not be possible.

References

Edward Aftandilian, Raluca Sauciuc, Siddharth Priya, and Sundaresan Krishnan. Building useful program analysis tools using an extensible java compiler. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, pages 14–23. IEEE, 2012.

M. Coblenz, W. Nelson, J. Aldrich, B. Myers, and J. Sunshine. Glacier: Transitive class immutability for java. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 496–506, May 2017. doi: 10.1109/ICSE.2017.52.

Michael Coblenz, Joshua Sunshine, Jonathan Aldrich, Brad Myers, Sam Weber, and Forrest Shull. Exploring language support for immutability. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 736–747, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3900-1. doi: 10.1145/2884781.2884798. URL `http://doi.acm.org/10.1145/2884781.2884798`.

Torbjörn Ekman and Görel Hedin. The jastadd extensible java compiler. *ACM Sigplan Notices*, 42(10):1–18, 2007.

Matthew Finifter, Adrian Mettler, Naveen Sastry, and David Wagner. Verifiable functional purity in java. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 161–174, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-810-7. doi: 10.1145/1455770.1455793. URL `http://doi.acm.org/10.1145/1455770.1455793`.

IDEA IntelliJ. the most intelligent java ide. *JetBrains[online].[cit. 2016-02-23]. Dostupné z: https://www. jetbrains. com/idea/# chooseYourEdition*, 2011.

Larry Koved, Bilha Mendelson, Sara Porat, and Marina Biberstein. Mutability analysis in java, August 2 2005. US Patent 6,925,638.

Douglas Lea. *Concurrent programming in Java: design principles and patterns*. Addison-Wesley Professional, 2000.

Jesper Öqvist. Extendj: extensible java compiler. In *2nd International Conference on Art, Science, and Engineering of Programming, Programming 2018*, pages 234–235. Association for Computing Machinery (ACM), 2018.

Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, et al. Object and reference immutability using java generics. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 75–84. ACM, 2007.