# Detecting Safe Transformation of Immutable Objects using Escape Analysis

Atulan Zaman
Electrical and Computer Engineering
University of Waterloo
a3zaman@uwaterloo.ca

***Abstract*—Immutable objects are preferred in Object Oriented Programming because of their safety and simplicity. However declaring all objects and data structures as immutable often causes heavy performance overhead for software programs. This is especially true for compilers where the entire Abstract Syntax Tree has to be rebuilt everytime the tree needs to be mutated. Therefore the work of this project looks at using static analysis to detect cases where objects can be safely transformed to be mutable from being immutable without compromising functionality and security. The basis of the analysis is using *Escape Analysis* to detect escaping objects using intra-procedural dataflow analysis and Object Representatives.**

CONTENTS



## I. INTRODUCTION

Object immutability is a feature of object oriented programming that restricts objects from being mutated after they are instantiated. This is generally a recommended practice because of simplicity and safety of programs to prevent bugs. Object immutability comes in different forms. One of the forms is variable immutability. In Java for example, this is enforced by declaring variables as `final` so that it can only be set during object construction and cannot be mutated afterwards using mutator method. Second form of immutability is reference immutability. This ensures that external methods referencing to a mutable parameter does not have permissions to mutate the parameter. There is much research aimed at enforcing this form immutability, most notably the work of Huang et. al [1] in creating a type system to ensure reference immutability when the referenced object is stored in a local variable. The work of Tschantz et. al on the tool *Javari* [2] is another research tool that helps programmers enforce reference immutability. At the class level, an object can be declared to be mutable if two design choices are maintained. Firstly, all the attributes of the class are declared to be immutable during declaration. Secondly all the mutator methods of the class retain the immutable nature of the object by creating a different instance of the object with the mutated attribute and returning that object instead of directly mutating the instance.

This is generally desirable because that causes methods sharing the object to not behave erroneously. Another application of this is in concurrent programming where threads sharing objects will have their own instances of the object. However, sometimes this causes overhead because everytime an object needs to be mutated, the a new instance has to be reconstructed from the existing instance. It is a standard safety vs performance tradeoff in software engineering. For large instance manipulation such as in manipulating abstract syntax tree construction in compilers this causes high memory and performance footprint during the construction process. Due to safety recommendations of object oriented programming, programmer usual choose to make object instances immutable to ensure safety of execution. However in cases of object sharing, infered using escape analysis [3][4][5] it is provable that program safety can be ensure without enforcing object immutability. The work of this project focuses on using intra-procedural escape analysis techniques [**?** ] to detect cases where it is safe make objects mutable without any side effects.

This techniques uses intra-procedural dataflow analysis using object representatives [6] detect locals and references that share a common heap pointer. Object representatives is a technique for detecting variables that share the same heap pointer, although they have different stack pointers, without doing inter-procedural analysis. After the object representative mapping is created using a may alias analysis, our escape analysis heuristics are used to analyze program points to detect locals and references that might have escaped. The intra-procedural analysis in this project is implemented using the Soot framework for Java bytecode analysis [7].

Much of program analysis research has been focused on detecting lack of immutability in program to detect potential sources of bugs and problems. Some work has been done on creating type systems to detect variables that behave like immutable although they are not declared as `final`[8]. This work in a way is going against the flow of traditional research

in the sense that it tries to escape from the paranoia of safety by trying to make immutable objects immutable.

## II. Example

In this section the motivating example of the work in explained. First a immutable class is presented, a case is shown when it is safe to transform the class to an immutable class. Following the positive example, a case is presented which explains when it is not possible to transform an immutable class to mutable. The point of the analysis is to recognize the cases when an object cannot be transformed to mutable due to the nature of its usage by the caller of the object.

Fig. 1: Immutable Object

```
public class ObjectImmutable {

    private final ImmutableList<String> tokens;

    public ObjectImmutable() {
        this.tokens = ImmutableList.of();
    }

    public ObjectImmutable(final ImmutableList<String> tokens) {
        this.tokens = tokens;
    }

    /**
     * Construct a new Object with an extra token.
     * @param token
     * @return
     */
    public ObjectImmutable append(final String token) {
        // don't mutate this
        // construct a new list and a new Object
        return new ObjectImmutable(tokens.append(token));
    }
```

Fig. 2: Client without object escaping

```
public final class ClientNoEscape {
    public static void main(final String[] args) { //TODO
        ObjectImmutable a = new ObjectImmutable(); // create a1
        a = a.append("x"); // create a2, discard a1
        a = a.append("y"); // create a3, discard a2
        a = a.append("z"); // create a4, discard a3
        System.out.println(a); // read a4
    }
}
```

In this example object shown in figure 1 it is evident that the object is immutable by the fact that the `token` attribute is *final* and that the mutator method, `append` returns an instance of the mutated object rather than mutating the subject instance and returning `void`. Now let's look at figure 2 which shows the instance of an client object calling the immutable object. In this method all that is happening is that the client is appending to the initially created instance and finally it prints the most recent instance to the console. In this case, since the initialized immutable object is not assigned to other globally accessible varible such as a `static` variable, or that it is not being passed as a parameter to another method, it is evident that within this caller, the object is not "escaping". The conditions for escape analysis that are used in this project are discussed later on in section III-C. Therefore if the this is the only client calling the the immutable object then it would be safe to transform the immutable object to an mutable object, because a reference of the object is not assigned to the heap before the end of execution of the method.

Fig. 3: Transformed Mutable Object

```
public final class ObjectMutable {
    private ImmutableList<String> tokens;

    public ObjectMutable() {
        this.tokens = ImmutableList.of();
    }

    public ObjectMutable(final ImmutableList<String> tokens) {
        this.tokens = tokens;
    }

    public void append(final String token) {
        // mutate our list of tokens
        tokens.add(token);
        // for client compatibility
    }
```

Fig. 4: Transformed Client Object

```
public final class ClientTransformed {
    public static void main(final String[] args) {
        ObjectMutable a = new ObjectMutable(); // create a1
        a.append("x"); // mutate a1
        a.append("y"); // mutate a1
        a.append("z"); // mutate a1
        System.out.println(a);
    }
}
```

A possible transformation of the immutable object can be viewed in figure 3. In figure 3 it is evident that that only difference the immutable object and this are three transformations:

- The local variable has been changed from `private final` to just `private`.
- The mutator method `append` has been changed to return a `void` type.
- The mutator method directly appends to the `ImmutableList`.

Because of the change in the object implementation. All the calling sites of the object also needs to be refactored in order to make this change effective. In figure 4 the transformed client is shown. The changes that happen to the client are the program points where the client calls the mutator method of the transformed mutable object.

Fig. 5: Client with Escaping Object

```
public final class ClientEscape{

    static Object escaper = null;

    static void main(final String[] args) {
        ObjectImmutable a = new ObjectImmutable(); // create a1
        escape(a); // save a1 for later
        a = a.append("x"); // create a2, DO NOT discard a1 (saved in escaper)
        a = a.append("y"); // create a3, discard a2
        a = a.append("z"); // create a4, discard a3
        System.out.println(a); // read a4
        System.out.println(escaper); // read a1
    }

    public static void escape(Object o) { //TODO
        // go stick this object in the heap somewhere
        escaper = o;
        return;
    }
```

Figure 5 presents an example of a client where an object is escaping the stacking and is creating a pointer to the heap. The points to note here is that at the second line of the metho, the instantiated Immutable object is being passed as a parameter to a method. Inside the `escape()` method, the immutable object is assigned to a static global variable. Since global variable can be accessed multiple methods, this causes the immutable object to have a reference to the heap which causes it to be globally accessible. This is a case when an object is said to have "escaped". Turning attention to the main method again, it can be noticed that at the end of the method, the method is using the assigned static global variable `escaper` which stores a reference in the heap for the first instance of the local variable `a`. Now since the local object was immutable in nature, the heap reference stores the original instance, and when the global variable is referenced at a subsequent program point it successfully returns the original instance of the object. However, if the local object was mutable instead, it would lose its original instance through the mutator methods between the program point where the object is escaping and when the escaped object is being referenced at a later program point. So an erroneous value for the local object would be returned. The objective of this analysis is to detect this use cases through escape analysis and detect which objects in a method can be transformed to mutable.

## III. Overview of Analysis

For the scope of ECE750, only the intra-procedural analysis of the complete program analysis is completed. This mean that, for this project, the functionality of the program is only limited to doing intra-procedural analysis on a given method to detect whether an immutable local variable or a Field Reference escapes at a certain program point in a method, and checks whether a variable with the same *Object Representative* is used at a later program point. If such a case does happen, then because of the usecase of the immutable object in the respective method, the object cannot be transformed to mutable. First this section explains the intra procedural dataflow analysis that happens in the program. Second it explains the reason for the usage of Object Representatives [6] in the scope of the data flow analysis. Lastly the escape analysis rules used in the project to detect object escaping is explained. The intra-procedural analysis is done using the dataflow analysis API provided by the Soot framework [7].

### A. DataFlowAnalysis

The dataflow analysis is a "may-alias" forward analysis, where each unit in the analysis graph stores a HashMap of the localvaribles and fieldreferences at the respective program point. Each of the design choices for the data flow analysis are explained below.

**Direction of Analysis:** The direction of the analysis is chosen to be forward analysis because in each step of the flow through function, the information about the variables in previous program points is of interest to subsequent program points. This is important because in the escape analysis step of the algorithm followed by the dataflow analysis step, the information about the object representatives of previous variables is used to determine whether the it is safe to make the object immutable or not.

**Flowthrough Reaching Condition:** The reaching condition for the flowthrough of information in a unit done according to the following rules. Each unit of the analysis graph stores a HashMap that stores an instance of the Local variable of a field reference along with its Object Representative that is generated along the flowthrough function.

**Algorithm 1** Flowthrough algorithm for dataflow analysis

```
Copy InSet to OutSet
if Stmt is DefinitionStmt then
    if rhs is LocalVariable then
        OutSet.put(lhs, rhs.ObjectRep)
    else
        generateObjectRep
        OutSet.put(lhs, new ObjectRep)
    end if
end if
```

**Merge Condition:** The merge condition for this analysis is done in the form of MayAnalysis where a variable attained different ObjectRepresentatives in different, both the object representative for the object are store with respect to the program point in which the merge is happening. This is necessary because if the object escapes within the branch, and in a subsequent program pointer if either of the ObjectRepresentatives aliases with one of the variables being used in the Stmt, then the object is still considered to be non-transformable to mutable because the used object "may" have escaped.

### B. Object Representatives

In this analysis Object Representatives need to be used because in this analysis objects aliasing with respect to their pointers to the heap are of interest. Generally in order to infer object reference to the heap, an inter-procedural analysis is necessary. However object representatives are a technique

of dereferencing heap pointers to local objects in an intra-procedural analysis [6].

This is evident in our analysis of the client with the escaping object shown previously in figure 5. In that example, the escaped instance of `a` would have the same object representative as the field `escaper` object that is called later in the program although they were assigned to the heap in a seperate method from the subject method. If instead, na integer value assignment for the object instance and the field was used during the dataflow analysis, then this analysis would no be possible because they would have different value assignments. Without dereferencing pointers to the heap there is no way for programs to do this aliasing without using object representatives.

### C. EscapeAnalysis

In this program, escape analysis is used to detect objects that are escaping the stack to heap, which causes objects to be globally accessible and hence prevent tranformation. There has been much work centered around escape analysis in the last decade, most of which has been towards its application in creating thread safe programs and optimizing multithreaded programming. The papers by Choi et al. and Vivien et. al [4][5] were some of the first papers to introduce the escape analysis in the context of inter-procedural analysis. Both the papers were focused on the application of escape analysis to minimize object allocation to the heap with the objective of reducing sychronization operation for threads.

Surprisingly, escape analysis is seldom used in the context of intraprocedural analysis. The only work that relates to this project with respect to escape analysis is its application in dynamic compilation and deoptimization [9] which uses escape analysis both in the context of intraprocedural analysis interprocedural analysis. Much of the intraprocedural inference techniques used in this work has been inspired by the works presented there.

There are many programming use cases which causes local objects inside methods to escape the stack and get allocated to the heap, however in the scope of this research only some rules apply to the analysis. The applicable rules of escape analysis that apply in this context are strictly those related to object instance sharing and accessibility. According to [9] there are two categories of object escaping: *global escape* & *method escape*. In the context of this research on the global escape of objects is relevant. Objects that suffer from method escape are also called *thread-local* objects which means that the object has escaped the context of the method, but is local with respect to the thread running the method. Since the focus of this analysis is not to optimize synchronization calls with respect to threading, this type of object escape not relavant for this analysis.

The cases of *global escape* relevant to this research is listed below:

`T.sf = a` : A local variable escapes its local allocation on the stack to the heap when it is assigned to a static field of the class. Being assigned to the static field of a class allow the object instance to be referenced by multiple methods and that is causes the object to escape. In terms of static analysis, this can checked trivially by checking that the left operand is a member of the field deference list and checking that the right operand is a local variable, in which case the assignee is the variable escaping ie. `a`.

`p.f a` : When local variable is assigned to the field of another local variable, the assigned local variable is defined as escaped. The reason for this is that, if the assignee is immutable, and it is assigned to the field of and it assigned to field of another local variable, any changes to the field of the local variable will cause the assigned variable to be changed as well. This is shown in the sample code segment in figure 6.

Fig. 6: Object escape with local field assignment

```
public static void main(String[] args) {
    T t1 = new T(1);
    System.out.println("This is t1 first: "+t1.getb( ));
    R t2 = new R();
    t2.a = t1;
    t2.a.set(3);
    System.out.println("This is t2: "+t2.a.getb());
    System.out.println("This is t1 after mutating t2: "+t1.getb());
}
```

```
This is t1 first: 1
This is t2: 3
This is t1 after which was changed by t2!!: 3
```

In figure 6 two local variable objects `t1` and `t2` are created, where `t2` owns an attribute with the same type as `t1`. `t1` is assigned to the field `a` of t2 and followed by that the field of `t2` is mutated. After the mutation, it is seen from the output that due to the change in the field of `t2` the original variable also gets changed. It is difficult for to make this aliasing analysis in the dataflow analysis because the dataflow analysis only store the object representatives of the local variables, but it does not store the object representatives of the fields of the local variables. Therefore, it is conservatively assumed in this analysis that whenever a local is assigned to the field of another variable, the assignee is marked as "escaped" irrespective of whether the field of the local variable actually gets mutated or not. This could have been avoided if the implementation of the type of `t1` was immutable, so that the `set()` method of object returned a new instance of the object, rather mutating the existing attribute.

`foo(a)` : A local variable is declared as escaped when it is passed as a parameter to another method. The reason for this is that the calling method could possibly mutate the object. From just intra procedural analysis it is impossible to know whether the calling method will actually mutate the object or not, and therefore for this analysis it is conservatively assumed that when an object is passed as a parameter to another method it has escaped the stack and is therefore deemed as escaped. Ideally it would be possible to check the operations done the passed in variable through an interprocedural analysis where one can transitively analyze whether any of the pointers of the object are being mutated by the calling methods. However, the inter-procedure analysis is beyond the scope of this project

and is therefore a possible candidate for future work. This case of *global escape* is being exhibitted in figure 5 in section II. In that example, the local variable was being passed as a parameter to the method `escape()` and therefore at that program point that specific instace of the variable `a` is flagged as "escaped".

## IV. Implementation and Evaluation

The implementation of the analysis was completely done with the Soot framework[7] for Java bytecode analysis. Below in figure 2 the wrapper algorithm that uses the escaped analysis and the result of the dataflow analysis is presented.

**Algorithm 2** Wrapper algorithm for detect mutable safety

```
Do dataflow analysis
for Each node n in the dataflow graph do
    rhs = Value for Stmt.rhs
    if rhs is escaping in n then
        flag rhs as "escaped"
    end if
    rhsObjRep = Object Representative for rhs
    if rhsObjRep equals Object Representative for existing
local in the Map then
        if Any of the aliasing locals are flagged as escaped
then
            Add the aliasing tuple to a flagged list
        end if
    end if
end for
Filter the locals from the tuple set
Flag the locals as unsafe to transform
```

The analysis below returns a list of tuples from the intra-procedure and escape analysis of a method. The each tuple in the list indicate a set of local variables that have the same object representative pointer to the heap, with atleast one of them escaping and being read by the method at a subsequent program point. The tuples contain not just the locals and fields of the method but also many temporary variables that are introduced by Soot's intermediate form for bytecode analysis. Therefore all the temporary variables are filtered from the tuples sets, and all the Locals from the tuple set are stored in a separate list and this elements of this list indicate which varaibles in this method are unsafe to be transformed to mutable from their immutable declaration.

The result of the analysis by soot returns a list of variables for each method of the analyzed class that are unsafe to be transformed to mutable. In the bigger scheme of things, what we really interested in is which variables are not in that list because we care more about the variables that can be transformed. For now the analysis has only been tested on toy test cases created during the development of the analysis technique and it has not been run on widely used Java library classes. The limitations of this analysis is that it only addresses the core analysis behind detecting when it is safe to transform immutable object using escape analysis. However for the analysis to be useful, it first needs to infer which objects are immutable to begin with, because we don't care about the variables that are already mutable. Existing research, for example the work of C.Unkel et. al [8] can be used to inference objects with immutable declarations. This would be used to further filter from the list of flagged variables that is returned by this analysis, because currently is returns the list of all variable in a method that are escaping and being read afterwards, whether they are mutable or immutable.

The current analysis returns correct results within the use case domain explored during the development of the analysis. However, the analysis must run on actual useful code in order to validate its utilty.

## V. Future Work

Currently the analysis is strictly intra-procedural and returns the variables that are unsafe to transform within the scope of the methods inside the class. However, for the analysis to be scalable a inter-procedural analysis needs to take place prior to the intra-procedural analysis to generate a call graph for a certain object that is of interest, and then analyze the methods that define the call sites using the analysis defined in this project. This way, the user can test specific immutable objects whether it is safe to transform them to mutable and the interprocedural analysis will transitively analyse the calling methods of the object to make the inference.

After the inference is made whether an object can be made mutable or not, certain program transformation needs to take place to make this change happen. Broadly are two abstraction in which the transformations will take place. One of them is the object itself, where the object attribute declarations need to be changed to mutable rather than immutable. Also in object, the mutator methods need to be change mutate the instance itself rather than returning a new instance of the object. The abstraction where the transformation needs to take place are the call sites of the respective object. The call site need to be transformed to reflect the changes that have been made to the object as a result of the analysis. For this to happen certain formal transformation rules need to be defined, and this in itself would have its own program analysis.

After the interprocedural analysis and the transformation rules are finalized, the program analysis will finally be ready testing on production level code. In order to instrument the contribution and utility of this analysis there are two metrics that would be interesting to consider after it has been run through benchmark programs:

1) Finding what percentage of immutable object declarations actually qualify as "safe to transform" according to this analysis
2) After the transformation, what is the performance and memory usage improvement for the subject programs.

## VI. Conclusion and Project Experience

The work of this project has the potential to improve program performance by minimizing redundant object reconstruc-

tion overhead for data structures. The intra-procedural analysis presented in this project is the core of the analysis, however there is much room for work to analyze the effectiveness of this analysis and transformation technique. The static program analysis concepts relevant to this project are intra-procedural dataflow analysis, escape analysis, object representatives and the exploring the Soot framework. One of the main challenges of this project was incorporating the inter-procedural analysis needed for dereferencing object pointers to the heap that was essential for this analysis. The usage object representatives were the solution to that problem although it was not an obvious veness of this analysis and transformation technique. The static program analysis concepts relevant solution because purely intra-procedural analysis do not always care about heap references. Secondly the exploration of escape analysis was another major checkpoint in the course of the project because it incorporates some of the key elements of the analysis. However it was fun exploring how escape analysis could be incorporated in the context of this intra-procedural analysis such as this project, because from most paper review about escape analysis it was evident it was most commonly used for inter-procedural analysis and thread safety optimization domains. Lastly and most importantly, the learning curve for Soot was both challenging and rewarding. "The Soot Survival Guide" [10] by Einarsson and Nielsen was a life saver in the initial stages as well as the blogs from Eric Bodden. It was my first time using a static analysis framework, therefore learning the datflow analysi framework and studying the API library of Soot for hours to find the right commands was great learning experience. It was rewarding because experience in using a robust static analysis tool is great to have and also studying the Soot API helped me understand some of the general features of modern program analysis techniques. Assistance from Derek Rayside was also notable towards progress during the project thanks to his regular constructive feedback and help in forming the key motivational example for this analysis.

## REFERENCES


[1] W. Huang, A. Milanova, W. Dietl, and M. D. Ernst, "Reim & reiminfer: checking and inference of reference immutability and method purity," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, ser. OOPSLA '12. New York, NY, USA: ACM, 2012, pp. 879–896. [Online]. Available: http://doi.acm.org/10.1145/2384616.2384680

[2] M. S. Tschantz and M. D. Ernst, "Javari: adding reference immutability to java," in *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 211–230. [Online]. Available: http://doi.acm.org/10.1145/1094811.1094828

[3] J. Whaley and M. Rinard, "Compositional pointer and escape analysis for java programs," in *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '99. New York, NY, USA: ACM, 1999, pp. 187–206. [Online]. Available: http://doi.acm.org/10.1145/320384.320400

[4] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff, "Escape analysis for java," *SIGPLAN Not.*, vol. 34, no. 10, pp. 1–19, Oct. 1999. [Online]. Available: http://doi.acm.org/10.1145/320385.320386

[5] F. Vivien and M. Rinard, "Incrementalized pointer and escape analysis," *SIGPLAN Not.*, vol. 36, no. 5, pp. 35–46, May 2001. [Online]. Available: http://doi.acm.org/10.1145/381694.378804

[6] E. Bodden, P. Lam, and L. Hendren, "Object representatives: a uniform abstraction for pointer information," in *Proceedings of the 2008 international conference on Visions of Computer Science: BCS International Academic Conference*, ser. VoCS'08. Swinton, UK, UK: British Computer Society, 2008, pp. 391–405. [Online]. Available: http://dl.acm.org/citation.cfm?id=2227536.2227569

[7] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, ser. CASCON '99. IBM Press, 1999, pp. 13–. [Online]. Available: http://dl.acm.org/citation.cfm?id=781995.782008

[8] C. Unkel and M. S. Lam, "Automatic inference of stationary fields: a generalization of java's final fields," *SIGPLAN Not.*, vol. 43, no. 1, pp. 183–195, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1328897.1328463

[9] T. Kotzmann and H. Mössenböck, "Escape analysis in the context of dynamic compilation and deoptimization," in *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, ser. VEE '05. New York, NY, USA: ACM, 2005, pp. 111–120. [Online]. Available: http://doi.acm.org/10.1145/1064979.1064996

[10] A. Einarsson and J. D. Nielsen. (2008) A survivor's guide to java program analysis with soot. [Online]. Available: http://www.brics.dk/SootGuide/sootsurvivorsguide.pdf